

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems

ATMOS 2019, September 12–13, 2019, Munich, Germany

Edited by

Valentina Cacchiani

Alberto Marchetti-Spaccamela



Editors

Valentina Cacchiani

DEI, University of Bologna, Italy
valentina.cacchiani@unibo.it

Alberto Marchetti-Spaccamela

Sapienza University of Rome, Italy
alberto@diag.uniroma1.it

ACM Classification 2012

Theory of computation → Design and analysis of algorithms; Mathematics of computing → Discrete mathematics; Mathematics of computing → Combinatorics; Mathematics of computing → Mathematical optimization; Mathematics of computing → Graph theory; Applied computing → Transportation

ISBN 978-3-95977-128-3

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-128-3>.

Publication date

November, 2019

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 <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0):
<https://creativecommons.org/licenses/by/3.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/OASlcs.ATMOS.2019.0

OASlcs – OpenAccess Series in Informatics

OASlcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASlcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 1868-8969

<https://www.dagstuhl.de/oasics>

■ Contents

Preface	
<i>Valentina Cacchiani and Alberto Marchetti-Spaccamela</i>	0:vii

Railway Optimization

A Cut Separation Approach for the Rolling Stock Rotation Problem with Vehicle Maintenance	
<i>Boris Grimm, Ralf Borndörfer, Markus Reuther, and Thomas Schlechte</i>	1:1–1:12
New Perspectives on PESP: <i>T</i> -Partitions and Separators	
<i>Niels Lindner and Christian Liebchen</i>	2:1–2:18
On Sorting with a Network of Two Stacks	
<i>Matúš Mihalák and Marc Pont</i>	3:1–3:12

Robust Optimization

Routing in Stochastic Public Transit Networks	
<i>Barbara Geissmann and Lukas Gianinazzi</i>	4:1–4:18
Robust Network Capacity Expansion with Non-Linear Costs	
<i>Francis Garuba, Marc Goerigk, and Peter Jacko</i>	5:1–5:13

Delay Management

The Trickle-In Effect: Modeling Passenger Behavior in Delay Management	
<i>Anita Schöbel, Julius Pätzold, and Jörg P. Müller</i>	6:1–6:15
Vehicle Capacity-Aware Rerouting of Passengers in Delay Management	
<i>Matthias Müller-Hannemann, Ralf Rückert, and Sebastian S. Schmidt</i>	7:1–7:14

Shortest Paths

A Priori Search Space Pruning in the Flight Planning Problem	
<i>Adam Schienle, Pedro Maristany, and Marco Blanco</i>	8:1–8:14
Exploiting Amorphous Data Parallelism to Speed-Up Massive Time-Dependent Shortest-Path Computations	
<i>Spyros Kontogiannis, Anastasios Papadopoulos, Andreas Paraskevopoulos, and Christos Zaroliagis</i>	9:1–9:18
More Hierarchy in Route Planning Using Edge Hierarchies	
<i>Demian Hesse and Peter Sanders</i>	10:1–10:14

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Routing Problems

Maximizing the Number of Rides Served for Dial-a-Ride <i>Barbara M. Anthony, Ricky Birnbaum, Sara Boyd, Ananya Christman, Christine Chung, Patrick Davis, Jigar Dhimar, and David Yuen</i>	11:1–11:15
A Graph- and Monoid-Based Framework for Price-Sensitive Routing in Local Public Transportation Networks <i>Ricardo Euler and Ralf Borndörfer</i>	12:1–12:15
Mode Personalization in Trip-Based Transit Routing <i>Vassilissa Lehoux and Darko Drakulic</i>	13:1–13:15

ATMOS’19 Best Paper Award

An Asymptotically Optimal Approximation Algorithm for the Travelling Car Renter Problem <i>Lehilton L. C. Pedrosa, Greis Y. O. Quesquén, and Rafael C. S. Schouery</i>	14:1–14:15
--	------------

■ Preface

Running and optimizing transportation systems give rise to very complex and large-scale 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 19th ATMOS symposium (ATMOS'19) was held in connection with ALGO'19 and hosted by Technische Universität München in Munich, Germany, on September 12-13, 2019. Topics of interest 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 modelling 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, and traffic guidance. Of particular interest were papers applying and advancing techniques like 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, and simulation tools.

All submissions were reviewed by at least two referees and most of them by three members of the program committee, and judged on originality, technical quality, and relevance to the topics of the symposium. Based on the reviews, the program committee selected fourteen submissions to be presented at the symposium, which are collected in this volume. Together, they quite impressively demonstrate the range of applicability of algorithmic optimization to transportation problems in a wide sense. In addition, Dorothea Wagner kindly agreed to complement the program with an invited talk on *Traffic Assignment in Transportation Networks*.

Based on the program committee's reviews, Lehilton L. C. Pedrosa, Greis Yvet Oropeza Quesquén and Rafael Schouery won the Best Paper Award of ATMOS'19 with their paper *An Asymptotically Optimal Approximation Algorithm for the Travelling Car Renter Problem*.

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'19, all the authors who submitted papers, Dorothea Wagner for accepting our invitation to present an invited talk, the members of the Program Committee and the additional reviewers for their valuable work in selecting the papers appearing in this volume, and the local organizers for hosting the symposium as part of ALGO'19. 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'19 in its OASICS series.

August 2019

Valentina Cacchiani
Alberto Marchetti-Spaccamela

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ Organization

Program Committee

Valentina Cacchiani (co-Chair)	University of Bologna, Italy
Julian Dibbelt	Apple Inc., United States
Giuseppe Italiano	LUISS, Rome, Italy
Natalia Kliewer	Freie University of Berlin, Germany
Spyros Kontogiannis	University of Ioannina, Greece
Jesper Larsen	Technical University of Denmark, Denmark
Marco Laumanns	Bestmile SA, Switzerland
Alberto Marchetti-Spaccamela (co-Chair)	Sapienza University of Rome, Italy
Juan Mesa	University of Seville, Spain
Matúš Mihalák	Maastricht University, The Netherlands
Matthias Müller-Hannemann	University of Halle-Wittenberg, Germany
Grammati Pantziou	University of Western Attica, Athens, Greece
Marie Schmidt	Erasmus University, Rotterdam, The Netherlands
Sebastian Stiller	Technical University Braunschweig, Germany
Sabine Storandt	University of Würzburg, Germany

Steering Committee

Alberto Marchetti-Spaccamela	Sapienza University of Rome, Italy
Anita Schöbel	Technical University of Kaiserslautern, Germany and Fraunhofer ITWM, Kaiserslautern, Germany
Dorothea Wagner	Karlsruhe Institute of Technology (KIT), Germany
Christos Zaroliagis (Chair)	University of Patras, Greece

Local Organizing Committee

Susanne Albers (Chair)	Technical University of Munich, Germany
Ernst Bayer	Technical University of Munich, Germany
Gabriele Doblander	Technical University of Munich, Germany



A Cut Separation Approach for the Rolling Stock Rotation Problem with Vehicle Maintenance

Boris Grimm

Zuse Institute Berlin, Germany

<http://www.zib.de>

grimm@zib.de

Ralf Borndörfer

Zuse Institute Berlin, Germany

<http://www.zib.de>

borndorfer@zib.de

Markus Reuther

LBW Optimization GmbH, Berlin, Germany

<http://www.lbw-optimization.com>

reuther@lbw-optimization.com

Thomas Schlechte

LBW Optimization GmbH, Berlin, Germany

<http://www.lbw-optimization.com>

schlechte@lbw-optimization.com

Abstract

For providing railway services the company's railway rolling stock is one if not the most important ingredient. It decides about the number of passenger or cargo trips the company can offer, about the quality a passenger experiences the train ride and it is often related to the image of the company itself. Thus, it is highly desired to have the available rolling stock in the best shape possible. Moreover, in many countries, as Germany where our industrial partner DB Fernverkehr AG (DBF) is located, laws enforce regular vehicle inspections to ensure the safety of the passengers. This leads to rolling stock optimization problems with complex rules for vehicle maintenance. This problem is well studied in the literature for example see [8, 9], or [5] for applications including vehicle maintenance. The contribution of this paper is a new algorithmic approach to solve the Rolling Stock Rotation Problem for the ICE high speed train fleet of DBF with included vehicle maintenance. It is based on a relaxation of a mixed integer linear programming model with an iterative cut generation to enforce the feasibility of a solution of the relaxation in the solution space of the original problem. The resulting mixed integer linear programming model is based on a hypergraph approach presented in [3]. The new approach is tested on real world instances modeling different scenarios for the ICE high speed train network in Germany and compared to the approaches of [10] that are in operation at DB Fernverkehr AG. The approach shows a significant reduction of the run time to produce solutions with comparable or even better objective function values.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Railway Operations Research, Integer Programming, Infeasible Path Cuts, Cut Separation, Rolling Stock Rotation Problem

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.1

Funding This work has been supported by the Research Campus MODAL Mathematical Optimization and Data Analysis Laboratories funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM). All responsibility for the content of this publication is assumed by the authors.



© Boris Grimm, Ralf Borndörfer, Markus Reuther, and Thomas Schlechte;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 1; pp. 1:1–1:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In a liberalized market companies that offer products or services have to compete with competitors for market shares. In an increasing number of countries the railway sector is one of these markets and becomes liberalized more and more. Thus railway companies have to face the challenging problem to offer the best product possible, i.e., punctual, reliable, fast, and comfortable train rides for a reasonable price requiring being as cost efficient as possible. This leads to a wide variety of closely connected optimization problems. Typically these problems differ in the grade of detail and scope of the planning horizon for which decisions have to be made or optimized. One of these problems is the Rolling Stock Rotation Problem (RSRP) where the offered passenger trips of the timetable have to be covered by a set of available rolling stock vehicles in a most cost efficient way. This assignment has to be made with respect to a lot of operational requirements, i.e., the assigned vehicle should not exceed the platform length along a trip's path, electrical powered vehicles require electrified tracks, or the assigned number of coaches should match the expected number of passengers. Although the railway market is liberalized and open for competitors there are many laws the companies have to comply with. Some of them rule mandatory maintenance checks the rolling stock fleets have to pass to be able to transport passengers. A regular vehicle maintenance scheme has also direct and indirect effects on the value of the train journey a customer perceives. They result in a more reliable level of services and in a better shape of the vehicles which is directly recognized by the passengers and linked to the company's image. In Germany, at DB Fernverkehr AG (DBF) our industrial partner, exists a complex schedule of increasing maintenance schedules beginning with short checks of parts of the vehicle and ending in a more or less complete reassembly of the vehicle. Thus, integrating maintenance constraints in the rolling stock rotation problem is a natural choice. The main contribution of this paper is a novel solution approach to the rolling stock rotation problem with vehicle maintenance that differs from the one currently in operation at DBF. It relies completely on infeasible path cuts to take care of maintenance constraints instead of modelling a linked resource flow as it is done in the current approach at DBF. Adding these resource constraints to the arc based RSRP model used at DBF results in a significant increase of problem complexity and run time thus sophisticated algorithms that reduce that increase are very welcome at DBF. Another advantage of the new approach is that it produces several rather different incumbent solutions that are close to optimality which is a feature that planners like. The drawback of this feature and a disadvantage of the approach is that there is more a step wise than a monotone improvement of the solution quality during the solution process.

Rolling stock rotation problems were studied extensively under different names, in various level of detail, and with varying focus in literature the last decades. Since the focus of this paper is on vehicle maintenance we restrict the literature review to papers that consider vehicle maintenance rules to some extend. One of the earliest works that apply to this was done by [5] where locomotives and cars were assigned to passenger trains for scenarios of VIA Rail in Canada. Cyclic timetables were covered with detailed schedules. Vehicle maintenance was considered by a time discretization approach to schedule the maintenance service stops. In [8, 9] the authors proposed solution approaches to re-optimize vehicle schedules for so called urgent trains that require maintenance services within the next 1 to 3 days. Mixed integer programming models based on multi commodity flows are used to tackle the problems for real world scenarios of the Dutch railway operator NS Reizigers. A heuristic solution approach based on an integer linear programming formulation for the Rolling Stock Rotation Problem with integrated optimization of seat capacity of the assigned

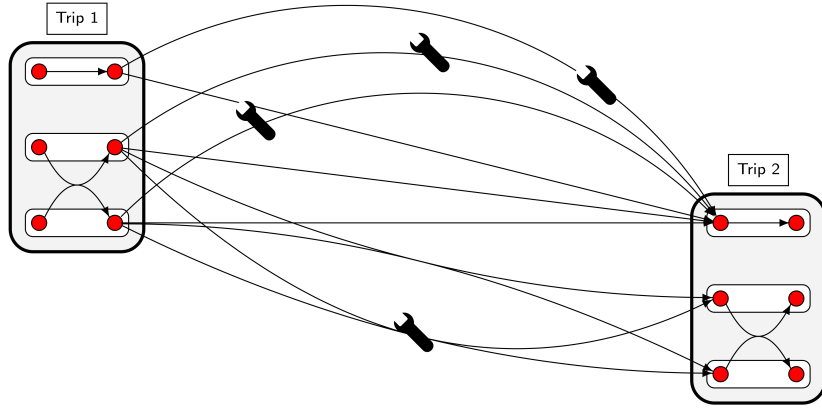
vehicle configurations is presented in [4]. Maintenance rules were added in a mileage based fashion and the approach is evaluated on scenarios of a regional train operator from Italy. The authors of [6] report a significant reduction of deadheading trips, respectively empty train runs, for their solutions computed by a mixed integer linear program using a commercial MILP-solver to find maintenance feasible Hamiltonian cycles in cyclic network wide instances of Trenitalia. The same approach was later used by [13] to compute maintenance feasible rosters for a case study in Japan. In [11] a nice overview of the research done in this field is presented. Moreover the authors describe heuristic solution approaches to solve the RSRP with integrated optimization of seat capacity for the DSB S-tog network in Copenhagen. Compared to the operated rolling stock schedules the heuristically constructed solutions were proven to be economically more attractive. A solution approach for a re-scheduling version of the RSRP is presented in [12]. After a disruption has happened re-optimized vehicle schedules have to be computed that have to consider maintenance appointments of the effected trains to be feasible. The authors evaluate their mixed integer programming formulation on instances of the Netherlands Railways. A path based branch-and-price algorithm to solve a re-scheduling variant of the RSRP is presented in [7]. In this approach maintenance constraints could be applied naturally to the vehicles due to the model's path based structure. The algorithm is designed for applications at on DSB S-tog network in Copenhagen. All this shows that algorithmic approaches to solve rolling stock scheduling problems with maintenance rules for real world applications is a growing area of interest for the scientific community as well as for the railway industry.

The paper is organized as follows, we define the Rolling Stock Rotation Problem with vehicle maintenance the way it is addressed in this paper in Section 2. This is followed by a solution approach for this problem based on an iterative cut separation procedure using infeasible path constraints in Section 3.2. We evaluate the performance of the presented approach in Section 4 with a comparison to the actual solution approach that is in operation at DBF. Finally, we conclude our results in Section 5.

2 Problem Definition

In this section we consider the *Rolling Stock Rotation Problem* as it is modeled in [3] and refer to the paper for additional technical details. In the following we shortly recapitulate the main modeling ideas.

Let T be the set of all passenger trips of a given timetable and V be a set of *nodes* representing departure and arrival events of dedicated vehicles operating passenger trips of T . Trips that could be operated with two or more vehicles have the appropriate number of arrival and departure nodes. Let further $A \subseteq V \times V$ be a set of directed standard arcs, and $H \subseteq 2^A$ a set of *hyperarcs*. Thus, a hyperarc $h \in H$ is a set of standard arcs and includes always an equal number of tail and head nodes, i.e., arrival and departure nodes. A hyperarc $h \in H$ covers $t \in T$ if each standard arc $a \in h$ represents an arc between departure and arrival of t . Each of the standard arcs a represents an individual vehicle that is required to operate t as part of the chosen vehicle configuration the hyperarc models. We define the set of all hyperarcs that cover $t \in T$ by $H(t) \subseteq H$. By defining hyperarcs appropriately, vehicle composition rules and regularity aspects can be directly handled by the model. In more detail for a single trip there are multiple different hyperarcs to chose from with different operational costs, i.e., for a trip that could be operated with one or two vehicles there exist hyperarcs that contain one, respectively, two directed arcs and thus a larger cost coefficient if two arcs, respectively, vehicles are involved. Moreover there are



■ **Figure 1** An example of hyperarcs to model two trips and maintenance services between them.

hyperarcs between two trips containing more than one directed arc if the vehicle composition does not change between the two trips. If the time between two trips is large enough to couple or decouple vehicles there are hyperarcs to model this as well. At DBF regularity is an important aspect during optimization so there are additional hyperarcs that contain hyperarcs that model exactly the same trip that is operated on multiple days of the week with exactly the same vehicle configuration. This regularity hyperarc is slightly cheaper than choosing the individual hyperarcs. Hyperarcs that contain arrival and departure nodes of different trips are used to model deadhead trips between the operation of two trips. With this construction it is possible to set up a cost function $c : H \mapsto \mathbb{Q}^+$ for the hyperarcs that includes a wide spectrum of different operational costs that have to be addressed by the model, i.e., costs for energy consumption, vehicle usage, coupling and combining of train units, short turn penalties, or even artificial cost for modelling regular vehicle movements. The RSRP *hypergraph* is denoted by $G = (V, A, H)$. We define sets of hyperarcs coming into and going out of $v \in V$ in the RSRP hypergraph G as $H(v)^- := \{h \in H \mid \exists a \in h : a = (u, v)\}$ and $H(v)^+ := \{h \in H \mid \exists a \in h : a = (v, w)\}$, respectively. One major challenge in rolling stock planning and optimization is vehicle maintenance. At DB Fernverkehr AG there are several different maintenance rules for the different ICE high speed train fleets that all have to be considered. In this paper we focus on maintenance rules that are based on the accumulated kilometers a vehicle is operated between two maintenance services. We denote the upper bound on the total mileage between two maintenance services by R . Maintenance services could only be performed at special maintenance locations $m \in M$. The kilometers a vehicle is moved during an operation modelled by a chosen hyperarc is given by a function $r : H \mapsto [0, R]$. By $|h|$ the number of standard arcs $a \in h$ required to model h is defined. Thus $|h| \cdot r(h)$ gives the aggregated kilometers of all vehicles modelled by h . This includes necessary deadhead trips to reach maintenance facilities or turn around trips to change the orientation of the vehicle. To model maintenance services in the RSRP *hypergraph* additional maintenance service hyperarcs were defined for each pair of trips if it is possible to visit a maintenance facility and perform a service between the operation of the two trips. The cost for the additional deadhead trip and the cost for the maintenance service is added to the cost of the hyperarc. In this sense, a cycle in G is called maintenance feasible, if and only if the accumulated kilometers of all trips and deadhead trips along the sub-paths between each two hyperarcs with a maintenance service of a cycle is smaller or equal than R .

In Figure 1 two trips are shown that could be operated by either one or two (red) vehicles. Thus, there is a red node for each arrival and departure event of a single vehicle. These are connected by either a single arc hyperarc or a double arc hyperarc to model the operation with one, respectively, two vehicles. Between the two trips there are hyperarcs that model maintenance events. These are marked by the wrench symbol containing single or double vehicle configurations. Additionally there are single arc hyperarcs to model single vehicle transitions between the two trips. The double arc hyperarc models a transition without changing the vehicle configuration.

► **Definition 1.** *Let G be a graph based hypergraph, c its associated cost function, and r a maintenance resource function with its upper bound R . The **Rolling Stock Rotation Problem (RSRP)** is to find a cost minimal, maintenance feasible set of hyperarcs $H_x \subseteq H$ such that H_x is a set of cycles that covers all trips $t \in T$ by a hyperarc $h \in H_x$.*

3 Solution Approaches to the RSRP

In the following section, we show how the RSRP is modelled and solved in the current application of RotOR, which is the optimization software used at DBF. After that the new approach, which is also implemented in RotOR, using infeasible path constraints in order to iteratively separate maintenance infeasible sub-paths, is presented.

3.1 Resource Flow Based Solution Approach

The current solution approach to the RSRP implemented in RotOR solves a mixed integer programming formulation of a hyperflow model with linked resource flow to model the vehicle maintenances. All details and sophisticated algorithms to solve this model can be found in [3, 10]. Using a binary decision variable x_h for each hyperarc and continuous variables w_a for the linked resource flow, the resource flow based MILP-formulation of the RSRP can be stated follows:

$$\min \sum_{h \in H} c_h x_h, \quad (\text{Flow})$$

$$s.t. \quad \sum_{h \in H(t)} x_h = 1 \quad \forall t \in T, \quad (1)$$

$$\sum_{h \in H(v)^-} x_h = \sum_{h \in H(v)^+} x_h \quad \forall v \in V, \quad (2)$$

$$w_a \leq \sum_{h \in H(a)} R x_h \quad \forall a \in A, \quad (3)$$

$$\sum_{a \in A(v)^+} w_a - \sum_{a \in A(v)^-} w_a = \sum_{h \in H(v)^+} r(h) x_h \quad \forall v \in V, \quad (4)$$

$$\sum_{a \in A(m)^+} w_a = \sum_{h \in H(m)^+} r(h) x_h \quad \forall m \in M, \quad (5)$$

$$w_a \in [0, R] \subset \mathbb{Q}_+ \quad \forall a \in A, \quad (6)$$

$$x_h \in \{0, 1\} \quad \forall h \in H. \quad (7)$$

The objective function of (Flow) minimizes the sum of the operational cost of all chosen hyperarcs. This includes all cost for operating a trip, deadhead trips, performing maintenances, and costs to penalize irregularities. The first constraints (1) ensure the covering of each

trip. Equations (2) take care about the (hyper)flow conservation. The following four sets of constraints deal with the vehicle maintenance. First, the maintenance variables w were coupled to the hyperarc variables allowing only those to be used for which a hyperarc was chosen. Followed by equations (4) which ensure the correct summation of the maintenance resource consumption. The constraints (5) state the possibility to reset the resource flow at maintenance service locations. Finally, the variable domains are given by (6) and (7). The presence of the constraints (3)-(6) makes the model way more difficult to solve as it implicitly implies a tracing of each individual vehicle while the other parts of the model can be seen as a vehicle type or fleet based formulation. This is one of the reasons to be interested in novel powerful algorithmic approaches to solve these kinds of problems.

3.2 Infeasible Path Cut Separation Approach

Besides the mentioned reason for a different approach to tackle the RSRP with included vehicle maintenance, the fact that maintenance service locations are often closely located to overnight parking depots is another one. This fact leads to the situation that solutions of the RSRP without consideration of the maintenance are often trivially maintenance feasible, respectively could be easily made maintenance feasible by replacing overnight parking hyperarcs with maintenance hyperarcs. Therefore we developed a new integer programming model that replaces the continuous resource flow by a rough bound on the overall resource consumption and a set of infeasible path constraints to forbid maintenance infeasible sub-paths in the chosen cycles. This class of cuts is well known and studied in the (asymmetric) travelling salesman community as for example in [1]. However, we are not aware of any publication applying this technique to rolling stock scheduling. Thus, the following integer program relies completely on the binary variables for choosing hyperarcs to be part of the solution. To set up the model, we define by P the set of all maintenance infeasible sub-paths in the underlying directed graph $D = (V, A)$ of the hypergraph G .

$$\min \sum_{h \in H} c_h x_h, \quad (\text{Cut})$$

$$s.t. \quad \sum_{h \in H(t)} x_h = 1 \quad \forall t \in T, \quad (8)$$

$$\sum_{h \in H(v)^-} x_h = \sum_{h \in H(v)^+} x_h \quad \forall v \in V, \quad (9)$$

$$\sum_{h \in H \setminus M} |h| r(h) x_h \leq \sum_{h \in M} |h| R x_h, \quad (10)$$

$$\sum_{h \in P_i} x_h \leq |P_i| - 1 \quad \forall P_i \in P, \quad (11)$$

$$x_h \in \{0, 1\} \quad \forall h \in H. \quad (12)$$

The objective function of (Cut) and the constraints (8), (9), and (12) are completely identical to the ones in the (Flow) formulation and model all technical aspects of the problem with the exception of the vehicle maintenance rules. These rules are implied by the other two sets of constraints. The first set (10) forces the solution to contain a sufficient number of maintenance service hyperarcs. Utilizing that the total resource consumption of the vehicles, i.e., the left hand side of constraints (10), is bounded from above by the number of maintenance arcs chosen times the product of the upper bound of the resource consumption and the number of maintained vehicles modelled by the chosen hyperarc. The

obvious argument for this becomes clear if we divide (10) by R . Note that the total resource consumption of the vehicles is not fixed a priori due to deadhead trips. The second set (11) then forbids to include maintenance infeasible sub-paths into the cycles of hyperarcs of the solution.

Trivially, one critical aspect of this model formulation is that it might contain an exponential number of infeasible path constraints (11). Therefore we solve this model by adding the infeasible path cuts dynamically to the model during the solution process. Nevertheless, it is easy to see that for the formulation containing all infeasible path constraints the following lemma holds.

► **Lemma 2.** *Let X^{Flow} , X^{Cut} be the sets of all integer feasible solutions of Cut, respectively Flow restricted to the x variables. It holds*

$$\text{conv}(X^{Flow}) = \text{conv}(X^{Cut}).$$

3.2.1 Dynamic Infeasible Path Constraint Separation

As mentioned before, the idea behind the algorithm is to generate the infeasible path constraints 11, respectively to separate maintenance infeasible solutions of the Cut approach dynamically. Algorithm 1 provides the respective pseudo code. It works as follow: The algorithm starts with a solution H_x of the Cut formulation, without any infeasible path constraint so far, computed by a commercial mixed integer solver with a limit on the optimal IP tolerance of $\frac{\epsilon}{2}$. We denote the problem Cut_P with $P := \emptyset$ in the following as *maintenance relaxation* of the original problem. It gives the algorithm a feasible solution to the RSRP without considering the maintenance constraints and therefore a valid lower bound on the objective function (trivially, since each maintenance feasible solution is still contained in the solution space). After that, the cycles of the directed arcs contained in the chosen hyperarc variables were constructed. By tracking each cycle once, beginning at an arbitrarily chosen maintenance arc, the algorithm checks the maintenance feasibility of the solution by summing up the resource consumption along the cycle resetting it every time a maintenance arc is passed. If the sum of the aggregated mileage of the arcs exceeds the upper bound R of the resource, the chosen cycle is proven to be infeasible and a valid infeasible (sub-)path constraint is generated automatically by the set of hyperarcs passed since the last maintenance arc, i.e., including the arc itself. We denote this set by $P_i \subseteq H$. In a maintenance feasible solution it is not possible to chose all $|P_i|$ hyperarcs from this set. The algorithm collects all of these constraints, denoted by \hat{P} that could be generated from H_x . If no infeasible path constraint is generated, the solution is maintenance feasible and optimal since the objective function value equals the lower bound of the maintenance relaxation. In the opposite case where H_x was proven to be maintenance infeasible a neighborhood search is applied to substitute hyperarcs of the solution with their counterparts that include additional maintenance services to construct a maintenance feasible solution \hat{H}_x . Therefore we define two hyperarcs to be maintenance equivalent by the definition given in 3. In a nutshell, two hyperarcs are maintenance equivalent, denoted by $h \simeq g$, if they model the same hyperarc with and without performing a maintenance service in between. With this definition we can set up the so called *Maintenance Assignment Model (MAM)* as shown in MAM. This model is then solved by a commercial mixed integer solver. If it is feasible the algorithm constructs a solution \hat{H}_x that is feasible to Cut_P and Cut as well. This solution is then checked whether its quality in sense of the gap between its objective function value and the best known lower bound is

1:8 A Cut Separation Approach for the RSRP

small enough, i.e., smaller than ϵ . If this is the case the algorithm terminates returning \hat{H}_x as found solution. Otherwise Cut_P , respectively P is updated by $P = P \cup \hat{P}$. The algorithm then restarts the path separation loop by solving Cut_P until a fixed number of iterations are passed or a sufficiently good solution was found.

```

1 Input : Hypergraph  $G$ , resource function  $r : H \mapsto \mathbb{Q}$ , resource limit  $R$ 
2 Output: Maintenance feasible solution  $x \subseteq H$ 
3 {
4    $x := \emptyset$ 
5    $i := 0$ ;  $P := \emptyset$ 
6   do
7   {
8      $H_x := \text{MILPSolve}(Cut_P)$ ;
9      $\hat{P} := \text{generateInfeasiblePathCuts}(H_x)$ ;
10     $P := P \cup \hat{P}$ ; % update infeasible path cut set
11    if ( $\hat{P} = \emptyset$  or  $H_x = \emptyset$ )
12    {
13      return  $H_x$  % maintenance feasible and optimal or infeasible
14    }
15    else
16    {
17       $\hat{H}_x := \text{solveMAM}(H_x)$ ;
18      if ( $\hat{H}_x \neq \emptyset$ )
19      {
20        %Solution  $\hat{H}_x$  is a maintenance feasible solution for  $Cut$ 
21         $x := \hat{H}_x$ 
22      }
23    }
24     $i++$ ;
25  }
26  while(  $\frac{c(x) - c(H_x)}{c(H_x)} < \epsilon$  or  $i < I$  )
27
28  return  $x$ 
29 }
```

■ **Algorithm 1** Infeasible Path Constraint Separation Algorithm.

3.2.2 The Maintenance Assignment Model

To perform the neighborhood search for a maintenance infeasible solution H_x of the Cut_P model in the cut separation loop, the mixed integer programming model defined in MAM is solved. To set up the model, we formally define the maintenance equivalence relation for two hyperarcs as follows.

► **Definition 3.** *Two hyperarcs $h, g \in G$ are maintenance equivalent ($h \simeq g$) if and only if $A(h) = A(g)$.*

The MAM-model contains a binary decision variable x_h for each hyperarc $h \in H$ that is either included ($h \in H_x$) or maintenance equivalent to a hyperarc included in the actual solution ($h \simeq g \in H_x$). It also contains a continuous resource flow variable $w_v \in [0, R]$ for each node of V that is part of a chosen hyperarc $h \in H_x$.

$$\min \sum_{h \simeq h_x \in H_x} c_h x_h, \quad (\text{MAM})$$

$$s.t. : \sum_{h \simeq h_x} x_h = 1 \quad \forall h_x \in H_x, \quad (13)$$

$$w_a + \sum_{h \simeq h_x} r(h) x_h \leq w_b \quad \forall (a, b) \in h_x, h_x \in H_x, \quad (14)$$

$$x_h \in \{0, 1\} \quad \forall h \in H_x, \quad (15)$$

$$w_a \in [0, R] \quad \forall (a, b) \in h, h \in H_x. \quad (16)$$

The objective function of (MAM) minimizes the sum of the operational cost of all chosen hyperarcs in exactly the same way it is done in (Flow) or (Cut). The constraints (13) define that either a hyperarc that does or does not perform a maintenance service for each arc contained in a cycle of the solution H_x is chosen. If a hyperarc that contains a maintenance service is chosen, the respective constraint (14) ensures that the associated values for the w values are reset. In the opposite case they ensure the correct propagation of the resource consumption values to the next w -variables along the cycle. The last two sets of constraints (15) and (16) define the variable domains. We remark that solutions of this model may contain an increased number of maintenance arcs than the original solution H_x . But, if the model is feasible the computed solution is maintenance feasible and therefore a feasible solution for *Cut*. For our practical instances these models are very small and easy to solve by a commercial state of the art mixed integer solver.

4 Computational Results

This section presents the computational results for the presented solution approaches on a set of real world instances of DB Fernverkehr AG. These instances contain different scenarios for rolling stock rotation problems for the ICE high speed train vehicles operated by DBF. They differ in the number of contained timetable trips, operated fleets, and characteristics of the maintenance rules. All instances contain between 200 and 400 timetabled trips and a maximum of two coupled vehicles to operate a trip. All computations were performed on an Intel® Xeon(R) E3-1245 v5 @ 3.50GHz CPU with eight cores and Gurobi 8.1 as LP and sub-MILP solver.

Table 1 compares the solution process of the LP-relaxation of Flow and Cut, both implemented in RotOR. The LP-relaxation is solved with an algorithm called Coarse-2-Fine Column Generation which is described in detail in [2, 10] and out of scope of the paper to be explained in detail here. At this stage the only difference between the two solution approaches is the set of constraints that is given to the solver, i.e, the LP model formulation of (Cut) with no infeasible path cuts respectively the LP model formulation of (Flow). The first column of Table 1 identifies the instance while the second column shows the total number of hyperarcs required to model the instance. The following two blocks of three columns each headlined with Cut, respectively Flow show the solution characteristics of the associated approach. In detail, *Obj.* columns show the objective function values of the two approaches (times a factor of 10^{-z} , $z \in \mathbb{N}$), *CPU* columns give the total computation time of each approach in seconds, and the columns headlined with *Columns* mark the number of generated columns, respectively variables required to solve the LP-relaxation. The Cut approach shows a significant speed up and a lower number of generated columns for each of the instances, but for the price of a slightly weaker LP-bound.

1:10 A Cut Separation Approach for the RSRP

■ **Table 1** Computational results for the LP-relaxation of Cut and Flow.

Id	$ H $	Obj.	Cut		Flow		
			CPU(s)	Columns	Obj.	CPU(s)	Columns
1	229292	7062	5	16683	7064	16	30487
2	116206	6598	7	10881	6600	14	19496
3	1813512	7853	15	56118	7779	55	109303
4	1686124	8170	20	61276	8164	52	82726
5	275356	9340	8	16301	9288	13	21959
6	275356	8886	6	15009	8793	12	22359
7	363132	8976	9	18898	8994	13	25052
8	1452528	9011	30	76346	9007	47	80599
9	471056	10907	11	20883	10868	43	42830
10	471056	11229	15	23027	11216	40	40297
11	229634	7181	13	19829	7181	19	30109
12	226340	7204	6	14661	7212	17	31758
13	229634	7120	7	19501	7104	18	32360
14	226340	7044	5	16631	6977	18	33647

■ **Table 2** Computational results for Cut, Flow solved with RotOR, and Flow solved with Gurobi.

Id	$ H $	Cut		RotOR		Gurobi	
		Obj.	CPU(s)	Obj.	CPU(s)	Obj.	CPU(s)
1	229292	7087	301	7105	45	7090	48
2	116206	6622	6	6619	1	6610	17
3	1813512	7879	22	7937	190	7866	84
4	1686124	8220	78	8230	181	8221	352
5	275356	9405	9	9439	146	9401	18
6	275356	8961	17	8981	199	8939	195
7	363132	9079	19	9055	122	9048	145
8	1452528	9068	292	9091	255	9038	3390
9	471056	11099	23	11006	210	10977	327
10	471056	11350	102	11388	50	11336	47
11	229634	7182	7	7182	20	7182	20
12	226340	7212	6	7212	20	7212	18
13	229634	7183	40	7146	112	7147	26
14	226340	7081	16	7059	72	7116	46
\sum			939		1635		4734
geometric mean			27.68		81.9		77.5

In the following three different solution approaches to compute an integer solution for the RSRP were compared. The first approach is the Infeasible Path Constraint Separation Algorithm described in Section 3.2.1. The second one is the currently used algorithm in RotOR at DBF which is explained in detail in [2, 10]. In a nutshell, this algorithm makes use of a heuristic start solution and sophisticated problem specific branching rules fixing different types of variables at different times of the solution process to solve the remaining sub-MILPs by Gurobi. The third approach is the default version of Gurobi to solve the Flow-MILP-formulation. The three different approaches to solve the integer formulation of Cut or Flow were restricted to the variables generated during the column generation process. The numbers of generated variables are shown in Table 1. The RotOR and the Gurobi approach solve exactly the same MIP model containing the variables generated for the Flow model.

Table 2 shows the characteristics of the solutions computed by the three different approaches. The first column identifies the instance while the second column shows the total number of hyperarcs required to model the instance. The next two columns show the objective function values and computation times for each instance using the Infeasible Path Constraint Separation Approach, followed by the same values ordered in two columns for RotOR’s solution approach. Finally the objective function values and computation times of the default version of Gurobi are shown in the last two columns. For all solution approaches a limit of 1% on the gap between the best known upper and lower bounds was applied. All objective function values are shown times a factor of 10^{-z} , $z \in \mathbb{N}$. The values for the objective function values of all three approaches show that each of them is able to find solutions within the desired bounds. Although, struggling on two instances the Infeasible Path Constraint Separation Algorithm shows promising run times for the set of instances, especially for the largest instances 3, 4 and 8. The two instances 1 and 13 are instances with a rather small upper bound on the maintenance resource which leads to an increased number of maintenances in the final IP-solution compared to the LP-relaxation.

5 Conclusion

In this paper we presented an optimization algorithm based on infeasible path constraints to deal with the Rolling Stock Rotation Problem with integrated vehicle maintenance. The algorithm is capable to deal with practical sized real world instances. It shows promising results in terms of solution quality and computation time. For future research it might be interesting to generate a set of cuts in beforehand of the solution process or to couple cut generation to certain aspects of the different maintenance rules.

References

- 1 Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. A polyhedral study of the asymmetric traveling salesman problem with time windows. *Networks: An International Journal*, 36(2):69–79, 2000.
- 2 Ralf Borndörfer, Markus Reuther, and Thomas Schlechte. A Coarse-To-Fine Approach to the Railway Rolling Stock Rotation Problem. In *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 42, pages 79–91, 2014. doi:10.4230/OASIcs.ATMOS.2014.79.
- 3 Ralf Borndörfer, Markus Reuther, Thomas Schlechte, Kerstin Waas, and Steffen Weider. Integrated Optimization of Rolling Stock Rotations for Intercity Railways. *Transportation Science*, 50(3):863–877, 2015. doi:10.1287/trsc.2015.0633.

- 4 Valentina Cacchiani, Alberto Caprara, and Paolo Toth. Solving a real-world train-unit assignment problem. *Mathematical Programming*, 124(1):207–231, July 2010. doi:10.1007/s10107-010-0361-y.
- 5 Jean-François Cordeau, François Soumis, and Jacques Desrosiers. Simultaneous Assignment of Locomotives and Cars to Passenger Trains. *Oper. Res.*, 49:531–548, July 2001. doi:10.1287/opre.49.4.531.11226.
- 6 Giovanni Luca Giacco, Andrea D’Ariano, and Dario Pacciarelli. Rolling stock rostering optimization under maintenance constraints. *Journal of Intelligent Transportation Systems*, 18(1):95–105, 2014.
- 7 Richard M. Lusby, Jørgen Thorlund Haahr, Jesper Larsen, and David Pisinger. A Branch-and-Price algorithm for railway rolling stock rescheduling. *Transportation Research Part B: Methodological*, 99:228–250, 2017. doi:10.1016/j.trb.2017.03.003.
- 8 Gábor Maróti and Leo Kroon. Maintenance Routing for Train Units: The Transition Model. *Transportation Science*, 39:518–525, November 2005. doi:10.1287/trsc.1050.0116.
- 9 Gábor Maróti and Leo G. Kroon. Maintenance routing for train units: The interchange model. *Computers & OR*, pages 1121–1140, 2007.
- 10 Markus Reuther. *Mathematical optimization of rolling stock rotations*. PhD thesis, TU Berlin, 2017. doi:10.14279/depositonce-5865.
- 11 Per Thorlacius, Jesper Larsen, and Marco Laumanns. An integrated rolling stock planning model for the Copenhagen suburban passenger railway. *Journal of Rail Transport Planning & Management*, 5(4):240–262, 2015.
- 12 Joris C. Wagenaar, Leo G. Kroon, and Marie Schmidt. Maintenance Appointments in Railway Rolling Stock Rescheduling. *Transportation Science*, 51(4):1138–1160, 2017. doi:10.1287/trsc.2016.0701.
- 13 Takuya Shiina Jun Imaizumi Yuuta Morooka, Naoto Fukumura and Susumu Morito. Rolling Stock Rostering Optimization Based on the Model of Giacco et al.: Computational. Evaluation and Model Extensions. In *7th International Conference on railway operations modelling and Analysis (RailLille 2017)*, pages 709–725, 2017.

New Perspectives on PESP: T -Partitions and Separators

Niels Lindner

Zuse Institute Berlin (ZIB), Takustr. 7, 14195 Berlin, Germany
lindner@zib.de

Christian Liebchen

Technical University of Applied Sciences Wildau, Hochschulring 1, 15745 Wildau, Germany
liebchen@th-wildau.de

Abstract

In the planning process of public transportation companies, designing the timetable is among the core planning steps. In particular in the case of periodic (or cyclic) services, the Periodic Event Scheduling Problem (PESP) is well-established to compute high-quality periodic timetables.

We are considering algorithms for computing good solutions and dual bounds for the very basic PESP with no additional extra features as add-ons. The first of these algorithms generalizes several primal heuristics that have been proposed, such as single-node cuts and the modulo network simplex algorithm. We consider partitions of the graph, and identify so-called delay cuts as a structure that allows to generalize several previous heuristics. In particular, when no more improving delay cut can be found, we already know that the other heuristics could not improve either. This heuristic already had been proven to be useful in computational experiments [1], and we locate it in the more general concept of what we denote T -partitions.

With the second of these algorithms we propose to turn a strategy, that has been discussed in the past, upside-down: Instead of gluing together the network line-by-line in a bottom-up way, we develop a divide-and-conquer-like top-down approach to separate the initial problem into two easier subproblems such that the information loss along their cutset edges is as small as possible.

We are aware that there may be PESP instances that do not fit well the separator setting. Yet, on the RxLy-instances of PESPLib in our experimental computations, we come up with good primal solutions and dual bounds. In particular, on the largest instance (R4L4), this new separator approach, which applies a state-of-the-art solver as subroutine, is able to come up with better dual bounds than purely applying this state-of-the-art solver in the very same time.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Applied computing → Transportation; Mathematics of computing → Discrete optimization; Mathematics of computing → Integer programming

Keywords and phrases Periodic Event Scheduling Problem, Periodic Timetabling, Graph Partitioning, Graph Separators, Balanced Cuts

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.2

Funding *Niels Lindner*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – The Berlin Mathematics Research Center MATH+ (EXC-2046/1, project ID: 390685689).

Acknowledgements The authors want to thank Ralf Borndörfer for fruitful conversations.

1 Introduction

Traditionally, the planning process for public transportation companies is among the classical application areas of mathematical optimization. A very prominent general such success story had been established at Dutch railways [11]. At the borderline between service design and resource planning, timetabling is kind of in a central position of the entire planning process. This is one motivation why in the recent past there have been considered many “add-ons” to timetabling, e.g.,



© Niels Lindner and Christian Liebchen;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 2; pp. 2:1–2:18



OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- integrating decisions of line planning, sometimes even network design,
- considering the passengers' route choice as a function of the actual timetable,
- designing timetables that admit for efficient vehicle schedules and occasionally even crew schedules,
- computing delay-resistant timetables.

Nevertheless, most of these considered extensions share one limiting factor: computing efficient timetables in a core subroutine, at least.

Regarding timetables, there are several concepts around which design principles the timetable should follow, e.g., periodicity and symmetry [14]. In this paper, we are considering periodic timetables, i.e., those, in which the trips of the same line and into the same direction follow each other in a fixed time interval, which we denote the period time, or, the cycle time. In particular in Europe, these timetables are widely in use both for railways and for urban public transport.

To model periodic timetables, the Periodic Event Scheduling Problem, which had been formulated by Serafini and Ukovich [27] (see Section 2), can be considered as state-of-the-art. Notice that there are also further applications of PESP beyond periodic timetabling, such as traffic light signalling. Solution methods for PESP include (mixed) integer linear programming, constraint programming, satisfiability algorithms, as well as a couple of heuristics.

The contribution of this paper is to provide two new heuristics for computing good primal solutions for PESP instances relatively fast. On the one hand, the second heuristic does not fit for every PESP instance. On the other hand, if it fits, sometimes it can even be used to identify good dual bounds, too.

Interestingly, whereas several recent improvements to MIP performance touched on cycles within the graph model (in particular trying to improve the generally relatively weak dual bounds), both of our two heuristics deal with complementary structures, namely cutsets of a graph, in the second heuristic within the particular framework of so-called graph separators.

In Section 3, we invite the reader to think of PESP in terms of T -partitions. In particular, we introduce what we call *delay cuts* and these generalize the setting of several primal heuristics that had been considered earlier (e.g. single-node-cuts, modulo network simplex algorithm). By computing an optimal delay cut using a tailored MIP, we know that this locally optimal solution in particular is locally optimal for the other heuristics, too.

In Section 4, we propose a method to overcome some degeneracy that can sometimes be observed in a heuristic that had been dealt with in [23]. This heuristic starts, in a bottom-up manner, with optimum timetables for each line separately. Next, one combines (matches) those two clusters, between which in total find the largest weights and adjust the two separate timetables by shifting them against each other in order to synchronize the two line-clusters as good as possible. Here, sometimes it can be observed that from the moment on that one cluster becomes relatively large compared to the other clusters (still consisting of just one single directed line, in the extreme case), the heuristic degenerates simply to add – linearly – one line at a time.

This is why we are proposing to turn this procedure upside-down. At the very top level of the PESP constraint graph, we compute a separator in order to divide the instance into two essentially balanced subproblems. The two resulting PESP instances are then ideally much easier to solve to optimality. Keeping their relative structure within each of them, finally combine them to a timetable for the entire network by shifting them in a best possible way against each other. Right as in the previous approach in [23], the separator must not contain any arc that imposes a true restriction, e.g., of technical nature. Moreover, among the “free arcs”, we seek for a set of arcs (and their weights) between the two subproblems that is as small as possible, in order to lose only few information.

In Section 5, we report some computational results. We start by applying the separator heuristic from Section 4. For the subproblems, we apply the concurrent solver that has been presented recently in [1], and in which the MIP-based delay cut heuristic from Section 3 is implemented among other algorithms. In these experiments, it can be observed that in the result of some separation strategies, the two separated subproblems indeed can be solved with smaller average slack than the time-equivalent benchmark solution for the original complete problem. Unfortunately, at least on the instances that we are considering, this lead that is attained within the two subproblems turns out not to be enough to compensate the worse quality that finally appears on the arcs of the cutset that link the two subproblems when simply shifting the two pre-computed solutions of the two subproblems against each other.

2 Periodic Event Scheduling

The *Periodic Event Scheduling Problem* (PESP) is a mathematical optimization problem formulated by Serafini and Ukovich [27] that lies at the heart of periodic timetabling in public transport. The input for PESP consists of the following:

- A directed graph G with vertex set V and arc set A ,
- a period time $T \in \mathbb{N}$,
- lower and upper bounds $\ell, u \in \mathbb{Z}_{\geq 0}^A$ with $\ell \leq u$,
- weights $w \in \mathbb{Z}_{\geq 0}^A$.

We will only consider integer bounds and weights in this paper. A *periodic timetable* is a vector $\pi \in \{0, 1, \dots, T-1\}^V$. Any periodic timetable defines a *periodic slack* $y \in \mathbb{Z}_{\geq 0}^A$ by

$$y_{ij} := [\pi_j - \pi_i - \ell_{ij}]_T \quad \text{for all } ij \in A,$$

where $[\cdot]_T$ denotes the modulo T operator taking values in $[0, T)$. A periodic timetable π and its associated periodic slack y are called *feasible* if $y \leq u - \ell$ holds.

In the setting of periodic timetabling for public transport, think of a period time of, say, $T = 60$ minutes. The events correspond to either the set of arrivals or departures of trips of a certain line into a particular direction. A timetable π then assigns points in time within the period time T to each of these events. Finally, the arcs measure time distances between two adjacent events, and thus model time durations for trips, stops, headways, and many more.

Given an input as above, the *Periodic Event Scheduling Problem* is now to find a feasible periodic timetable π , in an optimization version we may in addition seek for a periodic timetable minimizing the weighted slack $\sum_{ij \in A} w_{ij} y_{ij}$.

The PESP has a natural formulation as a mixed integer linear program, namely

$$\begin{aligned} & \text{Minimize} && w^t y \\ & \text{s.t.} && y = B^t \pi - \ell + pT \\ & && 0 \leq \pi \leq T - 1, \\ & && 0 \leq y \leq u - \ell, \\ & && p \in \mathbb{Z}^A. \end{aligned}$$

Here, B^t denotes the transpose of the incidence matrix B of the directed graph G . Since B and hence B^t are totally unimodular [26, Example 19.2], we can w.l.o.g. relax π and y to be continuous variables.

Hence, a standard approach to solving PESP instances is to apply branch-and-cut procedures, as invoked by mixed integer programming solvers. To this end, several formulations and cutting planes have been presented [15, 17, 18, 19, 22]. Another solution strategy is to employ Boolean satisfiability methods [7, 6].

Exploiting the polyhedral structure of the problem, the modulo network simplex algorithm [20] is a rather fast local improving heuristic. Several methods for escaping local optima have been suggested [5]. We will unite these methods to a more global heuristic approach in Section 3.

Since the structure of public transportation networks is usually derived from lines, in the case when only few technical constraints have to be obeyed, a bottom-up matching approach has been introduced in [23, 12]. The idea is to cluster lines according to the importance of the transfers between them, increasing the number of lines as the matching heuristic proceeds. Doing so, it could happen that one cluster of lines is getting bigger and bigger and then, in fact, clustering only consists of a linear sequence in which the lines are added to the growing instance. In Section 4, in order to get several bigger subproblems that contain “most” of the information of the entire instance, we turn this approach upside-down: We develop a top-down divide-and-conquer strategy for PESP, i.e., we try to split the set of all lines into two parts of roughly the same size such that only a small amount of all transfers occurs between the parts. The idea is that on the intersection relatively few information is lost, whereas the practical tractability of the two subproblems improves significantly.

3 T -Partitions

In this section, we will present a view on periodic timetabling from the standpoint of cuts and partitions in graphs. Establishing a correspondence between periodic timetables and T -partitions, we translate several PESP strategies into the language of partitions. Finally, we present an improving heuristic for PESP in terms of maximum cuts, which subsumes several known local solving approaches in a single optimization problem. We identify the so-called delay cuts, as they have been already part of the computational framework presented in [1], as a useful device within the new concept of T -partitions.

3.1 Timetables and Partitions

Let (G, T, ℓ, u, w) be a PESP instance. Then any periodic timetable π naturally partitions the vertex set V of G into T sets, namely $\{i \in V \mid \pi_i = d\}$ for $d = 0, 1, \dots, T - 1$.

► **Definition 1.** A T -partition of a PESP instance with vertex set V and period time T is a T -tuple $\mathcal{V} = (V_0, V_1, \dots, V_{T-1})$ of pairwise disjoint subsets of V such that $\bigcup_{d=0}^{T-1} V_d = V$.

Note that the members of a T -partition might be empty. Clearly, there is a one-to-one correspondence between periodic timetables and T -partitions, identifying the sets in the T -partition of V with the preimages of the periodic timetable, when interpreted as a map $V \rightarrow \{0, \dots, T - 1\}$.

As periodic timetables can be thought of as maps taking values in the residue class group $(\mathbb{Z}/T\mathbb{Z}, +)$, there is a natural addition of timetables by componentwise addition modulo T . If π, π' are periodic timetables, we interpret π' as T -partition and obtain the sum as follows:

► **Definition 2.** Given a periodic timetable π and a T -partition $\mathcal{V} = (V_0, \dots, V_{T-1})$, define the periodic timetable $\pi^{\mathcal{V}}$ via

$$\pi_v^{\mathcal{V}} := [\pi_v + d]_T, \quad v \in V_d, \quad d = 0, \dots, T - 1.$$

We will now use T -partitions for optimizing a PESP instance. Let π^* be a timetable with minimum weighted slack. Given an initial timetable π , we can find π^* by looking for a T -partition \mathcal{V} with $\pi^{\mathcal{V}} = \pi^*$. In terms of periodic slacks on the arc set A , we have:

► **Lemma 3.** *Let π be a periodic timetable and let \mathcal{V} be a T -partition. If y and $y^\mathcal{V}$ are the periodic slacks associated to π and $\pi^\mathcal{V}$, respectively, then*

$$y_{ij}^\mathcal{V} = [y_{ij} - d + e]_T, \quad ij \in A \cap (V_d \times V_e), \quad d, e = 0, \dots, T - 1.$$

Note that since \mathcal{V} is a partition, this fixes $y_{ij}^\mathcal{V}$ for every arc $ij \in A$.

Proof. Plugging in the definitions,

$$y_{ij}^\mathcal{V} = [\pi_j^\mathcal{V} - \pi_i^\mathcal{V} + \ell_{ij}]_T = [\pi_j + e - (\pi_i + d) - \ell_{ij}]_T = [y_{ij} - d + e]_T. \quad \blacktriangleleft$$

► **Definition 4.** *Given a periodic timetable π on a PESP instance, the improvement of a T -partition \mathcal{V} is*

$$\iota(\pi, \mathcal{V}) := \sum_{d=0}^{T-1} \sum_{e=0}^{T-1} \sum_{ij \in A \cap (V_d \times V_e)} w_{ij} (y_{ij} - [y_{ij} - d + e]_T).$$

The MAXIMALLY IMPROVING T -PARTITION problem is to find for a given timetable π a T -partition \mathcal{V} such that $\iota(\pi, \mathcal{V})$ is maximum and $y^\mathcal{V} \leq u - \ell$, i.e., feasible.

► **Theorem 5.** *If π is a periodic timetable for a PESP instance I , then \mathcal{V} solves MAXIMALLY IMPROVING T -PARTITION for π if and only if $\pi^\mathcal{V}$ is an optimal solution to I .*

Proof. This follows directly from Lemma 3 and the definition of MAXIMALLY IMPROVING T -PARTITION. ◀

3.2 Delay Cuts

Assuming that an initial solution is available, so far we only have transformed PESP into the equivalent MAXIMALLY IMPROVING T -PARTITION problem. We will now focus on special classes of T -partitions to demonstrate the strength of this transformation. Again, we consider a PESP instance $(G = (V, A), T, \ell, u, w)$.

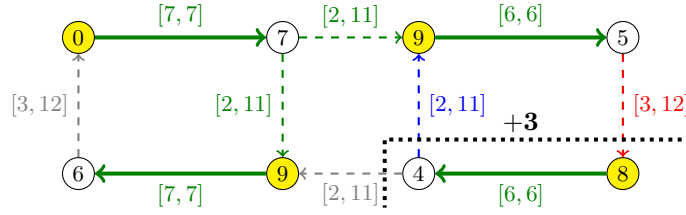
► **Definition 6.** *Let $S \subseteq V$ and $d \in \{1, \dots, T - 1\}$. The T -partition (V_0, \dots, V_{T-1}) with*

$$V_e := \begin{cases} S & \text{if } e = d, \\ V \setminus S & \text{if } e = 0, \\ \emptyset & \text{otherwise,} \end{cases} \quad e = 0, \dots, T - 1,$$

is called a delay cut (see [1]) with delay d and will simply be denoted by $\Delta(S, d)$. The restriction of MAXIMALLY IMPROVING T -PARTITION to delay cuts is called the MAXIMALLY IMPROVING DELAY CUT problem.

Intuitively, a delay cut $\Delta(S, d)$ delays – or shifts – all events in S by d . Delay cuts have been called *multi-node cuts* in [5], where the authors provide a way to escape from local optima produced by the modulo network simplex algorithm.

Starting with an initial timetable, an optimal timetable can be reached by decomposing a maximally improving T -partition $(V_0, V_1, \dots, V_{T-1})$ into the $T - 1$ delay cuts $\Delta(V_1, 1), \dots, \Delta(V_{T-1}, T - 1)$. From the perspective of T -partitions, delay cuts are hence natural building blocks. However, delay cuts themselves comprise several strategies:



■ **Figure 1** Fundamental delay cut: In this PESP instance with $T = 10$ and $w \equiv 1$, delaying the two vertices at the right lower corner by 3 produces a better (in fact, optimal) timetable: The overall slack is reduced from 7 to 4. This corresponds to the fundamental cut of the green spanning tree when removing the red arc. The modulo network simplex inner loop replaces the red arc with the blue arc at its lower bound.

1. *Modulo network simplex moves (“inner loop”)* [20]: The key insight behind the modulo network simplex method is that there is always an optimal PESP solution coming from a spanning tree structure: There is a spanning tree (or forest if the graph is not weakly connected) such that all tree arcs have either slack 0 or $u - \ell$. Starting from such a spanning tree structure, the algorithm tries to find a better solution by exchanging a tree arc with a co-tree arc, see also [13]. The delay cut then corresponds to the fundamental cut of the tree arc, the delay depends on the co-tree arc and whether the latter is considered with slack 0 or $u - \ell$. An example is depicted in Figure 1.
2. *Single-node cuts (“outer loop”)* [20], or *local improvements* [21]: These cuts are simply delay cuts $\Delta(S, d)$ with $|S| = 1$.
3. *Waiting edge cuts* [5]: If a vehicle dwells at a station where it is not terminating, then the dwell time is usually small. In particular, the difference $u - \ell$ is close to 0 and hence it seems reasonable to keep arrival and departure closely together and not to separate them by a cut. Waiting edge cuts are thus delay cuts $\Delta(S, d)$ with S consisting of the two vertices of an arc with small span $u - \ell$.

Since all these strategies rely on finding only a specific type of cut, solving MAXIMALLY IMPROVING DELAY CUT – searching the whole cut space – generalizes the above methods: If there is no improving delay cut, then also none of the approaches will be able to help. As the paper [5] only provided a randomized greedy procedure, we turn MAXIMALLY IMPROVING DELAY CUT into a genuine optimization problem.

► **Lemma 7.** *Let π be a periodic timetable. The improvement of a delay cut $\Delta(S, d)$ is*

$$\iota(\pi, \Delta(S, d)) = \sum_{ij \in \delta^+(S)} w_{ij}(y_{ij} - [y_{ij} - d]_T) + \sum_{ij \in \delta^-(S)} w_{ij}(y_{ij} - [y_{ij} + d]_T),$$

where $\delta^+(S)$ and $\delta^-(S)$ denote the sets of arcs leaving and entering S , respectively.

Proof. This is a simple computation from the definitions of delay cuts and the improvement of a T -partition. ◀

For a fixed delay d , we can transform MAXIMALLY IMPROVING DELAY CUT into a standard MAXIMUM CUT problem:

1. Construct the directed graph \bar{G} with vertex set $\bar{V} := V$ and arc set $\bar{A} := A \cup \{ji \mid ij \in A\}$, i.e., we add reverse copies of each arc if the reverse arc is not already present.
2. Initialize $c := 0 \in \mathbb{Z}^{\bar{A}}$.

3. For each arc $ij \in A$, set

$$c_{ij} := \begin{cases} c_{ij} + w_{ij}(y_{ij} - [y_{ij} - d]_T) & \text{if } [y_{ij} - d]_T \leq u_{ij} - \ell_{ij}, \\ -\infty & \text{otherwise.} \end{cases}$$

and

$$c_{ji} := \begin{cases} c_{ji} + w_{ij}(y_{ij} - [y_{ij} + d]_T) & \text{if } [y_{ij} + d]_T \leq u_{ij} - \ell_{ij}, \\ -\infty & \text{otherwise.} \end{cases}$$

4. Find the cut S in \overline{G} such that $c(\delta^+(S))$ is maximum.

Note that since y is given and d is fixed, this is indeed a MAXIMUM CUT problem with linear objective. As c does attain both positive and negative values, we are not able to transform our problem to a standard polynomial-time solvable MINIMUM CUT problem.

Although MAXIMUM CUT is NP-hard in general [9], our problem is still easy enough to be solved on reasonably large instances within a few minutes by a MIP solver, using, e.g., the formulation presented in Appendix A. An implementation of this program invoking the solver SCIP has been included into the concurrent PESP solver presented in [1], where it proved to be successful especially when faster heuristics already got stuck in local optima.

4 Graph Separators

This section introduces a novel divide-and-conquer approach to PESP. The core idea is to split the graph into two balanced parts, on the one hand losing as little information as possible, and on the other hand obtaining subproblems which are (much) easier to solve than the entire instance, see [25] for a recent application of this concept in the context of road networks, and references therein. We can then solve the PESP restricted to each half and combine the two solutions to a solution on the original instance. In order to avoid feasibility issues, we restrict ourselves to cut the original network at *free arcs*, i.e., arcs whose slack is allowed to take arbitrary values between 0 and $T - 1$. More formally, we want to find:

► **Definition 8.** Let (G, T, ℓ, u, w) be a PESP instance, $G = (V, A)$. Further let $\nu : 2^V \rightarrow \mathbb{R}_{\geq 0}$ be a measure on V , and let $\alpha \geq 1$ be an imbalance parameter. A (ν, α) -separator is a subset $S \subseteq V$ such that

- $\delta(S)$ consists only of free arcs, i.e., $ij \in A$ with $u_{ij} - \ell_{ij} \geq T - 1$,
- $w(\delta(S))$ is minimum,
- $\nu(V \setminus S) \leq \nu(S) \leq \alpha \cdot \nu(V \setminus S)$.

Here, $\delta(S)$ denotes the set of arcs in G with exactly one endpoint in S .

We will focus on the following two measures: At first, we consider balancing the number of vertices, i.e., $\nu(X) := |X|$ for $X \subseteq V$. Secondly, being a common indicator of the difficulty of a PESP instance, we will try to balance the cyclomatic number, i.e., the dimension of the cycle space of the graph, which equals $|A| - |V| + 1$ in the case of a connected graph. Of course, one could think of several more balancing criteria.

Since we are only allowed to cut through free arcs, our first step in creating a separator is to contract all non-free arcs. Note that these particular contractions are different from the commonly known PESP contractions which yield a simplified, but equivalent instance [4]. Doing so results in a multigraph, which can be resolved to a simple graph by adding up the weights of parallel arcs. The problem also permits to consider the underlying undirected graph. However, we need to keep track of the contracted vertices and the multiplicity of the arcs in order to calculate the correct measure ν , which lives on the uncontracted graph.

The structurally simplest PESP instances coming from public transit essentially contain two kinds of arcs: *Line* activities refer to driving a vehicle of a line between stations or dwelling in a station, and these activities come with only small allowed slack. *Transfer* activities are usually unconstrained in terms of slack, since one cannot expect all transfers in a network to be short, and restricting too many transfers might turn the problem infeasible. The weight on an arc is an estimate for the number of passengers using it. Recall from [16] that using PESP one is able to model a variety of other features from practice.

In this interpretation, the contraction process hence contracts all lines to single vertices. A separator then tries to divide the set of lines into two balanced parts such that the number of transferring passengers between the two parts is minimum.

We finally want to remark that finding separators is an NP-hard optimization problem in general [3]. However, it is possible to compute separators of good quality in a reasonable amount of time, and the literature is rich [2, 8, 10, 24].

4.1 Vertex-balanced Separators

By the above contraction process, finding a (ν, α) -separator balancing the number of vertices can be reduced to the following problem:

- **Problem 9.** Let (N, E) be an undirected graph with vertex multiplicities $m \in \mathbb{N}^N$ and edge weights $w \in \mathbb{Z}_{\geq 0}^E$. For a given imbalance $\alpha \geq 1$, find a subset $S \subseteq N$ such that
 - $w(\delta(S))$ is minimum,
 - $m(N \setminus S) \leq m(S) \leq \alpha \cdot m(N \setminus S)$.

- **Lemma 10.** Let $n := \sum_{i \in N} m_i$. Problem 9 is solved by the mixed integer linear program

$$\begin{array}{ll}
 \text{Minimize} & \sum_{ij \in E} w_{ij} x_{ij} \\
 \text{s.t.} & x_{ij} \geq z_i - z_j, \quad ij \in E, \\
 & x_{ij} \geq z_j - z_i, \quad ij \in E, \\
 & \sum_{i \in N} m_i z_i \geq \frac{n}{2}, \\
 & \sum_{i \in N} m_i z_i \leq \frac{\alpha \cdot n}{1 + \alpha}, \\
 & x_{ij} \in [0, 1], \quad ij \in E, \\
 & z_i \in \{0, 1\}, \quad i \in N.
 \end{array}$$

Proof. See Appendix B. ◀

4.2 Cycle-balanced Separators

We will now focus on balancing the cyclomatic number μ of the parts of a PESP instance (G, T, ℓ, u, w) . For a subset $X \subseteq V$, we will approximate the cyclomatic number by $\mu(X) := |A(G[X])| - |X| + 1$, where $A(G[X])$ denotes the set of arcs of the subgraph of G induced by X . This is the exact cyclomatic number if $G[X]$ is connected, and underestimates the true quantity by the number of connected components minus one otherwise.

Since contracting arcs does not change the difference between number of arcs and vertices, we do not need to remember the number of contracted vertices for computing μ . However, collapsing parallel arcs to a simple arc decreases the cyclomatic number, so that we keep track of the multiplicity of edges.

We hence consider the following problem:

► **Problem 11.** Let (N, E) be an undirected graph with edge multiplicities $m \in \mathbb{N}^E$ and edge weights $w \in \mathbb{Z}_{\geq 0}^N$. For a given imbalance $\alpha \geq 1$, find a subset $S \subseteq N$ such that

- $w(\delta(S))$ is minimum,
- $\mu(N \setminus S) \leq \mu(S) \leq \alpha \cdot \mu(N \setminus S)$.

► **Lemma 12.** *Problem 11 can be solved by the mixed integer linear program*

$$\begin{aligned}
 & \text{Minimize} && \sum_{ij \in E} w_{ij}(1 - \ell_{ij} - r_{ij}) \\
 & \text{s.t.} && \ell_{ij} \geq z_i + z_j - 1, && ij \in E, \\
 & && \ell_{ij} \leq z_i, && ij \in E, \\
 & && \ell_{ij} \leq z_j, && ij \in E, \\
 & && r_{ij} \geq 1 - z_i - z_j, && ij \in E, \\
 & && r_{ij} \leq 1 - z_i, && ij \in E, \\
 & && r_{ij} \leq 1 - z_j, && ij \in E, \\
 & && \mu_\ell = \sum_{ij \in E} \ell_{ij} - \sum_{i \in N} z_i + 1, \\
 & && \mu_r = \sum_{ij \in E} r_{ij} - \sum_{i \in N} (1 - z_i) + 1, \\
 & && \mu_\ell \geq \mu_r, \\
 & && \mu_\ell \leq \alpha \cdot \mu_r, \\
 & && \ell_{ij} \in [0, 1], && ij \in E, \\
 & && r_{ij} \in [0, 1], && ij \in E, \\
 & && z_i \in \{0, 1\}, && i \in N.
 \end{aligned}$$

Proof. See Appendix B. ◀

4.3 Combining Partial Solutions

Going back to PESP instances, it is clear that restricting a feasible periodic timetable to a subgraph results in a feasible periodic timetable, and the slack cannot increase. We summarize the converse for (ν, α) -separators: Let S be a (ν, α) -separator for a PESP instance $I = (G = (V, A), T, \ell, u, w)$. Let I^ℓ, I^r, I^m be the restrictions of I to the subgraphs induced by $S, V \setminus S$ and the shores of the cut induced by S , respectively (“left”, “right”, “middle”).

► **Theorem 13.** *Let S be a (ν, α) -separator, producing instances I^ℓ, I^r, I^m as above. Let π^ℓ, π^r be feasible periodic timetables for I^ℓ, I^r , respectively.*

(1) *The timetable π defined by*

$$\pi_i := \begin{cases} \pi_i^\ell & \text{if } i \in S, \\ \pi_i^r & \text{if } i \in V \setminus S \end{cases}$$

is feasible.

(2) *Moreover, if y^ℓ, y^r, y are the periodic slacks associated to π^ℓ, π^r, π , respectively, then*

$$w^t y = w^t y^\ell + w^t y^m + w^t y^r,$$

where y^m is the slack w.r.t. π of the arcs in I^m .

(3) If $\text{opt}(J)$ denotes the minimum weighted slack of a PESP instance J , then

$$\text{opt}(I^r) + \text{opt}(I^m) + \text{opt}(I^\ell) \leq \text{opt}(I) \leq \text{opt}(I^\ell) + \text{opt}(I^r) + W \cdot (T - 1),$$

where W stands for the weight of the cut, i.e., sum of the weights of all arcs in I^m .

Proof. Since a (ν, α) -separator cuts only through free arcs, (1) and (2) are clear. Since the optimal solution to I is feasible for the three parts I^ℓ , I^m , I^r , we obtain the left inequality. As we can combine optimal solutions to I^ℓ and I^r by (1) to a feasible solution to I , and the weighted slack increases at most by $W \cdot (T - 1)$ by (2), this shows the right inequality. ◀

Therefore, these separators produce as well primal and dual bounds for PESP instances. We will demonstrate the use of separators on large-scale timetabling instances in the next section.

5 Experiments

5.1 Set-up

We use the library `PESPlib`¹ as a benchmarking set. The library contains 20 hard timetabling instances, none of which is solved to proven optimality yet. The separator strategy does not seem to be suitable for the four bus timetabling instances: When removing all free arcs, the remaining network decomposes in only 2 (BL4) or 3 (BL1-BL3) components that cannot be separated further. In other words, there are only very few possible cuts. As a consequence, only the railway instances RxLy remain, which all show a similar structure, and this is why we will focus on the easiest instance R1L1 and the hardest instance R4L4.

At first, we compute vertex-balanced separators, choosing imbalance parameters $\alpha \in \{1.05, 1.1, 1.2, 1.5\}$. To this end, we use the fast graph partitioning software `METIS` [10] to generate an initial solution and apply the MIP solver `Gurobi 8.1`² to the program of Lemma 10 for 20 minutes. Secondly, we determine cycle-balanced separators with the same imbalance parameters as in the vertex case. Since `METIS` cannot handle the cycle balance constraints and its solutions usually violate it, we use only `Gurobi` on the MIP of Lemma 12 for 20 minutes. Of course, for both types of separators, we contract all non-free arcs in advance, and interpret the found separator on the original network again.

Having found a separator, we solve both parts with the concurrent PESP solver from [1], which integrates mixed integer programming, modulo network simplex and the `MAXIMALLY IMPROVING DELAY CUT` heuristic from Section 3. This solver computed the currently best bounds for all `PESPlib` instances, improving 10 former primal bounds in as little as 20 minutes using 7 parallel threads. We compare these results with our separator procedure by running each part for 10 minutes with the same number of threads. In particular, the computation time on the original instance equals the sum of running times of the two parts. The reason for the small running time limit is based on the good quality of the solutions produced by the concurrent solver, and the empirical observation that only minor improvements occur after the first 20 minutes [1]. Afterwards, we combine the timetables of both parts in an optimal way, i.e., we iterately shift the timetable of one of the parts by $0, 1, \dots, T - 1$ and choose the best combination.

¹ <https://num.math.uni-goettingen.de/~m.goerigk/pesplib>

² Gurobi Optimization LLC, <https://www.gurobi.com>

On the dual side, we compute dual bounds for each of the parts by running the concurrent solver for 10 minutes on 7 threads in pure MIP best bound mode with user cuts. We compare this with a 20-minutes run on the original instance with the same parameters.

In all PESP computations, CPLEX 12.8³ serves as underlying MIP solver. The experiments were carried out on an Intel Xeon E3-1270 v6 CPU at 3.8 GHz with 32 GB RAM. For an analysis of the impact of delay cuts, we refer to [1].

5.2 Separator Statistics

Vertex-balanced separators

In every case, Gurobi could improve the initial vertex-balanced separator found by METIS. For R1L1, the vertex separators are all optimal with respect to the given imbalance, whereas optimality gaps are around 70% for R4L4. In contrast to standard minimum cuts, the smaller part sometimes consists of several connected components, which is due to the balance constraint. However, this is no issue for solving PESP. Despite having almost equal number of vertices, especially the cyclomatic number and the weights turn out to be heavily imbalanced. The smallest cuts accumulate only 19% (R1L1) resp. 24% (R4L4) of the free weight of the original instance. Table 1 resp. Table 3 contain detailed statistics about the computed separators.

Cycle-balanced separators

As no fast initial solution is available, and the program from Lemma 12 is more difficult than in the vertex case, the best optimality gaps that we can achieve after 20 minutes are 26% (R1L1) resp. 86% (R4L4). The cuts are always heavier than in the vertex case, although the difference is much smaller for the large instance R4L4. On the plus side, the solutions are much better balanced with respect to other parameters such as number of vertices, number of arcs and the free weight.

5.3 Objective Values

R1L1

For the original instance R1L1, the concurrent PESP solver was able to compute a periodic timetable with weighted slack 30 861 021 (see Table 2 for details) within 20 minutes. We typically lose a weighted slack between 10 and 18 million in the cut, so there is little space for improvement on the two parts (left and right, see rows “cut” in column “primal objective value” in Table 2). Indeed, the timetable that is computed on the full instance is superior to all combined ones. The best combined timetable has weighted slack 34 669 413, coming from a cycle-balanced separator with imbalance parameter $\alpha = 1.2$. We note that the average weighted slack on the free arcs (in particular within the cut) – which have the largest impact on the primal objective value – is significantly higher on the combined timetables than on the original. In particular, along the free arcs within the cut, average slack values of almost 50% of the period time have to be accepted, whereas in those parts for which the concurrent solver computed the timetables (original, left, right), less than 25% of the period time can be achieved as average slack.

³ IBM ILOG CPLEX Optimization Studio, <https://www.ibm.com/analytics/cplex-optimizer>

The best dual bound computed from the sum of the two parts is 15 211 531, compared to 16 868 573. Again, the weight of the cut is the biggest hindrance, although optimality gaps are reasonably small on the parts. Due to the structure of the instance, assigning a slack of 0 to all free arcs in the cut is feasible, and we do not get any valuable lower bound from the “middle” part.

R4L4

Compared to an original primal bound of 40 706 349 after 20 minutes, we achieve 41 230 436 by a vertex-balanced separator with imbalance $\alpha = 1.2$ (see Table 4). However, all combined dual bounds (best: 11 428 968) are better than the original one (10 968 394). Thus it seems that the separator approach performs better on this larger instance. This is also due to the fact that the cuts comprise less weighted slack compared to R4L4. The good dual bound gives hope that separators might benefit to compute better lower bounds for PESP instances, which as to our experience is currently among the biggest obstacles in solving the PESPLib instances to optimality.

6 Conclusions

By considering T -partitions and introducing delay cuts for the PESP, we proposed a framework that generalizes several primal heuristics that had been known previously. In [1] the use of these cuts is already reported to contribute to the best known solutions for several instances of the PESPLib.

Regarding the separator heuristic, which can be regarded as an the entry point for a divide-and-conquer approach, so far, based on our first tuning of the computation of the separators, it is not able to come up with any better primal solutions for the instances of the PESPLib.

Nevertheless, we would not be surprised, if in the following settings the separator heuristic, too, could provide some added value:

- In contrast to the entire instance, the two resulting subproblems can be solved optimally.
- Apply the separation heuristic not only on one stage, but in a recursive, true divide-and-conquer mode.

Yet, be aware that along the edges of each separator – although being of minimal weight – we most often observed a relatively poor quality in the final solution (almost 50% of the period time).

- We believe that diving deeper into good algorithms for graph partitioning, e.g., by using better methods or simply more running time for the mixed integer programs, could overcome the difficulty that the separators are still too heavy to provide a trade-off for improving primal and dual objectives.
- Add some kind of post-processing “around” the separator: Instead of only shifting the fixed solutions of the two subproblems as a whole against each other, just keep fixed the slack values of those edges within them which are *not* incident with the separator. Then, optimize over those timetables in which the vertices that are endpoints of an edge of the separator can be shifted relative to the subproblem that they are actually belonging to.

References

- 1 Ralf Borndörfer, Niels Lindner, and Sarah Roth. A Concurrent Approach to the Periodic Event Scheduling Problem. Technical Report 19-07, Zuse Institute Berlin, 2019. To appear in RailNorrköping2019 – 8th International Conference on Railway Operations Modelling and Analysis.

- 2 Daniel Delling, Daniel Fleischman, Andrew V. Goldberg, Ilya Razenshiteyn, and Renato F. Werneck. An exact combinatorial algorithm for minimum graph bisection. *Mathematical Programming*, 153(2):417–458, November 2015.
- 3 M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.
- 4 Marc Goerigk and Christian Liebchen. An Improved Algorithm for the Periodic Timetabling Problem. In *ATMOS*, volume 59 of *OASICS*, pages 12:1–12:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
- 5 Marc Goerigk and Anita Schöbel. Improving the modulo simplex algorithm for large-scale periodic timetabling. *Computers & Operations Research*, 40(5):1363–1370, 2013.
- 6 Peter Großmann. *Satisfiability and Optimization in Periodic Traffic Flow Problems*. PhD thesis, TU Dresden, 2016.
- 7 Peter Großmann, Steffen Hölldobler, Norbert Manthey, Karl Nachtigall, Jens Opitz, and Peter Steinke. Solving Periodic Event Scheduling Problems with SAT. In He Jiang, Wei Ding, Moonis Ali, and Xindong Wu, editors, *Advanced Research in Applied Artificial Intelligence*, pages 166–175, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 8 Michael Hamann and Ben Strasser. Graph Bisection with Pareto Optimization. *J. Exp. Algorithmics*, 23:1.2:1–1.2:34, February 2018.
- 9 Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.
- 10 George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.
- 11 Leo Kroon, Dennis Huisman, Erwin Abbink, Pieter-Jan Fioole, Matteo Fischetti, Gábor Maróti, Alexander Schrijver, Adri Steenbeek, and Roelof Ybema. The new Dutch timetable: The OR revolution. *Interfaces*, 39(1):6–17, 2009.
- 12 Christian Liebchen. Optimierungsverfahren zur Erstellung von Taktfahrplänen. Master’s thesis, Technical University Berlin, Germany, 1998. In German.
- 13 Christian Liebchen. A Cut-Based Heuristic to Produce Almost Feasible Periodic Railway Timetables. In Sotiris E. Nikolettseas, editor, *Experimental and Efficient Algorithms, 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings*, volume 3503 of *Lecture Notes in Computer Science*, pages 354–366. Springer, 2005. doi: 10.1007/11427186_31.
- 14 Christian Liebchen. Optimization of passenger timetables: Are fully-integrated, regular-interval timetables really always the best? *European Rail Technical Review (RTR)*, 48(4):13–19, 2008.
- 15 Christian Liebchen. The first optimized railway timetable in practice. *Transportation Science*, 42(4):420–435, 2008.
- 16 Christian Liebchen and Rolf H Möhring. The Modeling Power of the Periodic Event Scheduling Problem: Railway Timetables—and Beyond. In *Algorithmic methods for railway optimization*, pages 3–40. Springer, 2007.
- 17 Christian Liebchen and Leon Peeters. Integral cycle bases for cyclic timetabling. *Discrete Optimization*, 6(1):98–109, 2009.
- 18 Christian Liebchen and Elmar Swarat. The Second Chvatal Closure Can Yield Better Railway Timetables. In Matteo Fischetti and Peter Widmayer, editors, *8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’08)*, volume 9 of *OpenAccess Series in Informatics (OASICS)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 19 Karl Nachtigall. *Periodic Network Optimization and Fixed Interval Timetables*. Habilitation thesis, Universität Hildesheim, 2008.
- 20 Karl Nachtigall and Jens Opitz. Solving Periodic Timetable Optimisation Problems by Modulo Simplex Calculations. In Matteo Fischetti and Peter Widmayer, editors, *8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’08)*, volume 9 of *OpenAccess Series in Informatics (OASICS)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

- 21 Karl Nachtigall and Stefan Voget. A genetic algorithm approach to periodic railway synchronization. *Computers & OR*, 23(5):453–463, 1996. doi:10.1016/0305-0548(95)00032-1.
- 22 Michiel A Odijk. A constraint generation algorithm for the construction of periodic railway timetables. *Transportation Research Part B: Methodological*, 30(6):455–464, 1996.
- 23 Julius Pätzold and Anita Schöbel. A Matching Approach for Periodic Timetabling. In *ATMOS'16*, volume 54 of *OASICS*, pages 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
- 24 Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.
- 25 Aaron Schild and Christian Sommer. On Balanced Separators in Road Networks. In Evripidis Bampis, editor, *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2015. doi:10.1007/978-3-319-20086-6_22.
- 26 Alexander Schrijver. *Combinatorial Optimization – Polyhedra and Efficiency*. Springer, 2003.
- 27 Paolo Serafini and Walter Ukovich. A mathematical model for periodic scheduling problems. *SIAM Journal on Discrete Mathematics*, 2(4):550–581, 1989.

A

 Maximum Cut MIP Formulation

► **Lemma 14.** Let $\bar{G} = (\bar{V}, \bar{A})$ and c be as constructed in Section 3.2. A cut S in \bar{G} maximizing $c(\delta^+(S))$ is computed by the following mixed integer linear program:

$$\begin{array}{ll}
 \text{Maximize} & \sum_{ij \in \bar{A}} c_{ij} x_{ij} \\
 \text{s.t.} & x_{ij} \leq z_i, \quad ij \in \bar{A}, \\
 & x_{ij} \leq 1 - z_j, \quad ij \in \bar{A}, \\
 & x_{ij} \geq z_i - z_j, \quad ij \in \bar{A} : c_{ij} < 0, \\
 & 0 \leq x_{ij} \leq 1, \quad ij \in \bar{A}, \\
 & z_i \in \{0, 1\}, \quad i \in \bar{V}.
 \end{array}$$

Here, the variables z_i with $z_i = 1$ will define the set S .

Proof. Let (x^*, z^*) be an optimal solution to the above program. Set $S := \{i \in V \mid z_i^* = 1\}$. If $x_{ij}^* = 1$ for an arc $ij \in \bar{A}$, then $z_i^* = 1$ and $z_j^* = 0$. On the other hand, $z_i^* = 1$ and $z_j^* = 0$ imply $x_{ij}^* \geq 1$ by the third constraint for arcs with negative c_{ij} and by maximization for $c_{ij} \geq 0$. i.e., S is a cut maximizing $c(\delta^+(S)) = \sum_{ij \in \bar{A}} c_{ij} x_{ij}^*$. Conversely, a maximum cut S^* produces a feasible solution to the mixed integer program of the same cost. ◀

B

 Proofs of Separator MIP Formulations

Proof of Lemma 10. The constraints for the minimum cut are straightforward: A vertex i lies in S iff $z_i = 1$ and an edge ij lies in $\delta(S)$ iff $x_{ij} = 1$. We just prove the balance constraints. In order to break symmetry, we can assume $m(S) \geq m(N)/2 = n/2$, as $m(S)$ is larger than $m(N \setminus S)$. Moreover, the condition $m(S) \leq \alpha \cdot m(N \setminus S)$ directly translates to

$$\sum_{i \in N} m_i z_i \leq \alpha \sum_{i \in N} m_i (1 - z_i),$$

which is equivalent to

$$(1 + \alpha) \sum_{i \in N} m_i z_i \leq \alpha n. \quad \blacktriangleleft$$

Proof of Lemma 12. We have $z_i = 1$ iff $i \in S$, $\ell_{ij} = 1$ iff ij has both endpoints in S (ℓ for “left”) and $r_{ij} = 1$ iff ij has no endpoint in S (r for “right”). The balance constraints are straightforward. ◀

C Tables

■ **Table 1** R1L1 separator statistics: The first column contains the type of the separator, the imbalance $\alpha \in \{1.05, 1.1, 1.2, 1.5\}$ and the optimality gap. Further columns: n – number of vertices, m – number of arcs, μ – cyclomatic number, w – weight, w_{free} – weight of all free arcs, $w \cdot (u - \ell)$ – maximum possible weighted slack. Rows: original – R1L1 instance as in PESPlib, contracted – after contraction of non-free arcs, left/right – parts of the separator, cut – subgraph induced by the arcs connecting left and right.

R1L1	part	n	m	μ	w	w_{free}	$w \cdot (u - \ell)$
	original	3 664	6 385	2 722	47 172 734	2 057 406	239 600 328
	contracted	106	2 230				
vertex	left	1 876	2 927	1 052	33 725 970	1 481 768	170 793 125
1.05	right	1 788	2 058	273	12 925 650	54 524	38 061 477
0.0%	cut	1 045	1 400	516	521 114	521 114	30 745 726
vertex	left	1 918	2 990	1 073	34 412 455	1 503 621	173 692 394
1.1	right	1 746	2 004	261	12 255 217	48 723	36 109 276
0.0%	cut	1 055	1 391	499	505 062	505 062	29 798 658
vertex	left	1 996	3 205	1 210	34 847 351	1 541 460	176 996 909
1.2	right	1 668	1 870	205	11 852 913	43 476	34 727 689
0.0%	cut	1 012	1 310	459	472 470	472 470	27 875 730
vertex	left	2 198	3 609	1 412	37 061 606	1 637 281	188 155 216
1.5	right	1 466	1 598	136	9 719 139	28 136	28 317 761
0.0%	cut	969	1 178	366	391 989	391 989	23 127 351
cycle	left	1 700	2 429	730	28 474 222	1 123 356	136 712 178
1.05	right	1 964	2 663	700	18 025 146	260 684	63 159 556
33.5%	cut	949	1 293	491	673 366	673 366	39 728 594
cycle	left	1 676	2 406	731	28 180 248	1 120 386	135 658 006
1.1	right	1 988	2 718	731	18 312 779	257 313	63 839 609
34.2%	cut	967	1 261	447	679 707	679 707	40 102 713
cycle	left	1 754	2 535	782	29 076 540	1 163 077	140 590 992
1.2	right	1 910	2 562	653	17 441 343	239 478	60 373 127
36.3%	cut	979	1 288	466	654 851	654 851	38 636 209
cycle	left	1 926	2 807	882	30 359 130	1 202 889	146 190 635
1.5	right	1 738	2 327	590	16 188 820	229 733	56 547 437
26.2%	cut	955	1 251	447	624 784	624 784	36 862 256

■ **Table 2** R1L1 objective values: Primal obj value – weighted slack of best found timetable, free % – contribution of free arcs to weighted slack, dual obj value – best lower bound, gap % – optimality gap. Rows: left, right, cut – as in Table 1, combined – optimal combination of partial timetables (primal) resp. sum of lower bounds (dual). The optimality gaps in the row combined are measured w.r.t. the best primal objective value, i.e., of the original instance.

R1L1	part	primal		average weighted slack			dual	
		obj value	free %	total	free	non-free	obj value	gap %
	original	30 861 021	87.68%	0.65	13.15	0.08	16 868 573	45.34%
vertex	left	24 894 427	88.26%	0.74	14.83	0.09	13 949 001	43.97%
1.05	right	409 562	100.00%	0.03	7.51	0.00	358 120	12.56%
	cut	14 322 886	100.00%	27.49	27.49	–	0	–
	combined	39 626 875	92.62%	0.84	17.84	0.06	14 307 121	53.64%
vertex	left	23 376 399	87.01%	0.68	13.53	0.09	14 106 622	39.65%
1.1	right	340 302	100.00%	0.03	6.98	0.00	295 224	13.25%
	cut	13 885 806	100.00%	27.49	27.49	–	0	–
	combined	37 602 507	91.92%	0.80	16.80	0.07	14 401 846	53.33%
vertex	left	22 842 193	86.11%	0.66	12.76	0.10	14 327 640	37.28%
1.2	right	297 141	100.00%	0.03	6.83	0.00	256 629	13.63%
	cut	12 879 922	100.00%	27.26	27.26	–	0	–
	combined	36 019 256	91.19%	0.76	15.97	0.07	14 584 269	52.74%
vertex	left	24 857 603	86.79%	0.67	13.18	0.09	15 068 169	39.38%
1.5	right	149 989	100.00%	0.02	5.33	0.00	143 362	4.42%
	cut	10 258 139	100.00%	26.17	26.17	–	0	–
	combined	35 265 731	90.69%	0.75	15.55	0.07	15 211 531	50.71%
cycle	left	16 382 907	85.07%	0.58	12.41	0.09	10 189 253	37.81%
1.05	right	3 193 192	89.61%	0.18	10.98	0.02	2 608 782	18.30%
	cut	18 264 680	100.00%	27.12	27.12	–	0	–
	combined	37 840 779	92.66%	0.80	17.04	0.06	12 798 035	58.53%
cycle	left	3 288 791	91.72%	0.18	11.72	0.02	2 482 699	24.51%
1.1	right	14 370 669	84.62%	0.51	10.85	0.08	10 110 491	29.64%
	cut	18 033 828	100.00%	26.53	26.53	–	0	–
	combined	35 693 288	93.04%	0.76	16.14	0.06	12 593 190	59.19%
cycle	left	15 029 848	86.12%	0.52	11.13	0.07	10 518 964	30.01%
1.2	right	2 985 689	89.43%	0.17	11.15	0.02	2 341 735	21.57%
	cut	16 653 876	100.00%	25.43	25.43	–	0	–
	combined	34 669 413	93.07%	0.73	15.68	0.05	12 860 699	58.33%
cycle	left	15 523 603	84.57%	0.51	10.91	0.08	10 809 272	30.37%
1.5	right	2 862 115	90.82%	0.18	11.31	0.02	2 218 792	22.48%
	cut	16 932 910	100.00%	27.10	27.10	–	0	–
	combined	35 318 628	92.47%	0.75	15.87	0.06	13 028 064	57.78%

■ **Table 3** R4L4 separator statistics: See Table 1 for a legend.

R4L4	part	n	m	μ	w	w_{free}	$w \cdot (u - \ell)$
	original	8 384	17 754	9 371	65 495 305	2 219 558	297 194 946
	contracted	265	8 257				
vertex	left	4 286	8 190	3 905	35 754 908	1 013 074	151 098 512
1.05	right	4 098	6 453	2 356	29 169 656	635 743	112 422 715
70.5%	cut	1 915	3 111	1 424	570 741	570 741	33 673 719
vertex	left	4 386	8 402	4 017	36 179 480	1 032 288	153 439 283
1.1	right	3 998	6 261	2 264	28 748 028	619 473	110 255 640
72.4%	cut	1 891	3 091	1 419	567 797	567 797	33 500 023
vertex	left	4 572	8 766	4 195	37 645 797	1 076 741	159 615 340
1.2	right	3 812	5 849	2 038	27 282 163	575 472	104 106 251
75.1%	cut	1 939	3 139	1 424	567 345	567 345	33 473 355
vertex	left	5 030	9 878	4 849	41 451 476	1 252 913	180 683 746
1.5	right	3 354	4 991	1 640	23 501 054	423 870	84 487 475
73.2%	cut	1 826	2 885	1 273	542 775	542 775	32 023 725
cycle	left	4 086	7 093	3 008	33 052 401	863 684	133 516 192
1.05	right	4 298	7 204	2 907	31 792 978	705 948	125 333 120
86.0%	cut	2 097	3 457	1 596	649 926	649 926	38 345 634
cycle	left	4 796	7 941	3 146	34 722 846	824 356	138 016 435
1.1	right	3 588	6 566	2 979	30 170 267	793 010	123 649 183
84.1%	cut	1 898	3 247	1 560	602 192	602 192	35 529 328
cycle	left	4 918	8 268	3 351	36 200 384	879 816	144 508 303
1.2	right	3 466	6 265	2 800	28 684 198	729 019	116 653 986
84.8%	cut	1 863	3 221	1 574	610 723	610 723	36 032 657
cycle	left	5 098	8 891	3 794	38 255 730	951 766	154 792 427
1.5	right	3 286	5 819	2 534	26 665 459	693 676	108 529 675
87.3%	cut	1 756	3 044	1 490	574 116	574 116	33 872 844

■ **Table 4** R4L4 objective values: See Table 2 for a legend.

R4L4	part	primal		average weighted slack			dual	
		obj value	free %	total	free	non-free	obj value	gap %
	original	40 706 349	94.69%	0.62	17.37	0.03	10 968 394	73.05%
vertex 1.05	left	15 887 937	94.18%	0.44	14.77	0.03	6 074 517	61.77%
	right	10 180 187	95.79%	0.35	15.34	0.02	5 248 237	48.45%
	cut	15 936 993	100.00%	27.92	27.92	–	0	–
	combined	42 005 117	96.78%	0.64	18.32	0.02	11 322 754	72.18%
vertex 1.1	left	16 630 820	94.15%	0.46	15.17	0.03	6 025 103	63.77%
	right	9 608 412	93.71%	0.33	14.53	0.02	5 141 257	46.49%
	cut	15 855 247	100.00%	27.92	27.92	–	0	–
	combined	42 094 479	96.25%	0.64	18.25	0.02	11 166 360	72.57%
vertex 1.2	left	16 923 159	94.45%	0.45	14.85	0.03	6 153 882	63.64%
	right	8 133 392	89.08%	0.30	12.59	0.03	4 994 591	38.59%
	cut	16 173 885	100.00%	28.51	28.51	–	0	–
	combined	41 230 436	95.57%	0.63	17.75	0.03	11 148 473	72.61%
vertex 1.5	left	20 449 436	90.50%	0.49	14.77	0.05	6 441 593	68.50%
	right	6 120 307	92.89%	0.26	13.41	0.02	3 880 625	36.59%
	cut	15 245 750	100.00%	28.09	28.09	–	0	–
	combined	41 815 493	94.31%	0.64	17.77	0.04	10 322 218	74.64%
cycle 1.05	left	12 822 538	93.21%	0.39	13.84	0.03	5 948 343	53.61%
	right	11 145 363	94.12%	0.35	14.86	0.02	5 480 625	50.83%
	cut	18 328 779	100.00%	28.20	28.20	–	0	–
	combined	42 296 680	96.39%	0.65	18.37	0.02	11 428 968	71.92%
cycle 1.1	left	13 982 046	95.43%	0.40	16.19	0.02	5 736 502	58.97%
	right	11 580 126	89.07%	0.38	13.01	0.04	5 460 355	52.85%
	cut	16 928 374	100.00%	28.11	28.11	–	0	–
	combined	42 490 546	95.52%	0.65	18.29	0.03	11 196 857	72.49%
cycle 1.2	left	14 648 967	94.74%	0.40	15.77	0.02	5 823 535	60.25%
	right	10 313 092	86.04%	0.36	12.17	0.05	5 307 285	48.54%
	cut	17 130 851	100.00%	28.05	28.05	–	0	–
	combined	42 092 910	94.75%	0.64	17.97	0.03	11 130 820	72.66%
cycle 1.5	left	16 400 078	95.60%	0.43	16.47	0.02	6 183 490	62.30%
	right	9 274 273	85.59%	0.35	11.44	0.05	5 051 562	45.53%
	cut	15 985 667	100.00%	27.84	27.84	–	0	–
	combined	41 660 018	95.06%	0.64	17.84	0.03	11 235 052	72.40%

On Sorting with a Network of Two Stacks

Matúš Mihalák 

Department of Data Science and Knowledge Engineering, Maastricht University, The Netherlands
matus.mihalak@maastrichtuniversity.nl

Marc Pont

Department of Data Science and Knowledge Engineering, Maastricht University, The Netherlands
m.pont@student.maastrichtuniversity.nl

Abstract

Sorting with stacks is a collection of problems that deal with sorting a sequence of numbers by pushing and popping the numbers to and from a given set of stacks. Multiple concrete decision or optimization questions are formed by restricting the access to the stacks. The motivation comes, e.g., from shunting train wagons in shunting yards, shunting trams in depots, or in stacking cargo containers on cargo ships or storage yards in transshipment terminals.

We consider the problem of sorting a permutation of n integers $1, 2, \dots, n$ using $k \geq 2$ stacks. In this problem, elements from the input sequence are pushed one-by-one (in the order of the elements in the sequence) to one of the k stacks. At any time, an element from a stack can be popped and pushed to another stack; such an operation is called a *shuffle*. Also, at any time, an element can be popped from a stack and placed to the output sequence. We can only place the elements to the output in the increasing order of their value such that at the end the output is the ordered sequence of the elements. The problem asks to minimize the number of shuffles in the process.

It is known that for $k \geq 4$, the problem is NP-hard, and that there is no approximation algorithm unless $P=NP$. For $k \geq 3$, it is known that at most $O(n \log n)$ shuffles are needed for any input sequence. For the case when $k = 2$, there exist input sequences that require $\Omega(n^{2-\varepsilon})$ shuffles, for any $\varepsilon > 0$. Nothing substantially more is known for the case of $k = 2$. In this paper, we study the following variant of the problem with $k = 2$ stacks: no shuffle and no placement to the output sequence can happen before every element is in one of the two stacks. We show that our problem can be seen as the MINUNCUT problem by providing a polynomial-time reduction, and thus we show that there exists a randomized $O(\sqrt{\log n})$ -approximation algorithm and a deterministic $O(\log n)$ -approximation algorithm for our problem.

2012 ACM Subject Classification Mathematics of computing \rightarrow Approximation algorithms; Theory of computation \rightarrow Approximation algorithms analysis

Keywords and phrases Sorting, Stacks, Optimization, Algorithms, Reduction, MinUnCut

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.3

Acknowledgements The authors would like to thank Peter Widmayer and Thomas Erlebach for fruitful discussions on the topic.

1 Introduction

In computer science, a *stack* is a fundamental list-based data structure. Stacks allow item insertions and removals at one end of the list, known as the *last-in, first-out* (LIFO) principle. For stacks, insertions and removals are, respectively, called *push* and *pop* operations. A queue is another fundamental list-based data structure. Item insertions and removals happen at opposite ends in a queue, and this operation modus is known as the *first-in, first-out* (FIFO) principle.

Besides being a fundamental data structure in computer science, both stacks and queues model a wide range of applications in the real world, and in logistics and production planning in particular. An example is reordering of train wagons on a shunting yard, which can be



© Matúš Mihalák and Marc Pont;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 3; pp. 3:1–3:12

OpenAccess Series in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

seen as sorting (reordering) with stacks (rail tracks). Typically, a set of n wagons, each having a unique identifier, need to be coupled to form a train that visits several customers along a fixed route. For every visit, the wagons determined for the respective customer need to be at the very end of the train, so that the wagons can be decoupled from the train and left at the customer. Thus, the n wagons need to be ordered according to a specific order (given by the order of the customers along the fixed route). The ordering of the wagons happens at a railway switching network, commonly known as a shunting yard, switch yard, marshalling yard, or classification yard. Typically, there are several so-called classification tracks, which can be accessed from only one end. The wagons can be pushed from a main track over a network of switches to any of the classification tracks. Subsequently, some of the wagons from a classification track can be pulled back to the main track, and the process can be repeated. Because the tracks can only be accessed from one end, the tracks can be modeled as stacks.

Inspired by such real-world scenarios, Knuth initiated the study of the problem of deciding what input sequences $\pi = (\pi_1, \dots, \pi_n)$ – permutations of the integers $1, 2, \dots, n$ – are *sortable* by a single stack or a queue [14]. In his setting, the elements of the input sequence are accessed sequentially (from π_1 to π_n), and are placed to the stack. At any time, either an element from the input sequence is pushed to the stack, or an element from a stack is popped, and then placed to the output sequence. The goal is to provide a sorted sequence at the output. In this setting, some sequences cannot be sorted, and the main focus of Knuth’s work was to characterize input sequences that can be sorted. Knuth also touched upon the question of using more than one stack and especially the question of the number of stacks that are needed to sort any input sequence [15].

Subsequently, Even and Itai [6] and Tarjan [21] picked-up from Knuth and studied how to sort an input sequence using several stacks or queues. Tarjan introduced and studied a general model for sorting that contains several stacks and queues [21]. In the model, the stacks and queues are connected by an underlying directed graph that additionally contains the input node s and the output node t , such that s contains the input sequence and t contains the output sequence. Initially, s contains a permutation $\pi = (\pi_1, \dots, \pi_n)$ of the first n integers, and t contains an empty sequence. Vertex s has no incoming edge, and vertex t has no outgoing edge. At any time, an element (an integer) can be taken from any vertex u (obeying the access rules of the underlying data structure – a stack or a queue), moved along any outgoing edge (u, v) of the graph, and stored to the data structure at v . In this context, the elements from s are obtained in the order $\pi_1, \pi_2, \dots, \pi_n$. The goal is to decide whether the input sequence can be *sorted*, i.e., whether there is a sequence of moves of the elements along the edges such that the elements arrive at t in the order $1, 2, \dots, n$. This is not always possible, and the posed question initiated the study of permutation classes, see, e.g. [4]. Much of the work along these questions have focused on structural results, characterizing and counting permutations that are sortable by a given acyclic network of stacks and queues.

However, as Tarjan observes, whenever the underlying graph contains a cycle (and the cycle is reachable from s , and vertex t is reachable from the cycle), any input permutation can be sorted. Thus, for the question “what permutations are sortable?”, such underlying graphs are trivial and thus not considered in the research along the question.

Optimization Variant

Much later, the observation of Tarjan was picked up, and two related *optimization* questions were asked for underlying graphs containing cycles [7, 17]: “How many moves do we need to sort a given input sequence?” and “What is the complexity of sorting with a minimum

number of moves?”. These questions have mainly been studied in the setting where the underlying graph is a complete graph, with the exception of the vertices s and t , which only have outgoing and incoming edges, respectively.

Felsner and Pergel show that for $k \geq 3$ stacks, any input sequence can be sorted with $O(n \log_{k-1} n)$ many moves [7]. This is asymptotically tight, as the worst-case inputs require at least $\frac{n}{2} \log_k n - \Theta(1)$ moves [7]. This is in strong contrast to the case when $k = 2$, as for this case, Felsner and Pergel show that for any $\varepsilon > 0$ there exists an input sequence which needs $\Omega(n^{2-\varepsilon})$ many moves. Interestingly, their example is also valid if we restrict the movements such that no move to t can be made before all elements from s are in one of the stacks. This restriction is called a *midnight constraint*, as it mimics the situation when trams need to be parked in a depot (tracks in the depot are the stacks in our model) at the end of the day, and can leave the depot only in the morning.

König and Lübbecke study the optimization version of the sorting problem [17]. Naturally, in any solution to the sorting problem, every item needs to move along an arc from s exactly once, and along an arc to t exactly once, so König and Lübbecke study the following optimization problem: sort the input sequence by a minimum number of *shuffles*, where a *shuffle* is a move along an arc that is not incident to s and also not incident to t . For this problem, a ρ -approximation algorithm, for $\rho > 1$, is a polynomial-time algorithm that sorts any input sequence π using at most $\rho \cdot \text{OPT}(\pi)$ many shuffles, where $\text{OPT}(\pi)$ is the minimum number of shuffles needed to sort the sequence π . König and Lübbecke show that it is NP-hard to approximate the minimum number of shuffles within $O(n^{1-\varepsilon})$, for any non-trivial,¹ even constant, $k \geq 4$. Their work is based on the work of Evan and Itai [6], and the relation of the problem of deciding whether a proper k -coloring of a given circle graph exists to the problem of deciding whether the input permutation can be sorted with k -stacks without shuffles. The former problem has been shown NP-complete for $k \geq 4$ by Unger [23].

We note that since for $k \geq 4$ deciding whether one can sort with zero shuffles is NP-hard, it follows that, for $k \geq 4$, there is no approximation algorithm for the problem of minimizing the number of shuffles, unless P=NP.

Optimization with Midnight Constraint

We further note that the optimization problem with $k \geq 4$ stacks remains NP-hard also for the midnight-constraint, since the midnight constraint can be imposed by appending the integer 0 to the input permutation π of integers $1, 2, \dots, n$, and considering the new problem on the resulting permutation π' of integers $0, 1, \dots, n$ without midnight constraint. Since the smallest integer 0 comes at the end of the input sequence π' , no move to t can happen before 0 is moved from s , and thus before all elements of π' are in one of the k stacks. However, the problem of deciding whether one can sort with zero shuffles becomes polynomial-time solvable for any k in the case of midnight constraint, since this problem is equivalent to the k -coloring of permutation graphs [6], which can be solved in polynomial time. Hence, the inapproximability result for the general case (i.e., no midnight constraint) and $k \geq 4$ does not carry over to the case with the midnight constraint.

Since for $k \geq 3$, at most $O(n \log_k n)$ shuffles are needed [7], it follows that there exists a $O(n \log_k n)$ -approximation algorithm for the minimization problem with midnight constraint and $k \geq 3$. For $k = 2$ and midnight constraint, no non-trivial approximation algorithm is known. (It is easy to see that for $k = 2$, no more than $O(n^2)$ shuffles is needed, which gives a trivial $O(n^2)$ -approximation algorithm.)

¹ E.g., $k = n$ is trivial, as every item can be placed on a unique stack, and thus no shuffle is required.

The complexity of the minimization problem for $k = 2$ and $k = 3$, however, is an open problem also for the optimization problem with the midnight constraint. Both Felsner and Pergel [7] and König and Lübbecke [17] asked, as an open problem, whether the sorting problem with midnight constraint is computationally easier. (There is a light evidence that the problem with the midnight constraint might be easier: recall that deciding whether one can sort with zero shuffles is NP-hard for $k \geq 4$ in general, but becomes solvable in polynomial-time for the midnight constraint.)

Our Contribution

In this paper, we address the open problems for the case with $k = 2$ stacks and make a progress for a special case of the midnight constraint. We study the minimization problem with the *strong midnight-constraint*, which we define as the midnight constraint, with an additional constraint where no shuffles are allowed before all items are moved away from s .

We show that the problem of minimizing the number of shuffles with the strong midnight-constraint can be seen (by a certain polynomial-time reduction) as the MINUNCUT problem on certain graphs, and thus inherits the same approximation-algorithm guarantees. In particular, as a corollary, we show that there exists a randomized $O(\sqrt{\log n})$ -approximation algorithm, and a deterministic $O(\log n)$ -approximation algorithm for the minimization problem with $k = 2$ stacks and with the strong midnight-constraint.

The result thus substantially improves upon the trivial $O(n^2)$ -approximation algorithm for $k = 2$. We note that the $O(n \log n)$ -approximation algorithm for $k \geq 3$ and for the midnight constraint carries over also to the case with the strong midnight-constraint. Our result thus gives, for the variant with the strong midnight-constraint, a better approximation algorithm for the case $k = 2$ than for the case $k \geq 3$.

Additionally, we show that in the setting with the strong midnight-constraint and $k = 2$ there exists an input sequence for which every algorithms needs $\Omega(n^2)$ shuffles. This improves upon the lower bound of $\Omega(n^{2-\varepsilon})$ shuffles that holds for any $\varepsilon > 0$, and was shown by Felsner and Pergel for the (normal) midnight constraint and $k = 2$, and that also applies to the setting with the strong midnight constraint [7].

We note that relating the number of shuffles to the number of edges that need to be deleted such that some auxiliary graph becomes bipartite (as is the case of the MINUNCUT problem) is not new. Motivated by the result of Evan and Itai [6] which states that one can sort with zero shuffles using k stacks if and only if there exists a k -coloring of the underlying circle graph, König and Lübbecke consider the question whether the number of shuffles is equal to the number of monochromatic edges in a k -coloring of the underlying circle graph, and provided an example demonstrating that this is not case [17]. In our work, we use a different auxiliary graph than the circle graph.

1.1 Further Related Work to Stack Sorting

Albeit Tarjan defined the problem of sorting with a network of stacks and queues for any underlying graph [21], mainly the following two graph classes have been studied: (i) a directed path from s to t ; this case is also referred to as *sorting with stacks/queues in series*, and (ii) graphs where every node other than s and t is connected only to s (in the direction from s) and to t (in the direction to t); this setting is also known as *sorting with stacks/queues in parallel*. Observe that the question “can we sort the input sequence with zero shuffles” where the underlying graph induced by the stacks is a complete graph is equivalent to the setting with stacks in parallel. Even and Itai study the setting with stacks in parallel and with

queues in parallel [6]. Besides the general setting, Even and Itai also study the variant where no movement to t can precede any movement from s . In their setting, this is equivalent to both the strong midnight-constraint and the midnight constraint.

As mentioned before, much of the previous work focused on structural results concerning permutations that can be sorted by stacks/queues in series/parallel. Bóna surveys much of the progress made till about 2003 [4]. Among the newer research that is of algorithmic flavor is the study by Smith that compares two greedy algorithms for sorting with stacks in series [20], the paper by Pierrot and Rossin that shows that deciding whether a given permutation is sortable by two stacks in series is polynomial-time solvable [18], and the paper by Biedl et al. [2] which studies the question of how the number of stacks influences the number of shuffles needed to sort any input sequence.

1.2 Related Work in the Application Domain

Over the years, optimization theory literature has mentioned numerous problems related to sorting with stacks. A few examples are: assigning trains, trams or buses to positions in a depot [3, 11, 9]; storing integrated steel slabs in order of processing [16]; sorting car bodies for paint processing [12]; and storage yard operations in container terminals [5, 22]. Each of these problems require items to be placed on stacks such that they can be retrieved in a desired order with minimum effort or shuffles. In practice it is not uncommon that additional constraints exist such as stack height or item placement.

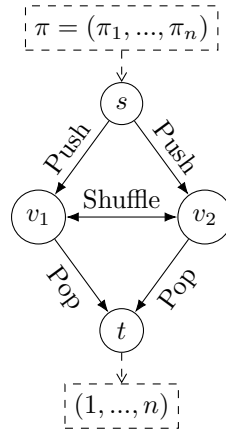
2 The Setting and Preliminaries

We study the RESTRICTEDBI-STACKSORTING problem, which is defined as follows, and illustrated in Figure 1. We are given a directed graph $G = (V, E)$ on four vertices and an *input sequence* π , which is a permutation of the numbers $1, \dots, n$. Each number in the permutation is also called an *element* of π . Vertex set V represents the four possible locations for items π_1, \dots, π_n , which are a *source* vertex s , two stack vertices v_1 and v_2 , and a *target* vertex t . The two stacks v_1 and v_2 exhibit the LIFO behavior. Edge set E consists of directed edges (i, j) to represent actions that move the first available item from vertex i to vertex j . These actions are *push*, represented by edge (s, v_1) and by edge (s, v_2) , *shuffle*, represented by edge (v_1, v_2) and by edge (v_2, v_1) , and *pop*, represented by edge (v_1, t) and edge (v_2, t) . There are no other edges in E . Items (π_1, \dots, π_n) arrive sequentially from s and may only traverse edges in E along the direction of the edge. The problem asks to move the elements in π from s to t such that at any time, only one item traverses along an edge, the items at v_1 and v_2 respect the LIFO behavior, the items leave s in the given order by π , and the items arrive at t in increasing order of value. Furthermore, we require that while there are items in s , no shuffle *and* no pop appears. We call this the *strong midnight-constraint*. The goal of RESTRICTEDBI-STACKSORTING is to minimize the number of shuffles along the process. The number of made shuffles by an algorithm is called the *shuffle count* (of the algorithm).

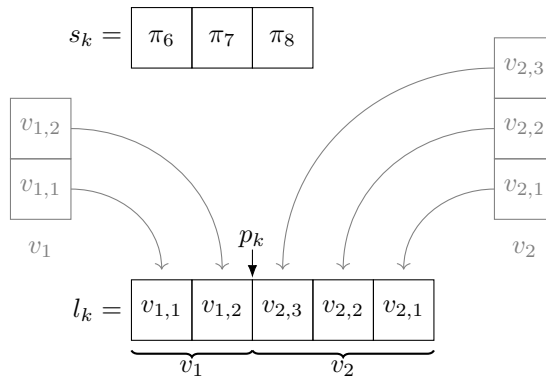
2.1 Useful State Representation

At any time of the sorting procedure, the state is fully determined by the remaining elements in s , and by the content of the two stacks in vertices v_1 and v_2 . We now describe an alternative description of a state, which is the crucial element in showing our main result.

We view any sorting procedure as an iterative process, which at any time step $k = 1, 2, \dots$, moves an element along an edge of G . A state S_k at time step k describes the situation



■ **Figure 1** Illustration of the underlying graph for sorting with two (fully connected) stacks.

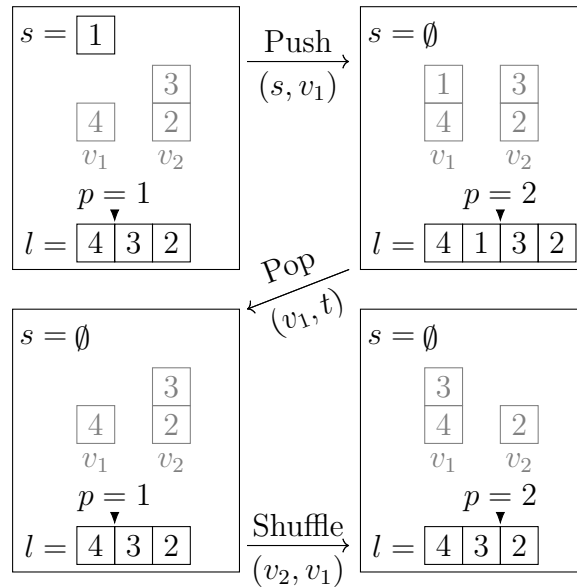


■ **Figure 2** Creating state representation $S_k = (s_k, l_k, p_k)$.

after the k -th move. State S_k is a tuple (s_k, l_k, p_k) , where s_k is the remaining input at vertex s , l_k is a list (array) that is the union of the two stacks, and p_k is a pointer to the list l_k , called *shuffler indexer*, which specifies the tops of the two stacks in the list l_k . To create list l_k , we concatenate the stacks $v_1 = (v_{1,1}, \dots, v_{1,|v_1|})$ and $v_2 = (v_{2,1}, \dots, v_{2,|v_2|})$ as list $l_k = (v_{1,1}, \dots, v_{1,|v_1|}, v_{2,|v_2|}, \dots, v_{2,1})$, such that $v_{i,1}$ is the bottom-most item of stack v_i and $v_{i,|v_i|}$ is the top-most item of stack v_i . Alternatively, $l_k = v_1 v_2^R$, where v_2^R is the reverse of v_2 . Finally, we set shuffler indexer p_k at location $p_k = |v_1|$, representing the top of both of our stacks. Figure 2 depicts our alternative state representation for RESTRICTEDBI-STACKSORTING. Figure 3 provides an example of how our state representation transitions during a push, a pop, and a shuffle action. State S_0 describes the situation before any move is made, and thus $S_0 = (\pi, (), 0)$, where $()$ denotes the empty list.

2.2 Relation of State with Shuffle Count

Recall that RESTRICTEDBI-STACKSORTING requires that all elements are first pushed from s before any shuffle or pop operation can happen. Observe that after all n elements are pushed from s , the shuffle count is determined: The first element that needs to be popped is 1. For this, all elements that are above element 1 need to be shuffled to the other stack.



■ **Figure 3** State transitions when applying a push, pop, and shuffle action.

After that, 1 can be popped, and the same procedure continues with element 2: all elements that are above 2 in the same stack need to be shuffled, and then element 2 can be popped. After that, the same procedure continues with elements 3, 4, . . . , n .

Using our alternative state representation, we can determine the shuffle count from the list l_n of state S_n , i.e., the state after all n elements have been pushed from s , as follows: We place an auxiliary value 0 at the position p_n in list l_n . Then, observe that to pop element 1, we need to shuffle all elements that appear in l_n between element 0 and element 1. Similarly, after we have popped element $i \geq 1$, we need to shuffle all elements that are the elements that lie in l_n between i and $i + 1$ and that are larger than $i + 1$. This follows because: (i) when element i is popped to t , it is on top of a stack, and to pop the next element $i + 1$, elements that lie above $i + 1$ need to be shuffled, and these lie between i and $i + 1$ in l_n ; (ii) any element in l_n that is smaller than $i + 1$ has been popped to t and thus does not need to be shuffled.

Let $sc(i, i + 1)$ denote the number of elements that lie in l_n between elements i and $i + 1$, and that are larger than $i + 1$. We have thus showed the following.

► **Lemma 1.** *The shuffle count is equal to $\sum_{i=0}^{n-1} sc(i, i + 1)$.*

2.3 Worst-Case Number of Shuffles

We now show that the worst-case number of shuffles for RESTRICTEDBI-STACKSORTING is $\Omega(n^2)$. For the strong midnight-constraint, this further strengthens the lower bound of $\Omega(n^{2-\varepsilon})$, for any constant $\varepsilon > 0$, of Felsner and Pergel [7], which also holds for the (normal) midnight constraint.

► **Theorem 2.** *There exists an input sequence π for RESTRICTEDBI-STACKSORTING for which every algorithm has a shuffle count of $\Omega(n^2)$.*

Proof. Consider the input sequence $\pi^* = (2, 4, 6, \dots, n, n - 1, n - 3, \dots, 5, 3, 1)$ for any even n . Notice that π^* consists of two sub-sequences: $\pi_{1, \frac{n}{2}}^*$ of all even values and $\pi_{\frac{n}{2}+1, n}^*$ of all odd values.

After $\pi_{1, \frac{n}{2}}^*$ is pushed to the stacks, one of the stacks will contain the majority of the items of $\pi_{1, \frac{n}{2}}^*$, and thus at least $\frac{n}{4}$ many items. Without loss of generality, assume that stack v_1 contains at least $\frac{n}{4}$ items from $\pi_{1, \frac{n}{2}}^*$.

Observe that every item in v_1 needs to be popped in the order from bottom to top. Also, observe that for every even value e that we pop from v_1 , we then need to pop the odd value $e + 1$ from $\pi_{\frac{n}{2}+1, n}^*$. Now, regardless of how the items in $\pi_{\frac{n}{2}+1, n}^*$ have been pushed to the stacks, for every bottom value e of v_1 , we need to shuffle at least the remaining even values on stack v_1 in order to pop value $e + 1$. Given that we have at least $\frac{n}{4}$ even values on stack v_1 , we see that the shuffle count is at least $\sum_{i=1}^{\frac{n}{4}} (\frac{n}{4} - i) = \frac{n^2 - 4n}{32} = \Omega(n^2)$. Thus, any algorithm for RESTRICTEDBI-STACKSORTING uses at least $\Omega(n^2)$ shuffles. ◀

3 Relation to MinUnCut

We will now show that RESTRICTEDBI-STACKSORTING can be seen as MINUNCUT. The problem MINUNCUT is given by an undirected graph $H = (V_H, E_H)$, and asks for a partition (S, T) of the vertices in V_H such that the number of edges with endpoints from the same part is minimized. The problem is the complement of the more famous MAXCUT problem, that asks for (S, T) such that the number of edges in the cut, i.e., edges with one endpoint in S and one endpoint in T , is maximized. It can be easily seen that an optimum solution to MAXCUT is also an optimum solution to MINUNCUT. MAXCUT was among the first computational problems shown to be NP-complete [13].

The two problems differ with respect to approximability. While for MAXCUT there exists a ρ -approximation algorithm where ρ is roughly 0.878 [10], the best approximation algorithm for MINUNCUT is a randomized $O(\sqrt{\log n})$ -approximation algorithm by Agarwal et al. [1] and a deterministic $O(\log n)$ -approximation algorithm by Garg et al. [8].

Our main technical result is stated in the following theorem.

► **Theorem 3.** *We can reduce any input instance of RESTRICTEDBI-STACKSORTING to an instance of MINUNCUT in polynomial time, such that the number of edges with endpoints in the same part of a solution (S, T) in the created instance of MINUNCUT is at most the shuffle count of a corresponding solution to the instance of RESTRICTEDBI-STACKSORTING, and the corresponding solution to RESTRICTEDBI-STACKSORTING can be computed in polynomial time.*

Proof. We create graph $H = (V_H, E_H)$ for MINUNCUT as follows. We put all vertices $1, 2, \dots, n$ to V_H , corresponding to the elements (integers) in π . Placing vertex i to S will correspond to pushing the element i to stack v_1 , and placing vertex i to T will correspond to pushing the element i to stack v_2 .

Recall that before any shuffle, we need to push all elements from s to the stacks. This will lead to the state $S_n = (s_n = \emptyset, l_n, p_n)$. Referring to Lemma 1, and especially to the count $sc(i, i + 1)$ for $i = 0, 1, \dots, n - 1$, we want to create edges between vertices in H that express the shuffles in $sc(i, i + 1)$.

Recall that $sc(i, i + 1)$ is the number of elements between i and $i + 1$ in list l_n that are larger than $i + 1$. Let x be any element that is larger than $i + 1$. Let us investigate the positions of the elements x, i , and $i + 1$ as they appear in the permutation π , and how these positions influence $sc(i, i + 1)$. **First**, observe that whenever x appears before i and $i + 1$ in π , then for any state l_n , x will never be between i and $i + 1$, and thus x will never be counted in $sc(i, i + 1)$. In this case, we do not create any edge between x and i nor any edge between x and $i + 1$ in graph H (as there should be no cost in the MINUNCUT problem). **Second**, assume now that x appears between elements i and $i + 1$ in the input permutation

π . In this case, either element i appears before x , or element $i + 1$ appears before x in π . Among i and $i + 1$, let z be the element that appears before x in π . Observe now that x appears between i and $i + 1$ in l_n (and thus contributes one to $sc(i, i + 1)$), if and only if x is placed on the same stack as z . Thus, in this case, we create an edge between z and x to account for the cost of one in every solution that places z and x to the same part in a partition (S, T) . **Third**, assume now that x appears after both elements i and $i + 1$ in the input sequence π . Observe now that x appears between i and $i + 1$ in the list l_n if and only if i and $i + 1$ are placed on two different stacks. For this reason, in this case, we create an auxiliary vertex $v(x, i, i + 1)$ and connect it to vertex i and to vertex $i + 1$ (and to no other vertex). Observe now that whenever vertices i and $i + 1$ are placed such that one is in part S and one is in part T , it does not matter to which part we place vertex $v(x, i, i + 1)$: any placement will incur cost exactly one, which exactly corresponds for x appearing between i and $i + 1$ in the list l_n (and thus corresponds for one count in $sc(i, i + 1)$). Also, observe that whenever i and $i + 1$ are placed to the same stack, there is no shuffle of x encountered in the cost $sc(i, i + 1)$, and this can be reflected by placing the vertex $v(x, i, i + 1)$ to the opposite part in which vertices i and $i + 1$ are, which leads to zero count for MINUNCUT.

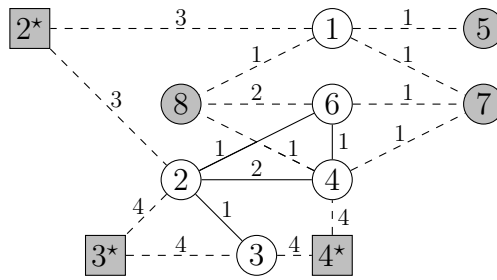
Now, to finish the proof, consider an instance of RESTRICTEDBI-STACKSORTING and the corresponding graph H created by the reduction above. Consider a solution (S, T) to MINUNCUT given by instance H . If in the solution there are vertices $i, i + 1$, and $v(x, i, i + 1)$ placed in the same part, we modify the solution by moving $v(x, i, i + 1)$ to the other part. This decreases the cost of the solution for MINUNCUT by two, and only affects the two edges incident to $v(x, i, i + 1)$. We repeat this process until no such three vertices $i, i + 1$, and $v(x, i, i + 1)$ can be found. This takes at most $O(n^2)$ many steps (one for every value i and every value $x > i$), and results in a new solution (S', T') to MINUNCUT of cost not larger than the original solution (S, T) to MINUNCUT. We can create a solution to the RESTRICTEDBI-STACKSORTING as follows: push element i to stack v_1 if and only if i is in S' (and otherwise, when i is in T' , push it to stack v_2). Observe now that every edge in the new solution (S', T') that has both endpoints in the same part corresponds to exactly one shuffle in the corresponding solution to RESTRICTEDBI-STACKSORTING, and thus the number of shuffles in the created solution for RESTRICTEDBI-STACKSORTING is the cost of the solution (S', T') for MINUNCUT. ◀

The construction from the proof is illustrated in Figure 4. The existence of approximation algorithms for RESTRICTEDBI-STACKSORTING with the claimed approximation ratios now comes as a direct corollary.

► **Corollary 4.** *There exists a randomized $O(\sqrt{\log n})$ -approximation algorithm for the problem RESTRICTEDBI-STACKSORTING. There exists a deterministic $O(\log n)$ -approximation algorithm for the problem RESTRICTEDBI-STACKSORTING.*

3.1 Beyond Two Stacks

It is a natural question to try to adapt our approach also to the cases $k \geq 3$. However, this is not possible (in a direct way). Our approach crucially depends on the alternative state representation introduced in Section 2.2 and on its relation to the shuffle count as expressed by Lemma 1. In our alternative state representation, we heavily used the fact that two stacks after step k can be merged into a linear list l_k . For three or more stacks, it is not clear how such a list could be created. Clearly, it is one of the interesting open problems to provide better approximation algorithms for the case $k = 3$.



■ **Figure 4** Graph H for the input $\pi = (2, 4, 3, 6, 1, 8, 7, 5)$. The circles denote the vertices of the permutation, the squares denote the auxiliary vertices $v(*, i, i + 1)$, and the edge labels denote the multiplicity of the edges between the vertices. The white vertices are in part S and grey vertices are in part T . Such a partition induces an uncut of size 5, and thus a solution to the sorting problem with 5 shuffles. This is optimum for this permutation.

3.2 Shuffling Before Midnight

Clearly, for $k = 2$, one can use our alternative state representation not also for the strong midnight-constraint, but also for the (normal) midnight constraint, where one can shuffle even if not all elements are pushed from s to the two stacks. For the strong midnight-constraint, the first n moves are only pushes from s , which results in some state S_n , which then uniquely induces what the algorithm does and the shuffle count the algorithm requires. In some sense, for the strong midnight-constraint, the only decisions to be made are to which stack shall we push element i , $i = \pi_1, \pi_2, \dots, \pi_n$. These decisions can be reflected by the auxiliary graph H that we create in the proof of Theorem 3. However, if we allow to shuffle before all n elements are pushed from s to the stacks, it is not clear how to create an auxiliary graph H which would reflect (via the MINUNCUT problem) the shuffle count for this case. In some sense, allowing shuffles before all elements from s are pushed would move the position of the shuffler index while the push operations are made, possibly changing the shuffle count impact of past and future push operations. To reflect this in H , it seems that we would need to update, add, and remove labeled edges in graph H . It is not clear that we can create H that would reflect such a dynamic behavior. A more extensive discussion on this topic can be found in the Master thesis of Pont [19]. We leave it a prominent open problem to settle the complexity of the sorting problem with $k = 2$ stacks and midnight constraint.

4 Conclusions

Motivated by the open problem of addressing the complexity of minimizing the shuffles when sorting with two stacks with or without the midnight constraint, we introduced the *restricted midnight-constraint* and studied the resulting RESTRICTEDBI-STACK SORTING problem. We showed that our problem is closely related to the MINUNCUT problem. This shows that the problem admits non-trivial approximation algorithms, which is in strong contrast to known approximability and inapproximability results to the other variants of the optimization problems that have been considered so far.

There are several open problems left by our paper. One of the most important ones is to settle whether the problem is NP-hard. Beyond the topic of this paper, i.e., sorting with two stacks with the strong midnight-constraint, one of the most interesting open problems is to investigate whether non-trivial approximation algorithms exist for the general, unrestricted case of $k = 2$ stacks.

References

- 1 Amit Agarwal, Moses Charikar, Konstantin Makarychev, and Yury Makarychev. $\mathcal{O}(\sqrt{\log n})$ Approximation Algorithms for min UnCut, min 2CNF Deletion, and Directed Cut Problems. In *Proc. Annual ACM Symposium on Theory of Computing (STOC)*, pages 573–581. ACM, 2005.
- 2 Therese Biedl, Alexander Golynski, Angèle M. Hamel, Alejandro López-Ortiz, and J. Ian Munro. Sorting with networks of data structures. *Discrete Applied Mathematics*, 158(15):1579–1586, 2010. doi:10.1016/j.dam.2010.06.007.
- 3 Ulrich Blasum, Michael R Bussieck, Winfried Hochstättler, Christoph Moll, Hans-Helmut Scheel, and Thomas Winter. Scheduling Trams in the Morning. *Mathematical Methods of Operations Research*, 49(1):137–148, 1999.
- 4 Miklós Bóna. A survey of stack-sorting disciplines. *The Electronic Journal of Combinatorics*, 9(2):A1.1–A1.16, 2003. URL: <http://eudml.org/doc/123106>.
- 5 Héctor J Carlo, Iris FA Vis, and Kees Jan Roodbergen. Storage Yard Operations in Container Terminals: Literature Overview, Trends, and Research Directions. *European journal of operational research*, 235(2):412–430, 2014.
- 6 S. Even and A. Itai. Queues, stacks, and graphs. In *Proc. International Symposium on the Theory of Machines and Computations*, pages 71–86. Academic Press, 1971.
- 7 Stefan Felsner and Martin Pergel. The Complexity of Sorting With Networks of Stacks and Queues. In *Proc. European Symposium on Algorithms (ESA)*, pages 417–429. Springer, 2008.
- 8 Naveen Garg, Vijay V Vazirani, and Mihalis Yannakakis. Approximate max-flow min-(multi) cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, 1996.
- 9 Brady Gilg, Torsten Klug, Rosemarie Martienssen, Joseph Paat, Thomas Schlechte, Christof Schulz, Senan Seymen, and Alexander Tesch. Conflict-Free Railway Track Assignment at Depots. *Journal of rail transport planning & management*, 8(1):16–28, 2018.
- 10 Michel X. Goemans and David P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *J. ACM*, 42(6):1115–1145, 1995. doi:10.1145/227683.227684.
- 11 Mohamed Hamdouni, Guy Desaulniers, Odile Marcotte, François Soumis, and Marianne Van Putten. Dispatching Buses in a Depot Using Block Patterns. *Transportation Science*, 40(3):364–377, 2006.
- 12 Stephan A Hartmann and Thomas A Runkler. Online Optimization of a Color Sorting Assembly Buffer Using Ant Colony Optimization. In *Operations Research Proceedings 2007*, pages 415–420. Springer, 2008.
- 13 Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proc. Symposium on the Complexity of Computer Computations*, pages 85–103. Springer, 1972. doi:10.1007/978-1-4684-2001-2_9.
- 14 Donald Ervin Knuth. *The Art of Computer Programming*, volume 1, chapter 2.2.1, pages 238–243. Addison-Wesley, 2nd edition, 1987.
- 15 Donald Ervin Knuth. *The Art of Computer Programming*, volume 3, chapter 5.2.3, page 168. Addison-Wesley, 2nd edition, 1998.
- 16 Felix G König, Macro Lübbecke, Rolf Möhring, Guido Schäfer, and Ines Spenke. Solutions to Real-World Instances of PSPACE-Complete Stacking. In *Proc. European Symposium on Algorithms (ESA)*, pages 729–740. Springer, 2007.
- 17 Felix G König and Marco E Lübbecke. Sorting With Complete Networks of Stacks. In *Proc. International Symposium on Algorithms and Computation (ISAAC)*, pages 895–906. Springer, 2008.
- 18 Adeline Pierrot and Dominique Rossin. 2-Stack Sorting is Polynomial. *Theory of Computing Systems*, 60(3):552–579, 2017. doi:10.1007/s00224-016-9743-8.
- 19 Marc Pont. The Bi-Stack Sorting Problem. Master’s thesis, Department of Data Science and Knowledge Engineering, Maastricht University, 2019.

3:12 On Sorting with a Network of Two Stacks

- 20 Rebecca Smith. Comparing Algorithms for Sorting with t Stacks in Series. *Annals of Combinatorics*, 8(1):113–121, 2004. doi:10.1007/s00026-004-0209-3.
- 21 Robert Tarjan. Sorting Using Networks of Queues and Stacks. *J. ACM*, 19(2):341–346, 1972.
- 22 Kevin Tierney, Dario Pacino, and Rune Møller Jensen. On the Complexity of Container Stowage Planning Problems. *Discrete Applied Mathematics*, 169:225–230, 2014.
- 23 W. Unger. On the k -colouring of circle graphs. In *Proc. International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 61–72. Springer, 1988.

Routing in Stochastic Public Transit Networks

Barbara Geissmann

Department of Computer Science, ETH Zurich, Switzerland
barbara.geissmann@inf.ethz.ch

Lukas Gianinazzi

Department of Computer Science, ETH Zurich, Switzerland
lukas.gianinazzi@inf.ethz.ch

Abstract

We present robust, adaptive routing policies for time-varying networks (temporal graphs) in the presence of random edge-failures. Such a policy answers the following question: *How can a traveler navigate a time-varying network where edges fail randomly in order to maximize the traveler's preference with respect to the arrival time?* Our routing policy is computable in near-linear time in the number of edges in the network (for the case when the edges fail independently of each other).

Using our robust routing policy, we show how to travel in a public transit network where the vehicles experience delays. To validate our approach, we present experiments using real-world delay data from the public transit network of the city of Zurich. Our experiments show that we obtain significantly improved outcomes compared to a purely schedule-based policy: The traveler is on time 5-11 percentage points more often for most destinations and 20-40 percentage points more often for certain remote destinations. Our implementation shows that the approach is fast enough for real-time usage. It computes a policy for 1-hour long journeys in around 0.1 seconds.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Route Planning, Public Transit Network, Temporal Graphs

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.4

1 Introduction

Real-World networks, such as communication and transportation networks, change over time and can be subject to delays and edge failures. Routing a traveler (or packet) through such a stochastic and time-dependent network (from a source vertex to a destination vertex) is challenging and requires *robust routing policies* that not only recommend a fixed route, but give alternatives in case the intended route becomes infeasible.

What constitutes a “best” policy depends on the preference the traveler has with respect to the arrival time. For example, one goal could be to minimize the expected time of arrival. Another goal could be to maximize the probability to arrive before a given deadline. These goals are distinct in the stochastic setting because they express different attitudes towards risk. In particular, when the goal is to arrive before a given deadline, as the available time to reach the destination becomes smaller, the best policy might have to become more and more risky (i.e. choose edges that are more likely to fail) in order to have a chance to arrive on time. In contrast, if only the expected arrival time has to be minimized, a route is chosen that is “good in most cases”. These preferences of the traveler can be expressed as *utility functions* on the arrival times.

Inherently time-dependent networks can be well-represented by *temporal graphs* [12, 1, 10], which model time explicitly: every edge is only available at certain points in time (consider Figure 1). For example, the connections of a public transit network are only available at certain points in time (when a vehicle departs).



© Barbara Geissmann and Lukas Gianinazzi;
licensed under Creative Commons License CC-BY

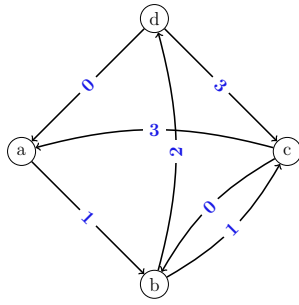
19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 4; pp. 4:1–4:18

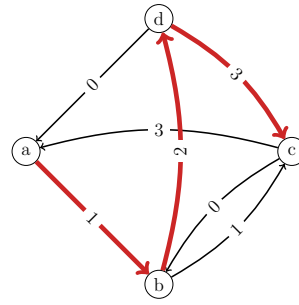
OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Every edge in a temporal graph can only be traversed at a certain *availability time*. In the example, there is an edge from vertex b to vertex d at availability time 2.



■ **Figure 2** A *strictly time-respecting path* has increasing edge availability times. The bold path is the only strictly time-respecting path from a to c and has *arrival time* 3.

A core characteristic of traveling inside a public transit network is the possibility of missing a *transfer connection*. This can occur because of delays of a vehicle or even because the connecting vehicle leaves too early. When a connection breaks, the traveler needs to change their route during the journey. Therefore, small delays of a vehicle can cause larger delays for a traveler’s journey.

This motivates adding *random edge failures* to a temporal graph, and thus obtaining a *faulty temporal graph*. For this setting, we show how to provide *optimal robust routing policies* for *any* efficiently computable *utility function* in near-linear time in the size of the temporal graph. In this paper, we show how public transit networks can be modeled as faulty temporal graphs and we give an algorithm for robust routing in public transit networks. We validate our robust routing policies by using *real-world* public transportation data. Note that our focus is not on catastrophic network failures due to accidents or other highly disruptive events, but on failures due to *everyday delays* caused for example by traffic congestion. In principle, our approach could also be applied to other time-varying and failure-prone networks such as ad-hoc or mobile phone networks.

1.1 Preliminaries

Temporal Graphs

To represent a network where edges can only be taken at a certain moment in time, we can use a temporal graph [12, 1, 10, 20, 8]. Formally, a temporal graph $G = (V, E, T)$ has vertices V and temporal edges E , where E is a multiset of directed edges on V . Each temporal edge $e = (u, v)$ has a nonnegative integer availability time $T(e)$. The semantic of the temporal graph is that at time $T(e)$, the edge $e = (u, v)$ can be used to go from vertex u to vertex v (see Figure 1). Note that there can be multiple temporal edges going from u to v . The number of edges that are incident to a vertex v is the degree $\text{deg}(v)$ of v .

Note that in a variant of temporal graphs (so-called interval graphs [10]) every edge is available during an *interval of time*. This is *not appropriate* for our purposes, as in our main application (for public transport), *connections are only available at discrete points in time*.

A *strictly time-respecting path* $p = e_1, \dots, e_k$ in a temporal graph (V, E, T) is a path in the graph (V, E) where the edges have increasing availability times (according to T). That is, if two edges e_i and e_{i+1} follow each other in the strictly time-respecting path p , then $T(e_i) < T(e_{i+1})$ (see Figure 2). We call the availability time of the last edge e_k in a time-respecting path p the *arrival time* of the path p .

Faulty Temporal Graphs

When the connections of a network change probabilistically over time, we can model this using a temporal graph with edges that fail at random. Formally, we augment the definition of a temporal graph with a failure distribution F over the edges.

A *faulty temporal graph* $\mathcal{G} = (V, E, T, F)$ has vertices V , directed edges from a multiset E , discrete nonnegative edge availability times $T : E \mapsto \mathbb{N}$, and a failure distribution F . The number of vertices is $n = |V|$ and the (maximum) number of edges in the faulty temporal graph is $m = |E|$. A faulty temporal graph defines a random variable whose outcomes are temporal graphs with vertices V , edges that are a subset of E , and with availability times given by T . Specifically, for an edge $e \in E$ we say it is *potentially available* at the fixed *availability time* $T(e)$. The edge fails with probability p_e and $F(e)$ is the indicator random variable for the event that edge e fails. If not stated otherwise, we assume that the edges fail independently of each other. This assumption is relaxed in Section 2.3, where the edge distributions follow a kind of Markovian assumption.

Robust Adaptive Routing

In the *robust adaptive routing* problem, a traveler in a faulty temporal graph starts out at a designated starting vertex at time 0. Whenever the traveler arrives at a vertex i at a time t , the traveler picks a temporal edge $e = (i, j)$ with availability time $T(e)$ larger than t . At that time $T(e)$, the traveler tries to go across this edge. If the edge does not fail, the traveler succeeds and arrives at the endpoint j of that edge at time $T(e)$. If the edge fails, the traveler remains at vertex i and must pick a new edge to take with availability time larger than $T(e)$. Note that this means that the traveler traverses a strictly time-respecting path in the faulty temporal graph using only edges that did not fail. The goal of the traveler is to maximize the expectation of a computable *utility function* of the time at which a destination vertex is reached. For example, they might want to arrive at a destination vertex before the *deadline* x with the largest possible probability. Then, the utility is 1 if the traveler arrives on time and 0 otherwise.

The algorithmic question that solves the robust adaptive routing problem is to preprocess the faulty temporal graph such that we can quickly answer the following *routing query*:

“When arriving at vertex i at time t , where should the traveler go next?”

A set of answers to these routing queries is called a *routing policy*. We are interested in *optimal routing policies* in the sense that they maximize the *expected utility* of the traveler.

If desired for a certain application, a routing policy could also be used to generate a temporal path a-priori (such a path represents the journey in case no connection breaks and can be thought of as an optimistic preview). This path would be followed until one of the edges fails. In this event, the policy would be queried again to compute a new path.

We continue with a more formal statement of the routing problem. We are given a faulty temporal graph $\mathcal{G} = (V, E, T, F)$ and a set of destination vertices $S \subseteq V$. A *policy* P maps every (non-destination) vertex $i \in V - S$ and every time t to an edge $e' = (i, j)$ with larger availability time $T(e') > t$, or to a special symbol \perp in case no edge $e' = (i, j)$ with availability time larger than t exists. The semantics of the policy are such that if the traveler is at vertex i at time t , they choose the edge $e = P(i, t)$ to traverse next according to the policy. If the edge $e = (i, j)$ does not fail, the traveler goes to the other endpoint j of e at time $T(e)$ and continues to choose an edge from there. Otherwise, the traveler stays at vertex i but the time also changes to $T(e)$. The traveler *stops* as soon as they reach a destination vertex or once the policy returns \perp , which means that the traveler is *stuck*.

We formalize the preference of the traveler with respect to the arrival time in a *utility function*. Such a *utility function* $U_i(t)$ maps every vertex $i \in S$ and every time t to a real-valued utility, where higher values correspond to a higher preference of the traveler. This terminology highlights the relation to Stochastic Optimal Control [2], where an agent tries to make decisions that optimize their expected utility. The idea is that if we arrive at vertex i through an edge e with availability time $t = T(e)$, the utility for the traveler is $U_i(t)$. For example, to maximize the *on-time arrival probability*, the utility is 1 if the traveler arrives at a destination vertex before a given deadline and 0 otherwise. If the traveler ever gets stuck, the utility obtains a smallest possible value that we denote by U^0 . For example, if the utility corresponds to the probability to arrive at a destination on time, then $U^0 = 0$.

The *utility of a policy* P starting from starting vertex i and starting time t is the value of the utility function at the vertex and time where the traveler stops. Note that the utility of a policy is a random variable. We consider the *expected utility* of the policy P , where the expectation is over the random failures of the edges of the faulty temporal graph.

1.2 Related Work

There is a vast variety of approaches to path finding problems in stochastic networks. We can categorize approaches based on the following criteria:

- **A-priori or Adaptive.** Does the traveler decide upfront which way to go (*a-priori*) [7, 15, 17] or can they change the route along the way (*adaptive*)?
- **Time-dependent or Time-independent.** Does the network change with time (*time-dependent*) or not (*time-independent*)? In a time-dependent network, the time it takes to go between two vertices changes depending on the time the traveler attempts to do so. An extreme case is when certain links are only available at discrete points in time.
- **Scoring Criterion.** How are different outcomes scored for the traveler? For example, does the traveler of the *stochastic* network want to maximize the *on time arrival* probability (SOTA) or does the traveler want to obtain a *least expected arrival time* (LET). Note that a utility function is not the only way to score a path. Alternatives include approaches which search for paths that are *pareto-optimal* with respect to multiple criteria [5, 16].
- **Runtime.** Is the solution obtained in polynomial time, pseudo-polynomial time (i.e. it depends on the number of time steps in the problem), or super-polynomial time?

Adaptive and Time-Independent

There are two motivations to take an adaptive approach as opposed to an a-priori approach. First, a-priori probabilistic path problems have only been solved in polynomial time for special cases (like for affine and exponential utility functions [15]) and hence it is pragmatic to take a different approach. Second, the outcome for the traveler can be improved if “live” information can be incorporated into the decision making process.

Fan and Nie [6] show termination for an algorithm to solve the adaptive SOTA problem (in the continuous time domain). They propose a set of (integral) equations which are solved iteratively, starting out with a trivial approximation, then using the approximation of the last iteration to compute the next iteration of the utility functions. Although they show convergence of the approximation to the true value, the algorithm can take an exponential number of iterations.

Samarayake et al. [19] observe that there is a minimum time that it takes to traverse an edge. They obtain pseudo-polynomial runtime (in the number of such traversals that can occur within the time-budget). Instead of computing the utility functions iteratively,

they construct parts of the functions one after the other. They discretize the problem by dividing the time-domain into y evenly sized pieces (y is a parameter such that the pieces are smaller than the minimum time it takes to traverse an edge). On this discretized version, they obtain a runtime of a $O(my^2)$, where m is the number of edges.

Because the SOTA utility functions are not of some nice form that can be integrated efficiently, the surveyed approaches all eventually discretize the time-domain. Samarayake et al. use evenly spaced time-intervals and do not provide bounds on the approximation quality as a function of the number of time-interval. This downside is addressed by Hoy and Nikolova [11], who give a polynomial-time approximation scheme for the SOTA and LET problem on *acyclic directed graphs* that obtains an additive error of $1/\epsilon$ in $O(mn^2/\epsilon^2)$ time. Similar to our approach, they can handle general scoring functions that depend on the arrival time of the traveler.

Adaptive and Time-Dependent SOTA

Transit networks consisting of trains and buses are different from street networks because vehicles that connect physical locations in a transit network are only available at a specific point in time (before the vehicle leaves a station), whereas streets remain available most of the time (although delays and infrequent disruptions are possible). In particular, missed transfer connections between distinct vehicles can play a crucial role in transit networks.

Keyhani et al. [13] looks at estimating the reliability of transfers and fixed (a-priori) paths in a train network. Keyhani [14] deepens this work on the reliability of train transfers and connections. They present an adaptive approach to solving SOTA in a similar transit network problem setting as ours. However, they allow the traveler to change the route depending on the arrival time at every vertex. This requires a model of the arrival-time distributions and leads their algorithms to have pseudo-polynomial runtime in the size of the support of the arrival-time distributions. Another difference to our work is that Keyhani does not represent the schedule as a temporal graph, but use their own problem-specific model.

Adaptive and Time-Dependent LET

In the *bus network problem* [3], the traveler decides whether to take a bus whenever it arrives. The traveler has access to the statistics of the bus arrivals, but they do not know exactly when a bus will actually arrive, until it arrives. The goal is to reduce the *expected time* that a policy takes to move a traveler from the start station to the destination station.

Boyan and Mitzenmacher [3] present results for the case when the buses arrive independently of each other and satisfy additional conditions (in particular they can be distributed according to exponential, uniform, or normal distributions). They generalized a previous more limited result by Datar and Ranade [4]. In order to compute a policy minimizing the expected travel time in polynomial time, they need to be able to compute the expected arrival time of a bus given that it has not arrived yet. However, exact computation of these expectations involves a convolution that can take polynomial time in the number of time steps considered and it is not shown how an approximate solution to the expectations impacts the accuracy of the result.

In contrast to our problem setting, the traveler in the bus network problem can change their decision whenever a bus arrives (whereas Keyhani [14] and in our model we only allow a decision when the traveler arrives at a station or a connection breaks). On the other hand, in our model we allow more general delay distributions and *our runtime is strongly polynomial*.

1.3 Our Contribution

We obtain a near-linear runtime for computing an optimal robust adaptive routing policy in a faulty temporal graph. For a faulty temporal graph with m edges that fail independently of each other, computing an optimal routing policy takes $O(m \log m)$ time. We need $O(m)$ space to store the policy and take $O(\log m)$ time per query. We allow any utility function that can be evaluated in $O(\log m)$ time.

When the edge failure probability depends on the last edge the traveler attempted to traverse, we compute an optimal routing policy in $O(m \log m + \sum_{i \in V} \deg^2(i))$ time.

As an application of our model, we represent traveling inside a *public transit network that is subject to delays* as robust adaptive routing in a faulty temporal graph. We transform a timetable with N entries where at most d vehicles run through any station into a faulty temporal graph with $O(Nd)$ edges. This gives $O(Nd \log N)$ time to compute a robust routing policy for a transit network with independent delays. We evaluate our routing policy using real-world transit network delay data from the public transit network of the city of Zurich. We compare our approach to a traveler that travels to arrive as early as possible using only the schedule provided by the city of Zurich and to a traveler that has perfect knowledge of all future delays. Our evaluation shows that our model is *accurate in predicting the probability of being on time* and our routing policy provides (in less than 0.1 seconds) significant improvements over an approach that neglects delays.

2 Robust Adaptive Routing

The efficiency and generality of our approach is enabled by two observations. First, the problem has an acyclic nature, since the traveler navigates strictly time-respecting paths. Hence, a dynamic programming formulation emerges. This initial dynamic program (presented in Section 2.1) is, however, too slow because it depends on the largest availability time. Second, only certain points in time matter (those where there is an edge with that availability time). This leads to an improved dynamic program (described in Section 2.2) that achieves near-linear runtime.

2.1 Pseudo-Polynomial Time Algorithm

We start out with a basic dynamic program to compute a routing policy for any faulty temporal graph (with independent edge failures).

For every vertex $i \in V$ and every time $t \in \mathbb{N}$, we denote the computed expected utility starting from vertex i at time t with $u_i(t)$. The basic idea is to find a recursion for $u_i(t)$, parameterized by the current vertex i and the current time t . Since the traveler traverses a strictly time-respecting temporal path, the traversed edges have increasing availability time. Therefore, the subproblems overlap in an acyclic way and this gives a dynamic program.

Let us start with the base cases. If the vertex i is a destination vertex, the expected utility $u_i(t)$ coincides with the value of the utility function, hence we set $u_i(t) = U_i(t)$.

Next, we recursively describe the best decision to take being in vertex i at time t . Intuitively, the idea is to try every incident outgoing edge and then take the best such edge. For this, we need to compute the expected utility given that we take a particular edge. For each such edge e with an availability time $T(e)$ larger than the current time t , we condition on the event that the edge e fails. Using the law of total expectation, we relate the expected utility at time t to an expected utility at some time larger than t . If an edge $e = (i, j)$ fails,

the traveler stays at vertex i and the (conditional) expected utility is $u_i(T(e))$. Otherwise, the traveler reaches vertex j and the (conditional) expected utility is $u_j(T(e))$. We can express this in the following recursive formula (for any vertex i that is not a destination):

$$u_i(t) = \max_{\substack{e=(i,j) \in E \\ T(e) > t}} \left((1 - p_e) \cdot u_j(T(e)) + p_e \cdot u_i(T(e)) \right) .$$

For any time t and vertex i , the expected utility $u_i(t)$ at vertex i and time t only depends on values $u_j(t')$ with $t' > t$. Hence, we can process the expected utilities $u_i(t)$ in order of decreasing time t . When the traveler is at some vertex i at some time t , the routing policy is to take the edge $e = (i, j)$ which obtains the maximum value in the expression for $u_i(t)$ (take any edge if several edges are tied for the same value). If there is no edge e with $T(e) > t$, then $u_i(t) = U^0$. In that case, the traveler is stuck and cannot reach a destination at all (i.e., the policy returns \perp). See Appendix A.1 for an inductive correctness proof of the algorithm.

Let x be the largest availability time that occurs in T , then the runtime of this approach is $O(m x)$. Initializing the base cases takes time $O(x + n)$. Afterwards, each of the vertices needs to compute at most x different values (the utility for all times larger than x is trivially U^0 and does not need to be computed). To compute the value for a particular vertex and time, we need to look up an already computed value for each of the neighbors. Thus, a vertex i with degree $\deg(i)$ takes $O(\deg(i) x)$ time to find its best decision. The runtime is thus $O(\sum_{i \in V} \deg(i) x) = O(m x)$. As we explicitly store the result to all routing queries, the routing policy uses $O(n x)$ space and each routing query takes $O(1)$ time.

2.2 Near-Linear Time Algorithm

The problem with the basic dynamic program is that its runtime and space depends on the largest availability time x . This value is not polynomial in the input size and so the basic dynamic program runs in pseudo-polynomial time. In practice, this means that increasing the time-resolution of the data (say from measuring in minutes to seconds) also increases the runtime of the algorithm proportionally. We now show how to reduce the runtime of the basic algorithm by improving the order in which we evaluate the recursion and by leaving out redundant points in time. The new algorithm runs in near-linear time.

Observe that only those times are relevant for the traveler where there is some incident edge. More precisely, if there is no edge *leaving* vertex i inside some time interval t_1, \dots, t_2 , then the expected utility for vertex i is the same for all those times. This is because when the traveler is at vertex i , the traveler cannot take any new decision in that time range and the traveler does not learn any new information. It thus suffices to compute the expected utility for t_2 . We store the computed utilities sorted by increasing times for each vertex. To query the expected utility at a certain time, we do a binary search for the next largest time that has a computed value.

A closer look at the recursive equation reveals that the expected utility for vertex i and time t is the maximum of the expected utility at time $t + 1$ and the maximum possible expected utility given that we take an edge leaving at time $t + 1$. We therefore process the *edges* in decreasing order of availability times. For each edge $e = (i, j)$, we compute the best possible expected utility $u(e)$ given that the traveler plans to take this edge e using the expression

$$u(e) = (1 - p_e) \cdot u_j(T(e)) + p_e \cdot u_i(T(e)) .$$

After processing all edges at time $t + 1$, we update all adjacent vertices. For each vertex i that has some edge leaving at time $t + 1$, we set

$$u_i(t) = \max \left(\left(\max_{\substack{e=(i,j) \in E \\ T(e)=t+1}} u(e) \right), u_i(t+1) \right) .$$

If the maximum expected utility is obtained by taking some edge e at time $t + 1$, this edge is chosen for time t . Otherwise, the same edge is chosen as for time $t + 1$. Recall that we do not explicitly store the choices and utilities for all times and vertices, but only remember decisions for vertices and times, where the vertex has an outgoing edge.

► **Theorem 1.** *Computing an optimal routing policy in a faulty temporal graph with independent edge failures takes $O(m \log m)$ time. The policy uses $O(m)$ space and a routing query takes $O(\log m)$ time.*

Proof. Sorting the edges takes $O(m \log m)$ time. Then, each edge e is processed once to compute $u(e)$, which uses two already computed utilities. Looking up those utilities takes $O(\log m)$ time (by using a binary search for the successor). Updating a vertex at a certain time takes time proportional to the number of edges at that time, so $O(1)$ per edge. Inserting a new expected utility value into the sorted array takes $O(1)$ amortized time by using standard array doubling (store the array in decreasing order of time so that inserting a new expected utility always occurs at the end of the array). ◀

2.3 Last-Edge Markovian Failures

So far, we assumed the edges to fail completely independently of each other and independently of time. We can also consider the situation when the probability for an edge to fail depends on the last edge the traveler planned to take (they either attempted to traverse this edge and failed or succeeded to traverse this edge). By replacing our independence assumption by a Markovian independence assumption given the last edge the traveler attempted to take, we obtain *Last-Edge-Markovian failures*. As detailed in Appendix B, we can modify our dynamic program to obtain the following result:

► **Theorem 2.** *Computing an optimal routing policy in a faulty temporal graph with Last-Edge-Markovian failures takes $O(m \log m + \sum_{i \in V} \deg^2(i))$ time. The policy uses $O(\sum_{i \in V} \deg(i))$ space and a routing query takes $O(1)$ time.*

3 Applications in Public Transit Networks

Traveling in a public transit network in the presence of delays can be modeled as robust adaptive routing in a faulty temporal graph, as we show in this section. In a public transit network, there are several *lines* of buses, trains, trams, and other vehicles. Each of those lines connects a series of *stations* in a predetermined order. Along each line, vehicles run according to a *schedule* which prescribes when a vehicle is supposed to arrive and to leave a station. The schedule contains N tuples that contain the line of the vehicle, the departure time, departure station, arrival time, and arrival station.

We assume that the traveler leaves the start station s_{start} at the starting time and wants to arrive at the destination station s_{dest} while maximizing their preference with respect to the arrival time: this preference is expressed as a *utility function* $u(t)$ of the arrival time and the goal is to maximize the expected value of this utility function (the *expected utility*).

At every station, the traveler can get off the current vehicle and attempt to *transfer* onto another vehicle (which succeeds if the latter vehicle departs *after* the arrival time of the current one plus the time it takes to transfer between the two vehicles).

Our model restricts traveling to routes that are feasible according to the schedule. This is somewhat pessimistic in that certain connections infeasible in the schedule could be feasible in practice due to delays and early arrivals. However, we argue that being slightly pessimistic is compatible with our goal of giving robust routes. Moreover, recommending such infeasible routes might be counter-intuitive to users of a transit system.

3.1 Public Transit Network Model

Temporal Graph Model

We map the transit network onto a temporal graph $G = (V, E, T, F)$. Note that our model is related to the time-expanded-graph model in [18], where each edge has a weight instead of an availability time. Without modeling failure probabilities, the latter model can be used to compute a route which minimizes the earliest arrival time.

To construct the temporal graph, we first add the vertices and edges that correspond to a single-hop ride with a vehicle. Say that, according to the schedule, some vehicle of line l leaves station s at time t and arrives at station s' at time t' . Then, there is a *departure vertex* (dep, l, s, t) and an *arrival vertex* (arr, l, s', t') connected by a temporal edge with availability time t' . Next, we add the connections that correspond either to transfers or to staying in a vehicle. In particular, there is an edge connecting every arrival vertex (arr, l, s', t') to every departure vertex $(\text{dep}, l', s', t'')$ at the same station s' with larger departure time than arrival time ($t'' > t'$). Finally, there is a special extra vertex *start*. The *start* vertex is connected to every departure vertex $(\text{dep}, l, s_{\text{start}}, t)$ at the start station s_{start} with a temporal edge at time $t - 1$ equal to the starting time minus 1. This shifting by one is necessary since the traveler traverses strictly time-respecting paths.

The temporal graph has $n = 2N + 1$ vertices (recall that N is the size of the schedule). Because of the transfer edges, the number of edges of the constructed graph depends on the largest number of vehicles that pass through a station. Let d be this maximum number of vehicles per station. Then, the temporal graph contains $m = O(Nd)$ edges.

Every arrival vertex $(\text{arr}, l, s_{\text{dest}}, t)$ at the destination station s_{dest} is a destination vertex. The *utility* at such a *destination vertex* $(\text{arr}, l, s_{\text{dest}}, t)$ should correspond to an estimate of the expected utility when using the vehicle v that arrives at that vertex. Given a list of observed arrival times t_1, \dots, t_k for vehicle v at station s_{dest} , compute $U_{s_{\text{dest}}} = \left(\sum_i^k U_{s_{\text{dest}}}(t_i) \right) / k$, the average value of the utility function for those arrival times (Justification in Appendix A.2).

We now describe how to set the *edge failure probabilities* based on the probability that vehicles are delayed. For simplicity, we assume that neither edges corresponding to traveling (these go from a departure vertex to an arrival vertex) nor edges that correspond to staying inside a vehicle can fail. This means that only transfer edges, which go from an arrival vertex of some line l to a departure vertex of some other line l' can fail completely. Intuitively, the probability for this edge to fail is the probability that we are too late to catch the connection. We are given samples for the arrival time of a vehicle a , departure time of vehicle b , and transfer time between the two platforms. Then, the failure probability for the transfer edge assuming independent vehicle travel is estimated as the fraction of samples where the transfer is infeasible (i.e. the arrival time plus transfer time is larger than the departure time).

Note that we do not require to model the actual delay of vehicles (which can be time-dependent and congestion-dependent [9]), since we are only interested in the probability to miss a connection.

Computation of the Policy

We can apply our policy construction algorithm from Section 2.2 to our model of a transit network. We call such a policy a *robust transit network routing policy* and we obtain the following bounds, which follow from Theorem 1 since the number of edges is in $O(Nd)$:

► **Corollary 3.** *For a schedule of size N where at most d vehicles run through any station, computing a robust transit network routing policy with independent edge failures takes $O(Nd \log N)$ time and uses $O(Nd)$ space.*

3.2 Experimental Methodology

We evaluate our algorithm by applying it to the public transportation network of the city of Zurich (ZVV network). We investigate the performance of our routing policy on real transit network delay data for the year 2018. Throughout all experiments, we consider a traveler who wants to arrive at a given destination station at a given (hard) *arrival deadline* time x , within some *time budget* b (i.e., the traveler starts the journey at time $x - b$ in some station). We considered three main questions for our study:

1. **Quality of the solution.** How well does the computed policy compare to a deterministic *schedule-based* policy and how much room for improvement is there compared to an *oracle policy* with perfect knowledge of the future?
2. **Model error.** How well do the predicted utilities (according to the policy) match the simulated utilities when following the policy? The model error evaluates the underlying model assumptions empirically (e.g., that edges fail independently).
3. **Runtime.** How does the time to compute a policy scale with the time budget b ?

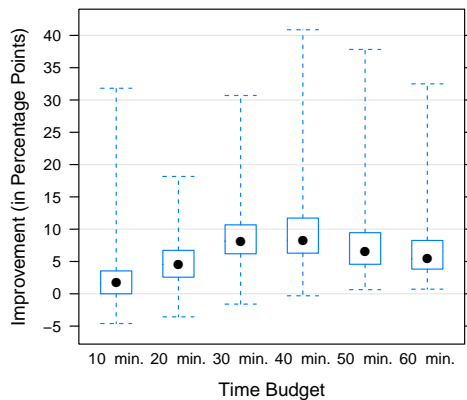
We evaluated these questions for time budgets between 10 and 60 minutes in 10-minute increments. For our policies, we train an edge failure model that uses the *last two weeks of delay data* before the first evaluated day.

Evaluation Approach

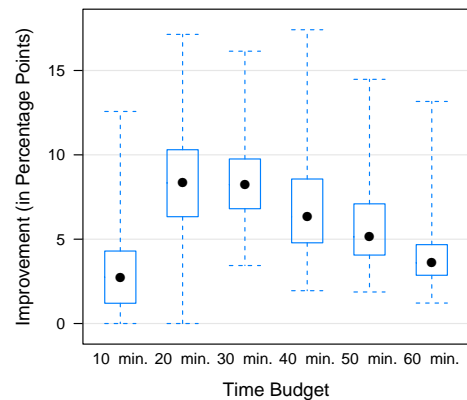
We evaluate the performance of the candidate policies for 38 destinations (and for each destination for every possible departure station) and 13 random destination deadlines between 7am and 6pm.

For the evaluation, we first compute the frequency (during a 1 month period) with which we can reach a given destination from a given source station at a given time using a given candidate policy (and we repeat this for every such set of parameters). Then, we compute the desired quantity (either an error value or some difference in utility) by averaging over all source stations for the given destination. For the error metrics we take the average over those source stations which reach the destination with nonzero probability (using the oracle policy). Note that for the other source stations the error metric is trivially 0.

In Figures 3 – 6 we use box plots to show the results, where the dot indicates the median of the values, the boxes indicate the lower and upper quartile, and the whiskers indicate minima and maxima. Note that each plot summarizes data for different destination stations and deadlines (and thus the variation is not due to probabilistic reasons only, but mainly due to differences depending on the destination stations). See Appendix C for a more detailed description of our experimental setup.



■ **Figure 3** Average improvement over the schedule-based policy, plotted by time budget.



■ **Figure 4** Average potential for improvement of our policy with respect to the oracle policy, by time budget.

3.3 Results

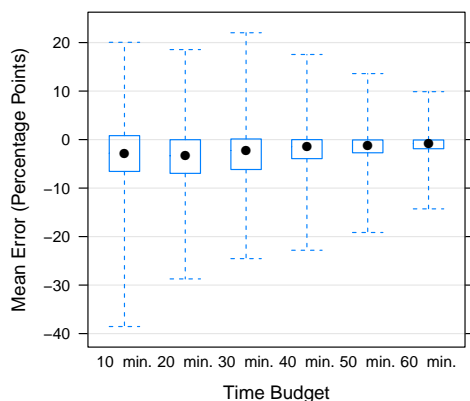
Quality of the Solution

Figure 3 shows that the largest improvement (with respect to the schedule-based policy) occurs for time budgets of 30 and 40 minutes, where it is 7 – 11 percentage points depending on the destination station and deadline. This makes sense, as for too short time budgets there are few possible routes and there is not much to improve, while for very long time budgets, the choices matter less as the traveler has enough slack time for delays. For the longer configurations the median improvement is around 5 percentage points. For the very short configurations (i.e., 10 minutes) the median improvement is only slight with 1 – 2 percentage points. Note that there are several destination stations where our improvement over the deterministic policy is especially high (i.e., 10-40 percentage points). These are the stations reachable by a single bus, where delays have a larger impact. Figure 4 shows that the maximum potential for improvement over our policy lies between 2 and 8 percentage points, where our policy is closer to optimal for longer time budgets. In conclusion, we see significant improvements for the vast majority of configurations. Improvements are on the order of improving the probability to be on time by around 5 – 11 percentage points.

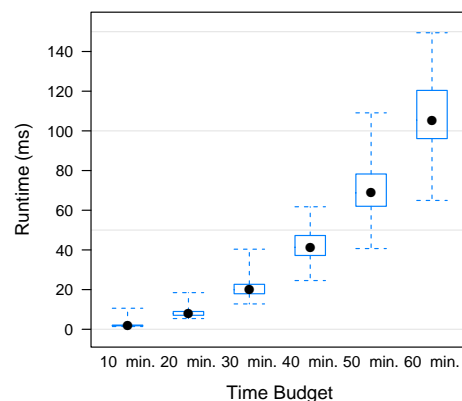
Model Accuracy

Figure 5 shows that the mean model error is small (less than 5 percentage points for all time budgets). With increasing time budget, the mean error decreases slightly. Note that the mean error is negative for most of the cases, which means that the simulated utility from the policy is better than the expected utility from the model. Since the mean model error is relatively small, this shows that in practice, *the assumption that the edges fail independently of each other does not significantly impact the applicability of the approach.*

Note that the obtained error is nontrivial, as, interestingly, using 5 months old delay data would yield a median error of 25 percentage points (and provide no improvement w.r.t. the schedule-based policy): Our approach is able to capture seasonal variations in delay distributions.



■ **Figure 5** Mean difference between the computed utility according to the policy and the observed utility.



■ **Figure 6** Time to compute a policy for varying time budgets.

Runtime

As seen in Figure 6, the median runtime is around 0.02 seconds for the 30 minutes time budget and around 0.105 seconds for the 60 minutes time budget. As the time budget b increases, we expect the runtime to grow slightly faster than proportional to b^2 . This is because the number of possible transfers (and hence the size of the graph) increases quadratically with the time budget. The observed runtimes roughly follow the predicted trend.

4 Conclusion

We showed an approach to robust adaptive routing in time-dependent networks that both is tractable (computable in near-linear time in the size of the network) and yields useful improvements in practice over a purely schedule-based routing despite our simplifying assumptions (as exemplified by our analysis of travel inside a public transit network).

One next step could be to try variations on how to train the edge probabilities. It would be interesting to investigate other types of dependencies between the edge failure distributions and how they affect the quality of the solution.

We saw that the age of the training data affects the results. In principle, one could alter the model every day and always use the most up-to-date delay data (say over the last one or two weeks) instead of changing the model every month as we did in our experiments. Another extension would be to include other forms of travel, for example travel by foot. This would not require a fundamental change in the model, but just a way to estimate travel durations for these trips.

In terms of theory, it would be interesting to know how the error in approximating the edge failure probabilities affects the error in the quality of the solution.

Future work could also include looking at applications of our approach for routing in *ephemeral or ad-hoc communication networks*. In that context, a distributed computation of the policy might be interesting.

References

- 1 Eleni C. Akrida, Jurek Czyzowicz, Leszek Gasieniec, Lukasz Kuszner, and Paul G. Spirakis. Temporal Flows in Temporal Networks. In *Algorithms and Complexity - 10th International Conference, CIAC 2017, Athens, Greece, May 24-26, 2017, Proceedings*, pages 43–54, 2017. doi:10.1007/978-3-319-57586-5_5.
- 2 Dimitri P. Bertsekas. *Dynamic programming and optimal control, 3rd Edition*. Athena Scientific, 2005. URL: <http://www.worldcat.org/oclc/314894080>.
- 3 Justin A. Boyan and Michael Mitzenmacher. Improved results for route planning in stochastic transportation. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 895–902, 2001. URL: <http://dl.acm.org/citation.cfm?id=365411.365803>.
- 4 Mayur Datar and Abhiram G. Ranade. Commuting with delay prone buses. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA.*, pages 22–29, 2000. URL: <http://dl.acm.org/citation.cfm?id=338219.338228>.
- 5 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria Shortest Paths in Time-Dependent Train Networks. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, pages 347–361, 2008. doi:10.1007/978-3-540-68552-4_26.
- 6 Yueyue Fan and Yu Nie. Optimal Routing for Maximizing the Travel Time Reliability. *Networks and Spatial Economics*, 6(3):333–344, September 2006. doi:10.1007/s11067-006-9287-6.
- 7 H. Frank. Shortest Paths in Probabilistic Graphs. *Operations Research*, 17(4):583–599, 1969. doi:10.1287/opre.17.4.583.
- 8 Viswanath Gunturi, Shashi Shekhar, and Arnab Bhattacharya. Minimum Spanning Tree on Spatio-Temporal Networks. In *Database and Expert Systems Applications, 21th International Conference, DEXA 2010, Bilbao, Spain, August 30 - September 3, 2010, Proceedings, Part II*, pages 149–158, 2010. doi:10.1007/978-3-642-15251-1_11.
- 9 Dirk Heidemann. Queue length and delay distributions at traffic signals. *Transportation Research Part B: Methodological*, 28(5):377–389, 1994. doi:10.1016/0191-2615(94)90036-1.
- 10 Petter Holme and Jari Saramäki. Temporal Networks. *CoRR*, abs/1108.1780, 2011. arXiv: 1108.1780.
- 11 Darrell Hoy and Evdokia Nikolova. Approximately Optimal Risk-averse Routing Policies via Adaptive Discretization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI'15*, pages 3533–3539. AAAI Press, 2015. URL: <http://dl.acm.org/citation.cfm?id=2888116.2888207>.
- 12 David Kempe, Jon M. Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 504–513, 2000. doi:10.1145/335305.335364.
- 13 Mohammad H Keyhani, Mathias Schnee, Karsten Weihe, and Hans-Peter Zorn. Reliability and delay distributions of train connections. In *OASIS-OpenAccess Series in Informatics*, volume 25. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.
- 14 Mohammad Hossein Keyhani. *Computing Highly Reliable Train Journeys*. PhD thesis, Technische Universität, Darmstadt, 2017.
- 15 Ronald Prescott Loui. Optimal Paths in Graphs with Stochastic or Multidimensional Weights. *Commun. ACM*, 26(9):670–676, September 1983. doi:10.1145/358172.358406.
- 16 Matthias Müller-Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-criteria Pareto Search. In *Algorithmic Methods for Railway Optimization, International Dagstuhl Workshop, Dagstuhl Castle, Germany, June 20-25, 2004, 4th International Workshop, ATMOS 2004, Bergen, Norway, September 16-17, 2004, Revised Selected Papers*, pages 246–263, 2004. doi:10.1007/978-3-540-74247-0_13.

- 17 Mehrdad Niknami and Samitha Samaranyake. Tractable Pathfinding for the Stochastic On-Time Arrival Problem. In *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, pages 231–245, 2016. doi: 10.1007/978-3-319-38851-9_16.
- 18 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Experimental Comparison of Shortest Path Approaches for Timetable Information. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Combinatorics and Combinatorics, New Orleans, LA, USA, January 10, 2004*, pages 88–99, 2004.
- 19 S. Samaranyake, S. Blandin, and A. Bayen. A tractable class of algorithms for reliable routing in stochastic networks. *Transportation Research Part C: Emerging Technologies*, 20(1):199–217, 2012. Special issue on Optimization in Public Transport+ISTT2011. doi: 10.1016/j.trc.2011.05.009.
- 20 B Bui Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, 2003.

A Correctness Proofs

We show that the computed policies indeed achieve the largest possible utility. Moreover, to justify our transit network model, we slightly generalize our notion of utility functions to *probabilistic utility functions*. Then, we show that computing an optimal policy with respect to the expectation of this probabilistic utility suffices to obtain an optimal policy with respect to the probabilistic utility function.

We denote the expectation of a random variable X as $\mathbf{E}[X]$, denote its conditional expectation given another random variable Y with $\mathbf{E}[X|Y]$, and denote the probability of an event E with $\mathbf{P}[E]$.

A.1 Deterministic Utility Functions

We show that the algorithm presented in Section 2.1 computes an optimal policy. The algorithm in Section 2.2 is equivalent, as already argued therein.

► **Theorem 4.** *The algorithm from Section 2.1 computes a policy that obtains the largest possible expected utility for all start vertexes and start times.*

Proof. The proof is by strong induction with decreasing time. The basic idea is that a policy with largest expected utility starting from vertex i at time t must try to use some edge e leaving i at time larger than t and use a policy that maximizes the expected utility for each of the two possible outcomes (edge e fails or does not fail).

Let $u_i^*(t)$ be the largest possible expected utility of any policy starting at vertex i at time t . That is, $u_i^*(t)$ gives the *true optimal utility*, whereas $u_i(t)$ is the *computed utility*. The proof consists of showing the two are equal.

For any time t , the induction hypothesis $H(t)$ is that for all times $t' > t$ and all vertexes i , we have that $u_i(t') = u_i^*(t')$. Assume that $H(t)$ holds for some $t > 0$. We show that $H(t - 1)$ holds. Consider some vertex i . If i is a destination vertex, then $u_i(t) = U_i(t) = u_i^*(t)$ holds by construction. By $H(t)$, then $u_i(t') = U_i(t') = u_i^*(t')$ holds for any $t' > t$.

Next, consider the case where i is not a destination vertex. If there is no edge leaving vertex i at a time larger than t , then $u_i(t) = U^0 = u_i^*(t)$, as there is no way to reach the destination. Otherwise, the following equation is used to compute $u_i(t)$:

$$u_i(t) = \max_{\substack{e=(i,j) \in E \\ T(e) > t}} \left((1 - p_e) \cdot u_j(T(e)) + p_e \cdot u_i(T(e)) \right).$$

By induction hypothesis, all the utilities appearing on the right-hand side correspond to the largest possible expected utilities:

$$u_i(t) = \max_{\substack{e=(i,j) \in E \\ T(e) > t}} \left((1 - p_e) \cdot u_j^*(T(e)) + p_e \cdot u_i^*(T(e)) \right).$$

Let P_e be the policy that uses edge e first and continues using the optimal policy after that. Let $U(P_e)$ be its utility starting from vertex i at time t . Then, we can see that:

$$\begin{aligned} u_i(t) &= \max_{\substack{e=(i,j) \in E \\ T(e) > t}} \left(\mathbf{P}[F(e) = 1] \cdot \mathbf{E}[U(P_e) \mid F(e) = 1] \right. \\ &\quad \left. + \mathbf{P}[F(e) = 0] \cdot \mathbf{E}[U(P_e) \mid F(e) = 0] \right) \\ &= \max_{\substack{e=(i,j) \in E \\ T(e) > t}} \mathbf{E}[U(P_e)] \\ &= u_i^*(t), \end{aligned}$$

where the last step follows because among all possible policies P_e we choose the one with largest expected utility. Finally, note that $H(t)$ implies that $u_i(t') = U_i(t') = u_i^*(t')$ holds for any $t' > t$. ◀

A.2 Probabilistic Utility Functions

Recall that in our application to public transit networks in Section 3.1, we set the utility at a destination vertex to the average utility for the observed arrival times. We proceed to justify this as a way to compute optimal policies with respect to *probabilistic utility functions*.

For each destination vertex i , we introduce a *random variable* \mathbb{U}_i that gives a utility distribution at the destination vertex i (given that we arrive at vertex i). In order to be able to define the expected value of a policy with respect to such utilities, we require that these random variables have finite expectation. Moreover, we assume that each random variable \mathbb{U}_i is independent of the edge failure variables F . As before, we require that there is a smallest (deterministic) utility U^0 with the property that if the traveler gets stuck at a non-destination vertex j , their utility is always $\mathbb{U}_j = U^0$.

The *expected utility* of a policy with start vertex i and start time t is now the expected value of the utility function at the vertex where the traveler stops. Here, the expectation is both over the random edge failures and the outcome of the utility functions.

We prove that if we *replace the probabilistic utility functions with their expectations* and compute a policy with maximum value with respect to these deterministic utilities, we obtain a policy with maximum expected utility with respect to the probabilistic utilities.

► **Lemma 5.** *For every destination vertex i , set the utility $U_i(t)$ to the expected value $\mathbf{E}[\mathbb{U}_i]$, (for all t). Let P be a policy with maximum expected utility with respect to $U_i(t)$. Then, P is a policy with maximum expected utility with respect to the probabilistic utilities \mathbb{U}_i .*

Proof. Let the start vertex j and start time t be arbitrary.

Let P' be some policy, let $\mathbb{U}(P')$ be its utility with respect to the probabilistic utilities, and let $U(P')$ be its utility with respect to the deterministic utilities $\mathbf{E}[\mathbb{U}_i]$. The goal is to show that $\mathbf{E}[\mathbb{U}(P')] = \mathbf{E}[U(P')]$. Let $\text{STOP}(P')$ be the random variable that denotes the vertex where the traveler stops. By conditional expectation, we have that

$$\begin{aligned} \mathbf{E}[\mathbb{U}(P')] &= \sum_{i \in V} \mathbf{E}[\mathbb{U}(P') \mid \text{STOP}(P') = i] \cdot \mathbf{P}[\text{STOP}(P') = i] \\ &= \sum_{i \in V} \mathbf{E}[\mathbb{U}_i \mid \text{STOP}(P') = i] \cdot \mathbf{P}[\text{STOP}(P') = i] \\ &= \sum_{i \in V} \mathbf{E}[\mathbb{U}_i] \cdot \mathbf{P}[\text{STOP}(P') = i] \\ &= \sum_{i \in V} \mathbf{E}[U(P') \mid \text{STOP}(P') = i] \cdot \mathbf{P}[\text{STOP}(P') = i] \\ &= \mathbf{E}[U(P')] \quad . \end{aligned}$$

where we can leave out conditioning on $\text{STOP}(P')$ because the utilities \mathbb{U}_i do not depend on the edge failures (which are the only thing that affects where the traveler stops). We can see that a policy P' that maximizes the expected utility $\mathbf{E}[U(P')]$ with respect to the deterministic utilities also maximizes the expected utility $\mathbf{E}[\mathbb{U}(P')]$ with respect to the probabilistic utilities. ◀

B Last-Edge Markovian Failures

To compute an optimal policy for the case of Last-Edge-Markovian edge failures, we condition the expected utility equations on the *last edge that the traveler planned to take*. Since we are in a temporal graph, this implicitly also encodes the time at which the last edge was taken. Notice that a decision only needs to happen at the times when there is an incident edge.

Note that the traveler is always aware of the last edge they planned to take when the next decision needs to be taken. It would not yield any benefits for the traveler to condition on an event the traveler cannot observe (as they could not gather the necessary information to decide which case to use). If more global information was available, one could also condition on the complete past at the cost of an explosion in runtime (the state space grows exponentially with the number of past edges considered).

Proof (of Theorem 2). Let us describe the new dynamic program to compute an optimal robust routing policy in a faulty network with Last-Edge-Markovian edge failures. The approach is very similar to before, except that now we have different probabilities and we cannot apply the optimization that reduced the computation time to $O(1)$ per utility value.

The starting vertex receives a special dummy loop edge (at the starting time 0) so that all equations have the same form. For each vertex i and each incident edge $\tilde{e} = (k, i)$ or (i, k) , we define the expected utility $u_i(\tilde{e})$ as the largest possible expected utility that can be obtained starting from vertex i , given that \tilde{e} is the last edge which the traveler planned to take. Furthermore, we denote by $p_{e|\tilde{e}}$ the probability that e fails conditioned on \tilde{e} .

The base cases are as follows. The expected utility $u_i(\tilde{e})$ for the case where i is a destination vertex i is initialized as $U_i(T(\tilde{e}))$. In any case, the expected utility $u_i(\tilde{e})$ is U^0 for every vertex i where there is no edge e that leaves after \tilde{e} arrives.

For all other cases, the expected utility equation is (by conditional expectation):

$$u_i(\tilde{e}) = \max_{\substack{e=(i,j) \in E \\ T(e) > T(\tilde{e})}} \left((1 - p_{e|\tilde{e}}) \cdot u_j(T(e)) + p_{e|\tilde{e}} \cdot u_i(T(e)) \right).$$

We evaluate the dynamic program in decreasing order of the edge availability times. This works because the expected utility of an edge \tilde{e} only depends on the utilities of edges e that have strictly larger availability times, i.e., $T(e) > T(\tilde{e})$.

Each vertex i has $\deg(i)$ entries that need to be computed. Each such entry depends on $O(\deg(i))$ other values. Hence, the runtime is $O(m \log m + \sum_{i \in V} \deg^2(i))$. Note that $O(\sum_{i \in V} \deg^2(i)) = O(m \max_i \deg(i)) = O(m^2)$. ◀

C Experimental Setup

C.1 Data

We use the publicly available data set “Fahrzeiten 2018 der VBZ im SOLL-IST-Vergleich”¹ from the VBZ (which is available via Open Data Zurich). It includes the *actual* (i.e. measured) departure and arrival times for all buses and trams in the Zurich transit network. The data also includes the scheduled times for those events. All times are reported in seconds (although the measured accuracy may vary and is not specifically documented). Initial testing revealed that a very small number (around 3 – 8 per day) of departure/arrival pairs are erroneous such that the departure time is larger than the arrival time. We ignore these clearly incorrect data points in our study.

C.2 Algorithms

All evaluated approaches follow the basic idea of modeling the problem as a temporal graph and computing a policy (in the sense of Section 2) that maximizes the utility given some edge failure probabilities. For training our *probabilistic policy*, we use the delay data over the last two weeks before the first evaluated week.

We compare our algorithm to the *deterministic policy* that follows the schedule to find a journey with the *earliest arrival time*. This is equivalent to computing a policy using our algorithm by setting the failure probabilities based on the schedule times and using a utility that is zero minus the arrival time.

Moreover, we compare our algorithm to the *oracle policy* which has perfect knowledge of which connections break and which do not. This means that the failure probabilities are set based on the actual arrival and departure times. Note that the oracle policy will never miss a connection and always arrives on time if that is possible at all.

C.3 Evaluation

We evaluated the performance of the candidate policies for 38 destinations (and for each destination for every possible departure station). The stations are spread throughout the city and are of varying size. Some are exclusively tram or bus stations, others run both. The sample includes very centrally located stations and also more remote stations that often are terminal stations.

¹ https://data.stadt-zuerich.ch/dataset/vbz_fahrzeiten_ogd_2018, on 11.04.2019

4:18 Routing in Stochastic Public Transit Networks

For each destination station, we evaluated the policies during three 1-month evaluation periods in the year 2018 (namely in February, May, and November). All three periods have the same schedule. During this initial evaluation we focussed on 30 and 40 minute time budgets. As the experiments did not show any qualitative differences between the three time-periods, our final reporting focuses on the period in May (but reports on a larger variety of time budgets, where we found larger differences).

During the implementation and pre-evaluation, we used only 18 of the evaluated destinations and older data from the years 2015 and 2016. This helped us to avoid over-fitting our implementation to the evaluation periods and chosen destination stations.

In each evaluation period, we considered the average performance of the policies over the *weekdays*. For each configuration (which is given by an evaluation day, arrival deadline, time budget, and destination station), we simulate travel using the candidate policies. In the simulation, the feasibility of every transfer is determined based on the actual travel data for that day. The time for a transfer is assumed to be fixed and known. When a transfer fails, the policy is queried for a connection with time larger than the schedule time of the missed connection.

Finally, we measured the runtime on the Euler compute cluster using nodes equipped with Intel Xeon E5-2680 v3 processors (a 2.5 Ghz, 12-core processor with 30 MB last-level cache). Our computations required at most 4 GB of RAM.

Robust Network Capacity Expansion with Non-Linear Costs

Francis Garuba¹

Department of Management Science, Lancaster University, United Kingdom
f.garuba@lancaster.ac.uk

Marc Goerigk

Network and Data Science Management, University of Siegen, Germany, Germany
marc.goerigk@uni-siegen.de

Peter Jacko

Department of Management Science, Lancaster University, United Kingdom
p.jacko@lancaster.ac.uk

Abstract

The *network capacity expansion problem* is a key network optimization problem practitioners regularly face. There is an uncertainty associated with the future traffic demand, which we address using a scenario-based robust optimization approach. In most literature on network design, the costs are assumed to be linear functions of the added capacity, which is not true in practice. To address this, two non-linear cost functions are investigated: (i) a linear cost with a fixed charge that is triggered if any arc capacity is modified, and (ii) its generalization to piecewise-linear costs. The resulting mixed-integer programming model is developed with the objective of minimizing the costs.

Numerical experiments were carried out for networks taken from the SNDlib database. We show that networks of realistic sizes can be designed using non-linear cost functions on a standard computer in a practical amount of time within negligible suboptimality. Although solution times increase in comparison to a linear-cost or to a non-robust model, we find solutions to be beneficial in practice. We further illustrate that including additional scenarios follows the law of diminishing returns, indicating that little is gained by considering more than a handful of scenarios. Finally, we show that the results of a robust optimization model compare favourably to the traditional deterministic model optimized for the best-case, expected, or worst-case traffic demand, suggesting that it should be used whenever computationally feasible.

2012 ACM Subject Classification Networks; Theory of computation → Network optimization; Theory of computation → Continuous optimization

Keywords and phrases Robust Optimization, Mobile Network, Network Capacity Design & Expansion, Non-linear Cost, Traffic and Transport Routing

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.5

1 Introduction

Network design and capacity planning has always been of strategic importance in most organization. This implies that it needs to be decided far ahead of time based on the estimation of future traffic demand. Projection for future traffic is usually done using traffic measurements and population statistics in combination with other marketing data. This often results in a large discrepancy between planned and actual carried traffic volume and distribution.

¹ Corresponding author



© Francis Garuba, Marc Goerigk, and Peter Jacko;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 5; pp. 5:1–5:13



OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To provide a more detailed motivation and positioning of our paper, we focus on the telecommunications field (other network design applications, such as line planning for public transport, are also well within the scope of this work). Here, this discrepancy could be as large as 10% according to [3]. Hence, the re-forecasting and re-planning becomes a continuous exercise using traffic measurements and traffic optimization tools, which are often based on deterministic concepts assuming the traffic demand is estimated without error.

The demand for capacity in mobile wireless networks has seen an ever-growing trend in the last couple of decades and growth rate is expected to be even higher going into the future. This explosion in demand for data is coming at a lower cost rate. This means that in order to provide an acceptable quality of service, capacity will need to be regularly extended with optimal investment in capital expenditure. This balancing act of traffic volume, quality of service and capital expenditure has made network capacity expansion a key strategic function resulting in high global telecoms investment. Similar capacity expansion challenges are present to network designers and operators in other types of networks as well, such as transport networks. The *network capacity expansion problem* can hence be considered one of the key network optimization problems practitioners are expected to regularly face in present and future.

To have a network that is robust against uncertain estimated traffic demand, this uncertainty needs to be factored in already during the planning and design process, which we address using a scenario-based robust optimization approach. This methodology is geared towards producing results that are insensitive to the uncertain demand, by solving the problem using two separate stages. In the first stage, we determine the capacity expansion, and in the second stage, demand scenarios are realized. The resulting mixed-integer programming model is developed with the objective of minimizing costs.

In most literature on network design, costs are assumed to be linear functions of the added capacity, which is not true in practice. Real-world costs typically follow a volume discount regime which is reflected by a non-linear function and can be attributed to bulk buy. To address this, two non-linear cost functions are investigated in this paper: (i) a linear cost with a fixed charge that is triggered if any arc capacity is modified, and (ii) its generalization that is piecewise-linear in added capacity.

To the best of our knowledge, this is the first paper that includes non-linear cost functions in the robust network capacity planning problem. This extension leads to a more computationally-demanding model than the one with linear cost. The contributions of our paper are as follows: We show that networks of realistic sizes can be designed using non-linear cost functions in a practical amount of time within negligible suboptimality. We present the benefits of considering a robust optimization model (even with two scenarios) instead of the traditional deterministic model, and present the benefits of considering non-linear costs instead of the usual linear costs. It is illustrated that including additional scenarios approximately follows the law of diminishing returns, indicating that little is gained by considering more than a handful of scenarios. Finally, we show that the results of a robust optimization model compare favourably to the traditional deterministic model optimized for the best-case, expected, or worst-case traffic demand, suggesting that it should be used whenever computationally feasible.

The rest of this paper is organized as follows. Section 2 presents a literature review of related research. In Section 3, we then introduce the problem description of robust network capacity expansion and mathematical models. Experimental results using networks from the SNDLib (see [21]) are discussed in Section 4. Finally, Section 5 concludes our work and points out future research directions.

2 Literature Review

2.1 Robust Optimization in Network Design

In robust optimization, we assume that all possible data scenarios are given in form of an uncertainty set. For general surveys, we refer, e.g., to [13, 14]. The classic approach aims at finding a solution that is feasible for all scenarios from the uncertainty set, while optimizing a worst-case performance. This approach is relaxed through two-stage robust optimization, where not all decisions need to be taken in advance, see [6]. Instead, one distinguishes between “here and now” decisions that need to be fixed in advance, and “wait and see” variables that are determined once a scenario has been revealed. Two-stage robust optimization problems are also known as adjustable robust counterparts.

Adjustable robust optimization has been applied to wireless telecommunication services in the area of network design and expansion. This helps to model decisions that are delayed in time, e.g., traffic needs to be routed only once the demand scenario is known. Three closely related problems are the radio network design problem, the radio network loading problem and the virtual private network problem [17].

In telecoms, the long term strategic network planning can be viewed as the first stage “here and now” decision making, while the traffic redistribution that occurs after the realisation of the traffic demand pattern would be the second stage “wait and see” adjustment decision. Unrestricted second stage recourse in robust network design is called dynamic routing, see [7]. Most applications of adjustable robust optimization have focused on approximations that put a restriction on the recourse.

A special type of recourse restriction based on a specific type of uncertainty model (Hose model) has been proposed independently by [11] and [12] for an asynchronous transfer mode and broadband traffic network. They also introduced the concept of static routing, which [5] applied under their generalized polyhedral uncertainty model using a column and constraint generation algorithm. [20] investigated network capacity expansion under demand and cost uncertainty and recently, [23] used a cutting plane algorithm while taking into consideration the outsourcing costs for unmet demand. Some papers use an affine decision rule to restrict the recourse decisions, thus creating a tractable robust counterpart. [22] introduced affine routing in their robust network capacity planning model, while [24] and [3] used polyhedral uncertainty sets. On the other hand, [2] study the problem in detail by exploiting the underlying network structure.

2.2 Related Work on Non-linear Cost Functions

In general, routing costs, transportation costs or capacity costs can be a non-linear functions of traffic flows. In the following, we review literature on fixed-charge costs and piecewise-linear costs.

2.2.1 Fixed-Charge Cost Models

In a network with fixed-charge costs, an initial outlay cost is incurred to make an arc available. In this setting, one needs to pay a fixed initial cost in addition to the arc expansion cost. The fixed costs could be the installation costs, cabinet outlay costs, additional energy or utility costs and line replacement costs. Applications are found in wide areas of network design problems and not limited to energy networks, transportation and communication. A survey is provided by [16] that demonstrate many applications in logistics, transportation and communications. The fixed-charge cost network design problem (FCND) has been found to be NP-hard, see [16, 19].

Literature on the FCND has concentrated on solution algorithms for the different model variants. [8] addressed the multi-commodity capacitated FCND using a cutting plane algorithm with an improvement on the mixed-integer programming (MIP) formulation. [9] presented a detailed survey on the use of Benders decomposition to solving a wide range of FCND's which includes two facility networks. This can be viewed as a two-commodity network with a variant that introduces a quality of service measure. In [1], a heuristic approach for separating and adding violated partition inequalities was implemented. [26] solved a FCND using a variant of Benders decomposition which they referred to as the Bender-and-cut technique. The closest work to our model is [18]. Here, they formulate a robust network design problem with both transportation cost and demand uncertainty. Investment in arc capacity is modeled as a binary decision (i.e., expansion or no expansion). The model is approximated using an affine decision rule.

2.2.2 Piecewise-Linear Cost Models

The piecewise-linear cost model (PLC) can be used to model costs with economies of scale. In general, optimization problems involving PLC arise in domains including transportation, communications networks, large scale integrated circuits, supply chain management and logistics planning. They are usually modeled as MIPs, see [25]. The problem has been proven to be NP-hard for general concave cost objective functions, see [15].

As is the case for fixed-charge costs, most literature in this domain tends to focus on solution algorithms, see [10]. A continuous relaxation technique for solving network design with piecewise-linear costs was presented by [19]. [15] noted that exact techniques based on dynamic programming and branch and bound are only efficient for specific subclasses of the problem. A number of MIP model formulations exist for piecewise-linear functions. The names for these were unified in [27], which also provides a performance comparison. In terms of execution speed, they recommended the use of Multiple Choice Model (MCM) by [4] or the Incremental approach for a small number of segments.

3 Problem Formulation

We consider a multi-commodity network design problem where capacities are to be added on top of existing ones on a subset of arcs, with the aim of minimizing the total cost involved and so that routing of traffic for the different commodities over the arcs subject to design and network constraints is possible. We call this problem the *Robust Network Capacity Expansion Problem* (RNCEP). We first introduce the basic problem version with linear costs, before introducing two non-linear cost extensions.

3.1 RNCEP with Linear Costs

A communications network topology can be represented by a directed connected graph $G = (\mathcal{V}, \mathcal{A})$. Each of the arcs $a \in \mathcal{A}$ has an original capacity u_a . The original capacity on each arc a can be expanded at a cost c_a per each additional unit of capacity. A set of commodities \mathcal{K} represents potential traffic demands. A commodity $k \in \mathcal{K}$ corresponds to node pair $(s^k, t^k) \in \mathcal{V} \times \mathcal{V}$ and a demand $d^k \geq 0$ for traffic from s^k to t^k . The actual demand values are considered to be uncertain and depend on random scenarios $\xi \in \Xi$. We assume a finite set $\Xi = \{\xi^1, \dots, \xi^N\}$ of possible demand scenarios and write $d^k(\xi)$ for the demand of pair (s^k, t^k) in scenario ξ .

The robust network capacity expansion problem is to find a minimum-cost installation of additional capacities while satisfying all traffic demands $d^k(\xi)$ for all $k \in \mathcal{K}$ and all $\xi \in \Xi$. In this respect, RNCEP is a two-stage robust program. The additional capacity we install on arc $a \in \mathcal{A}$ is denoted by x_a and is a first stage decision variable, which has to be fixed before observing a demand realization $\xi \in \Xi$. Once the demand scenario ξ becomes known, traffic is routed through a multi-commodity flow with variables $f_a^k(\xi)$.

Let $\delta^+(v)$ and $\delta^-(v)$ denote the sets of outgoing and incoming arcs at node $v \in \mathcal{V}$, respectively. The problem can now be formulated as the following linear program.

$$\min \sum_{a \in \mathcal{A}} c_a x_a \quad (1)$$

$$\text{s.t.} \quad \sum_{a \in \delta^-(v)} f_a^k(\xi) - \sum_{a \in \delta^+(v)} f_a^k(\xi) = \begin{cases} -d^k(\xi) & \text{if } v = s^k \\ d^k(\xi) & \text{if } v = t^k \\ 0 & \text{otherwise} \end{cases} \quad \forall v \in \mathcal{V}, k \in \mathcal{K}, \xi \in \Xi \quad (2)$$

$$\sum_{k \in \mathcal{K}} f_a^k(\xi) \leq u_a + x_a \quad \forall \xi \in \Xi, a \in \mathcal{A} \quad (3)$$

$$f_a^k(\xi) \geq 0 \quad \forall k \in \mathcal{K}, \xi \in \Xi, a \in \mathcal{A} \quad (4)$$

$$x_a \geq 0 \quad \forall a \in \mathcal{A} \quad (5)$$

Objective function (1) is to minimize the total cost of capacity expansion subject to flow conservation constraint (2), while constraint (3) imposes that the amount of flow does not exceed the sum of existing and added arc capacity.

3.2 RNCEP with Fixed-Charge Costs

We now introduce an extension of the previous model, where a fixed charge occurs if the capacity of an arc is modified. To this end, let p_a be this fixed charge associated with arc $a \in \mathcal{A}$.

We introduce a new variable $h_a \in \{0, 1\}$ to denote if the capacity of arc a is modified. The *RNCEP with fixed-charge costs* can then be formulated as the following mixed-integer program:

$$\min \sum_{a \in \mathcal{A}} (c_a x_a + h_a p_a) \quad (6)$$

$$\text{s.t.} \quad x_a \leq M_a h_a \quad \forall a \in \mathcal{A} \quad (7)$$

$$h_a \in \{0, 1\} \quad \forall a \in \mathcal{A} \quad (8)$$

$$\text{Constraints (2) – (5)} \quad (9)$$

Here, M_a for all a are constants that are sufficiently large not to restrict the solution. For instance, taking any $M_a \geq \max_{\xi \in \Xi} \sum_{k \in \mathcal{K}} d^k(\xi)$ for all a is valid.

3.3 RNCEP with Piecewise-Linear Cost

We further extend the RNCEP by introducing a piecewise-linear cost function. To this end, we apply the multiple choice model (MCM) as mentioned in the literature review. We assume that for every arc, there are up to S segments with different slopes in the cost function. Let us write $\mathcal{S} = \{1, \dots, S\}$. For every arc a and segment s , let b_a^s denote the load breakpoint, with an additionally defined $b_a^0 := 0$. Let c_a^s denote the cost slope of segment s , and p_a^s its y -intercept.

In addition to the variables of RNCEP, we introduce two new sets of auxiliary variables. Variables h_a^s are binary variables that select the cost segment where the added capacity x_a falls in. Variables x_a^s denote the amount of capacity that is added within each cost segment. This gives the following mixed-integer programming formulation for the *RNCEP* with *piecewise-linear costs*:

$$\min \sum_{s \in \mathcal{S}} \sum_{a \in \mathcal{A}} (c_a^s x_a^s + h_a^s p_a^s) \quad (10)$$

$$\text{s.t. } x_a = \sum_{s \in \mathcal{S}} x_a^s \quad \forall a \in \mathcal{A} \quad (11)$$

$$b_a^{s-1} h_a^s \leq x_a^s \leq b_a^s h_a^s \quad \forall a \in \mathcal{A}, s \in \mathcal{S} \quad (12)$$

$$\sum_{s \in \mathcal{S}} h_a^s \leq 1 \quad \forall a \in \mathcal{A} \quad (13)$$

$$x_a \leq M_a \sum_{s \in \mathcal{S}} h_a^s \quad \forall a \in \mathcal{A} \quad (14)$$

$$x_a^s \geq 0 \quad \forall a \in \mathcal{A}, s \in \mathcal{S} \quad (15)$$

$$h_a^s \in \{0, 1\} \quad \forall a \in \mathcal{A}, s \in \mathcal{S} \quad (16)$$

$$\text{Constraints (2) – (5)} \quad (17)$$

4 Experimental Study

We implemented the fixed-charge cost model and the piecewise-linear cost model using instances from the SNDLib library by [21]. Network parameters characteristics on the four considered networks from SNDLib are presented in Table 1.

■ **Table 1** Network parameters characteristics (rounded to integers).

Network	Janos26	Janos39	Sun27	Node39
$ \mathcal{V} $	26	39	27	39
$ \mathcal{A} $	84	122	102	172
$ \mathcal{K} $	650	1,482	67	1,471
d^k (mean±SD)	123±198	69±243	28±16	5±2
u_a (mean±SD)	64±0	1,008±0	40±0	160±0
c_a (mean±SD)	468±225	313±162	19±10	23±11

Models were implemented using Julia and Gurobi version 7.5 on a Lenovo desktop machine with 8 GB RAM and Intel Core i5-6500 CPU at 2.50Ghz on 4 Cores. We used a time limit of 4000s for each problem instance and optimality is achieved once the optimality gap is below 0.01%.

4.1 Experimental Setup

Both the fixed-charge cost and the piecewise-linear cost models were implemented with one scenario (single-scenario) and with two scenarios (double-scenario). The base demand scenario was provided from the SNDLib library, which we randomly modified to generate additional demand scenarios. The amount of modification is controlled by a parameter λ , the maximum deviation of modified demand from the base demand. The parameter λ is chosen to be a fraction of the mean base demand \hat{d} ; we consider $\lambda = \text{round}(0.3\hat{d})$ and

■ **Table 2** Experimental setup for generating 120 problem instances for each network.

Parameters	# options	Options
Number of scenarios	2	1 (single) / 2 (double)
Scenario variability λ	2	$0.3\hat{d}$ / $0.6\hat{d}$
Fixed-charge factor P	3	0 / 10 / 100
Number of runs	10	—

■ **Table 3** Proportion of instances not solved to optimality within the time limit (rounded to one decimal).

Network	Janos26	Janos39	Sun27	Node39
Total	0.0%	24.2%	35.0%	66.7%
$P = 0$	0.0%	0.0%	0.0%	0.0%
$P = 10$	0.0%	0.0%	12.5%	100.0%
$P = 100$	0.0%	72.5%	92.5%	100.0%
Single-scenario	0.0%	15.0%	28.3%	66.7%
Double-scenario	0.0%	33.3%	41.7%	66.7%

$\lambda = 2 \cdot \text{round}(0.3\hat{d})$, corresponding to small uncertainty and large uncertainty, respectively. The value is then used as a bound for uniformly generating the modified demands around the base demand of every arc.

We summarize the experimental setup in Table 2. For each of the four networks, we consider the single-scenario and the double-scenario case, as well as small and large uncertainty. Additionally, for fixed-cost models we use three different fixed-charge factors P . These are used to calculate the fixed charges p_a of arc a by setting $p_a = Pc_a$. With $P = 0$, we recover the basic linear cost model without fixed charge. All networks and parameter settings are run 10 times to reduce variability in the results. In total, this gives $4 \cdot 2 \cdot 2 \cdot 3 \cdot 10 = 480$ optimization problem instances that need to be solved for the fixed charge case. For the piecewise-linear case, we follow the same setup with $4 \cdot 2 \cdot 2 \cdot 10 = 160$ instances. Each arc has three cost segments where the cost of each segment is calculated as ratio of the nominal arc cost. This gives segment costs as $c_a^s = c_a \cdot r_s$ where $r \in \{1.00, 0.90, 0.75\}$.

4.2 Results for RNCEP with Fixed-Charge Cost

4.2.1 Single- and Double-Scenario Results

Table 3 summarizes the results of the 480 problem instances, reporting the proportion of instances that were not solved to optimality within the time limit. We can see the optimization performance of problem instances in total, for different values of P , and for different number of scenarios. This performance measure gives a high-level summary of the hardness of particular instances. We can conclude that the instances become harder to solve as P increases, or as the number of scenarios increases.

Other performance metrics are presented in more detail in Table 4 and Table 5, where each cell gives an average and standard deviation from a sample of 20 problem instances. *Optimality gap* refers to the sub-optimality estimated and reported by Gurobi using the built-in procedure for lower-bounding the objective. *Solution time* is the time reported by Gurobi, capped by the time limit. *Capacity added* is the overall network capacity added on top of the original capacity (which can be calculated as $|\mathcal{A}|u_a$ from Table 1).

■ **Table 4** Single-scenario results (rounded to one decimal).

		Janos26	Janos39	Sun27	Node39
Optimality gap	$P = 0$	0.0%	0.0%	0.0%	0.0%
	$P = 10$	0.0%	0.0%	0.0%	$7.7 \pm 2.9\%$
	$P = 100$	0.0%	$0.3 \pm 0.6\%$	$5.0 \pm 2.8\%$	$51.9 \pm 4.8\%$
Solution time	$P = 0$	6.5 ± 0.5	156.9 ± 17.0	0.3 ± 0.1	536.4 ± 82.2
	$P = 10$	7.4 ± 0.6	227.1 ± 86.0	201.7 ± 201.4	$4,000.1 \pm 0.0$
	$P = 100$	10.8 ± 2.1	$3,120.9 \pm 1,088.0$	$3,694.8 \pm 815.9$	$4,000.1 \pm 0.1$
Capacity added	$P = 0$	$268,698 \pm 23,970$	$331,864 \pm 57,041$	$3,043 \pm 271$	$1,194 \pm 357$
	$P = 10$	$270,931 \pm 23,195$	$329,330 \pm 54,751$	$2,925 \pm 412$	$1,204 \pm 281$
	$P = 100$	$275,409 \pm 23,476$	$321,808 \pm 53,261$	$3,652 \pm 447$	$1,167 \pm 357$

■ **Table 5** Double-scenario results (rounded to one decimal).

		Janos26	Janos39	Sun27	Node39
Optimality gap	$P = 0$	0.0%	0.0%	0.0%	0.0%
	$P = 10$	0.0%	0.0%	$0.1 \pm 0.2\%$	$11.0 \pm 1.8\%$
	$P = 100$	0.0%	$1.3 \pm 0.5\%$	$10.8 \pm 1.4\%$	$57.1 \pm 3.3\%$
Solution time	$P = 0$	88.4 ± 25.1	$1,285.6 \pm 349.5$	1.2 ± 0.2	$2,256.6 \pm 317.9$
	$P = 10$	92.2 ± 21.0	$2,373.9 \pm 770.5$	$1,729.0 \pm 1,418.2$	$4,000.2 \pm 0.1$
	$P = 100$	189.0 ± 57.7	$4,000.3 \pm 0.2$	$4,000.1 \pm 0.1$	$4,000.2 \pm 0.1$
Capacity added	$P = 0$	$278,358 \pm 8,988$	$363,225 \pm 26,348$	$4,399 \pm 304$	$1,185 \pm 154$
	$P = 10$	$278,031 \pm 7,857$	$367,324 \pm 18,522$	$4,635 \pm 329$	$1,286 \pm 254$
	$P = 100$	$282,467 \pm 9,830$	$368,547 \pm 19,887$	$5,668 \pm 503$	$1,236 \pm 254$

Interestingly, network Sun27 shows large variability in solution time, for both single-scenario and double-scenario settings. While with $P = 0$ it is the quickest to solve out of all networks, for larger values of P it is roughly similar to Janos39, despite dealing with a smaller number of commodities. On the other hand, solution time of Janos26 is affected very little by different values of P .

Comparing the solution time reported in Table 4 and Table 5, the double-scenario model, as expected, takes longer to solve to optimality as the goal here is to factor in robustness into the solution. On average, this double-scenario model resulted in 7.39% additional capacity across the networks for instances that were solved to optimality. The average increase in solution time across the instances that were solved to optimality is 828.24%.

We also note that capacity added is highly network dependent. The capacities of Janos26 and Janos39 are expanded dramatically due to the high variability in the demand, which for some commodities significantly exceeds the original capacity (see Table 1). On the other hand, the demands in Sun27 and Node39 are small compared to the original capacity, so the capacity added is relatively small.

Not reported elsewhere is the effect of scenario variability λ : the solution time becomes smaller if the uncertainty is larger, i.e., on the average for all the networks and parameter settings, the $0.6\hat{d}$ variability results in lower solution times than for the $0.3\hat{d}$ variability. This was also found to be the trend when looking at single networks. This is summarized in Table 6.

Overall, it is possible to solve most of the problem instances to optimality within the time limit, and even most of those not solved to optimality report very small optimality gap. The only settings that would significantly benefit from an increased time limit are Sun27 at $P = 100$ and Node39 at $P = 10$ and $P = 100$.

4.2.2 Effect of Number of Scenarios

While the previous discussion focused only on single- and double-scenario instances, it is also of interest to understand how an increased number of scenarios affects the performance measures. Considering more scenarios is expected to lead to a solution which in practical terms guarantees the network ability to accommodate a higher level of demand variation and provides additional capacity.

To illustrate that, we tested network Janos26 with fixed charge $P = 10$. We started with a single-scenario instance, where the base scenario considered reflects the *expected* demand (this is the original demand from SNDLib). We then generated and gradually added additional scenarios by randomly perturbing all the demands of the base scenario within $\pm\lambda$, in the large uncertainty setting.

For comparison, we also considered the *optimistic* instance, which is a single-scenario instance in which the demand is generated by subtracting λ from the expected demand on every arc. This instance expands the capacity of the network to satisfy only the smallest demand scenario, and would be almost surely unable to satisfy the realized demand. Finally, we considered the *pessimistic* instance, which is a single-scenario instance in which the demand is generated by adding λ to the expected demand on every arc. This instance expands the capacity of the network to satisfy all the possible demand scenarios.

The results are presented in Table 7. These results are representative; similar results were obtained when we replicated the experiment with other randomly generated scenarios. The key observations are as follows: By gradually expanding the set of scenarios, the cost (our minimization objective) non-decreases; the added capacity follows a similar trend, but is not necessarily monotone (cf. 8 vs 9 scenarios); the solution time (reported in seconds and as a multiple of the expected scenario instance) increases exponentially; expansion by adding more scenarios approximately follows the law of diminishing returns in both the cost and added capacity: the increase is highest when expanding from 1 (expected) scenario to 2 scenarios (which includes the expected scenario and one randomly generated), with only a minor increase when considering more than 3 scenarios, indicating the value of considering a robust optimization approach even with few scenarios; the increase in both the cost and added capacity is dramatic (36.9%) when expanding from 1 (expected) scenario to 2 scenarios (which includes the expected scenario and one randomly generated), indicating that optimizing the network based on the expected scenario (i.e. on point forecasts) only may be an inappropriate approach, leading to a large amount of unsatisfied realized demand; optimizing the network for the pessimistic scenario is very expensive (the increase in both the cost and added capacity is about 115% compared to the expected scenario), indicating the value of

■ **Table 6** Effect of higher λ on solution time.

Solution Time	Single Scenario	Double Scenario
$\lambda = 0.3\hat{d}$	527.31	3,010.85
$\lambda = 0.6\hat{d}$	346.62	2,299.23
% Improvement	34.3%	23.6%

■ **Table 7** Results on Janos26 with fixed-charge cost ($P = 10$) for different numbers of scenarios.

# Scenarios	Cost (in 10^3)	Δ Cost	Added Capacity	Δ Added Capacity	Time (sec.)	\propto Time
1 (optimistic)	83,001	-10.9%	192,610	-9.2%	8	1x
1 (expected)	93,116	—	212,104	—	8	—
2	127,484	36.9%	292,893	38.1%	59	8x
3	129,804	39.4%	298,131	40.6%	376	50x
4	130,265	39.9%	300,426	41.6%	768	102x
5	130,272	39.9%	300,492	41.7%	1,080	143x
6	130,462	40.1%	300,913	41.9%	3,124	413x
7	130,753	40.4%	301,598	42.2%	2,488	329x
8	131,206	40.9%	301,936	42.4%	4,456	589x
9	131,255	41.0%	301,715	42.2%	8,869	1173x
1 (pessimistic)	200,593	115.4%	456,182	115.1%	8	1x

■ **Table 8** Solution results for piecewise-linear cost.

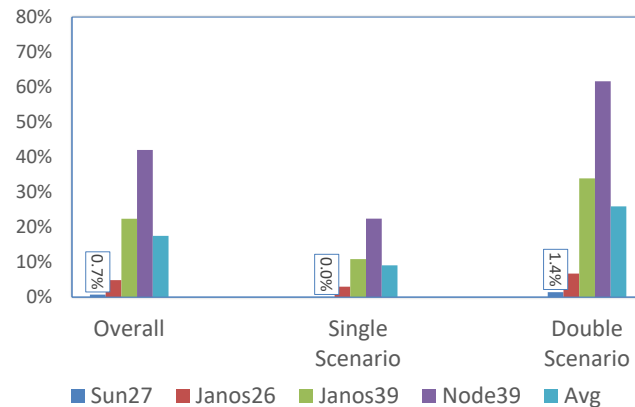
Single-Scenario	Sun27	Janos26	Janos39	Node39
Optimality Gap	0.00%	2.90%	10.43%	22.43%
Solution time	653.67 \pm 640.84	4000.22 \pm 0.11	4000.22 \pm 0.06	4000.16 \pm 0.04
Capacity Added	2,863 \pm 539	276,172 \pm 26,036	335,258 \pm 58,895	1,472 \pm 574
Double-Scenario				
Optimality Gap	1.43%	6.73%	37.44%	77.99%
Solution time	4000.04 \pm 0.01	4000.21 \pm 0.23	4000.10 \pm 0.03	4000.12 \pm 0.04
Capacity Added	4,380 \pm 278	296,354 \pm 11,398	472,889 \pm 110,491	4,117 \pm 2,601

considering a robust optimization approach even with few scenarios; optimizing the network for the optimistic scenario leads to savings (the decrease in both the cost and added capacity is about 10% compared to the expected scenario), but may not be acceptable in practice if the consequences of having practically no satisfied realized demand are non-negligible.

These results provide an indication of the ability of our model to become more robust by including more demand scenarios. We note that Gurobi was able to deal with up to approximately 200 scenarios for this network without giving an out-of-memory error, however, it would be unlikely to compute a close-to-optimal solution in a reasonable amount of time.

4.3 Results for RNCEP with Piecewise-Linear Costs

Next we consider the robust network capacity expansion problem with piecewise-linear costs. Overall, 12.5% of all problem instances were solved to optimality within the time limit, 77.5% returned a non-optimal solution, and 10% were timed out already during the root relaxation. None of the double-scenario problem instances reached optimality within the time limit. Only one of the networks, Sun27, reached optimality and this was for all the problem instances in the single-scenario case. Two networks, Janos39 and Node39, had instances timing out under the root relaxation phase.



■ **Figure 1** Optimality gap for piecewise-linear cost.

Table 8 presents more detailed results of this model for each network. The optimality gap is further illustrated in Figure 1, indicating that the optimality gap may be acceptable because of small values and small variability for Sun27 and Janos26 in the single-scenario setting and for Sun27 in the double-scenario setting. Better solutions can of course be achieved by increasing the time limit, which would be recommendable in the remaining settings.

The optimality gap provides insight into the increased difficulty of solving these problem instances, which also translates into longer solution time. It takes at least 512% more time to solve the double-scenario models compared to the single-scenario using Sun27 network, which is the easiest setting considering its very low optimality gap of 1.43% for the double-scenario instances. A further analysis was performed on the solution time using the paired sample t -Test which indicates no significant difference between solution time returned by $0.3\hat{d}$ and $0.6\hat{d}$ with a t -statistic of -0.2047 and a p -value 0.8423 .

5 Conclusions

In this paper, a robust approach to network capacity expansion with non-linear cost functions was investigated. We developed robust models with fixed-charge costs and with piecewise-linear costs. They were implemented on four networks taken from the SNDlib, [21], with results compared to using linear costs. In the experimental setup, a number of possible parameter configurations was considered, including different demand variability and fixed-charges.

When further increasing the number of scenarios, we found that results follow a law of diminishing returns. While objective values and added capacity change little beyond five scenarios, computation times increase considerably. This is an indicator that already few scenarios suffice to find solutions that are robust against uncertainty in demand. The next pursuit will be to further improve the solution time for these models testing a path-based flow formulation and by developing specialized algorithms based on column generation and Benders decomposition.

References

- 1 Y. K. Agarwal and Y. P. Aneja. Fixed charge multicommodity network design using p-partition facets. *European Journal of Operational Research*, 258(1):124–135, 2017.
- 2 A. Atamturk and M. Zhang. Two-Stage Robust Network Flow and Design Under Demand Uncertainty. *Operations Research*, 55(4):662–673, August 2007.
- 3 F. Babonneau, J.-P. Vial, O. Klopfenstein, and A. Ouorou. Robust capacity assignment solutions for telecommunications networks with uncertain demands. *Networks*, 62(4):255–272, 2013.
- 4 A. Balakrishnan and S. C. Graves. A composite algorithm for a concave-cost network flow problem. *Networks*, 19(2):175–202, 1989.
- 5 W. Ben-Ameur and H. Kerivin. Routing of Uncertain Traffic Demands. *Optimization and Engineering*, 6(3):283–313, September 2005.
- 6 A. Ben-Tal, A. Goryashko, E. Guslitzer, and A. Nemirovski. Adjustable robust solutions of uncertain linear programs. *Mathematical Programming*, 99(2):351–376, March 2004.
- 7 C. Chekuri, F.B. Shepherd, G. Oriolo, and M.G. Scutellá. Hardness of robust network design. *Networks*, 50(1):50–54, 2007.
- 8 M. Chouman, T. G. Crainic, and B. Gendron. Commodity Representations and Cut-Set-Based Inequalities for Multicommodity Capacitated Fixed-Charge Network Design. *Transportation Science*, 51(2):650–667, 2017.
- 9 A. M. Costa. A Survey on Benders Decomposition Applied to Fixed-Charge Network and Design Problems. *Computers and Operations Research*, 32(6):1429–1450, June 2005.
- 10 K. L. Croxton, B. Gendron, and T. L. Magnanti. Variable Disaggregation in Network Flow Problems with Piecewise Linear Costs. *Operations Research*, 55(1):146–157, 2007.
- 11 N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. Van der Merive. A flexible model for resource management in virtual private networks. *ACM SIGCOMM Computer Communication Review*, 29(4):95–108, October 1999.
- 12 J. A. Fingerhut, S. Suri, and J. S. Turner. Designing Least-Cost Nonblocking Broadband Networks. *Journal of Algorithms*, 24(2):287–309, August 1997.
- 13 V. Gabrel, C. Murat, and A. Thiele. Recent advances in robust optimization: An overview. *European Journal of Operational Research*, 235(3):471–483, June 2014.
- 14 M. Goerigk and A. Schöbel. Algorithm engineering in robust optimization. In *Algorithm engineering*, pages 245–279. Springer International Publishing, 2016.
- 15 G. M. Guisewite and P. M. Pardalos. Minimum concave-cost network flow problems: Applications, complexity, and algorithms. *Annals of Operations Research*, 25(1):75–99, December 1990.
- 16 T. L. Magnanti and R. T. Wong. Network Design and Transportation Planning: Models and Algorithms. *Transportation Science*, 18(1):1–55, February 1984.
- 17 S. Mattia. The robust network loading problem with dynamic routing. *Computational Optimization and Applications*, 54(3):619–643, August 2012.
- 18 S. Mudchanatongsuk, F. Ordóñez, and J. Liu. Robust solutions for network design under transportation cost and demand uncertainty. *Journal of the Operational Research Society*, 59(5):652–662, May 2008.
- 19 A. Nahapetyan and P. Pardalos. Adaptive dynamic cost updating procedure for solving fixed charge network flow problems. *Computational Optimization and Applications*, 39(1):37–50, January 2008.
- 20 F. Ordóñez and J. Zhao. Robust capacity expansion of network flows. *Networks*, 50(2):136–145, 2007.
- 21 S. Orłowski, R. Wessäly, M. Pióro, and A. Tomaszewski. SNDlib 1.0-Survivable Network Design Library. *Networks.*, 55(3):276–286, May 2010.
- 22 A. Ouorou and J.-P. Vial. A model for robust capacity planning for telecommunications networks under demand uncertainty. In *Design and Reliable Communication Networks, 2007. DRCN 2007. 6th International Workshop on*, pages 1–4. IEEE, 2007.

- 23 A. A. Pessoa and M. Poss. Robust Network Design with Uncertain Outsourcing Cost. *INFORMS Journal on Computing*, 27(3):507–524, August 2015.
- 24 M. Poss and C. Raack. Affine recourse for the robust network design problem: Between static and dynamic routing. *Networks*, 61(2):180–198, September 2012.
- 25 S. Sridhar, J. Linderoth, and J. Luedtke. Locally ideal formulations for piecewise linear functions with indicator variables. *Operations Research Letters*, 41(6):627–632, 2013.
- 26 V. Sridhar and J. S. Park. Benders-and-cut algorithm for fixed-charge capacitated network design problem. *European Journal of Operational Research*, 125(3):622–632, September 2000.
- 27 J. P. Vielma, S. Ahmed, and G. Nemhauser. Mixed-Integer Models for Nonseparable Piecewise-Linear Optimization: Unifying Framework and Extensions. *Operations Research*, 58(2):303–315, 2010.

The Trickle-In Effect: Modeling Passenger Behavior in Delay Management

Anita Schöbel

Technical University of Kaiserslautern, Germany

Fraunhofer Institute for Industrial Mathematics ITWM, Kaiserslautern, Germany

schoebel@mathematik.uni-kl.de

Julius Pätzold

University of Goettingen, Germany

j.paetzold@math.uni-goettingen.de

Jörg P. Müller

Department of Informatics, Clausthal University of Technology, Germany

joerg.mueller@tu-clausthal.de

Abstract

Delay management is concerned with making decisions if a train should wait for passengers from delayed trains or if it should depart on time. Models for delay management exist and can be adapted to capacities of stations, capacities of tracks, or respect vehicle and driver schedules, passengers' routes and further constraints. Nevertheless, what has been neglected so far, is that a train cannot depart as planned if passengers from another train trickle in one after another such that the doors of the departing train cannot close. This effect is often observed in real-world, but has not yet been taken into account in delay management.

We show the impact of this “trickle-in” effect to departure delays of trains under different conditions. We then modify existing delay management models to take the trickle-in effect into account. This can be done by forbidding certain intervals for departure. We present an integer programming formulation with these additional constraints resulting in a generalization of classic delay management models. We analyze the resulting model and identify parameters with which it can be best approximated by the classical delay management problem.

Experimentally, we show that the trickle-in effect has a high impact on the overall delay of public transport systems. We discuss the impact of the trickle-in effect on the objective function value and on the computation time of the delay management problem. We also analyze the trickle-in effect for timetables which have been derived without taking this particular behavioral pattern of passengers into account.

2012 ACM Subject Classification Applied computing → Transportation

Keywords and phrases Public Transport Planning, Delay Management, Integer Programming

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.6

Funding *Julius Pätzold*: supported by Simulation Science Center (SWZ) Clausthal-Göttingen.

1 Introduction

Delays constitute a major source of uncertainty when operating a railway or bus system. If a train is delayed, many rescheduling decisions have to be made, disturbing the nominal schedule of a public transport system. The question, whether an otherwise punctual train should wait for a delayed feeder train in order to allow transferring passengers to reach their connections, is known as *delay management problem* and has been studied extensively in the literature. The first papers dealing with this kind of question date back to [21, 23]. Integer programming models have been developed in [22, 5]. In order to make them more realistic, capacities along tracks have been included in [18], capacities at stations have been included



© Anita Schöbel, Julius Pätzold, and Jörg P. Müller;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 6; pp. 6:1–6:15



OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in [7] and passenger re-routing has been studied in [9, 20, 17]. Rescheduling of timetables, rolling stock and crew is studied in [8]. For all these cases algorithms have been developed, see [10] for a recent survey on the state of the art.

Delay management aims at minimizing passengers' delays by taking dependencies between delays into account (see [4]). Delays propagate along driving activities, i.e., if a train departs with some delay, then it also arrives at its next station with some delay – reduced by buffer time possibly included in the timetable. Delays also propagate along waiting activities in stations: If a train arrives at a station with some delay, it will probably also depart with some delay which again might have been reduced by buffer time. Finally, delays can propagate along changing activities as well. This is the case if a dispatcher decides that a connection from one train to another train should be maintained. Then the outgoing train will receive some delay by waiting for the delayed feeder train.

Nevertheless, there is an effect that has been neglected in the literature so far: A dispatcher may decide that a train should depart on time, but it may not be possible for the train to do so. To illustrate this issue, suppose a delayed train A arrives at a station and some of its passengers want to transfer at this station to another train B . The delay management problem requires a decision, whether train B should depart on time or wait for the passengers from train A .

- If train B is supposed to depart before train A has arrived, the delay management models work correctly. In this case, no delay propagates from train A to train B .
- If train B is supposed to wait long enough, the delay management models also work correctly and the delay propagates along the changing activity to the departure of train B .
- If, however, train B is supposed to depart shortly after train A has arrived without waiting for the passengers from train A , then the models fail. This is the case because normal delay management models assume that there is one common time that passengers need for walking from train A 's platform to train B 's platform. Instead, there may be quick and slow passengers. If the fastest passenger reaches train B before its departure, she can board. While getting onto train B , another fast passenger might arrive and while he boards, the next one will arrive, and so forth. In this way, all passengers might enter the train in a continuous stream preventing the train doors to close. Train B hence has to wait until finally even the slowest passengers from train A arrive and board train B . This effect has been simulated in [1] where it is called *trickle-in effect*.
- The same effect may also prolong the waiting time of train B in the case that B is supposed to wait for the passengers of train A since it may take longer to allow all passengers to trickle in than the lower bounds on the changing times suggest.

Note that the trickle-in effect is not only triggered by passengers not moving with the same speed, but also by the fact that passengers are not able to unboard train A instantaneously. Most readers will have experienced the situation of standing in a train corridor while waiting a decent amount of time for the passengers in front of them to unboard. This can result in a different transfer time of two passengers, even though they are able to walk with similar speed.

As a consequence, there exists a time interval in which train B is not able to depart, namely between the arrival of the fastest passenger and the arrival of the slowest passenger (assuming that there is no gap in speed of the passengers big enough to allow the doors of train B to close). We will call this interval *trickling interval*.

[1] show that the trickle-in effect, which can also be observed in many real-world situations, is in fact relevant. Our experiments (see Section 5) show that delay management decisions, which are optimal in the sense of classical delay management models, often schedule trains to depart in the “forbidden” trickling interval. If, for example, the trickling interval is $(2, 5)$

minutes and if all changing activities are distributed uniformly in $[2, 62)$ minutes (assuming a time period of 60 minutes), we can expect about 5% of all train departures to lie in the trickling interval. These departures are most likely not realizable and will cause additional delays. Hence, it is necessary to add this additional constraint to delay management models which is exactly what we do in this paper. We show how such an additional constraint can be included in classical delay management models, subsequently analyze the mathematical relation between the classical model and the one with the additional constraints, and finally show in experiments that delay management strategies change if the trickle-in effect is considered. We believe that by adding this detail we take a further step in bringing delay management models closer to practice.

The remainder of the paper is structured as follows. In Section 2 we recap the classical model for delay management. Section 3 models the trickle-in effect by introducing an additional constraint to the classical delay management model. We investigate theoretical consequences when adding the trickle-in effect to the classical delay management model in Section 4. Section 5 studies its practical effects in an experimental study on close-to-real-world data from LinTim [11, 19]. Integrating the trickle-in effect in models for (periodic) timetabling is identified as an extension and briefly discussed in Section 6, where we also conclude the contributions, discuss limitations of our work as well as venues of future research.

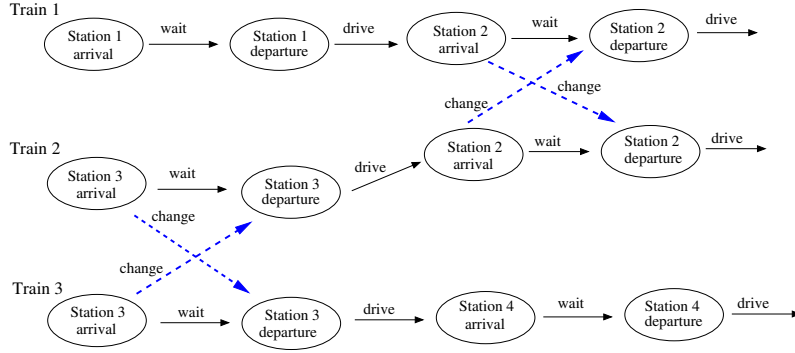
2 The Classical Delay Management Model

The *delay management problem* is defined as follows: Given an event-activity network, a timetable and some source delays, decide which connections should be maintained and which should be dropped such that the average delay of all passengers at their final destinations is minimal. The delay management problem was first introduced in [21], a recent overview is given in [10].

We hence have to first introduce the concept of *event-activity networks* (see [14] for its application in timetabling and [22] for its application in delay management). An event-activity network is a directed graph $\mathcal{N} = (\mathcal{E}, \mathcal{A})$, where \mathcal{E} consists of arrival and departure events \mathcal{E}_{arr} and \mathcal{E}_{dep} , respectively. A timetable $\pi \in \mathbb{N}^{|\mathcal{E}|}$ assigns each event $i \in \mathcal{E}$ to a time $\pi_i \in \mathbb{N}$. If a delay occurs, the given timetable π has to be updated to a so-called *disposition timetable* $x \in \mathbb{N}^{|\mathcal{E}|}$. To represent the constraints that have to be satisfied by a (disposition) timetable, we need the following types of activities, $\mathcal{A} = \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}} \cup \mathcal{A}_{\text{change}}$. Each of them is assigned to a minimal duration $L_a > 0$. The meaning of these activities is given as follows (see also Figure 1).

- *Driving activities* $\mathcal{A}_{\text{drive}} \subset \mathcal{E}_{\text{dep}} \times \mathcal{E}_{\text{arr}}$ model the driving of a train between two consecutive stations, i.e. a driving activity connects a departure event of some train with its next arrival event. The duration $L_a > 0$ of a driving activity $a = (i, j)$ represents the minimal necessary driving time between the departure event i and the arrival event j . Note that turnaround edges may be handled in the same way as driving activities.
- *Waiting activities* (also called *dwelling activities*) $\mathcal{A}_{\text{wait}} \subset \mathcal{E}_{\text{arr}} \times \mathcal{E}_{\text{dep}}$ represent the time period in which a train is waiting at a station to let passengers get on or off; a waiting activity hence connects an arrival event of some train with its next departure event. Its duration $L_a > 0$ describes the minimal time required to allow boarding and unboarding; sometimes it also includes exchanging train crews or other actions.

6:4 The Trickle-In Effect: Modeling Passenger Behavior in Delay Management



■ **Figure 1** An event-activity network with three trains and four stations. The solid (black) arcs represent driving and waiting activities of the trains. The dashed (blue) arcs represent changing activities which are possible between Train 2 and Train 3 at Station 3 and between Train 1 and Train 2 at Station 2.

If two events $i, j \in \mathcal{E}$ are connected by an activity $(i, j) \in \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}}$, then event i has to be performed before event j can take place. In particular, the disposition timetable x has to satisfy

$$x_j - x_i \geq L_a$$

for all $a = (i, j) \in \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}}$.

- *Changing activities* $\mathcal{A}_{\text{change}} \subset \mathcal{E}_{\text{arr}} \times \mathcal{E}_{\text{dep}}$ allow passengers to transfer from an incoming train to an outgoing train. Hence, a changing activity connects an arrival event of some train at some station with a departure event of another train at the same station. The lower bound $L_a > 0$ refers to the minimum time a passenger needs to transfer between both trains. In order to solve the delay management problem we have to decide for each changing activity if it should be kept or if it can be deleted. In case that a changing activity $a = (i, j)$ is kept, the disposition timetable x must satisfy $x_j - x_i \geq L_a$. If the changing activity is deleted, the outgoing train can depart without waiting for the incoming train and this inequality does not need to be satisfied anymore.

We remark that other types of activities such as *headway activities* or *turnaround activities* may be added, see [10] for the respective models. Notwithstanding that, in this work we focus on the classical model.

To formulate an integer programming model of the delay management problem, we next have to formally introduce the delays. We assume that a set of unexpected *source delays* is known, e.g., caused by signaling problems, construction work, accidents, or bad weather conditions. These source delays cause *secondary delays*, e.g., for the same train at subsequent stations or for other trains that wait for the delayed train. In our work we allow two types of source delays: The first type is a delay $d_i \in \mathbb{N}$ at an event $i \in \mathcal{E}$ (e.g., staff coming too late to their duty) referring to a fixed point of time. In this case, $x_i \geq \pi_i + d_i$ is required. The second type of source delay is a delay d_a which increases the duration of an activity $a = (i, j) \in \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}}$, e.g., an increase of traveling time between two stations due to construction work. Such a delay d_a has to be added to the minimal duration L_a of activity a . If an event or an activity has no source delay, we assume $d_i = 0$ or $d_a = 0$, respectively, to simplify the notation.

In the objective function we evaluate the disposition timetable from the passengers' point of view. To this end, let w_i be the number of passengers unboarding the train at event $i \in \mathcal{E}$ (thus, $w_i = 0$ for all $i \in \mathcal{E}_{\text{dep}}$) and w_a be the number of passengers who want to use

a changing activity $a \in \mathcal{A}_{\text{change}}$. We assume $w_a > 0$ for all $a \in \mathcal{A}_{\text{change}}$ – otherwise, the changing activity could be removed from the network, since nobody uses it. We further assume that all lines have a common period T , i.e., every line is served by a train every T minutes. Note that this assumption can be relaxed by introducing periods T_a for all changing activities $a \in \mathcal{A}_{\text{change}}$.

We can now state the integer programming formulation for the basic version of the delay management problem. To model the wait-depart decisions, i.e., whether some train should wait for some other train at a station or not, we introduce binary variables

$$z_a = \begin{cases} 0 & \text{if changing activity } a \text{ is maintained} \\ 1 & \text{otherwise} \end{cases}$$

for all changing activities $a \in \mathcal{A}_{\text{change}}$. The integer programming formulation then reads as follows:

$$\min \sum_{i \in \mathcal{E}} w_i(x_i - \pi_i) + \sum_{a \in \mathcal{A}_{\text{change}}} z_a w_a T \quad (\text{DM})$$

$$\text{s.t.} \quad x_i \geq \pi_i + d_i \quad \forall i \in \mathcal{E} \quad (1)$$

$$x_j - x_i \geq L_a + d_a \quad \forall a = (i, j) \in \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}} \quad (2)$$

$$M z_a + x_j - x_i \geq L_a \quad \forall a = (i, j) \in \mathcal{A}_{\text{change}} \quad (3)$$

$$x_i \in \mathbb{N} \quad \forall i \in \mathcal{E}$$

$$z_a \in \{0, 1\} \quad \forall a \in \mathcal{A}_{\text{change}}$$

where M is a fixed constant. The meaning of the objective function and of the constraints is explained next.

The first term of the objective function minimizes the sum of all delays of all events. If all connections were maintained, this would be the sum of delays for all passenger at their final destination. The second term adds the weighted sum of all missed connections with a penalty of one time period T (or T_a if we drop the assumption of a common period of all lines) a passenger has to wait for the next train of the same line. The objective function is hence an approximation of the sum of all delays over all passengers. It has been shown in [22] that it is not an approximation, but exactly computes the sum of all passengers' delays if the so-called never-meet property holds.

Constraints (1) and (2) ensure that the delay is passed on correctly along driving and waiting activities. (3) does the same for maintained changing activities (i.e. if $z_a = 0$). If, however, $z_a = 1$, constraints (3) get redundant if M is chosen big enough. If no capacity constraints are considered and $d_a = 0$ for all $a \in \mathcal{A}$, [22] shows that choosing M as the largest source delay $\max_{i \in \mathcal{E}} d_i$ is sufficient. Solution methods for (DM) mainly rely on integer programming, see [10] and references therein.

3 Modeling the Trickle-in Effect

In this section we adapt the classical delay management model (DM) by taking the following two phenomena into account:

1. Passengers do not change with the same speed. There may be fast and slow passengers and a decision for keeping a changing activity means practically that the train waits for all (even for the slowest) passengers.
2. Due to the trickle-in effect, trains are not able to depart while passengers are still boarding.

The first point is modeled by using a time interval (L_a^{\min}, L_a^{\max}) instead of a fixed time L_a to describe the duration of the changing activities. We hence replace L_a in constraints (3) by L_a^{\max} . The second point implies that a train can either depart before the fastest passenger has arrived or after the slowest one has boarded, i.e., it cannot depart in the interval $(x_i + L_a^{\min}, x_i + L_a^{\max})$. This restriction is modeled by adding new constraints as follows.

► **Lemma 1.** *Let $a = (i, j) \in \mathcal{A}_{\text{change}}$. There exists $z_a \in \{0, 1\}$ such that*

$$Mz_a + x_j - x_i \geq L_a^{\max} \quad (4)$$

$$M(z_a - 1) + x_j - x_i \leq L_a^{\min} \quad (5)$$

are both satisfied if and only if

$$x_j \notin (x_i + L_a^{\min}, x_i + L_a^{\max}). \quad (6)$$

Proof. Let (4) and (5) hold for some $z_a \in \{0, 1\}$. If $z_a = 0$, (4) reduces to $x_j \geq x_i + L_a^{\max}$. On the other hand, if $z_a = 1$ then (5) reduces to $x_j \leq x_i + L_a^{\min}$. In both cases, $x_j \notin (x_i + L_a^{\min}, x_i + L_a^{\max})$.

Vice versa, let $x_j \notin (x_i + L_a^{\min}, x_i + L_a^{\max})$. If $x_j \leq x_i + L_a^{\min}$ we choose $z_a = 1$ to see that both, (4) and (5) hold. On the other hand, if $x_j \geq x_i + L_a^{\max}$ then $z_a = 0$ guarantees that (4) and (5) are satisfied. ◀

The proof of Lemma 1 specifies two possible cases for a dispatcher:

- The changing activity is maintained ($z_a = 0$) if and only if the train departs after the last passengers have boarded $x_j \geq x_i + L_a^{\max}$.
- The changing activity is dropped ($z_a = 1$) if and only if the train departs before the first passengers have boarded $x_j \leq x_i + L_a^{\min}$.

The resulting model (DM-trick) is hence given as

$$\min \sum_{i \in \mathcal{E}} w_i(x_i - \pi_i) + \sum_{a \in \mathcal{A}_{\text{change}}} z_a w_a T \quad (\text{DM-trick})$$

$$\text{s.t.} \quad x_i \geq \pi_i + d_i \quad \forall i \in \mathcal{E} \quad (7)$$

$$x_j - x_i \geq L_a + d_a \quad \forall a = (i, j) \in \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}} \quad (8)$$

$$Mz_a + x_j - x_i \geq L_a^{\max} \quad \forall a = (i, j) \in \mathcal{A}_{\text{change}} \quad (9)$$

$$M(z_a - 1) + x_j - x_i \leq L_a^{\min} \quad \forall a = (i, j) \in \mathcal{A}_{\text{change}} \quad (10)$$

$$x_i \in \mathbb{N} \quad \forall i \in \mathcal{E}$$

$$z_a \in \{0, 1\} \quad \forall a \in \mathcal{A}_{\text{change}}$$

We remark that (DM-trick) contains (DM) as a special case by setting $L_a^{\max} := L_a$ and $L_a^{\min} := L_a - 1$ for all $a \in \mathcal{A}$, i.e., it is a proper extension of the classical delay management model: Constraint (10) may not be contained as an explicit constraint in (DM), but is implicitly contained. This is the case because for every transfer $a = (i, j) \in \mathcal{A}$ that is missed ($z_a = 1$) it needs to hold that $x_j - x_i \leq L_a - 1$ since otherwise there would have been enough time to connect event i with event j and due to the objective function z_a would have been set to 0.

Trickle-in constraints can also be combined with all other extensions known for delay management, i.e., it is possible to consider headway constraints as in [18], station capacity constraints as in [7], or passenger routing constraints as in [9]. For the sake of simplicity we compare (DM) and (DM-trick) in their basic versions as given above.

4 Analyzing the New Model

As already mentioned in Section 2, for the classical delay management problem it suffices to choose M as large as the largest source delay $D := \max_{i \in \mathcal{E}} d_i$ if all $d_a = 0$. This does not hold any more for (DM-trick), but still the size of M can be bounded. To this end, we need the following two lemmas, both dealing with the original timetable π_i , $i \in \mathcal{E}$. For this chapter, we assume that the original timetable π is feasible, i.e., that

$$\pi_j - \pi_i \in [L_a, U_a] \quad \forall a = (i, j) \in \mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}}, \quad (11)$$

for all driving and waiting activities. For the changing activities we assume that the trickling constraints (6) applied to the original timetable π

$$\pi_j \notin (\pi_i + L_a^{\min}, \pi_i + L_a^{\max}) \quad \forall a = (i, j) \in \mathcal{A}_{\text{change}} \quad (12)$$

are satisfied, i.e., either nobody can change or everybody can. However, *changing* activities are the ones that allow passengers to change, so the case $\pi_j - \pi_i \leq L_a^{\min}$ cannot hold. We hence may assume that

$$\pi_j - \pi_i \in [L_a^{\max}, T + L_a^{\min}] \quad \forall a = (i, j) \in \mathcal{A}_{\text{change}} \quad (13)$$

where the upper bound $T + L_a^{\min}$ holds since every line runs at least once per time period T .

In order to simplify the notation, we will sometimes use the *delay* y_i of an event $i \in \mathcal{E}$ in its disposition timetable, which is defined as

$$y_i := x_i - \pi_i.$$

► **Lemma 2.** *Let π_i , $i \in \mathcal{E}$ be a feasible timetable. If all $d_a = 0$, then there exists an optimal solution with $y_j \leq D$ for all $a = (i, j) \in \mathcal{A}$.*

Proof. The proof works by induction. Since the event-activity network does not contain any directed cycles, we can sort the events $i \in \mathcal{E}$ topologically. Let $i_1, \dots, i_{|\mathcal{E}|}$ be the resulting order. Then the delay y_{i_1} of the first event i_1 is given by $d_{i_1} \leq D$. Now take any other event j and consider all of its incoming activities $(i, j) \in \mathcal{A}$. We now estimate how large the delay of event j can be. Note that there exists an optimal solution in which no disposition time can be reduced (i.e., which does not contain any unnecessary delays). This means one of the inequality constraints (7), (8), (9) is sharp.

- If (7) is sharp we get $x_j = \pi_j + d_j$, hence $y_j = x_j - \pi_j = d_j \leq D$.
- If (8) is sharp for $(i, j) \in \mathcal{A}$ we have that $x_j = x_i + L_a$, i.e., the delay of event j can be computed as

$$\begin{aligned} y_j &= x_j - \pi_j = L_a + x_i - \pi_j \\ &= L_a + y_i + \underbrace{\pi_i - \pi_j}_{\leq -L_a} \\ &\leq y_i \leq D \text{ by induction hypothesis} \end{aligned}$$

where we have used feasibility of the timetable, see (11) and that event i is topologically smaller than event j .

- If (9) is sharp for $(i, j) \in \mathcal{A}$ we analogously have that $x_j = x_i + L_a^{\max}$, i.e., the delay of event j can be computed as

$$\begin{aligned} y_j &= x_j - \pi_j = L_a^{\max} + x_i - \pi_j \\ &= L_a^{\max} + y_i + \underbrace{\pi_i - \pi_j}_{\leq -L_a^{\max}} \\ &\leq y_i \leq D \text{ by induction hypothesis} \end{aligned}$$

where we have used the second feasibility constraint for the timetable, see (13) and again that event i is topologically smaller than event j . ◀

Under the same conditions as in the above lemma we can hence estimate the size of big M , which is a bit larger than in (DM) but still of moderate size.

► **Lemma 3.** *If $d_a = 0$ for all $a \in \mathcal{A}$, then $M = T + D$ is large enough to correctly solve Model (DM-trick).*

Proof. We have to find M that satisfies the following two conditions:

1. Constraint (4) should get redundant if $z_a = 1$, i.e., for an optimal solution we require that $M \geq L_a^{\max} + x_i - x_j$. We hence look for an upper bound of the right hand side:

$$\begin{aligned} L_a^{\max} + x_i - x_j &= L_a^{\max} + \pi_i + y_i - \pi_j - y_j \\ &= L_a^{\max} + \underbrace{\pi_i - \pi_j}_{\leq -L_a^{\max}} + \underbrace{y_i}_{\leq D} - \underbrace{y_j}_{\leq 0} \leq D, \end{aligned}$$

where we again used feasibility of the timetable, see (13).

2. Constraint (5) should get redundant if $z_a = 0$, i.e., for an optimal solution we require that $M \geq x_j - x_i - L_a^{\min}$. We again need an upper bound of the right hand side:

$$\begin{aligned} x_j - x_i - L_a^{\min} &= \pi_j + y_j - \pi_i - y_i - L_a^{\min} \\ &= \underbrace{\pi_j - \pi_i}_{\leq T + L_a^{\min}} + \underbrace{y_j}_{\leq D} - \underbrace{y_i}_{\leq 0} - L_a^{\min} \leq T + D, \end{aligned}$$

this time using the upper bound in (13).

We conclude that $M = D + T$ suffices for both constraints (4) and (5). ◀

In the case of $d_a > 0$, delays can increase for single trains and have to be bounded. This can theoretically be done by summing up all delays d_a or (better) by finding a longest path P in the event-activity network with respect to the weights d_a , see [18].

Let us now consider the case that the timetable is feasible according to its traditional definition *without* the trickle-in effect, i.e., it satisfies $\pi_j - \pi_i \in [L_a, T + L_a - 1]$ instead of (13) for some $L_a < L_a^{\max}$. Then the trickle-in effect may generate delays.

Let us illustrate this on a small example: Given a timetable π that schedules train A to arrive at 10:00 and train B to depart at 10:02 and given a trickling interval of (1, 3) minutes, then the trickle-in effect is observable. The first passengers only need a little bit more than one minute to catch the train, but then a continuous stream of passengers boards the train leading to a delayed departure of train j at 10:03, i.e., to a delay of one minute. Thus, there may occur delays due to the trickle-in effect without the existence of any source delays.

However, even in this situation we can use (DM-trick) to find optimal wait-depart decisions dealing with both, source delays and delays occurring due to trickling constraints, and even in this situation we can bound M . To this end, assume a changing activity $a = (i, j)$ from event i to event j for which we have $L_a < \pi_j - \pi_i < L_a^{\max}$. Then the transfer of all passengers may take longer than the timetable allows. Hence, the trickle-in effect leads to a new type of “source delay” on this changing activity, namely a delay of $L_a^{\max} - (\pi_j - \pi_i)$. In order to find a bound for M we hence need to search for a longest path P' with respect to the weights

$$w'_a := \begin{cases} \max(0, d_a) & \text{if } a \in \mathcal{A}_{\text{wait}} \cup \mathcal{A}_{\text{drive}} \\ \max(0, d_a, L_a^{\max} - (\pi_j - \pi_i)) & \text{if } a = (i, j) \in \mathcal{A}_{\text{change}} \end{cases}$$

and add its length to M .

Hence, we receive a bound of

$$M = D + T + \text{length}(P')$$

in this case. The computational experiments show that $M = D + T + \text{length}(P')$ is of reasonable size and hence a sufficient upper bound for M .

We now analyze the new model (DM-trick) with respect to the intervals $[L_a^{\min}, L_a^{\max}]$ for the changing activities. Varying both, the lower and the upper bound on the duration of the changing activities gives the following result.

► **Lemma 4.** *Let $I_a(k) = [L_a^{\min}(k), L_a^{\max}(k)]$ for all $a \in \mathcal{A}_{change}$ be a sequence of nested intervals with*

$$L_a^{\min}(1) \leq L_a^{\min}(2) \leq \dots \leq L_a \text{ and } L_a \leq \dots \leq L_a^{\max}(2) \leq L_a^{\max}(1)$$

and let $z^*(k)$ be the optimal objective function value for (DM-trick) with respect to the interval $I_a(k)$ and z^* be the optimal objective function value of (DM). Then

$$z^*(1) \geq z^*(2) \geq \dots \geq z^*.$$

Proof. Since $I_a(k+1) \subseteq I_a(k)$ for all $a \in \mathcal{A}_{change}$, (DM-trick) with respect to the intervals $I(k+1)$ is a relaxation of (DM-trick) with respect to the intervals $I(k)$ and the result follows. ◀

As a consequence, (DM) is a relaxation of (DM-trick) whenever the changing times L_a in the classical model (DM) satisfy $L_a \in [L_a^{\min}, L_a^{\max}]$ for all $a \in \mathcal{A}_{change}$. Hence, solving (DM) gives a lower bound on (DM-trick). In the experiments in Section 5 we compare the gap between this lower bound and the real solution. The best approximation of (DM-trick) by (DM) is given if we set $L_a := L_a^{\max}$ for all $a \in \mathcal{A}_{change}$, i.e., making sure that also the slow passengers are able to board their next train. This is shown in the next Lemma.

► **Lemma 5.** *Let $[L_a^{\min}, L_a^{\max}]$ for all $a \in \mathcal{A}_{change}$ be the given data for (DM-trick). Let $z^*(L_a)$ be the optimal objective function value for (DM) with data L_a for all $a \in \mathcal{A}_{change}$. Then an optimal solution to*

$$\max\{z^*(L_a) : L_a \in [L_a^{\min}, L_a^{\max}] \text{ for all } a \in \mathcal{A}_{change}\}$$

is provided by setting $L_a = L_a^{\max}$ for all $a \in \mathcal{A}_{change}$, i.e., the best lower bound obtainable from the classical model (DM) is provided by setting $L_a := L_a^{\max}$ for all $a \in \mathcal{A}$.

Proof. From Lemma 4 we already know that all $L_a \in [L_a^{\min}, L_a^{\max}]$ provide lower bounds. We hence have to show that the largest of them is obtained by setting $L_a := L_a^{\max}$ for all $a \in \mathcal{A}$. To this end, let $L'_a \leq L_a^{\max}$ for all $a \in \mathcal{A}$. Let (x, z) be a solution of (DM) with respect to L_a^{\max} . It hence satisfies (3) with L_a^{\max} on the right hand side and hence also with $L'_a \leq L_a^{\max}$ on the right hand side. Hence, (x, z) is also feasible for (DM) with respect to L'_a . We conclude that (DM) with respect to L'_a is a relaxation of (DM) with respect to L_a^{\max} , and hence

$$z^*(L'_a) \leq z^*(L_a^{\max}).$$

The computational results underline that using (DM) as a relaxation for (DM-trick) impose a good trade-off between computation time and (DM)'s quality as a lower bound.

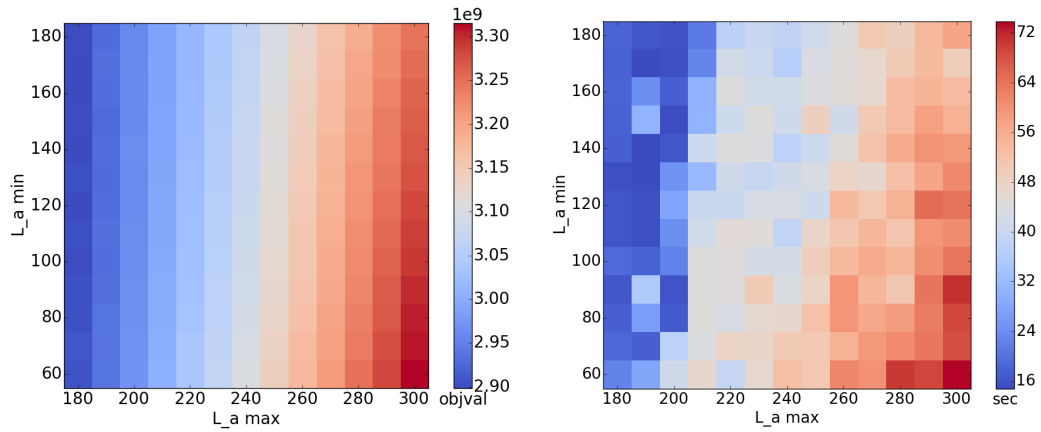
5 Experiments

In this section we investigate the effects of the trickle-in effect computationally. To this end, we implemented (DM-trick) in LinTim, an open source software framework for public transport optimization, see [19, 11]. We focus on solving the *bahn* dataset, consisting of 250 nodes and 326 edges, modeling the German ICE network, see Figure 3 in the appendix.

6:10 The Trickle-In Effect: Modeling Passenger Behavior in Delay Management

For quickly determining the timetable we use the MATCH heuristic as described in [16] since it is faster than the modulo simplex [12, 15] or integer programming approaches [13]. We roll out the periodic timetable for 4 hours and receive an aperiodic event-activity-network with around 20000 events and 40000 activities. For generating delays we use a LinTim procedure which is parameterized to choose 1000 activities and to generate source delays uniformly distributed between 1 and 900 seconds for each of the chosen activities. In order to calculate a sufficiently big M as described in Section 4, we calculate $\text{length}(P) = 3500$ seconds and $D = 0$ (because we generate source delays only on activities) and T was chosen to be 3600 seconds, leading to a choice of $M = 7100$. We implemented (DM-trick) using Gurobi 8.0 with a relative optimality gap of 1% and run the experiments on a compute server with 12 cores of Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz and 78 GB RAM.

In our first experiment we compare different trickling intervals with the lower bound L_a^{\min} ranging from 60 to 180 seconds and L_a^{\max} ranging from 180 to 300 seconds. The default minimum changing time L_a is assumed to be 180 seconds. The objective values, given in passengers times seconds, for solving (DM-trick) with these different intervals are depicted in Figure 2a.



(a) objective values in passengers \times seconds.

(b) runtimes in seconds.

■ **Figure 2** (DM-trick) for different trickling intervals (L_a^{\min}, L_a^{\max}) in seconds.

We see that the instance with the smallest trickling interval ([180, 180] seconds) has the lowest objective value of about $2.9 \cdot 10^9$, whereas the instance with the largest trickling interval ([60, 300] seconds) has the largest objective value ($3.3 \cdot 10^9$). This is consistent with the theory since small trickling intervals are a relaxation of larger trickling intervals, see Lemma 4. A higher objective value is equivalent to higher passenger delays in the event-activity-network which makes sense as a larger trickling interval potentially leads to longer waiting times for trains. In general, one can observe that a larger interval correlates with a higher objective value, and furthermore that a change in L_a^{\max} has a higher impact on the objective value than a change in L_a^{\min} .

Figure 2b now depicts the runtimes for different choices of the trickling interval.

Interestingly, also the instance with the smallest trickling interval has the lowest runtime and the instance with the largest trickling interval has the highest runtime. Also for the other instances the runtimes correlate primarily with the size of the trickling interval (although not as smoothly as the objective value) and a change in the upper bound L_a^{\max} has again a

higher impact on the runtime than a change in L_a^{\min} . The correlation between size of the trickling interval and runtime can be explained by the nature of integer programming. If the size of the “forbidden” trickling interval is increased, we get a weaker linear relaxation of the integer problem and hence need longer to solve it, e.g., via branch-and-bound.

The next experiment investigates the difference between a disposition timetable found by (DM) and a disposition timetable that respects the trickle-in effect. To this end, Figure 4 (in the appendix) depicts the number of changing activities of a disposition timetable from (DM) (with $L_a = 180$ seconds) that lie in the trickling interval. Hence, Figure 4 illustrates the difference between a disposition timetable found by solving (DM) and disposition timetables found by solving (DM-trick) for different trickling intervals. As can be seen in the figure, there exist up to 1194 infeasible change activities for a disposition timetable from (DM). In other words, if a disposition timetable from (DM) is found, it can be the case that 1194 changing activities (or about 6% of all changing activities) cause new delays due to the neglect of the trickle-in effect.

Furthermore, we investigate the results of Lemma 5, i.e., that solving (DM) with $L_a := L_a^{\max}$ yields the best approximation to (DM-trick).

One can see in Figure 5 (in the appendix) that the objective value indeed increases when L_a increases, culminating in a gap of only 3% if L_a is chosen to be L_a^{\max} . Hence, we get a reasonably good approximation of (DM-trick) by only solving an instance of (DM). Furthermore, it should be noted that solving (DM) takes only around 1 second, whereas solving (DM-trick) with trickling interval [60, 300] seconds took around 77 seconds to solve. Hence, we indeed get a decent trade-off between computation time and solution quality.

Finally, we investigate the difference between the disposition timetables from (DM) and (DM-trick) (with a trickling interval of [60, 300] seconds). We observe that the solution from (DM) schedules trains such that 566 connections are missed, whereas in the solution of (DM-trick) there are 684 missed connections. Interestingly, 513 of the missed connections coincide such that in this case considering the trickling effect yields to a change in 224 wait-depart-decisions. Thus, not only the objective values of the two models (DM) and (DM-trick) varies, but also the structure of the resulting delay management strategy.

As a final note, we also run the model (DM-trick) for the case if no source delays exist and received an objective value of around $5 \cdot 10^8$. This is roughly 15% of the objective value we encountered while working with the aforementioned 1000 source delays. Put differently, in this instance up to 15% of the delays might not be caused by source delays, but by the mere structure of the underlying periodic timetable and the trickle-in effect. Hence, the trickle-in effect has high relevance beyond delay management and should already be considered when planning a periodic timetable (which is not the case when using, e.g., MATCH for timetabling).

6 Conclusion and Suggestions for Further Research

In this paper we introduced the trickle-in effect, an observation on passenger behavior at train stations that highly influences delays in public transport. We introduced models for incorporating the trickle-in effect into standard delay management models and also showed how it already influences the periodic timetabling problem. We investigated mathematical properties of the resulting model and showed how (DM-trick) can be approximated best using the classical delay management problem. This allows to use approaches for classical delay

management (such as [3, 17, 8, 6]) for heuristically solving (DM-trick). The computational experiments underlined our hypothesis that the trickle-in effect has a high impact on delay management: Here, the objective value of (DM-trick) exceeds the objective value of (DM) up to 15%. Finally, since the computation times for (DM-trick) rise significantly, we still can get a decent approximation of (DM-trick) by solving a modified version of (DM).

Further research includes simulation approaches to better understand the behaviour of the passengers and to derive practically relevant trickling intervals. To this end, an agent-based simulation as in [2] is currently developed. We are also interested in adding the trickling constraints to more sophisticated delay management models including passengers' routing and capacity constraints.

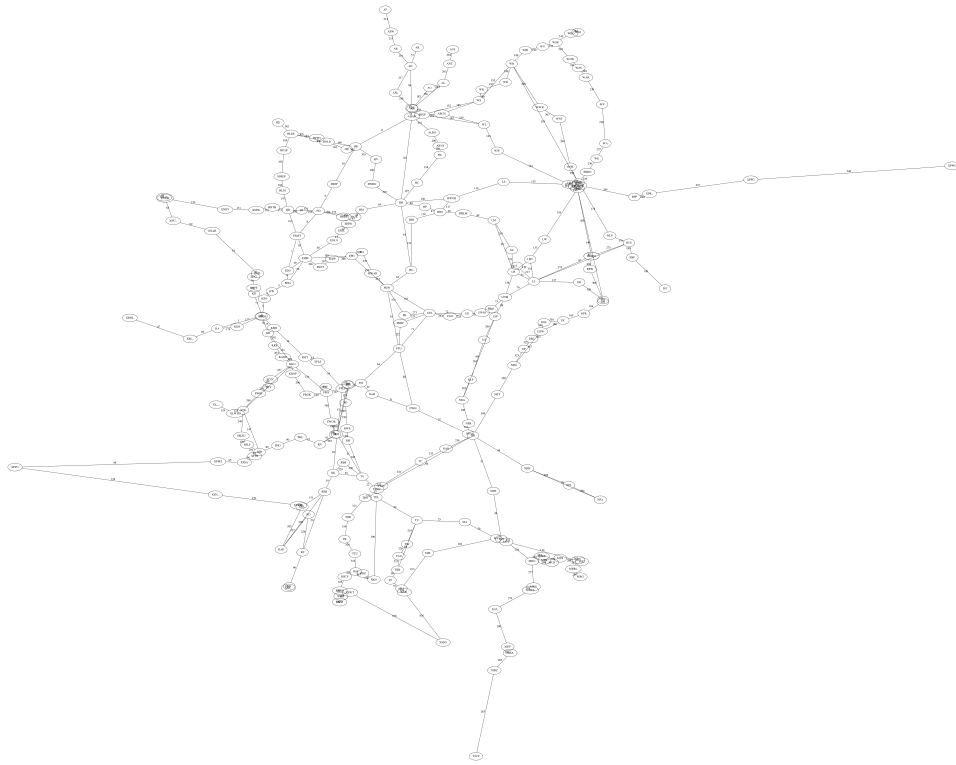
Finally, there is another line of research, namely adding trickling constraints to the timetabling problem. In Section 4 we have already seen that considering the trickle-in effect in a timetable that is not feasible with respect to (13) might cause source delays. The experiments justify this theoretical observation: a timetable might get significant delays just because of the trickle-in effect, i.e., even if no other source delays occur. We hence suggest to consider the trickle-in effect already in the timetabling phase. This means to add constraints of type (12) in timetabling such that either all passengers or none of the passengers can make a transfer. Hence, $\pi_j - \pi_i \in [L_a^{\max}, T + L_a^{\min}]$ needs to be respected for all changing activities (i, j) and even more general for all activities (i, j) from an arrival event of an incoming train to a departure event of (another) outgoing train. These constraints can be transferred also to periodic timetabling and considered as additional constraints in the periodic event scheduling problem (PESP). The analysis of them (runtime, impact on resulting timetable) are an interesting topic for future research which seems to be challenging and highly relevant from a practical point of view.

References

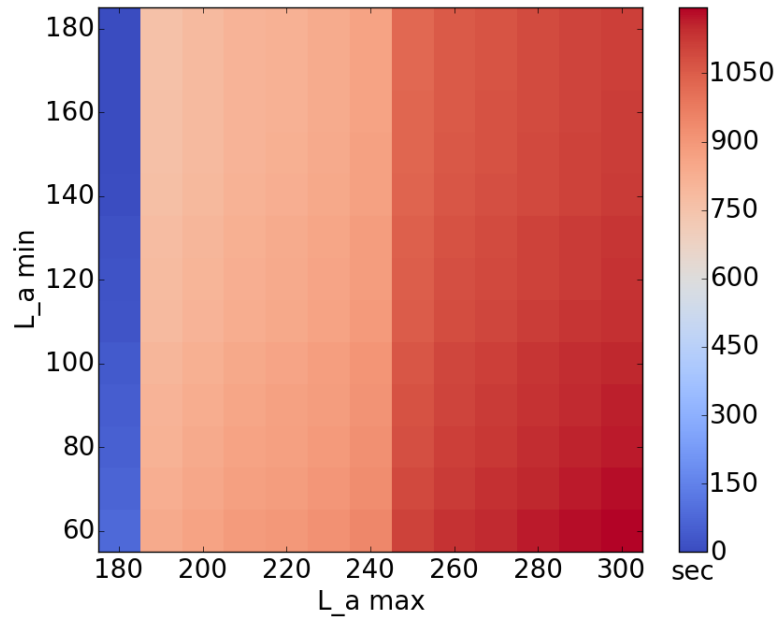
- 1 S. Albert, P. Kraus, J.P. Müller, and A. Schöbel. Passenger-induced delay propagation: Agent-based simulation of passengers in rail networks. In *Simulation Science*, volume 889 of *Communications in Computer and Information Science (CCIS)*, pages 3–23. Springer, 2018.
- 2 M. Aschermann, S. Dennisen, P. Kraus, and J.P. Müller. LightJason, a Highly Scalable and Concurrent Agent Framework: Overview and Application (demonstration paper). In M. Dastani, G. Sukthankar, E. Andre, and S. Koenig, editors, *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018)*, pages 1794–1796, 2018.
- 3 R. Bauer and A. Schöbel. Rules of Thumb — Practical online strategies for delay management. *Public Transport*, 6(1):85–105, 2014. URL: <http://num.math.uni-goettingen.de/preprints/files/2011-03.pdf>.
- 4 C. Conte and A. Schöbel. Identifying dependencies among delays. In *proceedings of IAROR 2007*, 2007. ISBN 978-90-78271-02-4.
- 5 L. De Giovanni, G. Heilporn, and M. Labbé. Optimization models for the single delay management problem in public transportation. *European Journal of Operational Research*, 189(3):762–774, 2008.
- 6 T. Dollevoet and D. Huisman. Fast Heuristics for Delay Management with Passenger Rerouting. *Public Transport*, 6(1-2):67–84, 2014.
- 7 T. Dollevoet, D. Huisman, L. Kroon, M. Schmidt, and A. Schöbel. Delay Management including capacities of stations. *Transportation Science*, 49(2):185–203, 2015.

- 8 T. Dollevoet, D. Huisman, L.G. Kroon, L.P. Veenturf, and J.C. Wagenaar. Application of an iterative framework for real-time railway scheduling. *Computers and Operations Research*, 78:203–217, 2017.
- 9 T. Dollevoet, D. Huisman, M. Schmidt, and A. Schöbel. Delay Management with Rerouting of Passengers. *Transportation Science*, 46(1):74–89, 2012.
- 10 T. Dollevoet, D. Huisman, M. Schmidt, and A. Schöbel. Delay propagation and delay management in transportation networks. In R. Borndörfer et al., editor, *Handbook of Optimization in the Railway Industry*. Springer, 2018.
- 11 M. Goerigk, M. Schachtebeck, and A. Schöbel. Evaluating Line Concepts using Travel Times and Robustness: Simulations with the Lintim toolbox. *Public Transport*, 5(3), 2013.
- 12 M. Goerigk and A. Schöbel. Improving the Modulo Simplex Algorithm for Large-Scale Periodic Timetabling. *Computers and Operations Research*, 40(5):1363–1370, 2013.
- 13 C. Liebchen, M. Proksch, and F.H. Wagner. Performances of Algorithms for Periodic Timetable Optimization. In *Computer-aided Systems in Public Transport*, pages 151–180. Springer, Heidelberg, 2008.
- 14 K. Nachtigall. *Periodic Network Optimization and Fixed Interval Timetables*. PhD thesis, University of Hildesheim, 1998.
- 15 K. Nachtigall and J. Opitz. Solving Periodic Timetable Optimisation Problems by Modulo Simplex Calculations. In *Proc. ATMOS*, 2008.
- 16 J. Pätzold and A. Schöbel. A Matching Approach for Periodic Timetabling. In Marc Goerigk and Renato Werneck, editors, *16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2016)*, volume 54 of *OpenAccess Series in Informatics (OASISs)*, pages 1–15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/OASISs.ATMOS.2016.1.
- 17 R. Rückert, M. Lemnian, C. Blendinger, S. Rechner, and M. Müller-Hannemann. PANDA: a software tool for improved train dispatching with focus on passenger flows. *Public Transport*, 9:307–324, 2017.
- 18 M. Schachtebeck and A. Schöbel. To wait or not to wait and who goes first? Delay Management with Priority Decisions. *Transportation Science*, 44(3):307–321, 2010. doi:10.1287/trsc.1100.0318.
- 19 A. Schiewe, S. Albert, J. Pätzold, P. Schiewe, A. Schöbel, and J. Schulz. LinTim: An integrated environment for mathematical public transport optimization. Documentation. Technical Report 2018-08, Preprint-Reihe, Institut für Numerische und Angewandte Mathematik, Georg-August-Universität Göttingen, 2018.
- 20 M. Schmidt and A. Schöbel. The complexity of integrating routing decisions in public transportation models. *Networks*, 65(3):228–243, 2015.
- 21 A. Schöbel. A Model for the Delay Management Problem based on Mixed-Integer Programming. *Electronic Notes in Theoretical Computer Science*, 50(1), 2001.
- 22 A. Schöbel. Integer Programming approaches for solving the delay management problem. In *Algorithmic Methods for Railway Optimization*, number 4359 in *Lecture Notes in Computer Science*, pages 145–170. Springer, 2007.
- 23 L. Suhl, C. Biederbick, and N. Kliewer. Design of customer-oriented Dispatching Support for railways. In S. Voß and J. Daduna, editors, *Computer-Aided Transit Scheduling*, volume 505 of *Lecture Notes in Economics and Mathematical systems*, pages 365–386. Springer, 2001.

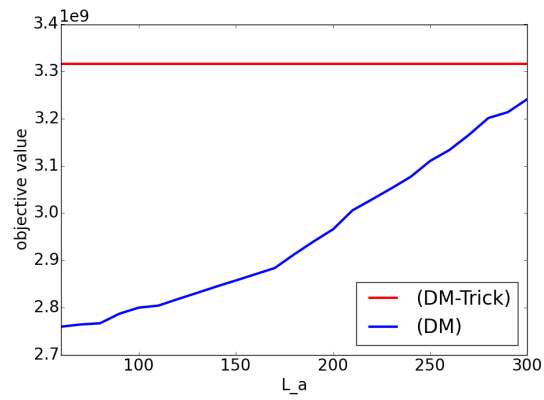
A Figures



■ **Figure 3** *bahn* dataset.



■ **Figure 4** number of infeasible changing activities for a timetable from (DM) for different trickling intervals.



■ **Figure 5** objective values for (DM) for different L_a .

Vehicle Capacity-Aware Rerouting of Passengers in Delay Management

Matthias Müller-Hannemann 

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg,
Von-Seckendorff-Platz 1, D 06120 Halle (Saale), Germany
muellerh@informatik.uni-halle.de

Ralf Rückert

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg,
Von-Seckendorff-Platz 1, D 06120 Halle (Saale), Germany
ralf.rueckert@informatik.uni-halle.de

Sebastian S. Schmidt 

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg,
Von-Seckendorff-Platz 1, D 06120 Halle (Saale), Germany
sebastian.schmidt@informatik.uni-halle.de

Abstract

Due to the significant growth in passenger numbers, higher vehicle load factors and crowding become more and more of an issue in public transport. For safety reasons and because of an unsatisfactory discomfort, standing of passengers is rather limited in high-speed long-distance trains. In case of delays and (partially) cancelled trains, many passengers have to be rerouted. State-of-the-art rerouting merely focuses on minimizing delay at the destination of affected passengers but neglects limited vehicle capacities and crowding. Not considering capacities allows using highly efficient shortest path algorithms like RAPTOR or the connection scan algorithm (CSA).

In this paper, we study the more complicated scenario where passengers compete for scarce capacities. This can be modeled as a piece-wise linear, convex cost multi-source multi-commodity unsplitable flow problem where each passenger group which has to be rerouted corresponds to a commodity. We compare a path-based integer linear programming (ILP) model with a heuristic greedy approach. In experiments with instances from German long-distance train traffic, we quantify the importance of considering vehicle capacities in case of train cancellations. We observe a tradeoff: The ILP approach slightly outperforms the greedy approach and both are much better than capacity unaware rerouting in quality, while the greedy algorithm runs more than three times faster.

2012 ACM Subject Classification Applied computing → Transportation; Theory of computation → Discrete optimization; Mathematics of computing → Network flows

Keywords and phrases Delay management, passenger flows, vehicle capacities, rerouting

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.7

Funding This work has partially been supported by a DFG grant for the research group FOR 2083, project MU1482/7-2.

Acknowledgements The authors wish to thank Deutsche Bahn for providing test data.

1 Introduction

Recent years have shown significant growth in passenger numbers on public transport services in many countries. Due to political efforts to increase utilization of public transport in support of sustainability goals, further growth is to be expected. While congestion in metro systems of mega-cities during peak hours has been recognized as a challenge for many years, awareness of increased in-vehicle density as a problem also for the management of passenger flows in long-distance trains started only recently.



© Matthias Müller-Hannemann, Ralf Rückert, and Sebastian S. Schmidt;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 7; pp. 7:1–7:14

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we combine delay and disruption management with scarce vehicle capacities. In large and complex public transportation systems delays as well as disruptions occur frequently. Typical reasons are technical defects, construction work, bad weather conditions, exceptionally many passengers, accidents, and the like. As a consequence, passengers miss planned transfers which results in a significant delay at their destinations, in considerable dissatisfaction, and ultimately in economic loss for the railway company. In delay management, train dispatchers have to decide which trains shall wait for delayed incoming trains in order to maintain connections for passengers. Such problems are challenging for several reasons: One has to deal with large-scale networks subject to dynamically changing, partially incomplete and imprecise information about current delays and their propagation, and solutions are required in almost real-time. In an on-going joint research project with Deutsche Bahn, we are working on the development of a decision support system for dispatchers which shall help to find optimal waiting decisions from a passenger's point of view. A key assumption is that detailed information about passenger flows is available, that is, for each passenger the planned route is known. Such passenger flows can be based on sold tickets or statistically validated demand models. In our long-term project, we have built a prototype for an optimized passenger-friendly disposition system, named PANDA [16, 19]. The acronym PANDA abbreviates Passenger-Aware Novel Dispatching Assistance. It is designed to provide train dispatchers with detailed real-time information about the current passenger flow and the multi-dimensional impact of waiting decisions in case of train delays and cancellations. If trains are cancelled or connections cannot be maintained, passengers have to be rerouted. State-of-the-art solutions determine new routes for passengers that are optimized subject to earliest arrival at the planned destination with few transfers as a secondary criterion. Technically, this requires the efficient solution of large-scale multi-criteria shortest path problems in suitably designed event-activity networks. Recent progress in shortest path algorithms for such applications allows to solve such problems in a few milliseconds per instance, for example by using RAPTOR [5] or the connection scan algorithm (CSA) [6, 7]. Capacity constraints, however, are widely neglected in previous work. Considering the available free capacity to avoid overcrowded trains leads to several, more challenging combinatorial optimization problems. With respect to capacities, we may distinguish between hard and soft capacities. For each vehicle, there is a designated number of available seats. This gives a soft capacity beyond which it becomes more and more uncomfortable to travel. At a certain threshold, the hard capacity, a vehicle becomes so crowded that it is not allowed to run anymore for security reasons.¹ Key drivers for crowding discomfort of passengers are dissatisfaction with standing and not being seated, fewer opportunities to make use of the time during the journey, and the physical closeness of other travellers per se [13].

Goals and contribution. The main use case and focus of this work are cancelled or partially cancelled trains where many (up to several hundred) passengers have to be rerouted simultaneously. A second use case are passengers with missed connections due to wait-depart decisions in delay management. For the most important train connections, also large numbers of passengers are affected by a single decision.

A crucial issue concerns the model how passengers behave. If we assume that passengers behave selfishly and inform themselves individually and independently about alternative routes, we have only limited capabilities to avoid overcrowded trains. All we can do in such a

¹ According to Richard Lutz, Chief Executive Officer (CEO) of Deutsche Bahn, overfull long-distance trains of Deutsche Bahn have to be stopped and evacuated half a dozen times a week (Handelsblatt of April 19, 2018).

scenario is to only recommend trains which have enough free capacity and to put preference on connections with lower seat occupation. To support such a goal, one can still solve shortest path problems. In an attempt to avoid violations of hard capacities, we can simply forbid all arcs in the event-activity model which would lead to overfull vehicles. Moreover, the objective function can be modified so that it prefers trains with larger free capacity.

Different optimization problems arise if we take the perspective of a recommending system which tries to achieve a system optimum, that is, a solution which minimizes the overall inconvenience for all affected passengers. Inconvenience can be expressed in several ways, the simplest version being the total delay at the destination, summed up over all passengers. Overcrowding of trains can be penalized with the help of convex cost functions. For such scenarios, we have to solve some large-scale integral minimum cost multi-commodity flow problems, where each group of passengers sharing the same origin and destination corresponds to some commodity. Since passengers groups want to travel together, we have to consider versions of unsplittable flow problems. Such problems are well-known to be NP-hard optimization problems. The most common approach to solve them is integer linear programming (ILP) with a path-based formulation and column generation. While the underlying network is fairly large, the number of commodities to be considered is typically of moderate size. Moreover, for each commodity, there is only a limited number of “reasonable” alternative paths to which one can heuristically restrict the search. We follow this general approach, and in addition, we will also consider a fast greedy rerouting scheme. With this work we want to tackle the following research questions:

1. How relevant is crowding-aware rerouting in long-distance trains already today? How much more important will it be with rising numbers of passengers?
2. Comparing a capacity-aware greedy rerouting with a minimum cost multi-commodity flow model, how much do we lose in quality if we use a greedy algorithm?
3. Can we solve the instances of unsplittable flow problems fast enough in practice?

Due to our cooperation with DB Fernverkehr, we concentrate on long-distance trains. Our main results are as follows. First, we observe that ignoring vehicle capacities would guide many passengers into overfull trains. This effect will become more severe with rising passenger numbers. Conversely, with our models we can reduce passenger inconvenience to a large amount. Second, the ILP solution is slightly better in quality than the greedy approach, but the greedy approach is about three times as fast. Third, while the ILP problems can be solved very easily within milliseconds, the required computation of alternative paths turns out to be the bottleneck. Severe cases of train cancellations with several hundred passenger groups require on average less than 85 seconds of computation time for our ILP approach.

Related work. Delay management has been studied very intensively in the literature, see the recent survey [10]. Based on event-activity networks most of these approaches model delay management by integer linear programming (ILP) and consider offline versions of the problem, where all delays are known before the optimization process starts [21, 22]. First approaches considered simplified versions and assumed that passengers who miss a connection wait for the next connection on the same line. Integrated passenger rerouting has been considered by [9, 10, 20]. The integration of passenger rerouting into an ILP formulation leads to a considerable blow-up, making these formulations very large. With today’s techniques for integer programming, the handling of several hundred thousands of passenger routes seems to be impossible in an online setting. Dollevoet and Huisman [8] propose several fast heuristics and introduce an iterative ILP approach which comes close to the exact solution but is significantly faster. However, it remains open whether their iterative ILP approach

scales well to large-scale networks. All previous delay management models have in common that they do not consider vehicle capacities and crowding. Models and algorithms for efficient passenger routing in public transport (timetable information) have intensively been discussed in the literature, see for example the surveys by Müller-Hannemann et al. [17] and the more recent one on efficient shortest path algorithms by Bast et al. [4]. Among the many recent approaches for shortest path computation in public transportation networks, most relevant for this work are those which are well suited to a dynamic online scenario. As they require no heavy preprocessing, the above-mentioned approaches RAPTOR [5] and an extension of CSA [6, 7] serve very well for minimizing earliest arrival time and the number of transfers. RAPTOR can be extended to determine the Pareto set of optimal solutions for additional criteria, for example reliability and ticket price (McRaptor). Discomfort of crowding as an additional criterion has to the best of our knowledge not been considered in a multi-criteria setting. There has been, however, related work, in load balancing of passenger flows. For example, Huang et al. [14] study route guidance for passengers as a means to reduce in-vehicle congestion. The problem of finding alternative routes has found quite some attention before. One possibility is to search for the top k shortest paths. Since event-activity networks are directed acyclic graphs, these paths can be found and output by Eppstein's algorithm in $O(n \log n + m + kn)$ time in networks with n vertices and m arcs [11]. As the top k shortest paths may be too similar, several attempts have been done to find sets of paths with limited overlap. One such approach is first to compute a large set of candidate paths and then to filter candidates with respect to some similarity measure, for example [1].

Multi-commodity flows and unsplittable flow problems have been extensively studied in the literature [2, 23]. In general, they are strongly NP-hard since many combinatorial problems, including disjoint paths, can be reduced to it. Classical applications of unsplittable flow problems include, for example, bandwidth packing problems in telecommunication networks or express package delivery problems in logistics [3]. For solving large-scale instances of unsplittable flow problems, path-based formulations have advantages over arc-based formulations [3]. Since the number of paths grows exponentially with the size of the graph, exact solutions usually require column generation. Barnhart et al. [3] provide seminal work on column generation and branch-and-price-and-cut algorithms for unsplittable flows. Fortz et al. [12] discuss models for piecewise linear cost versions of the unsplittable multi-commodity flow problem. Wang [23] provides a recent survey on solution methods for multi-commodity network flow problems.

Overview. The remainder of this paper is organized as follows. In Section 2, we present our multi-commodity flow model. We start with general considerations and model assumptions. Afterwards, we develop and explain step-by-step the underlying event-activity network, our modelling of capacities and cost functions, and the resulting integer programming formulation. Then, we present a fast greedy heuristic and, finally, we describe our approach for the computation of candidates for alternative paths. Our computational study with many instances from Deutsche Bahn is reported in Section 3. Finally, we summarize and conclude with future work.

2 Multi-Commodity Flow Model

In this section, we develop our approach for the simultaneous rerouting of passengers with respect to limited vehicle capacities.

2.1 Basic considerations and assumptions

Our model is based on the following considerations and assumptions:

- In case of a disruption, our task is to find valid alternative routes for *all* affected passengers whose planned connection becomes invalid. If we cannot find an “acceptable” alternative (say, with at most two hours of delay at the destination), this imposes a high cost for compensation.
- Only directly affected passengers are rerouted. We assume that all others who are not forced to change plans keep their planned route. In some cases, delayed trains may offer new opportunities for passengers to optimize their routes, but this issue is ignored in our model.
- Groups of passengers have planned to travel together (for instance, couples, families, school classes). They certainly have to stay together also in their new route. This implies that we have to consider unsplittable flow models. Moreover, many passengers share the same origin and destination even if they are personally unrelated. They may be treated as a group since recommending different alternatives to them might be hard to explain and communicate without personalized route guidance.
- Rerouting of passengers comes with several disadvantages. Passengers may lose their seat reservation and have to enter more crowded trains. The valuation of discomfort is very subjective and varies widely between passengers. It depends individually on personal circumstances (like age and healthiness), the reason for traveling, and several other factors [15]. Nevertheless, to keep the model simple enough, we use the same general utility functions for all passengers.
- For simplicity, we do not make a distinction between first and second class travelers.
- Train tickets are often bound to a specific connection for which they are booked. In case of disruption, we assume that passengers may choose any train and any connection (no restrictions due to ticket regulations apply).
- We restrict the set of eligible alternative paths to “reasonable” ones: i.e. we consider only paths where passengers arrive at their final destination at most 120 minutes after the earliest feasible arrival time. We also exclude paths with too many train changes, and the upper limit is at most six transfers.
- Train cancellations or missed transfers only become known at short notice, at a certain event-specific release time. Since passengers can react only after they become aware of a need to find an alternative route, we require that replanning can alter the original route only after this release time. Several cases are possible:
 1. The passenger has not yet started traveling. In this case, we assume that the passenger arrives at the station where the original route would have started more or less just in time for the planned train. Thus, an alternative route may not start earlier and should begin at the same station (although an initial footpath is allowed).
 2. The passenger is already traveling. Then, based on the current location (in some specific train or at a station) and time, an alternative connection must be compatible with the initial part of the original route.

Event-activity network

The given train schedule and the set of passenger routes can be modeled with the help of a so-called *event-activity network (EAN)* $N = (V, A)$, a directed acyclic graph with vertex set V and arc set A . The vertices of the network correspond to the set of all arrival and departure events of the given schedule. Arcs of the network model order relations between events. We distinguish between different types of arcs (“activities”):

7:6 Vehicle Capacity-Aware Rerouting

- *driving arcs*, modeling the driving of a specific train from one stop to its very next stop,
- *dwelling arcs*, modeling a train standing at a platform and allowing passengers to disembark or board the train, and
- *transfer arcs*, modeling the possibility for passengers to switch between two trains at the same or nearby stations.

Passenger routes correspond to paths in N from a departure event to some arrival event. Let K be the set of passenger groups which have to be rerouted due to a train cancellation or broken transfer. For $k \in K$, denote by d_k the size of group k . Denote by t_k the intended destination, i.e. the final station of the planned route. Likewise, denote by s_k the origin (“source”) of this group with respect to the time of replanning. The origin is the first station of the planned route if the journey has not yet started. Otherwise, it denotes the station where the group is currently waiting or the very next station at which they will arrive with their current train.

We extend the event-activity network N by adding a source and a target “event” for each passenger group $k \in K$. Each source s_k is connected to all departure events at the same station which can be reached by the group. If footpaths to nearby stations exist, we also connect s_k to the reachable departure events at these stations. At the target station, we connect all arrival events with t_k . In summary, we seek for each group a path which starts at source s_k and ends at target t_k . For instances with very high load and in particular for large groups of passengers, no feasible capacity-respecting path may exist. To make sure, that every instance has a feasible solution, we add for each pair (s_k, t_k) some direct “no-route” arc of infinite capacity but very high costs, so that such arcs are only chosen if no other path is available.

Capacities and cost functions

With every arc a which corresponds to a driving or dwelling activity of a train, we can associate a nominal seat capacity $cap(a)$. Recall that we do not distinguish between first and second class seats for simplicity. The hard capacity for such an arc is set as $\beta \cdot cap(a)$ where $\beta \geq 1$ is a parameter specifying the maximal overload acceptable for security reasons. For high-speed trains, choosing $\beta = 1.2$ may be a reasonable choice (and is used in our experiments).

For rerouting, we have to consider the *free capacity* which remains if we subtract the number of those passengers which are not affected by rerouting. Thus, if $load(a)$ denotes the current number of passengers on arc a , we obtain an upper bound $u_a = \max\{0, \beta \cdot cap(a) - load(a)\}$. (Arcs of overloaded segments with $u_a = 0$ are excluded from the model.) All arcs in A not corresponding to driving or dwelling activities of trains have unlimited capacity.

For arc $a \in A$, let $C_a(x_a)$ be a piece-wise linear convex cost function. For simplicity in notation, we assume that each cost function has exactly b linear segments, but we allow empty segments to model cost functions with fewer breakpoints. With $0 = u_a^0 \leq u_a^1 \leq u_a^2 \leq \dots \leq u_a^b = u_a$ we denote the breakpoints of the function. The cost varies linearly in the interval $[u_a^{i-1}, u_a^i]$ with slope c_a^i . Slopes are strictly increasing, i.e. $c_a^1 < c_a^2 < \dots < c_a^b$ (unless they are $+\infty$).

As a cost function, we use a kind of generalized or *perceived travel time* which penalizes transfers and crowding discomfort. For the latter, we use the time-multiplier method [18]. The cost function’s basic unit is travel time in minutes. Traveling, dwelling and transfer arcs each have a certain duration $dur(a)$. Train changes are penalized with α extra minutes per transfer. Traveling in the train (i.e. being on traveling or dwelling arcs) is penalized with respect to the load. For the segment of the piecewise linear cost function, we have penalty factors $\gamma_1 < \gamma_2 < \dots < \gamma_b$. Cost parameters are set as follows.

- For traveling and dwelling arcs we set:

$$c_a^1 = (1 + \gamma_1) \cdot dur(a), c_a^2 = (1 + \gamma_2) \cdot dur(a), \dots, c_a^b = (1 + \gamma_b) \cdot dur(a).$$

- Transfer arcs are uncapacitated and have only a single finite cost segment, thus we set

$$c_a^1 = dur(a) + \alpha, c_a^2 = +\infty, \dots, c_a^b = +\infty.$$

- “No-route arcs” have very high costs, say $c_a^1 = 10000, c_a^2 = +\infty, \dots, c_a^b = +\infty$.
- All remaining arcs have zero costs.

2.2 Integer linear programming formulation

We are now ready to formulate the unsplittable flow problem as an integer linear program. Denote by $P(k)$ be the set of all paths from s_k to t_k from which the group has to select exactly one. We use binary decision variables y_p^k where $y_p^k = 1$ if group k selects path $p \in P(k)$, and $y_p^k = 0$ otherwise. Let δ_a^p be an arc-path indicator variable that equals 1 if arc a is contained in path p . For arc $a \in A$, let x_a^k be the size of the flow on arc a of commodity k , and $x_a = \sum_{k \in K} x_a^k$ be the total flow on this arc.

A classical transformation of piecewise convex flows to standard flow with linear costs is to replace each arc a by a set of b parallel arcs [2]. The idea is to decompose the flow x_a into flows on the segments between neighboring breakpoints of the cost function. Define

$$f_a^i = \begin{cases} 0 & \text{if } x_a \leq u_a^{i-1} \\ x_a - u_a^{i-1} & \text{if } u_a^{i-1} < x_a \leq u_a^i \\ u_a^i - u_a^{i-1} & \text{if } x_a \geq u_a^i. \end{cases}$$

This implies $x_a = \sum_{i=1}^b f_a^i$ and $C_a(x_a) = \sum_{i=1}^b c_a^i f_a^i$. The path flow formulation is then:

$$\min \sum_{a \in A} \sum_{i=1}^b c_a^i f_a^i \tag{1}$$

subject to

$$\sum_{p \in P(k)} y_p^k = 1 \quad \text{for all } k \in K \tag{2}$$

$$\sum_{k \in K} \sum_{p \in P(k)} d_k y_p^k \delta_a^p = \sum_{i=1}^b f_a^i \quad \text{for all } a \in A \tag{3}$$

$$f_a^i \leq u_a^i \quad \text{for all } a \in A \text{ and all } i = 1, 2, \dots, b \tag{4}$$

$$f_a^i \geq 0 \quad \text{for all } a \in A \text{ and all } i = 1, 2, \dots, b \tag{5}$$

$$y_p^k \in \{0, 1\} \quad \text{for all } k \in K \text{ and all } p \in P(k) \tag{6}$$

Equations (2) ensure that exactly one path has to be chosen for each commodity. Equations (3) express that the total flow $x_a = \sum_{i=1}^b f_a^i$ on arc a equals the sum of chosen paths using this arc, weighted by the demands d_k of the commodities. Capacity constraints of all flow segments are given by Inequalities (4), while non-negativity of all flow variables is provided by Inequalities (5). The integrality of all flow variables f_a^i is implied by the integrality of the left-hand-side of Equations (3) and the strict increase in slope values c_a^i .

2.3 A fast greedy heuristic

The exact solution of the unsplittable flow problem is NP-hard, although the instances arising in our applications seem to be quite well solvable by state-of-the-art ILP solvers (see our experiments below). However, it requires the computation of many alternative paths in a first step which is computationally expensive.

Therefore, we suggest a simple, but fast greedy heuristic: process the passenger groups one after another (in some random order). For each group $k \in K$, compute the shortest alternative path with respect to the perceived travel time, subject to current free capacities. Assign the passengers of this group to the new route, update the capacities, and continue with the next group.

2.4 Path set computation

The set of all s - t -paths can be exponentially large. However, only a small number of them is sufficiently attractive for passengers so that they are actually used. Therefore, instead of working with the full set of paths $P(k)$ for each commodity, we heuristically restrict the search to a carefully selected small set of paths.

To this end, we first compute the Pareto set of shortest paths with respect to three criteria travel time, number of transfers, and some measure of inconvenience. We propose two variants:

1. PARETO1: we use the same perceived travel time function as inconvenience measure as in the greedy approach.
2. PARETO2: we consider a measure which focuses on load. The load of an arc with capacity restrictions (a *capacitated arc*) is defined with the same cost parameters as in the ILP. The load of a path is then defined as the sum of loads on its capacitated arcs.

In PARETO1, the first and third criteria are highly correlated, resulting in relatively small path sets. In contrast, the travel time and load are much less correlated (although the load costs of an arc also depend on its duration), leading to slightly larger Pareto sets. To increase the likelihood of finding feasible paths, we finally add the path set of the greedy approach. As mentioned above, the Pareto sets are pruned in both variants such that only paths with at most six transfers and at most two hours of extra delay are maintained. All other paths are considered as unacceptable.

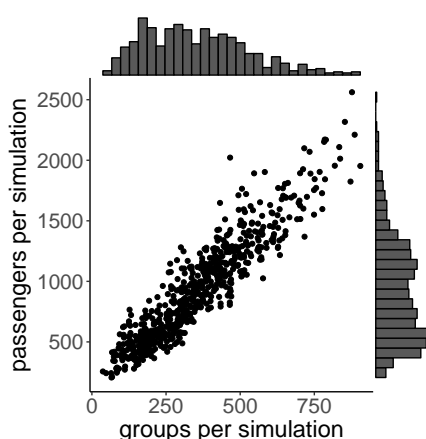
3 Experiments

In this section, we report our findings with experiments on many large-scale test instances.

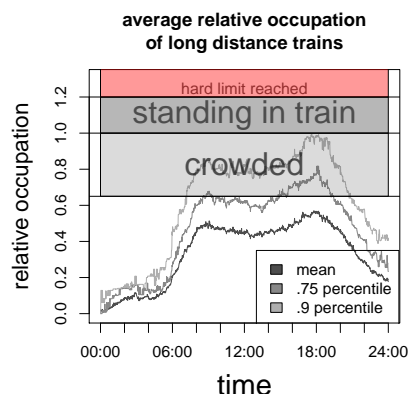
3.1 Test instances, implementation details, and experimental setup

Test instances

Our experiments are based on the timetable of Germany in 2019. For each day, passenger flow data for travelers using long-distance trains are provided by Deutsche Bahn. They are based on sold tickets until the day before and an estimation of short-term ticket buyers. Our capacity data is restricted to that of long-distance trains, for other trains we assume infinite capacity. We selected five test days in April 2019, in the week from Monday 22 to Friday 26. For the runtime measurements we only used the Friday. The average number of passengers in our passenger flow data is about 410,000 passengers (minimum 280,000 and maximum 540,000). In order to create meaningful and relatively hard test instances, we



■ **Figure 1** Number of groups and passengers per instance.



■ **Figure 2** Vehicle load over time of long-distance trains in Germany.

concentrate on train cancellations with many affected passengers and groups which have to be rerouted. To this end, we randomly selected a subset of trains with the property that their passenger load reaches at least 65% of its capacity. Each train cancellation is studied in isolation as an independent test instance. Overall, we have a test set composed of 653 train cancellations. Figure 1 shows the distribution of the number of affected passengers and groups per instance. The average number of affected groups is 351.4, the average number of affected passengers is 922.84.

Specific cost functions and parameters

The following parameter settings are used in all experiments. In our basic model, we use only three different segments for capacitated arcs. Recall that the hard limit u_a is chosen with respect to $\beta = 1.2$, i.e. 120% of the nominal vehicle capacity. The interval from 0 to the hard upper limit u_a is divided as follows:

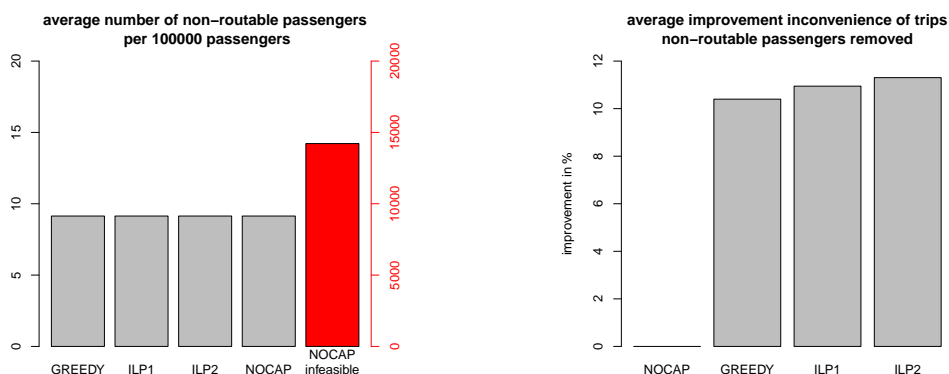
1. The train is currently occupied by at most 65% of its capacity. We set $u_a^1 = 65/120 \cdot u_a$. This is considered as a non-crowding scenario and imposes no crowding penalties, i.e. $\gamma_1 = 0$.
2. The current load is between 65% and 100% capacity. Every passenger finds a seat, but with limited choice. We set $u_a^2 = 100/120 \cdot u_a$. Here we impose a crowding penalty of $\gamma_2 = .2$, i.e. we impose .2 extra minutes per minute of travel time.
3. The available capacity is exceeded, some passengers have to stand. The penalty for standing is $\gamma_3 = 1$, i.e. one extra minute per minute of travel time.

Experimental setup

Our code has been written in C++, it is compiled with gcc 8.3 and run under Arch Linux x86_64 with packages from Mai 2019. All runs are executed on a four core plus hyper-threading Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz machine. Shortest path problems in the event-activity network are computed by an implementation of RAPTOR. ILPs are solved with Gurobi Optimizer 8.1².

² <http://www.gurobi.com>

7:10 Vehicle Capacity-Aware Rerouting



■ **Figure 3** The number of passengers ending up without a valid route. For NOCAP, we have to distinguish passengers with no route, and others with a route which violates hard capacities (last column, right scale).

■ **Figure 4** The mean improvement in inconvenience cost in %.

We compare the following four approaches:

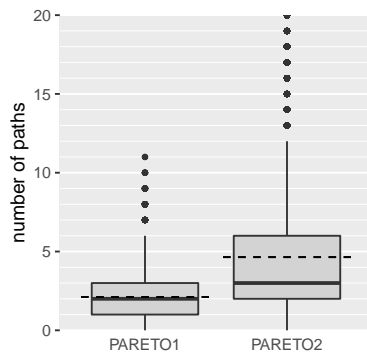
1. **NOCAP**: this refers to the approach of capacity-unaware rerouting. Passengers are simply rerouted to a path with earliest arrival time at their destination. Capacity constraints may be violated.
2. **GREEDY**: passenger groups are rerouted greedily as described in Section 2.3.
3. **ILP1**: this refers to the ILP model where the path set is chosen as PARETO1.
4. **ILP2**: the same ILP model is used but the path set is chosen as PARETO2.

3.2 Experimental Results

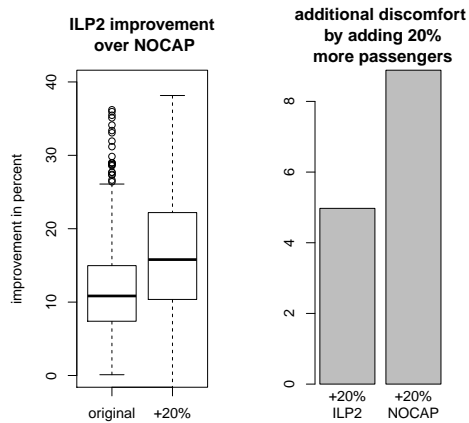
Question 1: How relevant is the consideration of scarce vehicle capacities in rerouting?

Considering the initial load (before rerouting) of long-distance trains, the average vehicle load of 44.71% (within the interval from 6am to 10pm) may falsely suggest that there is probably no severe problem with scarce capacities. However, if we look more carefully into the distribution we observe that the average load changes considerably during the course of the day, see Figure 2. Many of the most crowded trains run in the afternoon hours and close to their maximum seat capacity.

The problem with scarce capacities becomes apparent when we evaluate the traditional capacity-unaware rerouting scheme. Figure 3 shows the number of passengers for which we cannot find a (valid) route. For capacity-unaware rerouting (NOCAP), we have a few cases without any route (column 4), and many more cases, about 15% of all passengers, with an alternative route violating hard train capacities (column 5). In other words, that many affected passengers will be routed into some overfull train whenever capacities are ignored!



■ **Figure 5** Comparison of PARETO1 and PARETO2: Distribution of sizes of path sets.



■ **Figure 6** Comparison of original passenger flows and flows scaled by +20%. Left: The benefit of ILP2 over GREEDY. Right: comparison of additional discomfort ILP2 and NOCAP.

Question 2: How large is the improvement for passengers if we apply capacity-aware routing?

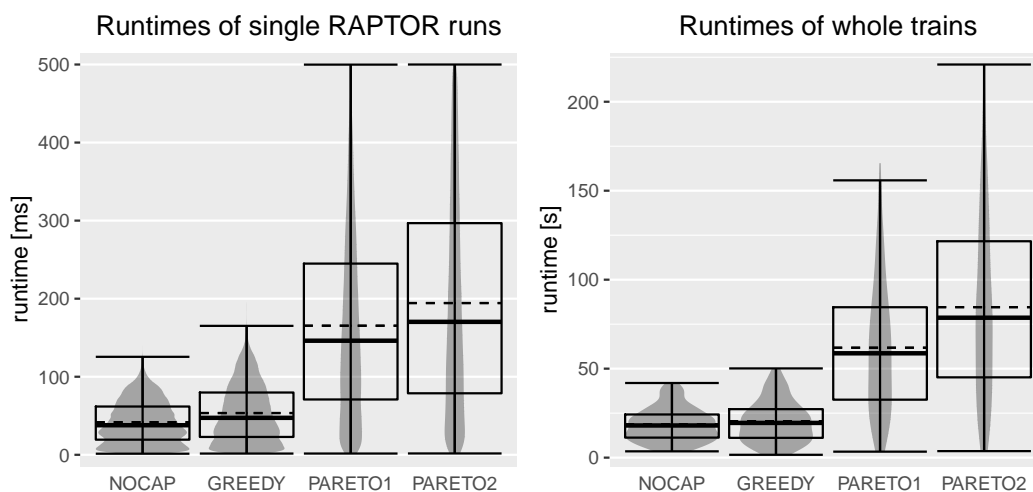
With the capacity-respecting variants, we can strongly reduce the number of passengers without a valid route, see again Figure 3. For less than 0.01% of the passengers no valid route exists (for all four algorithmic variants). Such cases occur for some rare connections without a feasible alternative within the next 24 hours.

We also evaluated the improvement in inconvenience costs over the NOCAP baseline version. Concentrating only on the subset of cases where all variants find valid routes for all passenger groups, we see that GREEDY reduces the total inconvenience costs by more than 10.5% in comparison with NOCAP. Even better mean improvements of about 11% and 11.5% over the baseline NOCAP are obtained with ILP1 and ILP2, respectively, see Figure 4.

The better average quality of ILP2 over ILP1 can be partially explained by the underlying path sets. As Figure 5 shows, the mean size of the PARETO1 path set is smaller than PARETO2, thus leading to fewer rerouting options within ILP1 in comparison with ILP2. Even better solutions can be expected if we further increase the path sets.

Question 3: What happens if passenger numbers increase by 20%?

To answer this question, we take the original passenger flows and scale them up by 20%. Figure 6 shows that the ILP2 solution is continuing to outperform the NOCAP solution on average by 15.0%. The change in total discomfort by passengers, however, is significant. Adding 20% more passengers increases the discomfort experienced in the NOCAP strategy by 9% while the ILP2 model does only produce 5% more discomfort for the same passengers. Thus, we conclude that capacity-aware routing will become more important with rising passenger numbers.



■ **Figure 7** Runtime distributions of single runs of RAPTOR (left) and for whole trains (right) for the four different variants NOCAP, GREEDY, PARETO1, and PARETO2.

Question 4: How efficient are the proposed approaches?

Let us start with some good news: When the underlying path sets are generated, it turns out that solving the ILP resulting instances is very easy for state-of-the-art solver gurobi, since all of them can be solved in few milliseconds.

The expensive part, however, is the computation of candidate paths for all passenger groups. Figure 7 shows runtime distributions as violin plots for single runs of RAPTOR for the four different variants NOCAP, GREEDY, PARETO1 and PARETO2. The mean runtimes are 42ms for NOCAP, 53ms for GREEDY, 166ms for PARETO1, and 228ms for PARETO2. The mean runtime to compute the path sets for all affected groups of a train cancellation is 19s for NOCAP, 20s for GREEDY, 62s for PARETO1 and 85s for PARETO2. We consider such runtimes as feasible for practical use. Improvements of running times are possible by further fine-tuning our implementations.

4 Conclusions and Future Work

In this paper we study the impact of limited vehicle capacities on the rerouting of passengers in case of train cancellations. We propose a convex cost unsplittable flow formulation. First experiments with restricted path sets already show significant improvements over previous, capacity unaware approaches.

In the present work, we solve the unsplittable flow problems with respect to a carefully selected fixed choice of paths. This could be extended by column generation (which amounts to solving a single-criterion shortest path problem for each commodity). The computational bottleneck is the efficient computation of path sets. With respect to our implementation there is certainly room for further improvement for the multi-criteria versions of RAPTOR. While the set of greedy paths has to be computed sequentially, the path set for passenger groups in the multi-criteria setting are independent and can be easily parallized.

We plan to extend our work in several ways. A first natural extension is to study different convex cost functions. Since our focus has been on the most extreme cases (up to several hundred affected passenger groups), a second extension concerns evaluations of real train

cancellations and wait-depart decisions, and similar use cases. Third, we are interested in the price of restricting to unsplittable flows. By how much can we improve solutions if we allow splitting of groups? We could either consider the linear programming relaxation of our ILP models as a lower bound or a closely related classical multi-commodity flow formulation.

References

- 1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. *J. Experimental Algorithmics*, 18:1.3:1.1–1.3:1.17, 2013. doi:10.1145/2444016.2444019.
- 2 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows*. Prentice Hall, Inc., 1993.
- 3 Cynthia Barnhart, Christopher A. Hane, and Pamela H. Vance. Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems. *Oper. Res.*, 48(2):318–326, 2000. doi:10.1287/opre.48.2.318.12378.
- 4 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016. doi:10.1007/978-3-319-49487-6_2.
- 5 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. *Transportation Science*, 49(3):591–604, 2015. doi:10.1287/trsc.2014.0534.
- 6 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, *Experimental Algorithms, SEA 2013*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013. doi:10.1007/978-3-642-38527-8_6.
- 7 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *ACM Journal of Experimental Algorithmics*, 23:1.7:1–1.7:56, 2018. doi:10.1145/3274661.
- 8 Twan Dollevoet and Dennis Huisman. Fast heuristics for delay management with passenger rerouting. *Public Transport*, 6(1-2):67–84, 2014. doi:10.1007/s12469-013-0076-6.
- 9 Twan Dollevoet, Dennis Huisman, Marie Schmidt, and Anita Schöbel. Delay Management with Re-Routing of Passengers. *Transportation Science*, 46(1):74–89, 2012. doi:10.1287/trsc.1110.0375.
- 10 Twan Dollevoet, Dennis Huisman, Marie Schmidt, and Anita Schöbel. Delay propagation and delay management in transportation networks. In Ralf Borndörfer, Torsten Klug, Leonardo Lamorgese, Carlo Mannino, Markus Reuther, and Thomas Schlechte, editors, *Handbook of Optimization in the Railway Industry*, pages 285–317. Springer International Publishing, 2018.
- 11 David Eppstein. Finding the K Shortest Paths. *SIAM J. Comput.*, 28(2):652–673, 1999. doi:10.1137/S0097539795290477.
- 12 Bernard Fortz, Luís Gouveia, and Martim Joyce-Moniz. Models for the piecewise linear unsplittable multicommodity flow problems. *European Journal of Operational Research*, 261(1):30–42, 2017. doi:10.1016/j.ejor.2017.01.051.
- 13 Luke Haywood, Martin Koning, and Guillaume Monchambert. Crowding in public transport: Who cares and why? *Transportation Research Part A: Policy and Practice*, 100:215–227, 2017.
- 14 Zhiyuan Huang, Ruihua Xu, Wei D. Fan, Feng Zhou, and Wei Liu. Service-Oriented Load Balancing Approach to Alleviating Peak-Hour Congestion in a Metro Network Based on Multi-Path Accessibility. *Sustainability*, 11:1293, 2019. doi:10.3390/su11051293.
- 15 Zheng Li and David A. Hensher. Crowding in Public Transport: A review of objective and subjective measures. *Journal of Public Transportation*, 16:107–134, 2013.

7:14 Vehicle Capacity-Aware Rerouting

- 16 Matthias Müller-Hannemann and Ralf Rückert. Dynamic Event-Activity Networks in Public Transportation — Timetable Information and Delay Management. *Datenbank-Spektrum*, 17:131–137, 2017. doi:10.1007/s13222-017-0252-y.
- 17 Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization*, volume 4395 of *Lecture Notes in Computer Science*, pages 67–89. Springer, 2007.
- 18 Adam J. Pel, Nick H. Bel, and Marits Pieters. Including passengers’ response to crowding in the Dutch national train passenger assignment model. *Transportation Research Part A: Policy and Practice*, 66:111–126, 2014. doi:10.1016/j.tra.2014.05.007.
- 19 Ralf Rückert, Martin Lemnian, Christoph Blendinger, Steffen Rechner, and Matthias Müller-Hannemann. PANDA: a software tool for improved train dispatching with focus on passenger flows. *Public Transport*, 9(1):307–324, 2017. doi:10.1007/s12469-016-0140-0.
- 20 Marie Schmidt. Simultaneous optimization of delay management decisions and passenger routes. *Public Transport*, 5:125–147, 2013. doi:10.1007/s12469-013-0069-5.
- 21 Anita Schöbel. A Model for the Delay Management Problem based on Mixed-Integer Programming. *Electronic Notes in Theoretical Computer Science*, 50(1), 2001.
- 22 Anita Schöbel. *Customer-oriented optimization in public transportation*. Springer, Berlin, 2006.
- 23 I-Lin Wang. Multicommodity Network Flows: A Survey, Part II: Solution Methods. *International Journal of Operations Research*, 15:155–173, 2018.

A Priori Search Space Pruning in the Flight Planning Problem

Adam Schienle

Zuse Institute Berlin, Berlin, Germany
schienle@zib.de

Pedro Maristany

Zuse Institute Berlin, Berlin, Germany
maristany@zib.de

Marco Blanco

Lufthansa Systems, Raunheim, Germany
marco.blanco-sandoval@lhsystems.com

Abstract

We study the Flight Planning Problem for a single aircraft, where we look for a minimum cost path in the airway network, a directed graph. Arc evaluation, such as weather computation, is computationally expensive due to non-linear functions, but required for exactness. We propose several pruning methods to thin out the search space for Dijkstra's algorithm before the query commences. We do so by using innate problem characteristics such as an aircraft's tank capacity, lower and upper bounds on the total costs, and in particular, we present a method to reduce the search space even in the presence of regional crossing costs.

We test all pruning methods on real-world instances, and show that incorporating crossing costs into the pruning process can reduce the number of nodes by 90% in our setting.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases time-dependent shortest path problem, crossing costs, flight trajectory optimization, preprocessing, search space

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.8

1 Introduction

With a looming climate change and an ever more connected world, it is imperative that aircraft routes are planned as efficient and efficiently as possible. Contrary to popular belief, aircraft cannot fly directly between their origin and destination airports, but have to adhere to the *Airway Network*, a directed graph. The nodes of the graph are called *waypoints*, whereas the arcs are called (*airway*) *segments*. For vertical separation, aircraft are stacked on 43 *flight levels*, which are mostly spaced 1 000 ft apart.

Airlines plan a flight by computing an optimal route according to their preferences, which may include minimum fuel, minimum time, minimum cost (e.g., in USD), or a combination thereof. Furthermore, one has to consider various overflight charges for countries, weight dependent fuel consumption functions and weather-dependent arc lengths. Due to the weather being time-dependent, this introduces an implicit time-dependency into the problem. On top, some air navigation service providers publish restrictions to prevent congestion, such as EUROCONTROL's *Route Availability Document*[12].

The **Flight Planning Problem** as we will discuss it is the problem of finding a minimum cost (in USD) trajectory given an origin and a destination airport, a departure time and a weather forecast, an aircraft and its consumption functions, and overflight charges for each country. Note that we take the airline's point of view, and only consider one aircraft at a time. A general introduction to this problem can for instance be found in [18]. We will in



© Adam Schienle, Pedro Maristany, and Marco Blanco;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 8; pp. 8:1–8:14

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the course of this paper disregard restrictions imposed by air navigation service providers (ANSPs), as they can usually only be enforced on a given route. In particular, the Flight Planning Problem as we see it is a base problem which may have to be solved a number of times in order to obtain a valid flight plan.

In practice, a flight plan is compiled a few hours before the aircraft takes off. It is then filed with ANSPs such as EUROCONTROL, who either accept or reject it. If rejected, it must be recomputed to comply with additional restrictions; if accepted, however, it must be flown as is, which requires the flight plan to be optimal – and discourages approximative algorithms. Since flight plans may have to be recomputed following a rejection by ANSPs, and since the base problem may have to be re-solved in order to obtain a solution which satisfies restrictions, it is necessary that the base problem can be solved fast.

In this paper, we aim to reduce the a priori search space before the query commences. To do so, we use a combination of upper and lower bounds in order to remove nodes from the graph which provably cannot lie on an optimal path. Since many countries (especially in Europe) allow more and more direct connections (so-called Free Route Areas) between any two nodes within their boundaries, the underlying graph tends to become denser in the future, which negatively affects runtimes.

1.1 Literature and Related Work

Reliant on a directed graph, the Flight Planning Problem shares similarities with the (Time-Dependent) Shortest Path Problem on road networks. The best-known algorithm to solve the non time-dependent version is Dijkstra’s Algorithm[8], which can be extended for the time-dependent setting, see [10]. However, the time-dependent version of Dijkstra’s Algorithm is only guaranteed to find an optimal solution if the FIFO property is satisfied; we say that a function $f: A \times \mathbb{R}_0^+$ satisfies the FIFO property, if

$$\tau < \tau' \Rightarrow f(a, \tau) \leq f(a, \tau') + (\tau' - \tau).$$

This property basically states that overtaking is not possible by waiting at nodes. Many speedup techniques for Dijkstra’s Algorithm exist for both the static and the time-dependent version; [1] provides a good overview. While some such as A*[16] use potential functions to guide the query at runtime, the methods which are most effective on road networks require one or more preprocessing step. Contraction Hierarchies [15] (CHs) and its time-dependent sibling Time-Dependent Contraction Hierarchies [2] (TCHs) assign ranks to the nodes, and look for a shortest path through an in- and then decreasing sequence of nodes.

In [6], the authors show that TCHs do not perform as effectively on the airway network graph as on road networks, being dominated both in preprocessing and in query times by A*. The authors also introduce a novel technique for underestimating traversal times in the flight planning problem. However, they do not consider overflight charges, which are a central component of this paper.

Jensen et al. introduce a geometric algorithm to solve the Free Flight Problem [17]. They partition any free flight airspace into rectangles of equal, constant wind. While similar to Dijkstra’s algorithm, their method sorts nodes based not on their costs, but on the costs of their cheapest successor, and lets nodes compete among each other for their successors, thus achieving a speedup over Dijkstra’s Algorithm or A*. One has to note, however, that their approach assumes airspace users can choose their waypoints freely. On the other hand, we will assume that all waypoints and segments are defined, and that one is not allowed to fly via self-defined waypoints. This represents the current point of view as expressed by air navigation service providers such as EUROCONTROL [11].

In terms of overflight charges or crossing costs, relatively little research has been conducted. Most notably, Blanco et al. define and analyse the Shortest Path Problem with Crossing Costs in [7], and solve a special case to optimality. In [5], the authors furthermore propose a cost projection onto the arcs. Dreves and Gerdtts [9] give an example on how to solve the problem using optimal control, albeit in a bounded region in Europe. None of the works cited deal with overflight charges considers the influence of weather. Both [6] and [5] only run queries on one layer of the airway network, whereas we use the full (3D) graph for our computations.

We seek to cut nodes with several different methods from the a-priori search space, pruning all those nodes which provably cannot lie on an optimal path. A similar technique is used in other applications of shortest path problems, e.g. in Electric Vehicle Routing [4], or in the algorithm Approximated Time-Dependent Contraction Hierarchies [3].

Crossing costs pose a particular problem, in that they do not always correlate to the arcs or nodes on the path, but rather depend on geometric information given by entry and exit points for the given regions. While all of the presented pruning methods bear similarities to A^* , to the best of our knowledge, there is no known underestimator for regional crossing costs of this particular type.

In section 2, we develop the theory on how to prune nodes prior to the query. Section 3 shows how to model the Free Flight Problem as a Time-Dependent Shortest Path Problem with Crossing Costs. We develop several different pruning methods for the Free Flight Problem in section 4, and show the results of real-world test cases in section 5.

2 Pruning Search Spaces in the Time-Dependent Shortest Path Problem

In this section, we consider the Time-Dependent Shortest Path Problem (TDSPP) defined as follows: We are given a graph $G = (V, A)$ together with a travel time function $T: A \times \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$, which maps an arc a and the time τ at which it is entered to the **travel time** $T(a, \tau)$ needed to traverse a . This function T will be one constituent of the total cost. For a path $P = (v_0, \dots, v_n)$ between two nodes v_0, v_n and a departure time τ_0 , we define the travel time for P as

$$T(P, \tau_0) := \sum_{i=0}^{n-1} T((v_i, v_{i+1}), \tau_i),$$

where $\tau_{i+1} = \tau_i + T((v_i, v_{i+1}), \tau_i)$ for every $i \in \{0, \dots, n-1\}$. We further impose that two nodes $s, t \in V$ exist, and seek to find the shortest path from s to t with respect to T , starting at time τ_0 .

In [6], the authors show that in flight planning, A^* outperforms even the most promising speedup technique to Dijkstra's Algorithm, Time-Dependent Contraction Hierarchies[2]. Since our pruning techniques in section 4 are similar to A^* , we limit ourselves to the discussion of pruning the search space for Dijkstra's algorithm.

In this section, we concentrate on how to prune the search space for Dijkstra's algorithm before the query begins. To this end, we use both lower bounds on the arc costs as well as an upper bound on the route costs. Usually, computing an upper bound on the route costs is easy by just computing any feasible solution.

Let P^* be a minimum cost path s - t path starting at τ_0 , and $c^* := c(P^*)$ its cost. We assume that we are given an upper bound \bar{c} on c^* , e.g., through a previously computed feasible solution.

8:4 Pruning Search Spaces

► **Theorem 1.** *Assume we are given a TDSPP as defined above, and an upper bound \bar{c} on the costs for a shortest path between s and t . Let $\underline{c}_s^t: V \rightarrow \mathbb{R}_0^+$ be a function which underestimates the costs for a shortest s - t -path via v . Any node $v \in V$ which violates the pruning inequality*

$$\underline{c}_s^t(v) \leq \bar{c}$$

cannot lie on an optimal path.

Proof. Let P^* be an optimal path, and assume that $v \in P^*$ violates the pruning inequality, so there exists a shortest s - t -path P_v via v such that $\underline{c}(P_v) > \bar{c}$. Then we find

$$c^* = c(P^*) \geq \underline{c}(P_v) > \bar{c},$$

contradicting the assumption that P^* was optimal to begin with. ◀

► **Remark 2.** In practice, we obtain a function $\underline{c}_s^t(v)$ as required in the above theorem by computing a lower bound function $\underline{c}: A \rightarrow \mathbb{R}_0^+$ such that

$$\underline{c}(a) \leq c(a, \tau) \quad \forall \tau \in \mathbb{R}_0^+.$$

This approach is common for shortest path problems (e.g. for A* or ALT, see [1]), and is also used in [6]. We write $\underline{G} := (G, \underline{c})$, and run a one-to-all Dijkstra from s and an all-to-one Dijkstra to t on \underline{G} . Note that \underline{G} carries static arc costs, hence we can grow the Dijkstra trees in $\mathcal{O}(|A| + |V| \log |V|)$.

If any node $v \in V$ is not contained in either of the two trees, it cannot be reached from s , or cannot reach t . In both cases, it can safely be eliminated, as it cannot lie on any optimal path. If a node $v \in V$ is not contained in both the forward and the backward tree, we will assume its respective costs to be ∞ .

Theorem 1 states that any path whose lower-bound costs are already higher than a pre-computed upper bound cannot be optimal. Note that this mimics the A* algorithm[16]. The key difference is that by using Theorem 1, nodes which cannot lie on an optimal path are eliminated a priori, rather than being discarded during the search. This means that it is a little weaker than A*: for any node $v \in V$, the latter adds an estimate of the remaining costs from v to t to the actual costs $c(P_s^v)$ from s to v , and sorts the node in the heap accordingly. Contrastingly, when applying Theorem 1, one adds two cost estimates and compares them to an upper bound. Unless the estimate is perfect, this leads to a gap between $\underline{c}(P_s^v)$ and $c(P_s^v)$. However, in practice it might be easier to compute an underestimation for the costs of a complete path, rather than just parts of it. Also, a pre-pruned search space can be advantageous for search algorithms which do not maintain a heap structure; e.g., by sorting the graph's nodes in topological order and run a search algorithm exploring the resulting node groups in their respective order.

► **Remark 3.** The applicability of Theorem 1 is dependent on both the quality of the lower bound as well as the quality of the upper bound solution. Clearly, lowering the upper bound will result in more nodes being eliminated, as will raising the lower bound.

As shown by Fredman et al.[14], the runtime for the static Dijkstra's Algorithm is $\mathcal{O}(|A| + |V| \log |V|)$ when a Fibonacci heap is used to store the unprocessed labels. The dynamic, i.e., time-dependent case with FIFO travel time functions can be solved using almost the same version of the algorithm [10]. The only difference is the evaluation of the arcs' cost: in the static case it can be in constant time but in the dynamic case the complexity of the evaluation depends on the shape of the functions [13]. This is why, even in the FIFO case, it is not guaranteed that the TDSPP is polynomially solvable and this is also the motivation for a restriction of the search space before the query commences.

3 Modelling the Free Flight Problem

We model the Free Flight Problem as a **Time-Dependent Shortest Path Problem with Crossing Costs**, or T-SPROC for short. As the name suggests, we introduce crossings costs to the Time-Dependent Shortest Path Problem to account for overflight charges. We proceed as follows:

We consider the airway network as directed graph $G = (V, A)$, and origin and destination airport as distinct nodes s and t in V . Each flight starts at a departure time $\tau_0 \in \mathbb{R}_0^+$. In our application, the cost functions comprises three components, to wit, the *fuel costs*, the *time costs*, and *overflight costs*. Fuel costs are defined by what an aircraft burns en route, while overflight costs are charges raised by countries' air navigation service providers. Time costs, on the other hand, comprise leasing costs, crew costs, and maintenance costs, and can in our case be considered a linear function of the time en route. Hence, we can think of time costs as being charged per arc, and introduce the *time cost function*

$$\begin{aligned} c_t: A \times \mathbb{R}_0^+ &\rightarrow \mathbb{R}_0^+ \\ (a, \tau) &\mapsto \alpha \cdot T(a, \tau). \end{aligned}$$

Fuel costs depend linearly on how much fuel an aircraft burns en route. The fuel burn is directly proportional to the distance relative to the surrounding air mass, or *air distance*. As in [17, 6], we assume that an aircraft flies with constant speed, which in turn renders the air distance proportional to the time en route. Therefore, we can again think of fuel costs as being charged per arc, and write

$$\begin{aligned} c_f: A \times \mathbb{R}_0^+ &\rightarrow \mathbb{R}_0^+ \\ (a, \tau) &\mapsto \beta \cdot T(a, \tau). \end{aligned}$$

Both fuel and time costs naturally depend on the travel time, which in turn is weather-dependent. We use the same travel time functions as given [6], assuming that weather is given for a discrete point set $\Delta \subset \mathbb{R}_0^+$ in time over a long enough interval to cover all flight durations. Usually, the break points are three hours apart; for any $\tau \notin \Delta$, we interpolate the closest two weather data objects to obtain a wind vector $\mathbf{w}(a, \tau)$ for an arc $a \in A$. It can be decomposed into its cross wind component $w_c(a, \tau)$ perpendicular to the direction of flight, and a track wind component $w_t(a, \tau)$ parallel to it. Together with the constant air speed, this leads to the travel time (c.f. [6, 17])

$$T(a, \tau) = \frac{\ell(a)}{\sqrt{v^2 - w_c^2(a, \tau) + w_t(a, \tau)}},$$

with $\ell(a)$ denoting the length over ground of the arc $a \in A$. In particular, our travel time is a non-linear function in prevailing the wind conditions.

Since both fuel and time costs are defined arc wise, we aggregate both functions into a single arc-based and time-dependent cost function $c^A: A \times \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$, defined as

$$\begin{aligned} c^A: A \times \mathbb{R}_0^+ &\rightarrow \mathbb{R}_0^+ \\ (a, \tau) &\mapsto c_f(a, \tau) + c_t(a, \tau). \end{aligned}$$

8:6 Pruning Search Spaces

For the definition of crossing costs for regions, we will closely follow the notation presented by Blanco et al. in [7]. We write $\delta^+(v)$ for out-arcs and $\delta^-(v)$ for in-arcs of the node v , and define $\delta(v) := \delta^+(v) \sqcup \delta^-(v)$. We assume that the arcs are partitioned into a set \mathcal{R} of k regions

$$A = R_1 \sqcup R_2 \sqcup \dots \sqcup R_k,$$

and we call $v \in V$ an **inner node** of R_i if $a \in R_i$ for all $a \in \delta(v)$. Stretching notational limits, we will also write $v \in R_i$ for an inner node v of R_i .

If, conversely, $a \notin R_i$ for all $a \in \delta(v)$, we call v an outer node. All nodes which are neither inner nor outer nodes are called **boundary nodes**, and we write $v \in \partial R$. We count airport nodes as boundary nodes. We emphasise that regions must not overlap arc-wise, yet they may share a common boundary. Without loss of generality, arcs do not cross more than one region: if an arc $a \in A$ did, we could subdivide it and insert a new boundary node at the border.

Write $t(P), h(P)$ for the first (or *tail*) and last (or *head*) node of a path P , and let $\mathcal{S}_R(P)$ denote the set of arc-maximal sub-paths of P in a region $R \in \mathcal{R}$. Then, for a sub-path $p \in \mathcal{S}_R(P)$, its tail $t(p)$ and head $h(p)$ are both elements of ∂R . We will denote the union of all boundary nodes by

$$\begin{aligned} \mathcal{B} &:= \{b \in V : b \in \partial R \text{ for some } R \in \mathcal{R}\} \cup \{v \in V : v \text{ is an airport}\} \\ &= \bigcup_{R \in \mathcal{R}} \partial R \cup \{v \in V : v \text{ is an airport}\}. \end{aligned}$$

Assume a metric $d: V \times V \rightarrow \mathbb{R}_0^+$. In our application, the natural metric arising from embedding $G = (V, A)$ on a spherical earth model is the great circle distance (gcd). We write \mathcal{P}_s^t for the set of all s - t paths. For a non-decreasing function $f_R: \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ we can now define the **crossing costs** $c_o^R: \mathcal{P}_s^t \rightarrow \mathbb{R}_0^+$ for a region R and an s - t -path P as

$$c_o^R(P) := \begin{cases} f_R \left(\sum_{p \in \mathcal{S}_R(P)} d(t(p), h(p)) \right) & \text{if } R \cap P \neq \emptyset, \\ 0 & \text{if } R \cap P = \emptyset. \end{cases}$$

Note that these costs do not rely on the time τ_0 at all. We can now define the Time-Dependent Shortest Path Problem with Crossing Costs (for short: T-SPROC) as follows:

Input: A directed graph $G = (V, A)$, nodes $s, t \in V$, a departure time $\tau_0 \in \mathbb{R}_0^+$, an arc-based cost function $c^A: A \times \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$, and a crossing cost function $c_o: \mathcal{P}_s^t \rightarrow \mathbb{R}_0^+$ as defined above.

Objective: Find an s - t -path P starting at τ_0 which minimises

$$c(P, \tau_0) := \sum_{i=0}^{n-1} c^A((v_i, v_{i+1}), \tau_i) + \sum_{R \in \mathcal{R}} c_o^R(P). \quad (1)$$

► **Remark 4.** T-SPROC is an extension of TDSPP. Clearly, the first sum constitutes the time-dependency, whereas the second one accounts for the crossing costs. Especially, if $c_o^R(\cdot) \equiv 0$, we re-obtain the TDSPP. Also, note that while the crossing costs c_o are not time-dependent, they are not defined per arc. This obvious difference to TDSPP poses an additional challenge.

We have shown in [19] that under certain conditions, the wind functions in our application satisfy the FIFO property; we will for the remainder of this paper presuppose the FIFO property for c^A . Blanco et al. developed the Two-Layer-Dijkstra algorithm in [7], which solves the shortest path problem with crossing costs to optimality in polynomial time.

We also observe that Theorem 1 requires an underestimation function for all nodes $v \in V$. Since we cannot even evaluate the crossing costs for non-boundary nodes $v \in V \setminus \mathcal{B}$, we cannot apply Theorem 1 to T-SPROC directly. However, since overflight charges are always non-negative, we can underestimate the function c_o by zero.

4 Preprocessing in Practice

In order to have as low runtimes as possible, we aim to prune the a-priori search space for the Flight Planning Problem. Regardless of which pruning algorithm we choose, the objective function will always be given by (1).

4.1 Dead-End elimination

We can pre-eliminate any nodes which either cannot reach t or cannot be reached from s . This can be done in $\mathcal{O}(|V| + |A|)$, as it suffices to run one forward breadth-first search from s and one backward breadth-first search from t . This pruning method is plainly graph-theoretical, and removes cul-de-sac nodes from the graph. We will compare all other pruning methods to this baseline both in terms of runtime and in terms of nodes in the search space.

4.2 The Tank-Capacity Pruning

This pruning method is the most intuitive one. Aircraft clearly cannot burn more fuel than they can carry with them; since fuel burn is proportional to the flight time, this means that there is an inherent maximum flight time for aircraft based on their tank capacity. Let $\bar{\Phi}$ denote the maximum fuel which the aircraft can carry, and $\varphi_s^t(v)$ the fuel consumption on a shortest path from s to t via v .

To underestimate $\varphi_s^t(v)$ for a node $v \in V$, we use the Super-Optimal Wind as introduced by Blanco et al. in [6]. The Super-Optimal Wind for an arc $a \in A$ is an artificial wind vector which never underestimates the travel time for an arc $a \in A$ arising out of an actual wind conditions. Given $a \in A$, it is obtained by separately minimising the cross wind and maximising the track wind, leading to the artificial wind vector

$$\mathbf{w}(a) = (\underline{w}_c(a), \bar{w}_t(a)),$$

where $\underline{w}_c(a) := \min_{\tau \in \mathbb{R}_0^+} |w_c(a, \tau)|$ and $\bar{w}_t(a) := \max_{\tau \in \mathbb{R}_0^+} w_t(a, \tau)$. Note that one can obtain better lower bounds by computing the Super-Optimal Wind $\mathbf{w}_i(a)$ per $\tau_i, \tau_{i+1} \in \Delta$. For a full discussion of the topic, we point the reader to [6]. As stated in the same work, Super-Optimal Wind can be pre-computed independent of the instance in a few seconds for all arcs $a \in A$.

We create a lower-bound graph (G, \underline{l}) , by computing the Super-Optimal Wind vector for each arc $a \in A$. This wind corresponds to a **minimum air distance** $\underline{l}_i(a)$ for each arc $a \in A$ and a given wind prognosis time $\tau_i \in \Delta$. Given an upper bound on the travel time (e.g., through a previously computed solution), we can determine $\tau_{k_0}, \tau_{k_j} \in \Delta$ such that the entire flight is in $[\tau_{k_0}, \tau_{k_j}]$. Hence, setting

$$\underline{l}(a) := \min_{i \in \{k_0, \dots, k_j\}} \underline{l}_i(a)$$

provides an effective lower bound on the air distance for the entire flight. We then run a one-to-all-Dijkstra from s and an all-to-one-Dijkstra to t on (G, \underline{l}) . Then, for any node $v \in V$, we obtain the minimum distance $\underline{l}_s^t(v)$ from s to t via v by setting

$$\underline{l}_s^t(v) := \underline{l}(s, v) + \underline{l}(v, t),$$

where $\underline{l}(u, w)$ denotes the length of a shortest path from u to w in G , with respect to \underline{l} . This value is a lower bound on the wind-corrected distance between s and t . We convert $\underline{l}_s^t(v)$ to fuel consumption by assuming an optimal flight profile on the given air distance, to obtain a lower bound $\underline{\varphi}_s^t(v)$ on the fuel consumption via v . Whenever $\underline{\varphi}_s^t(v)$ exceeds the tank capacity $\bar{\Phi}$, we can eliminate v from the search space. Note that while this is similar to Theorem 1, we do not rely on a precomputed upper bound; rather, the tank capacity is implicit in the input data. Since all arc costs are static, we can compute this lower bound for every node in $\mathcal{O}(|A| + |V| \log |V|)$.

4.3 Fuel and Time Pruning

Since both fuel and time costs are defined arc-wise, it makes sense to use both to create an arc-based underestimation – we will in this subsection underestimate crossing costs by zero.

Underestimations for time costs are easier to obtain than for fuel costs. We again make use of the Super-Optimal Wind: by employing the same strategy as above, we can obtain a lower bound $\underline{T}_s^t(v)$ on the travel time between s and t via any node v . As it turns out, both computations can be done in a single step by using that air distance and travel time are proportional via the constant air speed. Since we assume time costs $c_t(P)$ for a path P to be a linear function in the travel time, they are very easy to underestimate: in fact, the costs for the underestimated travel time are a good underestimation of the actual time costs. We define

$$\underline{c}_t(v) := c_t(\underline{T}(s, v) + \underline{T}(v, t))$$

Hence, given an upper bound solution with cost \bar{c} , the pruning inequality given in Theorem 1 evaluates to

$$\underline{c}_f(v) + \underline{c}_t(v) =: \underline{c}^A(v) \leq \bar{c}, \quad (2)$$

and we can eliminate any node $v \in V$ which violates it. This is essentially the application of Theorem 1 to the time-dependent part of T-SPROC, with crossing costs underestimated by the constant zero function. As in the tank capacity case, all data is static. In particular, we can compute this lower bound at the cost of Dijkstras, namely in $\mathcal{O}(|A| + |V| \log |V|)$.

4.4 Pruning Crossing Costs

The problem with the method presented in the previous section is that although we use the upper bound cost comprising crossing costs, we only sensibly underestimate the fuel and time components (i.e., the time-dependent part of T-SPROC). The crossing costs are underestimated by zero. While still a valid underestimation, overflight charges may account for up to one fifth of the total route costs, depending on the aircraft type. This already justifies the incorporation of these charges into our underestimation.

Overflight charges were already investigated in [5], where the authors project crossing costs for regions on the arcs using a heuristic which works very well in practice, but cannot guarantee an underestimation. Instead, we are going to pursue an exact solution. In [7], Blanco et al. introduce a macro graph which they use to keep track of the overflight costs. We will mimic this construction, and use the macro graph to underestimate the overflight costs.

So far, all cost components could be computed as a sum of individual arc costs. Recall that as per the definition, crossing costs are not defined per arc, but are only given at the boundaries of regions. Hence, sensible pruning can only occur at these boundaries. The idea is to eliminate as many boundary nodes as possible, and then prune the search space further by running an additional fuel/time pruning on the reduced search space.

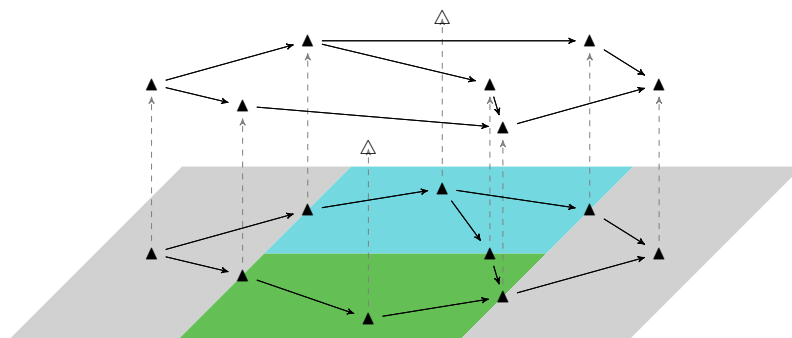
Since crossing costs are not time-dependent, we can precompute a lower bound on them by constructing a new graph $M = (V', A')$. We define this **macro graph** M as in [7]:

$$V' := \{v \in V : v \in \partial R \cap \partial R' \text{ for some } R, R' \in \mathcal{R}\} \cup \{v \in V : v \text{ is an airport}\},$$

i.e., the new nodes are all boundary nodes of the regions. We also count airport nodes as boundary nodes, to allow for crossing costs for flights beginning or ending in the interior of a region. While instead of all airports it would suffice to only add s and t (as done in [7]), our more general definition has the advantage of being independent of the instance. We can hence reuse the same graph structure for all flight instances.¹ We then set

$$A' := \{a = (u, v) \in 2^{V'} : \exists P = (u, n_1, \dots, n_{k-1}, v) \text{ such that } u, v, n_i \in R \forall i\}.$$

In other words, we insert an arc between two boundary nodes of R whenever there is a path connecting them which is entirely contained in R . We then endow M with the (not time-dependent) metric function d as defined in section 3. The macro graph for a set of airspaces is depicted in Figure 1.



■ **Figure 1** Macro graph with two airports in the gray regions.

Note that for all boundary nodes $b \in \mathcal{B}$, the value

$$\underline{c}_o(b) := c_o(s, b) + c_o(b, t)$$

is not only a lower bound on the crossing costs, but the actual crossing costs $c_o(b)$ via b . In particular, by running a one-to-all Dijkstra from s and an all-to-one Dijkstra to t , we can obtain the actual crossing costs from s to t via each $b \in \mathcal{B}$. By construction, $|A'| \in \mathcal{O}(|V'|^2)$, which means that running the Dijkstra algorithms takes at most $\mathcal{O}(|V'|^2)$.

We observe that a route which minimises the crossing costs need not be optimal in terms of the total costs – the minimum crossing costs c_o^* between s and t are always a lower bound on the actual crossing costs $c_o(P)$ for any s - t -path P . This means that we can safely underestimate the crossing costs by c_o^* instead of zero, without losing optimality in the ensuing query – thus raising the lower bound by a significant amount. This leads to the following procedure:

¹ While our definition is very similar to the one in [7], the authors use it to solve the shortest path problem with crossing costs to optimality, whereas we use it to prune the search space.

8:10 Pruning Search Spaces

1. Deactivate all boundary nodes $b \in \mathcal{B}$ for which $\bar{c} < \underline{c}^A(b) + \underline{c}_o(b)$. Just as for bidirectional Dijkstra[1], observe that whenever the fringes of both the one-to-all tree and the all-to-one tree meet at a node b , we obtain a candidate $c_o(b)$ for the minimum overflight costs between s and t . The total minimum overflight costs

$$c_o^* = \min_{b \in \mathcal{B}} c_o(b)$$

are therefore a natural byproduct of this step.

2. Lower \bar{c} to $\bar{c}' := \bar{c} - c_o^*$ (this is equivalent to raising the lower bound by c_o^*).
3. Run fuel/time pruning on the reduced search space with the upper bound \bar{c}' .

Through this procedure, we use the influence of overflight charges twice: first in actively removing boundary nodes, second by lowering the upper bound for the ensuing fuel/time pruning.

4.5 Pruning in Practice

The airway network is designed as a directed graph on the Earth's surface and the flight levels can be thought of as distinct, interconnected layers of this graph. To quicken the preprocessing step, we only consider the base layer of the airway network: we set an arc's underestimated length to the minimum lower bound length over all flight levels on which the arc is defined. This may weaken the effect of the pruning algorithms, but does not affect correctness. We use this procedure in our computations.

5 Computational Results

Both the Airway Network and the instance data were provided by Lufthansa Systems. The graph consists of 109 314 nodes and 838 114 arcs per level, on 43 such flight levels. The instance set consists of the 7 735 most often flown international connections, based on week 21 of 2014.² We limit ourselves to international relations only, since for some countries such as the US, overflight charges do not apply for domestic flights.

All considered flights connect cities which are at least 1 000 km apart on the great circle between the two cities; due to the structure of the Airway Network, instances with airports closer to each other do not benefit much from nuances in pruning algorithms.

We implemented all algorithms explained in Section 4 in C++ within our flight planning tool. We compiled the code with GCC, and all our tests were carried out on machines with 132GB of RAM, and an Intel(R) Xeon(R) CPU E5-2690 v4 processor with 2.60GHz and 35.8MB of cache. Queries were run in single-thread mode.

We will measure the quality of the pruning methods by comparing runtimes and the number of active nodes before pruning to afterwards both absolutely and relative to dead-end elimination, which will act as our baseline. A higher number always indicates a more effective method. The advantage of counting active nodes is that the speedup is purely algorithm- but not implementation-dependent. For a fixed instance, we will keep the same upper bound solution for each underestimator for all pruning algorithms. The names for the different pruning methods found in this section are listed in Table 1.

² single trip – only one direction is represented in the instance set.

■ **Table 1** Pruning methods.

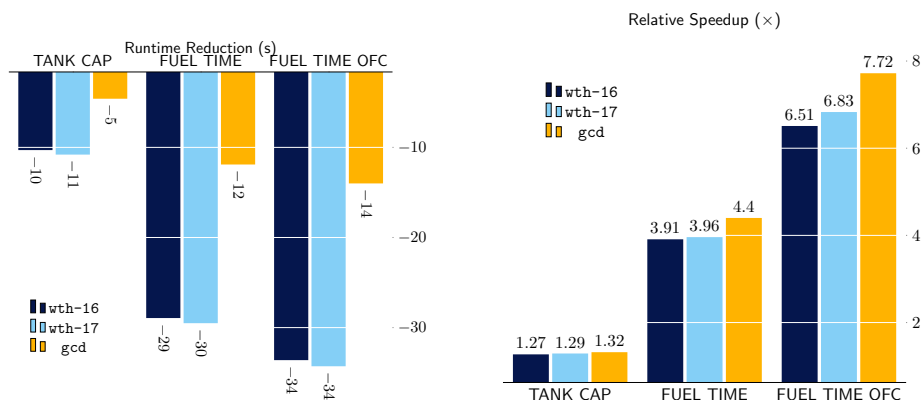
DEAD END	dead end elimination
TANK CAP	tank capacity pruning
FUEL TIME	fuel/time pruning
FUEL TIME OFC	fuel/time/overflight costs pruning

We investigate three different scenarios: we use two different weather prognoses, six months apart (`wth-16` and `wth-17`), and in a third case investigate the situation where there is no wind at all. This `gcd` case highlights the potential for FUEL TIME OFC pruning while eliminating unpredictable effects introduced by the wind. The absolute runtimes for each scenario, averaged over all 7735 connections, are presented in Table 2.

■ **Table 2** Average Query Times per Weather.

Weather	DEAD END		TANK CAP		FUEL TIME		FUEL TIME OFC	
	runtime (s)	nodes (#)	runtime (s)	nodes (#)	runtime (s)	nodes (#)	runtime (s)	nodes (#)
wth-16	44.01	31 115	33.70	9 213	15.05	4 125	10.38	3 042
wth-17	44.61	31 120	33.61	9 151	14.89	4 057	10.11	2 963
gcd	17.42	31 123	12.81	8 673	5.50	3 541	3.41	2 391

Recall that we investigate the base problem in flight planning, that of finding a 3D trajectory. Note, too, that we may have to recompute a solution given new restrictions imposed by ANSPs. With this in mind, it is imperative that each instance can be computed as fast as possible. To this end, we compare the query time of Dijkstra’s Algorithm after applying TANK CAP, FUEL TIME, and FUEL TIME OFC pruning to the query time after using only DEAD END. For each of the more than 7 000 flights, the speedup is recorded both absolutely and relatively, averaged over the instances, and then summarised for the current weather situation. Note that this approach is possible since our runtimes are in the order of seconds rather than milliseconds, which yields comparatively stable runtime measurements. The previous table already indicates the results, but we also visualise them in Figure 2.



■ **Figure 2** Absolute runtime reduction (left) and averaged relative speedup per instance (right), visualised per test set.

8:12 Pruning Search Spaces

One can see that TANK CAP pruning yields a speedup factor close to 1.3, FUEL TIME pruning a factor of almost 4, and FUEL TIME OFC pruning a relative speedup of just under 7 for the weather-dependent instance sets.

To highlight the quality of the respective pruning algorithms, we also recorded the absolute and relative reduction of nodes in the search space, which is only dependent on the pruning method but not on its implementation. The results are presented in Table 3. Recall that even though our computations took place in the full 3D setting, our pruning methods work on the projection of the graph onto a 2D layer. Hence, it makes sense measure the reduction of the search space in terms of deactivated 2D nodes.

■ **Table 3** Average 2D Search Space Reduction Per Instance.

Weather	TANK CAP		FUEL TIME		FUEL TIME OFC	
	absolute	%	absolute	%	absolute	%
wth-16	21 900	70.84	26 988	87.40	28 071	90.94
wth-17	21 969	71.03	27 063	87.60	28 157	91.18
gcd	22 448	72.45	27 581	89.11	28 731	92.85

As one would expect, FUEL TIME OFC pruning is the most effective method. Applying FUEL TIME OFC pruning to the search space yields a reduction of more than 90% of the active nodes. This reduction is also visible in terms of the average query time speedup, which is even ≈ 1.72 times better than with the second best method, FUEL TIME pruning. While the TANK CAP pruning is not quite as effective as the other methods, its inherent advantage is that it does not rely on an upper bound solution. Indeed, the upper bound is given by the input in terms of the aircraft’s tank capacity, which renders it a computationally light alternative – or a fallback method in case one cannot find a reasonable upper bound solution.

It becomes apparent that all pruning methods are more effective in the gcd case than with weather. This is logical since this case can be thought of as zero wind, whose Super-Optimal Wind underestimation is perfect – thus tightening the bounds on both fuel and time underestimation. Consequently, the influence of including the overflight fees in the estimation become more apparent. Therefore, it is also not surprising that FUEL TIME OFC pruning in the gcd case is the most effective of all methods, since it deals with the tightest lower and upper bounds possible.

6 Conclusion

We have investigated the Flight Planning Problem in more detail than what was covered before. To speed up the query, we have developed and presented three different pruning methods for an a priori search space reduction. In particular, we presented a way to incorporate crossing costs in the underestimation, thus tightening the lower bounds on the optimal costs for the Flight Planning Problem.

We showed both theoretically and computationally that each of the methods is effective. Clearly, including crossing costs in the underestimation yields a noticeable reduction of both the search space and the ensuing query time. While all pruning methods bear similarities to A^* , to the best of our knowledge, there is no known underestimator for this particular problem. This is partly due to the fact that A^* requires that one define a potential function at each node. Contrastingly, we use a two-stage pruning process to first eliminate boundary nodes and then deactivate inner nodes.

The added benefit of a priori search space reduction is that it can be used with non-heap-based algorithms, such as topological sorting, too.

References

- 1 Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. Technical report, Microsoft Research, 2015.
- 2 G. Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-dependent Travel Times with Contraction Hierarchies. *J. Exp. Algorithmics*, 18:1.4:1.1–1.4:1.43, April 2013.
- 3 Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Paola Festa, editor, *Experimental Algorithms*, pages 166–177, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 4 Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner. Speed-Consumption Tradeoff for Electric Vehicle Route Planning. In Stefan Funke and Matúš Mihalák, editors, *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 42 of *OpenAccess Series in Informatics (OASIS)*, pages 138–151, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIScs.ATMOS.2014.138.
- 5 Marco Blanco, Ralf Borndörfer, Nam Dung Hoang, Anton Kaier, Pedro Maristany de las Casas, Thomas Schlechte, and Swen Schlobach. Cost Projection Methods for the Shortest Path Problem with Crossing Costs. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, 2017.
- 6 Marco Blanco, Ralf Borndörfer, Nam Dung Hoang, Anton Kaier, Adam Schienle, Thomas Schlechte, and Swen Schlobach. Solving Time Dependent Shortest Path Problems on Airway Networks Using Super-Optimal Wind. In *16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2016)*, 2016.
- 7 Marco Blanco, Ralf Borndörfer, Nam Dung Hoang, Anton Kaier, Thomas Schlechte, and Swen Schlobach. The Shortest Path Problem with Crossing Costs. Technical report, ZIB, 2016.
- 8 Edsger W. Dijkstra. *Numerische Mathematik*, chapter A Note on Two Problems in Connexion with Graphs, pages 269–271. Springer, 1959.
- 9 Axel Dreves and Matthias Gerdt. Free Flight Trajectory Optimization and Generalized Nash Equilibria in Conflicting Situations. Technical report, Universität der Bundeswehr München, 2017.
- 10 Stuart E. Dreyfus. An Appraisal of Some Shortest Path Algorithm. *Operational Research*, 17(3):395–412, June 1969.
- 11 Eurocontrol. Free route airspace (FRA). Online, 2019. Accessed April 2019. URL: <https://www.eurocontrol.int/articles/free-route-airspace>.
- 12 Eurocontrol. Route Availability Document. Online, April 2019. URL: <https://www.nm.eurocontrol.int/RAD/>.
- 13 Luca Foschini, John Hershberger, and Subhash Suri. On the Complexity of Time-Dependent Shortest Paths. *Algorithmica*, 68(4):1075–1097, April 2014. doi:10.1007/s00453-012-9714-7.
- 14 Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.
- 15 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.
- 16 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, July 1968.
- 17 Casper Kehlet Jensen, Marco Chiarandini, and Kim S. Larsen. Flight Planning in Free Route Airspaces. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, 2017.

8:14 Pruning Search Spaces

- 18 Stefan E. Karisch, Stephen S. Altus, Goran Stojković, and Mirela Stojković. *Quantitative Problem Solving Methods in the Airline Industry*, chapter Operations, pages 283–383. Springer, 2012.
- 19 Adam Schienle. Shortest Paths on Airway Networks. Master's thesis, Freie Universität Berlin, 2016.

Exploiting Amorphous Data Parallelism to Speed-Up Massive Time-Dependent Shortest-Path Computations

Spyros Kontogiannis

University of Ioannina, Greece
Computer Technology Institute & Press, Rion, Greece
<http://www.cse.uoi.gr/~kontog/>
kontog@uoi.gr

Anastasios Papadopoulos

University of Patras, Greece
anpapad@ceid.upatras.gr

Andreas Paraskevopoulos

University of Patras, Greece
paraskevop@ceid.upatras.gr

Christos Zaroliagis

University of Patras, Greece
Computer Technology Institute & Press, Rion, Greece
<https://www.ceid.upatras.gr/webpages/faculty/zaro/>
zaro@ceid.upatras.gr

Abstract

We aim at exploiting parallelism in shared-memory multiprocessing systems, in order to speed up the execution time with as small redundancy in work as possible, for an elementary task that comes up frequently as a subroutine in the daily maintenance of large-scale *time-dependent* graphs representing real-world relationships or technological networks: the *many-to-all time-dependent shortest paths* (MATDSP) problem. MATDSP requires the computation of one time-dependent shortest-path tree (TDSPT) per origin-vertex and departure-time, from an arbitrary collection of pairs of origins and departure-times, towards all reachable destinations in the graph.

Our goal is to explore the potential and highlight the limitations of *amorphous data parallelism*, when dealing with MATDSP in multicore computing environments with a given amount of processing elements and a shared memory to exploit. Apart from speeding-up execution time, consumption of resources (and energy) is also critical. Therefore, we aim at limiting the work overhead for solving a MATDSP instance, as measured by the overall number of arc relaxations in shortest-path computations, while trying to minimize the overall execution time. Towards this direction, we provide several algorithmic engineering interventions for solving MATDSP concerning: (i) the compact representation of the instance; (ii) the choice and the improvement of the time-dependent single-source shortest path algorithm that is used as a subroutine; (iii) the way according to which the overall work is allocated to the processing elements; (iv) the adoption of the amorphous data parallelism rationale, in order to avoid costly synchronization among the processing elements while doing their own part of the work.

Our experimental evaluations, both on real-world and on synthetic benchmark instances of time-dependent road networks, provide insight how one should organize heavy MATDSP computations, depending on the application scenario. This insight is in some cases rather unexpected. For instance, it is not always the case that pure data parallelism (among otherwise totally independent processors) is the best choice for minimizing execution times. In certain cases it may be worthwhile to limit the level of *data parallelism* in favor of *algorithmic parallelism*, in order to achieve more efficient MATDSP computations.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms



© Spyros Kontogiannis, Anastasios Papadopoulos, Andreas Paraskevopoulos, and Christos Zaroliagis; licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 9; pp. 9:1–9:18



Open Access Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases amorphous data parallelism, delta-stepping algorithm, travel-time oracle, many-to-all shortest paths, time-dependent road networks

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.9

Funding *S. Kontogiannis, A. Papadopoulos, C. Zaroliagis*: Partially supported by the INTERREG V-A GR-IT programme 2014-2020 (project INVESTMENT).

A. Paraskevopoulos: Supported by the Hellenic Foundation for Research and Innovation, and the General Secretariat for Research and Technology of Greece (project 10110).

1 Introduction

Recent advances on hardware and algorithms for mining and analyzing a large data corpus, have unveiled an entire novel era of Algorithmic Data Science, which is perceived as the new revolution in Computer Science. Apart from the computational efficiency of executing elementary tasks, possibly numerous times and on huge data sets, another crucial aspect is the consumption of resources for these computations. For example, despite the huge improvements of Natural Language Processing via the exploitation of Artificial Intelligence and Machine Learning, there are some serious concerns about the environmental impact of these improvements: As demonstrated in [22], training a single AI model for NLP corresponds to the emissions of carbon of five cars in their entire lifetimes, including their own production phases.

The emphasis of the present work is on exploiting parallelism for speeding up *cautiously* (i.e., as work-efficiently as possible) the execution-time of an elementary but demanding task: computing earliest-arrival-times and/or the corresponding paths, from each of a collection of pairs of origins and departure-times, towards all reachable destinations, in large-scale graphs with time-dependent arc-traversal-time functions. Such instances represent real-world networks like road network infrastructures, social/friendship/collaboration networks, power grids, etc. This task, called the *Many-To-All Time-Dependent Shortest Path* (MATDSP) problem, appears quite often as a typical subroutine in the daily maintenance of such graphs, e.g., for the creation of metric related metadata.

Our motivation comes from the need for fast MATDSP computations when dealing with travel-time, landmark-based, oracles for large-scale road networks with time-dependent arc-traversal-time functions. A typical travel-time oracle preprocesses the instance as efficiently as possible, in order to create a carefully designed data structure (called *travel-time summaries*) of *subquadratic* space requirements. This data structure will then be exploited by a query algorithm which responds to arbitrary earliest-arrival-time queries in time *sublinear* in the size of the instance, and with *provable approximation guarantees* about the quality of the chosen path. During the preprocessing phase of such an oracle (e.g., of CFLAT [11] which is to date the most efficient travel-time oracle), MATDSP is repeatedly used while pre-computing approximate earliest-arrival-time functions, from selected origins (the landmarks) and for carefully selected departure-times, towards all reachable destinations. The MATDSP-algorithm for CFLAT in [11] was based on *a priori splitting* the overall workload of landmarks among the available processing elements, i.e., a static, *work-sharing* approach. Consequently, each processing element would employ a time-dependent shortest path subroutine to execute its own part of the job.

An algorithm for MATDSP would need as a subroutine an algorithm for the *Time-Dependent Shortest Path* (TDSP) problem of computing a *time-dependent shortest-path tree* from a given origin and departure-time towards all reachable destinations. Two classical algorithms for TDSP, at least when the instance obeys the celebrated *FIFO property* for the arc-traversal-times, are:

- the time-dependent variant (TDD) [8] of Dijkstra's (DIJ) algorithm [7], and
- the time-dependent variant (TDBF) [18] of the Bellman-Ford (BF) algorithm [4, 10].

The main difference of the time-dependent variants TDD and TDBF from DIJ and BF, respectively, concern the relaxations of arcs which can only occur as the result of an evaluation of an arc's traversal-time *function* at a given departure-time from its tail that is depicted by its current label. For the time-independent case (static arc-costs), several variants and hybrids of the classical DIJ and BF algorithms have appeared in the literature. Especially for DIJ, numerous priority queues have been also considered, e.g. see [12] and references therein. We proceed with an analogous experiment for TDD, on real-world time-dependent road network instances.

Our *first contribution* is an experimental evaluation of our own implementations of the TDD algorithm with (i) binary-heap (TDDbh), (ii) implicit-priority heap (TDDph) and (iii) sequence heap (TDDsh) in real-world road-network instances. Our experiments demonstrate that TDDsh is superior to both TDDbh and TDDph, at least for workloads coming from real-world road networks.

A notable hybrid of DIJ and BF is the Delta-Stepping (DS) algorithm [14], which connects smoothly these two extremes. Ideally, one would like to trade-off as smoothly as possible the (optimal) work of the essentially sequential DIJ algorithm, with the optimal completion time of the fully parallelizable BF algorithm. This is exactly what DS is doing, by organizing the arc relaxation requests to be served in a more loose partial order than that of DIJ, but certainly in a more structured way than BF which considers all the arc relaxation requests in an arbitrary order (which is exactly its main advantage with respect to parallelization). We adapt the DS algorithm to work for FIFO-abiding TDSP instances.

Our *second contribution* is the first (to our knowledge) implementation and experimental evaluation of a time-dependent variant (TDDS) of the DS algorithm. The experimental evaluation demonstrates that TDDS is more efficient than TDDsh, at least for road network instances.

When trying to exploit parallelism for MATDSP, there are several aspects one should take into account. The challenge is, given a small number P of parallel computing nodes, to achieve as much speedup as possible (ideally up to P times) in execution-time, compared to the most efficient sequential execution-time as a subroutine, consuming as small overhead in work as possible. Although many papers in the literature deal with speeding up many-to-all shortest path computations in time-independent instances, most of them exploit either massively-parallel architectures or GPU-computing (e.g., [6, 5]) to speedup the wallclock execution time, regardless of the work efficiency. On the other hand, our own approach tries to achieve as much speedup as possible, for a given amount of $P = 24$ threads at our disposal, approaching the optimum speedup of 24 as much as possible, and also with a small overhead in the overall work as measured by the number of arc-relaxation requests (the elementary operations in label-setting/correcting algorithms for shortest paths).

A parallel MATDSP algorithm must make two crucial strategic decisions: (i) allocate the overall work to the processing elements either statically (i.e., at the beginning of the computation) at the cost of possibly unbalanced portions of work, or dynamically (i.e., at runtime) at the cost of centrally controlling the pending work to be allocated; (ii) either to abide with the partial ordering of the arc-relaxation requests (as the sequential variants of TDD and TDDS would do), or else allow for cautious violations of these orderings, as is done according to the amorphous-data-parallelism (ADP) rationale [19].

Our *third contribution* is the consideration of the ADP rationale in the parallel implementations of both the TDDS algorithm for MATDSP, and the entire preprocessing phase of the CFLAT oracle.

Our parallel implementation of TDDS was based on the ADP implementation of DS in the Galois system [2, 16, 17], which was for *time-independent* shortest-path tree computations. We also applied the ADP and dynamic work allocation rationales to the preprocessing phase (and the query algorithm) of the CFLAT oracle, leading to a new, more efficient version which we call the OFLAT oracle. The workload in the preprocessing phase is always allocated dynamically (i.e., at runtime), both to processes (for landmarks) and threads (for chunks of arc-relaxation requests); as for the ADP rationale, it is adopted for the entire preprocessing phase, not just for the parallel implementation of TDDS. Moreover, we used static allocation of equal work shares for generic MATDSP instances, but fully dynamic allocation of work for the preprocessing phase of the OFLAT oracle since the exact work load cannot be accurately predicted in this latter case.

For our experimental evaluations we used two real-world road networks, plus one synthetic benchmark instance, with arc-traversal-time *functions* to determine the *time-dependent* cost of traversing each arc in the network at a particular time of the day. The real-world instances represent the metropolitan area of Berlin (BER) and the German road network (GER), respectively. The synthetic instance represents the road network of Europe (EUR). The graphs are assumed to be fixed, in the sense that the input data does not change. We have experimentally evaluated the performances of the following MATDSP solvers:

- The sequential MATDSP solvers with TDDbh, TDDph, TDDsh as TDSP subroutines: It turns out that, at least for random workload instances in road networks, TDDsh is the most time-efficient subroutine for a work-optimal sequential MATDSP solver.
- The sequential MATDSP solvers with TDDsh and TDDS as TDSP subroutines: TDDS is already more time-efficient than TDDsh, achieving speedups ranging from 1.036 for BER, 1.042 for GER, and 1.092 for EUR.
- A parallel MATDSP solver based on the ADP rationale, that employs P/Q independent processes (i.e., executables), each running an ADP variant of TDDS with Q threads, called TDDS(Q), as a subroutine. Each process gets a priori (i.e., statically) its own share of the the workload: the achieved speedups range from 18.86 for BER, 18.73 for GER and 16.56 for EUR, compared to our best sequential MATDSP solver using the TDDS(1) subroutine.

Our experimental evaluation for the preprocessing phase of OFLAT demonstrates that it actually pays off to combine data-parallelism with task-parallelism. For example, compared to the pure data-parallelism used in the original CFLAT oracle, the use of P/Q independent processes, each with its own dynamically allocated load share to serve with Q threads, we achieved (for appropriate choices of Q) further speedups in preprocessing by 1.46 times in BER, 1.38 times in GER and 1.49 times in EUR.

As for our travel-time query algorithm, OFCA, it is worth mentioning that parallelism only pays off for very large instances. For example, in the EUR instance OFCA achieves less than half the query time of CFCA [11], although it is still inferior to the query time of KaTCH [3], essentially due to space limitations. On the other hand, for the instances of BER and GER, OFCA is much faster than both CFCA and KaTCH.

2 Preliminaries and Notation

Let $G = (V, E)$ be a directed graph representing a road network. Such a graph is typically sparse (in particular, with constant maximum degree) and non-planar. $\forall uv \in E$, $D[uv] : [0, T) \mapsto \mathbb{R}_{\geq 0}$ is a continuous, piecewise-linear (pwl) function providing the *arc-traversal-times* from the tail u to the head v , for departure times from a given period $[0, T)$. It is assumed that this function has minimum slope greater than -1 , so as to abide with the strict FIFO (a.k.a. non-overtaking) property of road networks. $\forall uv \in E$,

$\forall t_u \in [0, T)$, $Arr[uv](t_u) = t_u + D[uv](t_u)$ is the corresponding function providing *arc-arrival-times* at the head v , for different departure-times from u . Let $\mathcal{P}_{u,v}$ denote all the (u, v) -paths in G . $\forall \pi = \langle x_0x_1, x_1x_2, \dots, x_{k-1}x_k \rangle \in \mathcal{P}_{u,v}$, $\forall t_u \in [0, T)$, $Arr[\pi](t_u) = Arr[x_{k-1}x_k](\dots Arr[x_0x_1](t_u) \dots)$ is the function of *path-arrival-times* from the origin $u = x_0$ to the destination $v = x_k$, when traveling via the (u, v) -path π . $D[\pi](t_u) = Arr[\pi](t_u) - t_u$ is the corresponding function of *path-travel-times* between u and v via π . $\forall t_u \in [0, T)$, $Arr[u, v](t_u) = \min_{\pi \in \mathcal{P}_{u,v}} \{ Arr[\pi](t_u) \}$ is the function of *earliest-arrival-times*, from the origin u to the destination v . $D[u, v](t_u) = Arr[u, v](t_u) - t_u$ is the corresponding function of *minimum-travel-times* from u to v , not necessarily always via the same (u, v) -path. $\forall \epsilon > 0$, the function $\bar{\Delta}[u, v]$ s.t. $D[u, v](t_u) \leq \bar{\Delta}[u, v](t_u) \leq (1 + \epsilon) \cdot D[u, v](t_u) \forall t_u \in [0, T)$, is a $(1 + \epsilon)$ *upper-approximation* of $D[u, v]$. Analogously, the function $\underline{\Delta}[u, v]$ s.t. $D[u, v](t_u)/(1 + \epsilon) \leq \underline{\Delta}[u, v](t_u) \leq D[u, v](t_u) \forall t_u \in [0, T)$, is a $(1 + \epsilon)$ *lower-approximation* to $D[u, v]$. For sake of succinctness in their representations, both $\bar{\Delta}[u, v]$ and $\underline{\Delta}[u, v]$ are also required to be continuous and pwl functions.

3 Algorithm-Engineering Interventions

In this section we provide a detailed overview of the main algorithmic-engineering interventions in order to exploit parallelism towards speeding up the execution times of either generic MATDSP computations, or the preprocessing phase (and secondarily the query algorithm) of the CFLAT oracle, without causing too much additional computational effort. We start with the presentation of the interventions concerning our first application scenario for generic MATDSP computations in time-dependent road networks. We then explain some additional interventions which are necessary for the work-efficient parallelization of the preprocessing phase in the CFLAT oracle. In particular, as we shall explain later, we had to redesign entirely the preprocessing phase, not just the MATDSP subroutine, in order to abide with the rationales of dynamic work allocation and amorphous-data-parallelism.

3.1 Algorithm-Engineering Interventions for Generic MATDSP Computations

In order to provide a time- and work-efficient parallel algorithm for generic MATDSP computations, we proceeded with the following algorithmic-engineering interventions:

3.1.1 Graph Representation

We reconsidered the representation of the time-dependent graph instances. Rather than using the graph type of PGL [13], a quite robust data type that was used in CFLAT's implementation, which aims to support *dynamic* updates of the graph structure, we adopted here its static version (with no empty slots), which coincides with the FORWARD-STAR graph data type [1]. The reason for this decision was that during the preprocessing phase of the oracle, the underlying road network infrastructure does not undergo any alteration. This change had a noticeable improvement on the memory consumption and the cache hit rate.

3.1.2 Optimizing the implementation of TDD

It is well-known that the (theoretically optimal) Fibonacci heap is not the best choice for an implementation of DIJ (or any of its variants), due to both the complications in its own implementation, and the fact that other priority queues (e.g., implicit binary heaps) are known to perform better in practice. Indeed, there have been numerous discussions on the choice of an efficient priority queue for DIJ (e.g. see [12] and references therein).

We departed from the standard implementation of TDD with an implicit binary heap (TDDbh), and tested also its execution time with the *implicit pairing heap*¹ (TDDph variant) and Sander’s implementation [20] of the *sequence heap*² (TDDsh variant). Our experiments demonstrated that, at least for time-dependent road-network workloads, it definitely pays off to adopt TDDsh as a work-optimal TDSP subroutine.

3.1.3 Alternative (sequential) TDSP Algorithms

We considered the substitution of TDD with the, quite efficient in practice, (sequential) TDDS algorithm, as a TDSP subroutine of our sequential MATDSP algorithm. As already mentioned, TDDS is a controllable label-correcting algorithm that allows the relaxation of arcs in a more loose order than that of TDD. The nice thing about TDDS is that it uses a bucket-based structure for the pending arc relaxations, without enforcing too much additional work for determining their total order because of its label-correcting nature within each bucket. Our experimental evaluation demonstrated a clear advantage of the (sequential) TDDS algorithm over the best implementation of TDD, which is TDDsh.

From now on, for sake of comparison, we use our most efficient implementation of a sequential MATDSP algorithm, which uses TDDS as a TDSP subroutine. All speedups due to parallelism are compared to the performance of this sequential MATDSP algorithm. As for the work overheads, these are compared to the optimal work of the sequential MATDSP algorithm with TDDsh as a subroutine.

3.1.4 Synchronous vs. Asynchronous Parallelism

A major burden in shared-memory environments is that one may have to periodically execute costly barrier-synchronization (SYNC) operations among the threads, when the parallelism is not only on data but also within the tasks. The parallel variants (with Q threads) of label-correcting algorithms for TDSP, e.g., TDBF(Q) and TDDS(Q), have a significant advantage: When an arc is tentatively relaxed, although it is not yet one of the arcs that TDD would choose next for relaxation, this (possibly redundant) tentative work cannot harm the *correctness* of the TDSP computation. Of course, the overall work is also a commodity that needs to be consumed with caution, especially when one is provided with a rather limited number of computational resources. In order to avoid as much as possible the need for SYNC operations, but without suffering too many unnecessary computations as in TDBF(Q), we adopt the rationale of *amorphous-data-parallelism* (ADP) [19] for our parallel implementation TDDS(Q) of TDDS. ADP lets each thread within a parallel algorithm, like TDBF(Q) or TDDS(Q), proceed with its own (eagerly allocated to it) computations independently of the other threads. The only indirect SYNC is done via the common pool of pending work, which is centrally handled in the shared memory of the system. TDBF(Q) would create a single pool of arc-relaxation requests, and each idle processor would then request to be allocated chunks of requests from this pool. TDDS(Q), on the other hand, organizes these pending arc-relaxation requests in buckets of different sizes (representing travel-time distances of the arcs’ tails from the origin), so that arcs which emerge from vertices belonging to buckets closer to the origin, are relaxed before arcs which emanate from vertices in buckets that are further away. Nevertheless, SYNC operations are limited on the threads’ time-dependent

¹ Source code: <https://code.google.com/archive/p/priority-queue-testing/>.

² Source code: <http://algo2.iti.kit.edu/sanders/programs/spq/>.

SPT³ frontiers. Each thread runs at its own speed and simply adopts this partial order for the arcs which are allocated to it. The only care that is taken is that every idle thread always pops from the shared memory a chunk of pending arc-relaxation requests for arcs whose tails belong to the first (in order) non-empty shared bucket.

3.1.5 Data vs. Task Parallelism

When a single process (i.e., executable) uses all the available threads (i.e., $Q = P$) for conducting a parallel computation, a major bottleneck is the shared-memory that is used by all of them. The more threads an algorithm has at its disposal for a parallel computation, the more likely it becomes to have conflicts in shared-memory access, e.g., due to *false sharing* and collisions in *atomic-write* operations. To tackle this difficulty, one could use $P/Q > 1$ processes, each with its own dedicated fraction of the shared-memory and Q threads at its disposal, leading to less bottleneck while accessing the shared-memory (which is fragmented among the processes). An extreme point (corresponding to pure data-parallelism) would be to have P independent processes, each running with a single thread at its disposal. For $Q > 1$ threads per process (and thus, less than P independent processes), there is some sort of blending between (i) data-parallelism among the processes which statically (for MATDSP) or dynamically (for CFLAT) share the overall workload, and (ii) task-parallelism within each process since an ADP implementation, TDDS(Q), of the TDDS algorithm as a TDSP algorithm.

On the contrary, for the CFLAT preprocessing scenario, even the allocation of work among the processes will be done dynamically, as it will be explained later.

We have experimented the hybrid approach between pure data-parallelism ($Q = 1$) and task-parallelism ($Q > 1$), for both for generic MATDSP computations, and for the preprocessing phase of CFLAT. In both cases we observed that it pays off to use more processes each with fewer threads, rather than a single process with all the available threads at its disposal.

3.2 Further Algorithm-Engineering Interventions for CFLAT

In this subsection we reconsider the parallelization of the preprocessing phase in the CFLAT oracle[11]. Our goal is to create an ADP-compliant variant of it, which also handles the allocation of work in a dynamic way. This variant is called OFLAT. Consequently, we also considered the exploitation of our parallel TDSP subroutine TDDS(Q) for the query algorithm of the oracle, called OFCA.

We focus on the main building block of CFLAT's preprocessing phase, which is the approximation algorithm CTRAP in CFLAT to preprocess travel-time functions from selected landmark-vertices towards all reachable destinations from them in the network. We proceed with the presentation of a novel approximation algorithm, called OTRAP, which adopts both the dynamic task-allocation and the ADP rationales in its implementation. Moreover, a different methodology is considered for creating upper-approximating functions between consecutive samples of the unknown travel-time functions.

³ In this work, when referring to an SPT we mean to say a time-dependent shortest SPT.

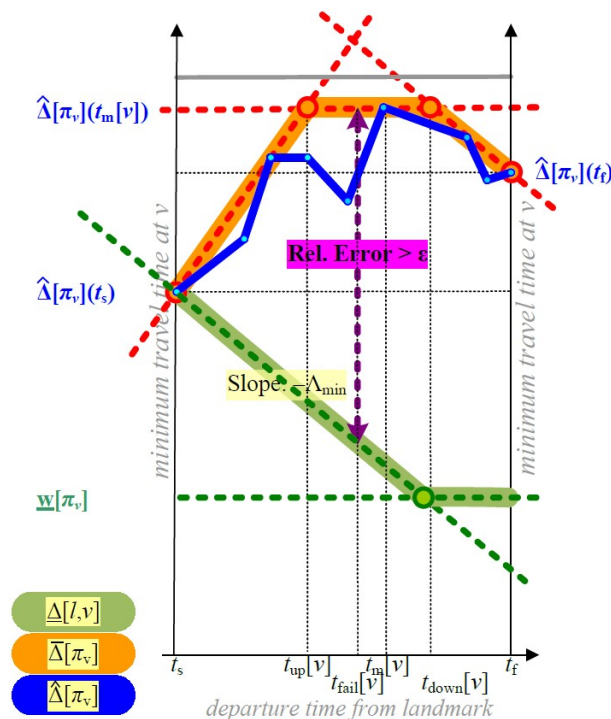
3.2.1 OTRAP: ADP Approximation Algorithm For Travel-Time Summaries

The goal of OTRAP is to compute continuous, piecewise-linear functions $\overline{\Delta}[\ell, v] : [0, T] \mapsto \mathbb{R}_{\geq 0}$ which are *upper-approximations* of an *unknown* minimum-travel-time function $D[\ell, v]$ from a given landmark (the origin) to each reachable destination $v \in V$. $D[\ell, v]$ can be efficiently *sampled* at specific departure-time points (breakpoints). We target for a sampling procedure that will produce a reasonable amount of breakpoints per approximating function, compared to any other upper-approximation that respects a required approximation guarantee of $1 + \epsilon$, or equivalently, a relative error of at most ϵ . Towards this direction, OTRAP mimics the steps of CTRAP, but with some notable differences. In particular, given a targeted relative error of $\epsilon > 0$, the algorithm starts from $t_s = 0$ and time-horizon $t_f = T$, and keeps sampling properly selected departure-times $t_{next} \in [t_s, t_f)$ from ℓ , until we can be sure that the already constructed upper-approximating travel-time functions from ℓ provide approximation guarantees no more than $1 + \epsilon$. The resulting oracle, which bases its own preprocessing on the OTRAP algorithm, is called OFLAT.

We proceed with a more detailed presentation of OTRAP, which is the most notable difference between OFLAT and CFLAT. OTRAP starts with the computation of an SPT T_ℓ^f from ℓ , under the *free-flow* metric according to which each arc $uv \in E$ has weight $w[uv] = \min_{t \in [0, T]} \{ D[uv](t) \}$. Assume inductively that we have already determined the required samples of departure-times and have constructed the corresponding time-dependent SPTs, by calling a TDSP algorithm from our origin ℓ , up to a last sample $t_s \in [0, T)$. Assume also that $t_f \in (t_s, T]$ is the current time-horizon of interest to our approximation (initially we set $t_s = 0$ and $t_f = T$). We have to decide whether to consider a new sample of departure-time from ℓ within (t_s, t_f) . For this, exactly as in CTRAP, we need two approximating functions of $D[\ell, v]$ (per destination v) within $[t_s, t_f)$, a lower-approximating function $\underline{\Delta}[\ell, v]$ and an upper-approximating function $\overline{\Delta}[\ell, v]$.

We begin with the construction of the upper-approximating functions. We first traverse all the tree arcs of the time-dependent SPT $T_\ell(t_s)$, in BFS order, so as to compute per arc $uv \in T_\ell(t_s)$ an upper-bounding function $\overline{A}[\pi_v](t)$ of the path-arrival-time function at v , $Arr[\pi_v](t)$, where π_v is the unique (ℓ, v) -path in $T_\ell(t_s)$. This is done in two steps: We construct, as a *function composition*, the function $\hat{A}[\pi_v](t) = \overline{A}[\pi_u](t) + D[uv](\overline{A}[\pi_u](t)) = Arr[uv](\overline{A}[\pi_u](t))$ of the already known (via the BFS visit order of the tree arcs on $T_\ell(t_s)$) upper-bounding function $\overline{A}[\pi_u](t)$, starting from $\overline{A}[\pi_\ell](t) = A[\pi_\ell](t) = t$, and the actual arc-arrival-time function $Arr[uv](t)$. The function $\hat{\Delta}[\pi_v](t) = \hat{A}[\pi_v](t) - t$ is already an upper-approximation of $D[\ell, v]$ within $[t_s, t_f)$, but with possibly too many breakpoints. We thus construct an upper-bounding function $\overline{\Delta}[\pi_v](t)$ of $\hat{\Delta}[\pi_v](t)$, within the considered departure-times subinterval $[t_s, t_f)$, which only imposes at most 4 breakpoints, the two extreme points and at most two intermediate breakpoints. In particular, $\overline{\Delta}[\pi_v](t)$ has a trapezoidal shape, as the lower-envelope of three lines: The constant line, parallel to the departure-times axis, that is tangent to the maximum point of $\hat{\Delta}[\pi_v](t)$ within $[t_s, t_f)$. Let $t_m \in (v)$ be the corresponding departure-time for this maximum value of $\hat{\Delta}[\pi_v]$. We construct two more lines: The first line passes via the point $(t_s, \hat{\Delta}[\pi_v](t_s))$ and is an upper-bounding tangent line to the left part of $\hat{\Delta}[\pi_v](t)$, i.e., for the subinterval $[t_s, t_m(v)]$. The second line passes via the point $(t_f, \hat{\Delta}[\pi_v](t_f))$ and is also an upper-bounding tangent line, to the right part of $\hat{\Delta}[\pi_v](t)$ this time, i.e., for the subinterval $[t_m(v), t_f)$. The required upper-approximating function $\overline{\Delta}[\pi_v](t)$, with the (at most) 4 breakpoints, is the lower-envelope of these three lines (cf. the solid-orange pwl function in Figure 1).

As for the lower-approximating functions $\underline{\Delta}[\ell, v]$ within $[t_s, t_f)$, these are constructed a bit differently from CTRAP: We consider a single line passing by $(t_s, D[\pi_v](t_s))$ and decreasing at slope $-\Lambda_{\min} = -1$ (i.e., the smallest possible slope, given the FIFO property for travel-time functions), and the constant line representing the free-flow distance $\underline{w}[\pi_v]$ from ℓ to v . Then, $\underline{\Delta}[\ell, v]$ is the upper-envelope of these two lines (cf. the solid-green pwl function in Figure 1).



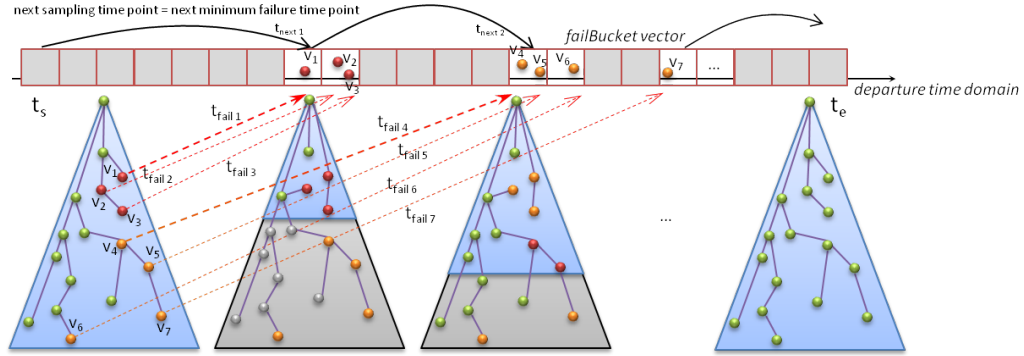
■ **Figure 1** The construction of the upper- and lower-approximating functions by OTRAP. Solid lines represent the pwl approximations. The upper-approximating function $\hat{\Delta}[\pi_v]$ is defined as the composition of the (already constructed, due to BFS order) function $\bar{\Delta}[\pi_u]$ and the exact arc-traversal-time function $D[uv]$. The relative error at time $t \in [t_s, t_f)$ is defined as follows: $RelError[v](t) = (\bar{\Delta}[\pi_v](t) - \underline{\Delta}[\ell, v](t)) / \underline{\Delta}[\ell, v](t)$. The failure-time for v is: $t_{fail}[v] = \inf\{t > t_s : RelError[v](t) > \epsilon\}$.

Given the two approximating functions per destination $v \in V$, OTRAP's next step is to determine the earliest departure time $t_{fail}[v] > t_s$ at which these two functions induce a relative error larger than ϵ . The next departure-time sample t_{next} is then equal to the minimum failure-time, $t_{next} = \min_{v \in V} \{ t_{fail}[v] \}$, among all reachable destinations from ℓ . OTRAP then moves t_s to t_{next} , updates the time-horizon t_f appropriately (see next subsection), and repeats until no destination has failure time within $[0, T)$.

In order to have fast access to the value of t_{next} , the OTRAP algorithm maintains a vector *failBucket* of failure-time slots, with index $1, 2, \dots, T$. Each destination $v \in V$ is then assigned to the bucket $i = \lfloor t_{fail}[v] \rfloor$. The earliest failure time is determined by the smallest index of a non-empty bucket. Moreover, for each sampled departure-time, rather than constructing a complete SPT, we only target at some *active* destinations whose current failure times (and possibly also the ones after the computation of the new SPT) are more likely to affect the determination of the next sampling point. In particular, if i_{next} is the next non-empty bucket, the vertices having the current earliest failure times

$t_{fail}[v] \in [i_{next}, i_{next} + 1)$ are marked as *active destinations*. The next sampling needs to be done at $t_{next} = \min_{v \in failBucket[i_{next}]} \{t_{fail}[v]\}$, constructing the next time-dependent SPT from (ℓ, t_{next}) until all active destinations are settled. Their settlement signifies an early-stopping condition for the TDSP algorithm (see Figure 2).

Let D_{max} be the maximum travel-time of these active destinations in the new tree. Some additional vertices, having failure-time less than $\zeta + t_{next} < T$, and travel-time less than $\gamma \cdot D_{max}$, for a given parameter $\gamma \geq 1$, are also marked as active destinations (red nodes in Figure 2). The inclusion of those “nearby” destinations reduces OTRAP’s computational cost, because it is most likely that those destinations are going to take part in the following sampling steps. The execution of the TDSP algorithm from (ℓ, t_{next}) then resumes, and continues until all these additional active destinations also become settled. The new subinterval of departure times to consider will be $[t_s := t_{next}, t_f := \min\{t_{next} + \delta \cdot D_{max}, T\})$, where $\delta \geq 1$ determines the width of the departure-times domain for the upper-approximating path-travel-time functions. The parameters γ and ζ adjust the depth of the shortest path tree that is required to be built, whereas δ adjusts the width of the departure-times interval, for the the upper-bounding functions. The algorithm terminates when all the destinations from ℓ have failure-times beyond T .



■ **Figure 2** The SPT sampling is done at the earliest failure-time point, among the vertices’ failure-times along their paths from ℓ in $T_\ell(t_s)$, in order to preserve the upper approximation guarantee. Red nodes denote vertices with earlier failure-time, which require an earlier sampling. Orange nodes denote vertices that have later failure-times, and thus require a sampling at a later step. Green nodes denote vertices that have achieved the required upper-approximation of the min-travel-time function over T and thus they don’t need further sampling.

As for the space requirements, OTRAP follows the same lossless compression scheme for the output data as in the case of CTRAP, with an additional procedure on the storage of predecessor-vertices’ IDs: In place of each predecessor-vertex ID, we choose to store the position-index of the corresponding arc in the vertex’s list of incoming arcs. Moreover, the predecessor-arcs indices are stored in bit-field arrays, thus further reducing the required number of bytes.

3.2.1.1 Dynamic Scheduling of Work

Due to the label-setting nature of TDSP(Q), and the the early termination of the sampling procedure when computing the upper-approximating functions $\bar{\Delta}[\ell, v](t)$, the actual work that corresponds to serving each of the landmarks during the preprocessing phase is not known a priori. For this reason, we chose to dynamically allocate work during runtime of OTRAP. We consider two different approaches towards this direction:

- Exploitation of joint data- and task-parallelism: We consider a number P/Q of independent processes (executables), each running on its own copy of the input. Each process claims pending (i.e., not yet assigned to other processes) landmarks dynamically, exploiting system routines for handling file descriptors: Each preprocessed information on behalf of a particular landmark has to eventually be stored in a file by the process handling it. Therefore, when a process becomes idle it claims a new landmark by first checking whether this file already exists. If not, it opens it and locks it for exclusive-write access. In order to avoid claiming already served landmarks, each process considers the landmarks in its own random order. Each process employs Q threads for serving the landmarks assigned to it, independently of the other processes.
- Dynamic allocation of work: Within each process (with Q threads at its disposal), every landmark is assigned dynamically to a unique thread that becomes idle, so that the overall work load is shared among the available threads as evenly as possible. In particular, each thread that becomes idle requests for an available landmark (i.e., not yet being assigned to another thread), and then calls OTRAP to preprocess it. Except for the indirect synchronization via the allocation of pending landmarks to idle threads, each thread's work is done independently of the other threads. The TDSP subroutine employed by OTRAP is the time-dependent and amorphous-data-parallel variant TDDS(Q) of the DS algorithm. OTRAP also makes a breadth-first-search (BFS) traversal of the arcs in the produced SPT, in ADP fashion. An amorphous-data-parallel variant BFS(Q) of a BFS traversal was implemented, which starts from the landmark ℓ at which the current SPT is routed. The traversal of all the destination vertices in the same BFS level can clearly be done independently and in parallel, offering an equivalent result as in the sequential BFS traversal. Even destinations at different levels can be processed independently of each other, provided that all their ancestors in the BFS tree have already been processed. This is exactly what is exploited by our ADP implementation BFS(Q) of BFS: Each thread is allowed to move deeper in the tree, so long as it is assured that all the predecessor have already been traversed.

3.2.1.2 Other improvements w.r.t. OTRAP

The OTRAP approximation algorithm achieves even better execution times when we apply the following classical optimizations for shortest-path computations:

- Vertex reordering: Similar to well-known observations concerning performance enhancements of DIJ [6, 21], we reorder the vertices of the graph so that neighboring vertices are actually located in adjacent memory blocks. This way, the Cache misses are reduced and the execution times are further decreased. For this re-ordering we used a variant of the depth first search (DFS) traversal of the graph: in each step we visit and insert into a LIFO queue all the adjacent vertices from the current vertex just popped out of the queue. That is, we adopt a traversal of vertices moving as much as possible in-depth first, and following a local-breadth scan (i.e., among sibling vertices) only when further depth is not possible for the moment. This order achieves a significant reduction on the Cache misses of both DIJ and DS(Q).
- Cache-friendly and compact data allocation: We order all the required variables (e.g., distances, predecessors, and sample containers) of both the OFCA (query) and OTRAP (preprocessing) algorithms for each vertex and arc, in order to enforce a contiguous memory allocation and thus reduce as much as possible the Cache misses whenever the algorithm needs to access the memory.

3.2.2 OFCA: The ADP query algorithm of OFLAT

The query algorithm OFCA of OFLAT is quite similar to the query algorithm CFCA of CFLAT, essentially following three main steps: In Step 1, a small time-dependent SPT ball is grown from the query’s origin pair (o, t_o) , until a given number of $N \in \{1, 2, 4, 6\}$ landmarks are settled. In step 2, starting from the query’s destination d , we recursively move backwards towards the origin o , following the preprocessed tree arcs in all the time-dependent trees routed at these settled landmarks, until some of the settled vertices from step 1 is reached. Step 3, finally, runs a TDSP subroutine in the subgraph induced by the arcs that have been marked in Step 2, in order to determine the best time-dependent od -path in this subgraph.

The difference between CFCA and OFCA lies only in Steps 1 and 3, and has to do with the choice of the TDSP subroutine. CFCA always uses the time-dependent variant of TDDbh, whereas OFCA considers either TDDsh or TDDS(Q), depending on the size of the network, which determines whether it really pays off to parallelize the query algorithm.

4 Experiments

4.1 Experimental Setup

All our algorithms were implemented in C++ (GNU GCC version 5.4.0) and Ubuntu Linux (16.04 LTS). All our experiments were conducted on a dual 6-core Intel Xeon CPU E5-2643v3 3.40GHz machine, with 128GB of RAM and 20MB SmartCache and 2 hardware threads per core. We used all 24 threads for the parallelization of both the MATDSP computation and the preprocessing phase of the CFLAT oracle.

Two real-world instances (BER,GER) and one synthetic instance (EUR) of road networks are used in our experiments, which have been provided to us for scientific purposes and are typical benchmarks for time-dependent speedup techniques. The instance of Berlin, kindly provided by TomTom in the frame of common R&D projects, describes the arc-travel-time functions taken from historical data of a typical working day (Tuesday). The instance of Germany, kindly provided by PTV AG in the frame of common R&D projects, describes a typical working day (TUE-WED-THU). The instance of Europe is based on the (static) road network instance of Western Europe provided in the 9th DIMACS challenge, which was equipped with synthetically generated travel-time functions [15]. Table 1 summarizes the description of the three benchmark instances:

■ **Table 1** Statistics of the benchmark instances.

Instance	#vertices	#arcs
BER	473.253	1.126.468
GER	4.692.091	10.805.429
EUR	18.010.173	42.188.664

4.2 Testing TDD with different priority queues in MATDSP Instances

It was common knowledge for many years that *implicit binary heaps* are quite efficient, and definitely more efficient than the theoretically optimal *Fibonacci heaps* for implementing DIJ. Nevertheless, recent studies have argued about the superiority of other heap variants, such as the *implicit pairing heaps* and the *sequence heaps*. For a comprehensive comparison of (static) DIJ’s performance on various workloads, when using different priority queue implementations,

the reader is referred to the excellent survey [12]. In our own experiment, we tested TDD’s performance w.r.t. three different priority queues, for randomly created generic MATDSP workloads that emerge in large-scale road networks. In particular, we have experimentally tested generic MTDASP computations, with the following variants of TDD:

- TDDbh, which is equipped with our own implementation of an implicit binary heap;
- TDDph, which uses an implicit pairing heap [9], as provided by [12]; and
- TDDsh, which uses Sanders’ implementation of a sequence heap [20].

We have always activated the DFS ordering of the vertices and the Cache-friendliness optimizations (cf. Section 3). We tested the construction of complete SPTs from randomly selected (origin,departure-time) pairs. The reported times are average times among independent random selections of (origin,departure-time) pairs. The sequence heap appears to have a clear advantage. Table 2 presents the results of our experimentation.

■ **Table 2** TDD’s implementation with different (implicit) priority queues: TDDbh for binary heap, TDDph for pairing heap, and TDDsh for sequence heap with fixed weight range (i.e., travel-times diameter) precomputed. Three MATDSP instances were created, with 1000 random queries for BER and GER, and with 100 random queries for EUR. For each query a complete time-dependent SPT was constructed. Only one thread was used for each of these experiments. In all cases the Cache-Friendliness and DFS ordering optimizations were used.

MATDSP EXECUTIONS WITH DIFFERENT IMPLEMENTATIONS OF TDD			
Complete SPTs from random (o,t) pairs -- Cache-Friendliness enabled -- DFS-ordering enabled			
#random queries @ INSTANCE			TDSP subroutine
1000 @ BER	1000 @ GER	100 @ EUR	
Avg SPT Time (msec)	Avg SPT Time (msec)	Avg SPT Time (msec)	#processes x PQ type (#threads)
193.43	2,213.05	9,065.11	1xTDDbh(1)
193.43	2,465.33	10,187.70	1xTDDph(1)
139.46	1,492.05	6,346.22	1xTDDsh(1)

4.3 Comparing Sequential and Parallel TDSP Subroutines in MATDSP Instances

Already for workloads on time-independent large-scale instances it was evident that the DS algorithm is in practice more efficient than DIJ. We conducted an analogous experiment for the time-dependent variants of the two algorithms, when used as subroutines of a sequential MATDSP solver.

As it is shown in the first two rows of Table 3, TDDS(1) is a more efficient algorithm compared to our best implementation TDDsh of the time-dependent Dijkstra’s algorithm with a sequence heap. The speedups of TDDS(1) over TDDsh ranges from 1.0363 for BER, to 1.042 for GER and 1.092 for EUR (these times are the inverses of the reported values the the last three columns of Row 1, in Table 3).

From now on we consider the execution-times reported in Row 2 of Table 3 as our baseline sequential performance, for comparison with the performances of the parallel MATDSP solvers that will be presented shortly. As demonstrated in Row 3 of Table 3, for the small BER instance the fastest MATDSP algorithm uses 24 parallel processes, each using a single thread for running its own TDDsh computations. On the other hand, Row 6 shows demonstrates that for the larger instances of GER and EUR the best MATDSP algorithm uses only 4 processes, each employing 6 threads for calling TDDS(6). I.e., it actually pays off to combine data-parallelism, e.g. statically splitting the workload among the 4 processes) with task-parallelism, e.g. using 6 threads for the execution of the ADP implementation TDDS(6).

■ **Table 3** Comparing execution times of generic MATDSP instances. Rows 1 and 2 concern sequential MATDSP algorithms (i.e., running on a single thread), which use TDDsh and TDDS as their TDSP subroutines, respectively. The times reported in Row 2 are used as the ground-truth for comparing the parallel algorithms' speedups over our best sequential MATDSP solver. Row 3 presents our fastest parallel MATDSP algorithm for BER, which employs 24 independent processes with a single thread. Row 6 and presents our fastest parallel MATDSP algorithm for BER, which employs 4 independent processes with 6 threads per process, for the instances of GER and EUR.

	MATDSP algorithm	Total Complete SPT Times (sec) MATDSP instance			Speedups wrt 1xTDDS(1) MATDSP instance		
		1000@BER	1000@GER	100@EUR	1000@BER	1000@GER	100@EUR
1	1xTDDsh(1)	139.46	1,492.05	634.62	0.965	0.959	0.916
2	1xTDDS(1)	134.57	1,431.39	581.04	1.000	1.000	1.000
3	24xTDDsh(1)	7.14			18.860		
4	12xTDDsh(2)		98.27			14.566	
5	12xTDDS(2)	7.66			17.571		
6	4xTDDS(6)		76.42	35.09		18.730	16.558

The observed speedups of our parallel MATDSP algorithms compared to our best sequential MATDSP algorithm, are 18.86 times for BER, 18.73 times for GER, and 16.558 times for EUR. It is reminded that for these generic MATDSP instances the workload is statically split among the different processes. It is also noted that the parallel implementation TDDS(Q) (for $Q > 1$) abides with the ADP rationale.

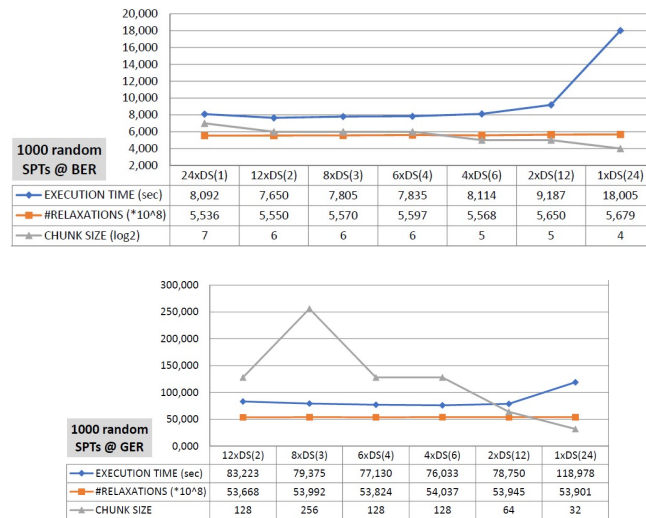
4.4 Sensitivity of TDDS(Q) to the choice of chunk sizes in MATDSP Instances

Since the ADP implementation of TDDS(Q) is dependent on the sizes of the chunks (with pending relaxation requests) that we consider, we conducted MATDSP experiments consisting of 1,000 randomly chosen (origin,departure-time) pairs, each of which is allocated to one of the P/Q processes (different executables) for construction of a complete time-dependent SPT. These processes get *statically* their own shares of work (i.e., $(Q/P) \cdot 1000$ pairs each).

Each process then employs the ADP implementation TDDS(Q) to serve its own workload sequentially. The allocation of chunks of arc-relaxation requests to the Q threads of the process is done dynamically this time: each thread that becomes idle claims the next available chunk of pending requests. Figure 3 shows the results of this experiment in BER and GER instances. In both cases the chunk size achieving the optimal execution time is decreasing with the number of processes that we use. This makes sense, since the more threads a process has at its disposal, the smaller chunks it should use in order to avoid having idle threads with no task to be allocated to them. It is evident also that the larger the instance, the larger the chunk size that we should use for TDDS(Q). It is finally noted that there is no significant variation of the overall work to be done (measured by the number of arc relaxations), as a function of the chunk size.

4.5 Data-Parallelism vs. Task-Parallelism in MATDSP Instances

Our next experiment was to determine the trade-off between pure data parallelism (that is exploited by 24 processes each running TDDsh), and algorithmic parallelism which is also exploited when using TDDS(Q): the work is (statically) among P/Q processes, each of them employs TDDS(Q) for serving its own work load, with the chunks-load again dynamically



■ **Figure 3** Experimenting with chunk sizes of TDDS(Q) when constructing 1000 random SPTs.

allocated to the Q threads within DS(Q). We have run an experiment of 1,000 random SPT queries in BER, and 1,000 random SPT queries in GER. Our first observation is that the best execution times for MATDSP with TDDS(Q) is indeed when $Q > 1$, in both cases. This implies that it pays off, at least for generic MATDSP instances, with TDDS(Q) as a subroutine, to mix data parallelism (i.e., how the queries are split among the processes) with algorithmic parallelism where each of the queries is handled by TDDS(Q). For the BER instance the best choice is to use 12 processes with 2 threads each, whereas for GER the best choice is to use only 4 processes with 6 threads each. When compared with the pure data-parallelism of 24 processes each using TDDsh (recall that TDDsh is superior to TDDS(1)), although this latter scenario is preferable for BER (the speedup of $12 \times$ TDDS(2) processes is less than 1), it is inferior to the scenario $4 \times$ TDDS(6) in the case of GER, achieving a speedup of more than 1.268. At the same time, the work overhead of both $12 \times$ TDDS(2) in BER and $4 \times$ TDDS(6) in GER over $24 \times$ TDDsh(1)'s optimal work, as measured by the total number of arc-relaxation requests, is at most 1.08.

■ **Table 4** Comparison of using TDDsh or TDDS(Q), during the execution of a MATDSP task.

TDDsh vs TDDS(Q) @ MATDSP		1000 random SPTs		4000 random SPTs	
instance	method	relaxations (*10 ⁶)	time (sec)	relaxations (*10 ⁶)	time (sec)
BER	24xTDDsh(1)	507.602	7.135	2,030.406	27.961
	12xTDDS(2)	554.875	7.659	2,218.709	30.238
	work overhead speedup	1.093	0.932	1.093	0.925
GER	12xTDDsh(2)	4,980.150	98.269	19,920.647	383.297
	4xTDDS(6)	5,377.436	76.423	21,486.132	302.268
	work overhead speedup	1.080	1.286	1.079	1.268

4.6 Effect of ADP Rationale in OFLAT

The effect of the ADP rationale was assessed with respect to the OFLAT oracle. In particular, we executed the preprocessing phase using our improved OTRAP approximation technique, both with one OTRAP process running 24 threads, each executing TDDsh (for which no

■ **Table 5** Comparison of preprocessing times of OFLAT, when using the amorphous-data-parallel implementation of OTRAP with: (i) the time-dependent DIJsh, and (ii) time-dependent DS(Q).

OFLAT PREPROCESSING			
#landmarks @ INSTANCE	Method	Time (min)	Speedup
1000 @ BER	8xTDDS(3)	7.70	1.46
	1xTDDsh(24)	11.28	
1000 @ GER	12xTDDS(2)	89.22	1.38
	1xTDDsh(12)	123.45	
900 @ EUR	4xTDDS(6)	3,549.75	1.49
	1xTDDsh(12)	5,293.33	

ADP is needed since each thread executes its own part), and with (P/Q) OTRAP processes, each running the ADP variant TDDS(Q) as a TDSP subroutine. In the latter case, it is noted once more that, apart from TDDS(Q) which was already implemented according to the ADP rationale, the entire preprocessing phase had to be redesigned and implemented under this rationale as well. We conducted measurements for the construction of preprocessed landmark information for BER, GER and EUR. Table 5 presents all these measurements. It is clear that even for the smaller BER instance, the task-parallelism of the ADP implementation TDDS(Q) pays off, compared to TDDsh, leading to speedups of 1.46 for BER, 1.38 for GER and 1.49 for EUR.

Table 6 presents a final experiment which demonstrates the efficiency of the query performance of OFCA, compared to those of CFLAT’s CFCA algorithm and the KaTCH speedup technique, in the GER and EUR instances. BER instance is omitted simply because already CFCA was superior to KaTCH for this instance [11].

We have tried both TDDsh and TDDS(Q) as TDSP subroutines for the first step of the query algorithm, and it became evident that GER is still small for parallelism to be useful. Indeed, the OFCA’s query performance was optimized with TDDsh. For EUR, on the other hand, even the query performance becomes non-negligible and parallelism of TDDS(Q) in OFLAT again pays off, compared to TDDsh of CFLAT. Compared to the query performance of KaTCH, OFCA is faster for the GER instance, but still slower for the EUR instance. Nevertheless, it is clearly faster than CFCA of CFLAT (even with the improved TDDsh variant).

5 Conclusions and Future Work

In this work we have attempted to explore the potential but also highlight the limitations of amorphous-data-parallelism and dynamic allocation of work, in generic MATDSP instances as well as in the CFLAT oracle.

Our findings demonstrate the significance of carefully using the available resources (hardware threads of the multicore environment) in order to achieve remarkable speedups with relatively small work overheads. For example, for generic MATDSP instances we have shown that the speedups of our parallel implementations range from 16.558 up to 18.860 in our experiments, compared to our most efficient sequential MATDSP solver 1xTDDS(1). At the same time, the workload overhead against the work-optimal (but not as efficient) sequential solver 1xTDDsh(1), is at most by no more than 1.08 times in the BER and GER instances.

■ **Table 6** Comparison of query response times among CFLAT, OFLAT and KaTCH, in GER and EUR instances. All reported times are average times of 50,000 independent trials.

QUERY RESPONSE TIMES (msec)		CFLAT ORACLE CFCA(N)	OFLAT ORACLE OFCA(N)		KaTCH
GER		4K SR Landmarks	5K SR Landmarks		0.820
	TDSP alg	1xTDDbh(1)	1xTDDS(1) [$\Delta=32$]	1xTDDsh(1)	
	N=1	0.582	0.692	0.4972	
	N=2	1.242	1.087	0.9612	
	N=4	2.413	1.926	1.8768	
	N=6	3.572	2.904	2.7515	
EUR		--	700 SR Landmarks		1.560
	TDSP alg	--	1xTDDS(12) [$\Delta=128$]	1xTDDsh(1)	
	N=1	--	3.332	7.998	
	N=2	--	4.678	15.463	
	N=4	--	7.688	30.008	
	N=6	--	10.444	44.652	

We have also seen the effectiveness of the ADP rationale for the parallelization of the CFLAT oracle. The improvement in both the preprocessing phase and the query performance of the OFLAT oracle, over CFLAT, is significant.

Even when compared to the prevailing speedup technique for TDSP, KaTCH, the OFCA query algorithm is quite competitive: Its query-time is already better than that of KaTCH in GER, but still inferior in EUR, mainly due to space limitations for the EUR instance in the preprocessing phase. We are currently in the process of further improving the space requirements of OFLAT's preprocessing, so that more landmarks are affordable for the EUR instance. This way, OFCA will become even faster for instances in the size of EUR.

References

- 1 R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory Algorithms and Applications*. Prentice Hall, Englewood Cliffs, 1993.
- 2 M. Amber-Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. *Sigplan Notices - SIGPLAN*, 46:3–12, 2011.
- 3 G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1):1–43, 2013. URL: <https://github.com/GVeitBatz/KaTCH>.
- 4 R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16, 1958.
- 5 V. T. Chakaravarthy, F. Checoniy, P. Murali, F. Petriniy, and Y. Sabharwal. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *IEEE Transactions on Parallel & Distributed Systems*, 28:2031–2045, 2017.
- 6 D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-accelerated shortest path trees. *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 921–931, 2011.
- 7 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- 8 S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

- 9 M. L. Fredman, R. Sedgwick, D. D. Sleator, and R. E. Tarjan. The pairing heap – A new form of self-adjusting heap. *Algorithmica*, 1:111–119, 1986.
- 10 L. R. Ford Jr. Network flow theory. Technical report, RAND CORP SANTA MONICA CA, 1956.
- 11 S. Kontogiannis, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis. Improved Oracles for Time-Dependent Road Networks. *Algorithmic Approaches for Transportation Modeling Optimization, and Systems (ATMOS)*, 2017.
- 12 D. Larkin, S. Sen, and R. E. Tarjan. A Back-to-basics Empirical Study of Priority Queues. *Algorithm Engineering & Experiments (ALENEX)*, pages 61–72, 2014.
- 13 G. Mali, P. Michail, A. Paraskevopoulos, and C. Zaroliagis. A new dynamic graph structure for large-scale transportation networks. *Conference on Algorithms and Complexity (CIAC)*, pages 312–323, 2013. LNCS 7878.
- 14 U. Meyer and P. Sanders. Δ -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.
- 15 G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional A* search on time-dependent road networks. *Networks*, 59:240–251, 2012.
- 16 D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. *ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.
- 17 D. Nguyen, A. Lenharth, and K. Pingali. Deterministic Galois: On-demand Portable and Parameterless. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 499–512, 2014.
- 18 A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.
- 19 K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Amber-Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Proutzos, and X. Sui. The tao of parallelism in algorithms. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 12–25, 2011.
- 20 P. Sanders. Fast Priority Queues for Cached Memory. *Journal of Experimental Algorithmics*, 5(7), 2000. ACM, New York, NY, USA.
- 21 P. Sanders, D. Schultes, and C. Vetter: Mobile route planning. Mobile route planning. *European symposium on Algorithms (ESA)*, pages 732–743, 2008.
- 22 E. Strubell, A. Ganesh, and A. McCallum. Energy and Policy Considerations for Deep Learning in NLP. *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019. [arXiv:1906.02243](https://arxiv.org/abs/1906.02243).

More Hierarchy in Route Planning Using Edge Hierarchies

Demian Hesse

Karlsruhe Institute of Technology, Germany

www.kit.edu

hesse@kit.edu

Peter Sanders

Karlsruhe Institute of Technology, Germany

www.kit.edu

sanders@kit.edu

Abstract

A highly successful approach to route planning in networks (particularly road networks) is to identify a hierarchy in the network that allows faster queries after some preprocessing that basically inserts additional “shortcut”-edges into a graph. In the past there has been a succession of techniques that infer a more and more fine grained hierarchy enabling increasingly more efficient queries. This appeared to culminate in contraction hierarchies that assign one hierarchy level to each *vertex*.

In this paper we show how to identify an even more fine grained hierarchy that assigns one level to each *edge* of the network. Our findings indicate that this can lead to considerably smaller search spaces in terms of visited edges. Currently, this rarely implies improved query times so that it remains an open question whether edge hierarchies can lead to consistently improved performance. However, we believe that the technique as such is a noteworthy enrichment of the portfolio of available techniques that might prove useful in the future.

2012 ACM Subject Classification Theory of computation → Shortest paths; Mathematics of computing → Paths and connectivity problems; Mathematics of computing → Graph algorithms

Keywords and phrases shortest path, hierarchy, road networks, preprocessing

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.10

1 Introduction

Computing shortest, fastest, or otherwise optimal routes in networks is a fundamental problem needed to be solved in many applications. For road networks alone there are multiple important applications, e.g., car navigation, traffic simulation, planning in logistics, etc. An important approach to fast route planning is to preprocess the network in such a way that subsequent queries are accelerated. In this paper we focus on point-to-point queries in road networks but note that other types of queries or networks might also be supported in a way analogous to previous applications of contraction hierarchies [12, 3].

A particularly successful class of preprocessing techniques for road networks is to exploit hierarchy in the network. An informal way to describe this is, that “usually”, the farther away we are from source or destination, the more important are the roads we use. Hierarchical route planning techniques had a history in becoming more aggressive in exploiting the hierarchy resulting in smaller and smaller search spaces. This began with early heuristics based on road categories [15, 16] and later used exact techniques that insert *shortcut edges*. Shortcuts encode that certain subpaths are important and, together with an appropriate query algorithm, ensure that optimal paths can be found using hierarchical routing techniques. Such techniques include overlay graphs [22, 7], reach based routing [14], highway hierarchies [20] and highway node routing [21] – so far culminating in contraction hierarchies (CHs) [11, 12, 9].



© Demian Hesse and Peter Sanders;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 10; pp. 10:1–10:14

OpenAccess Series in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

CHs order the *vertices* of the network by importance, i.e., we conceptually have n levels of hierarchy in a network with n vertices. By inserting appropriate shortcuts, CHs ensure that there exists an *up-down path* between any pair of vertices that is a shortest path. An up-down path progresses from the source vertex to more important vertices and then descends to less important vertices until reaching the destination. CHs are widely used since they are simple, allow fast preprocessing using little space and lead to very small search space.

In this paper we introduce *edge hierarchies* (EHs) as an even more fine grained way to define hierarchy in the network. EHs order *edges* rather than vertices by importance. They keep the concept of up-down paths resulting in a very simple query algorithm. Intuitively, this should further reduce search spaces. EHs – in contrast to CHs – only have to explore edges out of a vertex v that are more important than the edge leading to v in the current query. Also note that EHs are very close to the informal definition of hierarchical routing that we gave above.

After introducing basic terms and techniques in Section 2 and discussing further related work in Section 3, we describe EHs in detail in Section 4. While the basic query algorithm is simple by design, a preprocessing algorithm finding the “right” shortcuts turns out to be much more complicated. We also discuss some basic techniques for pruning the query search space.

In Section 5 we perform an experimental evaluation using large real world road networks and different cost functions. It turns out that EHs relax significantly less edges than CHs in particular for cost functions that are known to be difficult for CHs – with distance as the main optimization criterion and/or explicit modeling of turn penalties. Unfortunately, the overall query time is usually slightly worse than CHs and preprocessing time is considerably larger. Overall, EHs are thus an intriguing concept with considerable potential but they need further research to find out whether they will eventually be useful in some applications. In Section 6 we discuss possible research in this direction.

2 Preliminaries

In this paper, we consider directed and weighted graphs $G = (V, E, w)$, where V is a set of *vertices*, $E \subseteq V \times V$ a set of *edges* connecting vertices and $w : E \rightarrow \mathbb{R}_0^+$ a non-negative edge weight function. A path is a sequence of vertices (v_0, \dots, v_n) such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < n$. The *length* of a path is the sum of its edge weights. The length of a shortest path with source vertex s and target vertex t is also called the *distance* between s and t , or $\text{dist}(s, t)$.

The classical algorithm for finding shortest paths is Dijkstra’s algorithm [10]. It maintains a *distance label* (dist) for each vertex and repeatedly *settles* the vertex u with the currently smallest distance label among all unsettled vertices. It then *relaxes* all outgoing edges (u, v) by setting $\text{dist}(v) \leftarrow \min(\text{dist}(v), \text{dist}(u) + w(u, v))$. In the *bidirectional* version of Dijkstra’s algorithm, the *forward* search from s is complemented by a *backward* search from t that only considers incoming edges of the settled vertices.

A *shortcut* is an edge whose length corresponds to the length of some nontrivial path in the graph. For example, for edges $e_1 = (u, v)$ and $e_2 = (v, w)$, a shortcut $e_s = (u, w)$ with $w(e_s) = w(e_1) + w(e_2)$ can be added to the graph. Note that adding shortcuts does not change the distance for any pair of vertices in the graph. Also, by storing skipped vertices, we can recursively *unpack* shortcuts, e.g., by replacing e_s with e_1 and e_2 to find the corresponding path that only uses original (non-shortcut) edges.

Contraction Hierarchies [11, 12, 9] use shortcuts to build a hierarchy where every vertex is on its own level. Vertices are repeatedly removed from the graph in order of a measure of importance. If for any pair of incoming and outgoing neighbors u, w the removed vertex v is on the *only* shortest path (u, v, w) , then a shortcut (u, w) is added. Whether this shortcut is necessary is determined by a so-called *witness search* that runs a shortest path search starting at u on the *overlay graph*. The overlay graph consists of all vertices not yet removed and all edges incident to these vertices. The witness search can be restricted to stop after settling a small amount of vertices. This might add unnecessary shortcuts but does not affect correctness, while having the potential to speed up the algorithm. Vertex importance is usually determined by a combination of different measures. Metrics successfully implemented in previous work (and used in the implementation we compare against in our evaluation) are the amount of shortcuts added when a vertex were removed next, the number of *hops* represented by these shortcuts and an additional *level* metric that helps removing vertices uniformly throughout the graph. These numbers have in common that they only change when a neighbor of a vertex is removed from the graph. The algorithm therefore maintains all vertices in a priority queue with their importance as key. When a vertex is removed, the importance of its neighbors are updated. The query algorithm is a bidirectional Dijkstra search that only relaxes edges that connect a vertex to a more (less) important vertex in the forward (backward) search. Due to this, edges only need to be stored at the end point that is removed first.

3 More Related Work

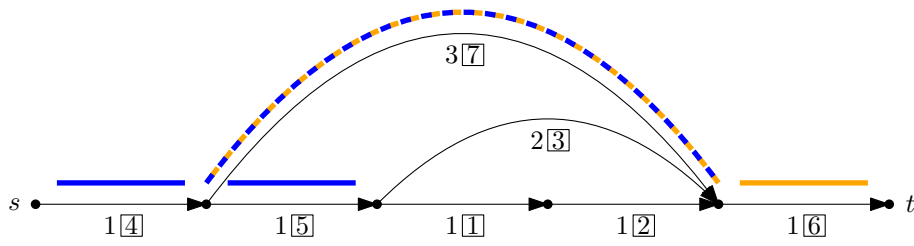
There has been a lot of work on route planning. Refer to [3] for a recent overview. Here we only give selected references to place EHs into the big picture. Besides hierarchical route planning techniques there are also techniques which direct the shortest path search towards the goal (e.g., landmarks [13], precomputed cluster distances [18], arc flags [19]). On road networks goal directed techniques are usually inferior to hierarchical ones since they need considerably more query or preprocessing time. However, combining goal directed and hierarchical route planning is a useful approach [13, 6]. We expect that this is also possible for EHs using the same techniques as used before. Other techniques allow very fast queries by building shortest paths directly from two (hub labeling [1]) or three (transit node routing [4, 2]) precomputed shortcuts without requiring a graph search. However, these methods require considerably more space than EHs.

4 Edge Hierarchies

The main idea of EHs is to provide a precomputed data structure that allows queries similar to those of CHs: All shortest paths can be found by a bidirectional Dijkstra search that only searches “upwards”. In contrast to CHs, which build a hierarchy of vertices, EHs build a hierarchy of edges. Let $r(u, v)$ denote the *rank* assigned to the edge (u, v) . Then, paths found by an EH query have the form $(s = v_0, \dots, v_m, \dots, v_n = t)$ with $r(v_{i-1}, v_i) \leq r(v_i, v_{i+1})$ for $0 < i \leq m$ and $r(v_{i-1}, v_i) \geq r(v_i, v_{i+1})$ for $m < i < n$ (allowing $s = m$ or $t = m$). In line with the terminology from CHs, we call such paths *up-down paths*.

The EH query is a modified version of the bidirectional variant of Dijkstra’s algorithm: In addition to the distance label dist , we maintain a rank label r at every node, set to 0 for s and t . When settling a vertex u , only edges with $r(u, v) \geq r(u)$ are relaxed. Whenever $\text{dist}(v)$ is updated while relaxing an edge (u, v) , $r(v)$ is set to $r(u, v)$. For a stopping condition, the

10:4 More Hierarchy in Route Planning Using Edge Hierarchies



■ **Figure 1** Search space of an EH Query. Blue edges are in the search space of the forward search, orange edges are in the search space of the backward search. Boxed numbers are edge ranks, unboxed numbers are edge weights.

■ **Algorithm 1** BuildEdgeHierarchy.

```

currentRank  $\leftarrow$  0;
while Unranked edges remain do
    Pick unranked edge  $(u, v)$ ;
     $r(u, v) \leftarrow$  currentRank++;
    for all unranked edges  $(u', u)$  do
        for all unranked edges  $(v, v')$  do
            if  $\text{dist}(u', v') = w(u', u) + w(u, v) + w(v, v')$  then
                Add shortcut  $(u', v)$  or  $(u, v')$ ; // Or adjust weight + unset rank
            end
        end
    end
end
end

```

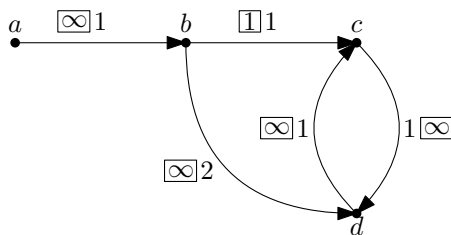
algorithm maintains an upper bound \bar{d} for $\text{dist}(s, t)$ (initially ∞) which is updated whenever a vertex is settled that has already been settled from the other direction. No edges leaving vertices with $\text{dist}(v) > \bar{d}$ are relaxed. Figure 1 illustrates the search space of an Edge Hierarchy Query. Note how the edges ranked 2 and 3 are not in the search space of the backward search, even though their target vertex is settled.

Algorithm 1 shows an algorithm template for constructing an EH. Initially, all edges are unranked (which we will treat as rank ∞). In iteration i , we pick an unranked edge (u, v) and set its rank to i . We then iterate over all unranked edges (u', u) and (v, v') and test whether (u', u, v, v') is a shortest path. If yes, we add either (u', v) or (u, v') as a shortcut. If either of these two edges already exists, we instead adjust its weight and reset its rank to ∞ , if it was already ranked before.

► **Theorem 1.** *For every pair of vertices s and t , such that there is a path from s to t in the input graph, Algorithm 1 assigns ranks and adds shortcuts such that there is a shortest up-down path from s to t .*

Proof. We prove this by showing the following: If at the beginning of iteration i , there is a shortest path from s to t that only uses unranked edges, then in iteration $j > i$, there exists an up-down-path p from s to t that only uses edges of rank $\geq i$. As at the beginning of the first iteration, all edges are unranked, this proves the theorem.

In iteration i , an edge e gets ranked. Let p be a shortest path from s to t consisting only of unranked edges. If e is not part of p , then p is still a shortest path that only uses unranked (rank ∞) edges (which is an up-down path by definition).



■ **Figure 2** Example showing that EH construction needs to calculate distances on the complete graph. Boxed numbers are edge ranks, unboxed numbers are edge weights.

If e is at neither end of p , then a shortcut is inserted that replaces two edges of p , so there still is a shortest path only using unranked edges from s to t .

If $e = (s, v)$ (the case $e = (v, t)$ is analogous) we distinguish two cases:

1. There still exists a shortest path of unranked edges from s to v : Then there is also a shortest path of unranked edges from s to t .
2. There is no shortest path of unranked edges from s to v : Then (s, v) gets assigned rank i and can never change its rank (note for this, that edges can only be inserted or assigned to a different rank if there is a shortest path of unranked edges between their endpoints). Furthermore, there is a shortest path of unranked edges from v to t . By induction, in every iteration $j > i$, there will be an up-down-path from v to t that uses only edges of rank $\geq i$. By adding the edge (s, v) to the beginning of that path, we get an up-down path from s to t .

As the induction basis, note that at the end of the algorithm, no edges are unranked, so the claim holds trivially. ◀

Note that from the induction in the proof above, it follows that we can use the EH query for the distance calculation in Algorithm 1.

The algorithm can also be slightly altered by only adding a shortcut if (u', u, v, v') is the *only* remaining unranked shortest path from u' to v' . However, preliminary experiments showed that the version presented here yields better results.

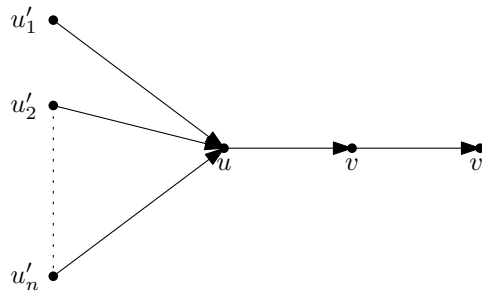
An important difference to CH construction is that Algorithm 1 has to calculate distances in the complete graph, whereas CH construction only has to query the overlay graph. See Figure 2 for an example why using the overlay graph does not suffice for EHs: If (b, d) is assigned rank 2, we need to check whether $p = (a, b, d, c)$ is a shortest path. If only the overlay graph were used for the distance calculation, then we would falsely assume that p is a shortest path and add a shortcut.

4.1 Shortcut Selection

The choice of the shortcut that is added in the inner loop of Algorithm 1 can make a significant impact on the total number of shortcuts added. For example, in Figure 3, we could either add the shortcut (u, v') or *all* of the shortcuts (u'_i, v) (assuming (u'_i, u, v, v') is a shortest path for all u'_i). In contrast, in CHs there is no choice of which shortcut to add. We minimize the number of shortcuts added using a solution to a minimum bipartite vertex cover problem for every iteration of the outer while-loop of Algorithm 1.

The problem $(U \cup V, E)$ is constructed as followed: Instead of directly adding one of the two possible shortcuts, we add the vertices u', v' to U, V respectively (if they have not been added before) and an edge between them.

10:6 More Hierarchy in Route Planning Using Edge Hierarchies



■ **Figure 3** When ranking (u, v) , we could either add *all* shortcuts (u'_i, v) or just (u, v') .

After all shortcut candidates for an iteration of the outer loop have been added to the bipartite graph, we compute a minimum Vertex Cover C . Note that this can be done in polynomial time via maximum cardinality bipartite matching using König's Theorem. We then add the shortcuts (u', v) for every $u' \in U \cap C$ and (u, v') for every $v' \in V \cap C$. It is easy to verify that for every pair of candidate shortcuts, one is added. Also, every set of shortcuts added implies a Vertex Cover for the graph above, so finding a *minimum* Vertex Cover minimizes the number of shortcuts added in every iteration of the construction algorithm, given the edge that is assigned a rank.

To further minimize the number of shortcuts added, we always prefer edges already present in the graph: if (u', v) or (u, v') is already in the graph (ranked or unranked), we change its weight accordingly and reset its rank. The pair (u', v') is then not added to the minimum Vertex Cover problem described above.

4.2 Edge Selection

In every iteration of Algorithm 1, an edge is selected to rank. Our heuristic to select these edges is guided by two goals: Adding a small number of shortcut edges to the graph, and ranking edges uniformly throughout the graph. Here, we present the version that produced the best results in our preliminary experiments. Other versions that resemble the vertex selection strategies used for CHs resulted in worse preprocessing and query times.

Our heuristic works in rounds: in the beginning of each round, a set of edges to rank is selected and fixed. Only when all edges selected are ranked, a new round is started and a new set of edges is selected. Edges are selected by counting for each unranked edge e the number of new shortcuts that would be added if e was ranked. This is done by simulating an iteration of the outer while-loop of Algorithm 1 without actually adding any shortcuts to the graph and resetting $r(u, v)$ to ∞ afterwards. Then, we select all edges that cause the minimum number of shortcuts among all their incident edges.

4.3 Stalling

A technique that significantly reduces query times for CHs is called *Stall on Demand*. The idea is to stall the search at vertices that do not lie on a shortest path from s to t by checking whether a shorter path can be found via incoming (outgoing) downward edges in the forward (backward) search. This can happen because CHs only guarantee shortest up-down paths between any pairs of vertices. The same is true for EHs. We present two stalling techniques that can be used with EHs.

Stall on Demand. In EHs any edge can be a downward or an upward edge depending on the rank of the edges leading to the source vertex of that edge. Stall on Demand checks *all* incoming (outgoing) edges in the forward (backward) search.

Stall in Advance. Stall on Demand may relax every edge twice: Once when settling the source (target) vertex and once for stalling when settling the target (source) vertex in the forward (backward) search. *Stall in Advance* relaxes every edge at most once: when settling a vertex u , we not only relax all outgoing (incoming) edges that are ranked higher than the path to u , but also all edges that are ranked lower. However, we do not update dist with the distance computed via the low ranked edges. Instead, we store it in a separate stallDist label. To check whether we can stall the search at vertex v , we compare $\text{dist}(v)$ with $\text{stallDist}(v)$. If stallDist is smaller, we can stall at v .

5 Experimental Evaluation

We implement EHs in C++ and compile with gcc 7.4.0 using full optimizations (`-O3`). Our implementation of the construction algorithm is relatively straight forward without much emphasis on optimizations. For queries, we use adjacency arrays for incoming and outgoing edges and sort all edges incident to a vertex in descending order of their rank. This way we can stop iterating over a vertex's neighborhood once we find an edge with a lower rank than allowed for the current path. Additionally, we reorder the vertices in depth-first-search-preorder for better memory locality. The EH *construction* algorithm uses CH queries to find the distance between two vertices. The source code is available on GitHub¹.

For comparison with CHs, we use the implementation from RoutingKit² [9] where queries use Stall on Demand.

The machine used for all experiments is equipped with 4 x Intel Xeon E5-4640 at 2.4 GHz and 512 GiB DDR3-PC1600 RAM but only a single core is utilized.

5.1 Data Sets

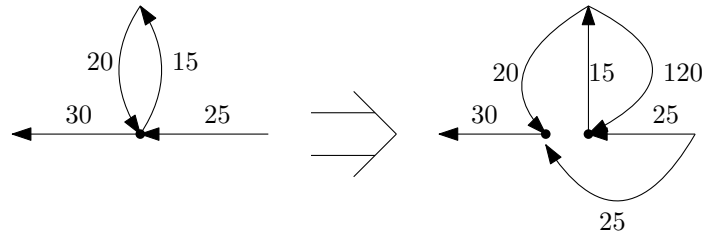
We evaluate EHs on two benchmark graphs from the DIMACS Challenge on Shortest Paths [8]: The road network of Western Europe from PTV AG with 18 million vertices and 42 million directed edges, and the TIGER/USA road network with 23 million vertices and 29 million undirected edges (resulting in 58 million directed edges), as well as smaller subsets of the TIGER/USA graph. Both graphs are available with edge weights corresponding to travel times or geographic distance.

In addition to these graphs, we also evaluate the performance on graphs that model the cost for taking turns at a crossing. We follow the approach used in [7, 3] to define simple turn costs that reportedly yield performance characteristics similar to truly realistic values: For the travel time metric, we assign costs of 100 seconds for U-turns (meaning an edge pair $(u, v), (v, u)$) and 0 for all other turns. For the distance metric, all turns are free. We explicitly model turns into our graphs. This can be done by splitting every vertex v into a number of vertices equal to its degree and connecting each new vertex to one of v 's incident edges. Then, edges between the new vertices are added: For each vertex incident to one of v 's incoming edges, an edge is added to each of the vertices incident to one of v 's outgoing edges. The weights of these new edges are set to the turn costs. We use a more compact

¹ <https://github.com/Hespian/EdgeHierarchies>

² <https://github.com/RoutingKit/RoutingKit>

10:8 More Hierarchy in Route Planning Using Edge Hierarchies



■ **Figure 4** Left: Original graph. Right: Graph with added turns. 100 seconds are added to the edge corresponding to a U-turn.

representation of the same concept: We only split a vertex into a number of vertices equal to its outgoing degree and connect incoming edges directly to these new vertices, adding the turn costs to the edge weights. Figure 4 shows an example for travel times. Table 1 lists all instances and their sizes used in our evaluation.

The distance metric as well as adding turn information are cases in which CHs were shown to perform significantly worse than with the travel time metric and without turn information (e.g. [7]).

■ **Table 1** Instances used in our evaluation. *With turns* are original instances with added turns.

Graph	Original		With turns	
	$ V $	$ E $	$ V $	$ E $
USA.BAY	321 270	794 830	794 830	2 279 208
USA.W	6 262 104	15 119 284	15 119 284	41 815 474
USA.CTR	14 081 816	33 866 826	33 866 826	93 609 832
USA	23 947 347	57 708 624	57 708 624	159 734 066
EUROPE	18 010 173	42 188 664	42 188 664	113 953 602

5.2 Choosing the Right Stalling Technique

In this section we evaluate the stalling techniques explained in Section 4.3. To get some insight in how stalling performs for other techniques, we compare to Stall on Demand for CHs. Tables 2 and 3 compare the query times, number of vertices settled and edges relaxed for different stalling techniques averaged over 100 000 random queries. The number of edges actually relaxed and the number of edges “relaxed” to check whether the search can be stalled are shown separately. We also count the number of vertices that are settled at their actual distance to the source vertex (min. vertices). This gives an insight into how many vertices would be settled with a *perfect* stalling technique. For the travel time metric, EHs with both Stall on Demand and Stall in Advance perform more stall checks than CHs, outweighing the savings in number of vertices settled and leading to longer query times than without any stalling. For the distance metric, Stall on Demand reduces the number of vertices settled for EHs to less than for CHs. The total of number of edges touched is also less for EHs. However, running times are still faster without stalling because less edges are relaxed (or considered for stalling) and thus less distance labels are touched. Due to the additional distance label, Stall in Advance significantly increases query times. The last column also shows that stalling holds more potential for CHs than for EHs. However, we also see that EHs already perform relatively well without stalling: CHs on the travel time metric would have to settle more than twice as many vertices as EHs if no stalling was used and even

■ **Table 2** Query results for different stalling techniques for Edge Hierarchies and Contraction Hierarchies on the EUROPE road network with the **travel time** metric and **turns**.

Algo.	Stalling	time [μ s]	settled	relaxed	stall checks	min. vertices
EH	-	199	906	1734	-	361
	S. on Demand	250	604	958	11920	
	S. in Advance	471	614	982	10563	
CH	S. on Demand	130	533	1969	2888	253
	-	338	1996	15500	-	

■ **Table 3** Query results for different stalling techniques for Edge Hierarchies and Contraction Hierarchies on the EUROPE road network with the **distance** metric and **turns**.

Algo.	Stalling	time [μ s]	settled	relaxed	stall checks	min. vertices
EH	-	608	2573	5586	-	638
	S. on Demand	642	1368	2276	29192	
	S. in Advance	1387	1439	2442	26959	
CH	S. on Demand	634	1943	16849	25007	704
	-	3403	12320	300758	-	

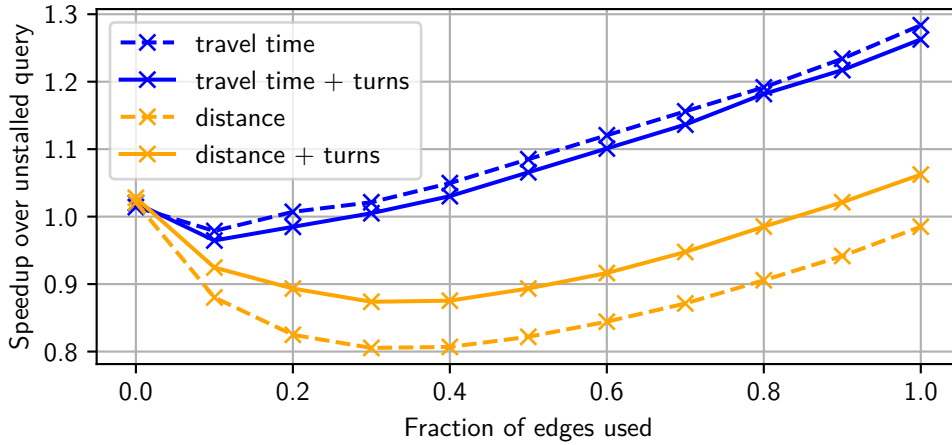
when not counting the stall checks, CHs with Stall on Demand relax more edges than EHs. For the distance metric, this is even more severe: Here, the search space for CHs without Stall on Demand increases so much that query times increase to over 3 ms. EHs already settle a reasonably small number of vertices without stalling.

These experiments show that the increased number of edges touched outweighs the decreased number of vertices settled. Thus, a stalling technique that only touches *some* more edges might lead to improved running times if it successfully stalls at enough vertices. Figure 5 shows the performance when only a fraction of the edges incident to a vertex are considered for Stall on Demand – going from high ranked edges to low ranked edges (note that this can be done efficiently in our implementation as edges are stored ordered by their rank). We are going to refer to this as *partial stalling* from here on. We see a slight increase in running time due to the associated calculations (see the data point for $x = 0.0$) but all instances shown benefit from partial stalling for some fraction (10% for travel times and 30% for distances).

5.3 Main Results

As EHs share similarities with CHs, both using similar query algorithms, we compare the two with respect to their preprocessing and query times as well as the number of vertices settled and edges relaxed during queries. Another interesting property is the number of edges in the hierarchy. Note however, that CHs only store each edge once, whereas EHs need to store each edge at both endpoints. Tables 4 and 5 show these numbers averaged over 100 000 random queries. We execute queries without Stall on Demand and with partial stalling in increments of 10%. The numbers reported here are for the best query times among these stalling configurations as indicated by the last column. In a real-world system the optimal configuration could be found as a part of the preprocessing step. Due to time restrictions, the construction was only run once for each algorithm and instance. Checking whether the search can be stalled at a vertex is essentially an edge relaxation (minus priority queue operations), so we combine these numbers here. We can see that EHs suffer less from adding

10:10 More Hierarchy in Route Planning Using Edge Hierarchies



■ **Figure 5** Speedup of query with partial stalling over unstalled query with different fractions of edges used for stalling. Times were measured on the EUROPE road network.

turns to the graphs than CHs. While the number of shortcuts added is comparable for EHs and CHs on the original graphs (with CHs even adding slightly fewer), CHs add significantly more when turns are added. This can also be seen in the number of edges relaxed: The number of edges relaxed with and without turns are very similar for EHs. For the distance metric, EHs perform even better when adding turns than on the original inputs. With turns, EHs almost always relax less than half as many edges as CHs. This shows that the intuition behind EHs – ranking *roads* (edges) rather than *junctions* (vertices) – helps to better prune roads that are irrelevant for the query. However, CHs usually settle between 2 and 3 times less vertices (except for the distance metric with turns where EHs often settle less vertices than CHs). Overall this leads to longer query times for EHs in most cases. For the distance metric with turns, query times for EHs are close to CHs – for the EUROPE instance EHs even achieve faster queries. The preprocessing step is much faster for CHs, partially due to our unoptimized implementation, but the CH vertex ranking also only updates the neighbors of a vertex after it was ranked. The edge ranking we use, on the other hand, simulates the ranking of every edge for each round of edge selection. The CH implementation in RoutingKit also limits the number of steps done for the witness search, giving additional speed up. As EHs have to find witnesses and (depending on the edge ranking technique) calculate importance values for every edge, compared to CHs having to do the same for every vertex, longer preprocessing times are to be expected.

The random queries used for the experiments above are long-ranged on average. However, real-world queries tend to be short-ranged. For this reason, Sanders and Schultes [20] introduce an evaluation methodology using *Dijkstra Ranks*. When running a Dijkstra query starting at some vertex in the graph, the i th vertex removed from the priority queue is assigned Dijkstra Rank i . Figures 6 and 7 show the number of vertices settled, number of edges relaxed, and query times for vertices of Dijkstra Ranks $2^6, \dots, 2^{\lceil \log |V| \rceil}$ from 1000 random starting vertices. This way, the performance of algorithms can be observed for both short-ranged and long-ranged queries (and everything in between). EHs use 10% and 30% partial stalling for travel times and distances, respectively. The comparison between number of vertices settled and query time shows that the algorithm that settles less vertices has the faster query time and edge relaxations play a less important role. This is likely due to

■ **Table 4** Running times and search space sizes of Edge Hierarchies and Contraction Hierarchies on different graphs with the **travel time** metric.

	Graph	Prepr. [s]		E [M]		Query [μ s]		settled		relaxed		stall. %
		EH	CH	EH	CH	EH	CH	EH	CH	EH	CH	
Original	USA.BAY	100	6	1.4	1.4	37	16	301	108	710	679	-
	USA.W	1785	153	27.5	27.4	96	37	538	193	1299	1386	-
	USA.CTR	4389	482	61.5	61.1	140	53	612	254	3132	2136	10
	USA	7145	674	104.5	104.0	153	60	643	271	3320	2253	10
	EUROPE	3171	453	70.3	70.3	138	75	607	356	2443	2967	10
With turns	USA.BAY	634	156	4.0	6.0	79	67	511	362	929	3253	-
	USA.W	9403	2730	69.9	105.1	165	124	748	564	1365	4810	-
	USA.CTR	25084	7316	159.3	239.2	240	172	885	700	3126	6530	10
	USA	45904	15462	270.3	404.3	250	186	900	737	3217	6792	10
	EUROPE	17822	4743	194.0	249.1	191	130	726	533	2662	4857	10

■ **Table 5** Running times and search space sizes of Edge Hierarchies and Contraction Hierarchies on different graphs with the **distance** metric.

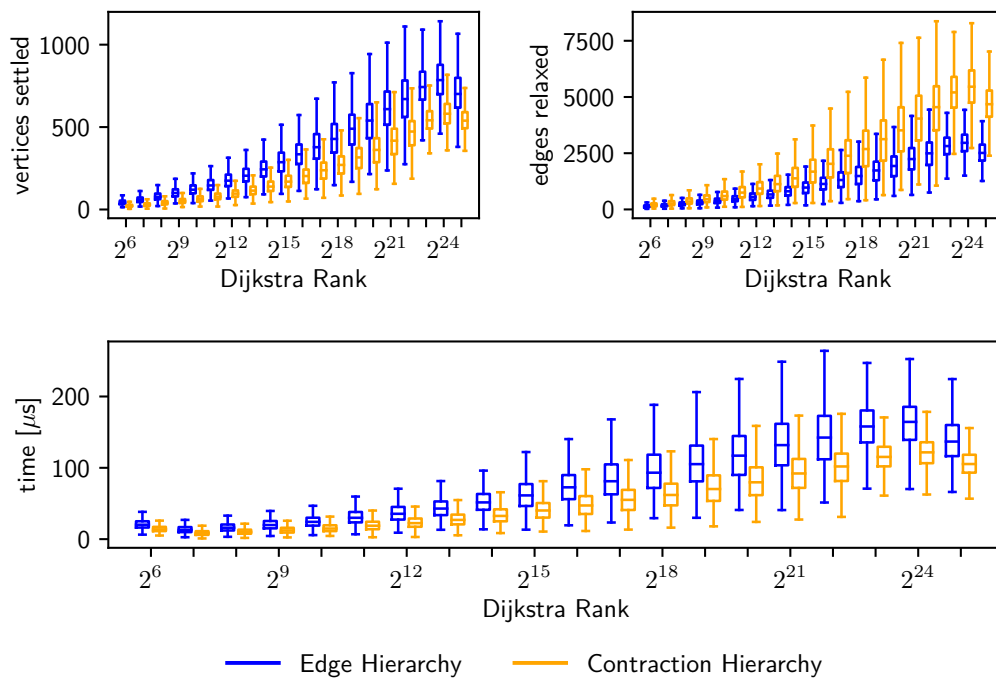
	Graph	Prepr. [s]		E [M]		Query [μ s]		settled		relaxed		stall. %
		EH	CH	EH	CH	EH	CH	EH	CH	EH	CH	
Original	USA.BAY	166	9	1.5	1.5	73	30	560	180	1440	1686	-
	USA.W	3435	243	28.6	28.5	254	96	1002	446	8183	6045	20
	USA.CTR	13062	1157	65.7	65.5	526	216	1697	832	20041	15561	30
	USA	21041	1537	110.8	110.7	573	235	1769	897	21461	16787	30
	EUROPE	14487	2152	79.6	79.6	538	355	1756	1179	19793	27807	30
With turns	USA.BAY	476	158	3.6	5.7	95	92	623	470	1149	4979	-
	USA.W	8452	3338	64.9	102.3	278	250	1289	993	2564	13402	-
	USA.CTR	30313	13629	148.5	235.7	556	537	1477	1743	15286	31629	40
	USA	58025	30869	251.1	398.3	604	580	1605	1849	13712	33436	30
	EUROPE	24757	13266	172.3	267.2	533	634	1543	1943	13355	41856	30

vertex accesses causing more cache misses than accesses to the edges of a single vertex. If one would improve the cache efficiency by better node orderings or other improvements, it seems possible that EHs decreased number of relaxed edges can outweigh the increased number of settled vertices.

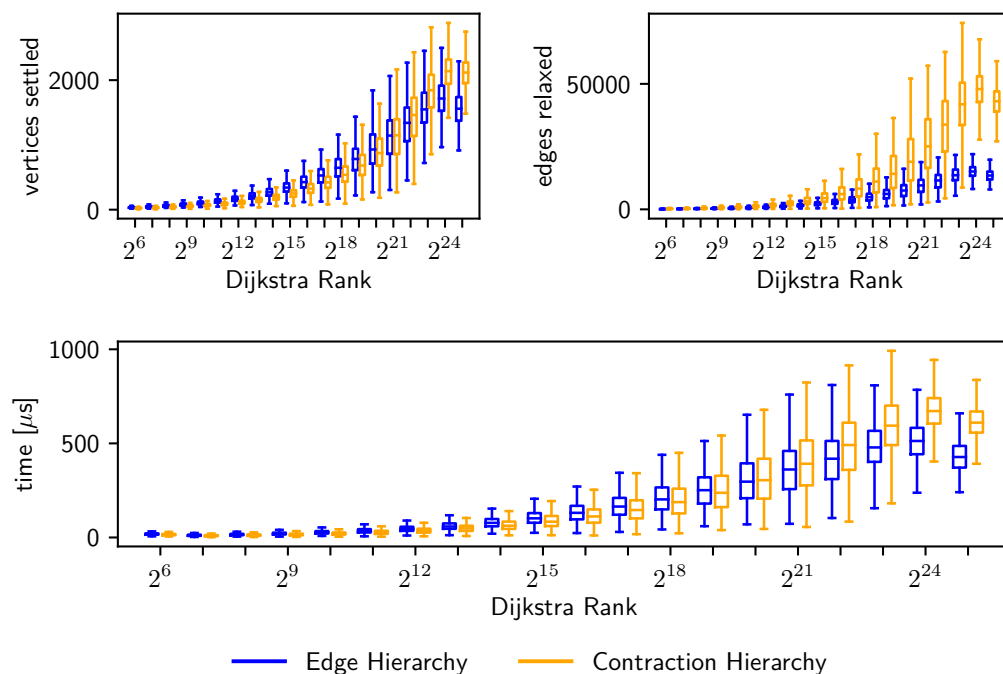
6 Future Work

For CHs there is a lot of experience with configuring the preprocessing process. The additional complications of EH preprocessing make it likely that much better versions are possible also for EHs. Trying different ways of cleaning up the distance labels for new queries might lead to some improvements as preliminary experiments showed some effect here. Due to EHs being less cache-efficient than CHs right now, we expect them to profit more from such changes. On the application side, we can look for networks with different characteristics where EHs might have advantages. For road networks, we might harvest the advantage in number of relaxed edges by looking at generalizations of static shortest path search where edge relaxations are expensive, e.g., time-dependent edge costs [5, 17] or multicriteria shortest paths.

10:12 More Hierarchy in Route Planning Using Edge Hierarchies



■ **Figure 6** Number of vertices settled and edges relaxed, and query times for different Dijkstra Ranks on EUROPE with the **travel time** metric and **turns**.



■ **Figure 7** Number of vertices settled and edges relaxed, and query times for different Dijkstra Ranks on EUROPE with the **distance** metric and **turns**.


References

- 1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *10th Symposium on Experimental Algorithms (SEA)*, volume 6630 of *LNCS*, pages 230–241, 2011. doi:10.1007/978-3-642-20662-7_20.
- 2 Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In *12th International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *LNCS*, pages 55–66. Springer, 2013.
- 3 Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer, 2016.
- 4 Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.
- 5 Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18, 2013. doi:10.1145/2444016.2444020.
- 6 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, 2010. doi:10.1145/1671970.1671976.
- 7 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2015.
- 8 Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Soc., 2009.
- 9 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21:1–5, 2016.
- 10 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- 11 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *7th Workshop on Experimental Algorithms (WEA)*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.
- 12 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.
- 13 Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *8th Workshop on Algorithm Engineering and Experiments (ALENEX)*, Miami, 2006.
- 14 Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 100–111, 2004.
- 15 Kunihiro Ishikawa, Michima Ogawa, Shigetoshi Azuma, and Tooru Ito. Map navigation software of the electro-multivision of the ’91 Toyota Soarer. In *Vehicle Navigation and Information Systems Conference*, volume 2, pages 463–473. IEEE, 1991.
- 16 George R. Jagadeesh, Thambipillai Srikanthan, and K. H. Quek. Heuristic techniques for accelerating hierarchical routing on road networks. *IEEE Transactions on intelligent transportation systems*, 3(4):301–309, 2002.
- 17 Spyros C. Kontogiannis, Dorothea Wagner, and Christos D. Zaroliagis. Hierarchical time-dependent oracles. In *27th International Symposium on Algorithms and Computation (ISAAC)*, pages 47:1–47:13, 2016.
- 18 Jens Maue, Peter Sanders, and Domagoj Matijević. Goal directed shortest path queries using Precomputed Cluster Distances. *ACM Journal of Experimental Algorithmics*, 14, 2009.

10:14 More Hierarchy in Route Planning Using Edge Hierarchies

- 19 Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 189–202, 2005.
- 20 Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *13th Annual European Symposium on Algorithms (ESA)*, pages 568–579. Springer, 2005.
- 21 Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In *6th Workshop on Experimental Algorithms (WEA)*, volume 4525 of *LNCS*, pages 66–79. Springer, 2007.
- 22 Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Using multi-level graphs for timetable information. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

Maximizing the Number of Rides Served for Dial-a-Ride

Barbara M. Anthony 

Southwestern University,
Georgetown TX 78626, USA
anthonyb@southwestern.edu

Sara Boyd

Southwestern University,
Georgetown TX 78626, USA

Christine Chung 

Connecticut College,
New London CT 06320, USA
cchung@conncoll.edu

Jigar Dhimar

Connecticut College,
New London CT 06320, USA

Ricky Birnbaum

Connecticut College,
New London CT 06320, USA

Ananya Christman 

Middlebury College,
Middlebury VT 05753, USA
achristman@middlebury.edu

Patrick Davis

Connecticut College,
New London CT 06320, USA

David Yuen

Kapolei HI 96707, USA
yuen888@hawaii.edu

Abstract

We study a variation of offline Dial-a-Ride, where each request has not only a source and destination, but also a revenue that is earned for serving the request. We investigate this problem for the uniform metric space with uniform revenues. While we present a study on a simplified setting of the problem that has limited practical applications, this work provides the theoretical foundation for analyzing the more general forms of the problem. Since revenues are uniform the problem is equivalent to maximizing the number of served requests. We show that the problem is NP-hard and present a $2/3$ approximation algorithm. We also show that a natural generalization of this algorithm has an approximation ratio at most $7/9$.

2012 ACM Subject Classification Theory of computation → Routing and network design problems

Keywords and phrases dial-a-ride, revenue maximization, approximation algorithm, vehicle routing

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.11

Acknowledgements The authors would like to thank Khanh Nghiem for helpful conversations regarding components of this work.

1 Introduction

Due to their practical applicability, Dial-a-Ride Problems (DARP) have been studied from the perspective of operations research, management science, combinatorial optimization, and theoretical computer science. There are numerous variants of the problem, but fundamentally all DARP variants require the scheduling of one or more vehicle routes and associated times to satisfy a collection of pickup and delivery requests, or rides, from specified origins to specified destinations. Each ride can be viewed as a request between two points in an underlying metric space, with the ride originating at a *source* and terminating at a *destination*. These requests may be restricted so that they must be served within a specified time window, they may have weights associated with them, details about them may be known in advance or only when they become available, and there may be various metrics to optimize. For most variations the goal is to find a schedule that will allow the vehicle(s) to serve requests within



© Barbara M. Anthony, Sara Boyd, Ricky Birnbaum, Ananya Christman, Christine Chung, Patrick Davis, Jigar Dhimar, and David Yuen;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 11; pp. 11:1–11:15



OpenAccess Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the constraints, while meeting a specified objective. Much of the motivation for DARP arises from the numerous practical applications of the transport of both people and goods, including delivery services, ambulances, ride-sharing services, and paratransit services.

We study offline DARP on the uniform metric (i.e. where the distance between any pair of locations is the same for all pairs) with a single server where each request has a source, destination, and revenue. The server has a specified deadline after which no more requests may be served, and the goal is to find a schedule of requests to serve that maximizes the total revenue. We furthermore assume uniform revenues, and refer to this problem as *URDARP*. *URDARP* is thus equivalent to maximizing the number of rides served by the deadline. Although this form of the problem has fewer practical relevance than more general forms, it can be applied to urban settings where it is reasonable to assume that a driver would like to serve as many requests as possible and these requests take roughly the same amount of time to serve. Our motivation for analyzing this basic form of the problem is that doing so would allow us to extend the analysis to more general versions, which have more practical applications, unlike this simplest form. However, we found in the course of our work that analyzing this basic form was challenging in itself. We found that even this fundamental variant is in fact NP-hard, and its analysis elusive. We expect it will provide the theoretical foundation for analyzing the more general forms of the problem.

In particular, we have found that any *DARP* algorithm for the nonuniform revenue variant that greedily chooses one request at a time to serve can be at best a $1/2$ -approximation. Lemma 1 in Section 2 details what happens when, for example, the greedy strategy is based on largest revenue. We have found a similar outcome for the variant of DARP on a non-uniform metric with uniform revenues. Lemma 2 in Section 2 details what happens when the greedy strategy is based on shortest request.

We therefore consider algorithms that give preference to sequences of requests that are “chained” together, i.e. such that each request in a sequence is served immediately after the previous request of that sequence. More formally, a chain of requests is defined as a sequence of requests such that (1) for all requests except the first, the source is the destination of the previous request and (2) for all requests except the last, the destination is the source of the next request. Specifically, we consider an algorithm we call *TWOCHAIN* that gives preference to requests that are in chains of length at least two. We focus on this algorithm and *URDARP* because, with an understanding of this fundamental setting, we can then work to break the barrier of $1/2$ in the setting with general revenues.

In sum, the focus of this work is on offline *URDARP* (i.e. DARP with a single vehicle, on the uniform metric, with uniform revenues). We begin by showing even this basic problem is NP-hard by a reduction from the Hamiltonian Path problem. We then show that our *TWOCHAIN* algorithm yields a $2/3$ -approximation, and exhibit an instance where *TWOCHAIN* serves exactly $2/3$ the optimal number of requests. Since *TWOCHAIN* yields a tight $2/3$ -approximation, with a matching lowerbound instance that is quite simple and clean, we expected that the natural generalization of the algorithm, an algorithm we call *k-CHAIN*, would yield a $k/(k+1)$ -approximation. Surprisingly, it does not. We exhibit an instance of *URDARP* where *k-CHAIN* earns at most $7/9$ times the revenue of the optimal solution. We conclude with a discussion of how the (non polynomial-time) algorithm that greedily always chooses the longest chain gives at most a $5/6$ -approximation.

1.1 Related Work

DARP has been extensively studied and there are numerous variations on the problem, including the number of vehicles, the objectives, the presence or absence of time windows, and how the request sequence is issued (i.e. offline or online). The 2007 survey *The dial-*

a-ride problem: models and algorithms [8] provides an overview of some of the models and algorithms, including heuristics, that have been studied. A decade later, *Typology and literature review for dial-a-ride problems* [9] focuses on classifying the existing literature based upon applicability to particular real-world problems, again including both algorithms with theoretical guarantees and heuristics. To our knowledge, despite its relevance to modern-day transportation systems, the version of the problem we investigate in this paper has not been previously studied, neither for the uniform nor general metric space.

However, there are a few variants that have similarities to our version. Our DARP variant is closely related to the Prize Collecting Traveling Salesperson Problem (PCTSP) where the server earns a revenue (or prize) for every location it visits and a penalty for every location it misses, and the goal is to collect a specified amount of revenue while minimizing travel costs and penalties. PCTSP was introduced by Balas [3] and studied by many others including Awerbuch et al. [2], who gave the first approximation algorithms with polylogarithmic performance. Bienstock et al. developed a 2-approximation for a version of PCTSP where there is a cost for each edge and a penalty for each vertex, and the goal is to find a tour on a subset of the vertices that minimizes the sum of the cost of the edges in the tour and the vertices not in the tour [4]. Blum et al. gave the first constant-factor approximation algorithm for the Orienteering Problem where the input is a weighted graph with rewards on nodes and the goal is to find a path that, starting at a specified origin, maximizes the total reward collected, subject to a limit on the path length [5]. The online variant of the PCTSP, where the cities arrive over time, has also been studied by Ausiello et al. [1] who presented a $7/3$ -competitive algorithm.

The online revenue-maximization variant of DARP, where requests have non-uniform revenue and a release time at which they become known to the server and the goal is to serve requests so as to maximize total revenue by a specified deadline, was also studied for the uniform metric in [7] and a non-uniform metric in [6], where Christman et al. presented competitive algorithms with ratios $1/2$, and $1/6$, respectively.

2 Preliminaries

The input to URDARP is a uniform metric space, a set of requests, and a time limit T . Each request has a source point and a destination point in the metric space, and a revenue, where the revenues are uniform. A unit capacity *server* starts at a designated location in the metric space, the *origin*. The goal is to move the server through the metric space, serving requests one at a time so as to maximize the revenue earned in T time units, which, with uniform revenues, is equivalent to maximizing the number of requests served. For an URDARP instance I , $\text{OPT}(I)$ denotes an optimal schedule on I .

We refer to a move from one location to another as a *drive*. If a request is being served then we refer to it as a *service drive* (sometimes referred to in the literature as a *carrying move*). If the drive is not serving a request and solely for the purpose of moving the server from one location to another we refer to it as an *empty drive* (sometimes referred to in the literature as an *empty move*). We refer to a sequence of one or more requests that are served without any intermediary empty drives as a *chain* and a sequence of two requests that are served without an empty drive in between as a *2-chain*.

We now provide two lemmas regarding a more generalized version of URDARP, where the revenues are nonuniform; we refer to this variant as RDARP. By an analysis similar to that of the online Greatest Revenue First (GRF) algorithm, studied by Christman and Forcier [7], it can be shown that the simple greedy algorithm that repeatedly finds and serves

11:4 Maximizing the Number of Rides for DARP

the highest-revenue request of those remaining is a $1/2$ -approximation for RDARP as well. We now give the matching bound, showing that this greedy algorithm can yield at best a $1/2$ -approximation.

► **Lemma 1.** *The approximation ratio of the greedy algorithm that repeatedly chooses a maximum-revenue request to serve for RDARP is no greater than $1/2$.*

Proof. Consider an instance with x requests chained together each with revenue r , and x individual requests, none of which are connected to other requests, each with revenue $r + \epsilon$ for some small $\epsilon > 0$. No requests start at the origin o . Let $T = x + 1$. OPT will serve all of the x requests that are chained together, earning xr revenue. An algorithm that greedily chooses one request at a time to serve will serve only the requests with revenue $r + \epsilon$, and can serve only $\lfloor x/2 \rfloor$ of them in time T , earning $\lfloor x/2 \rfloor(r + \epsilon)$. ◀

We assume for the remainder of this work that revenues are uniform. We now show that if we instead consider a non-uniform metric, the approximation ratio of a similar greedy algorithm is at most $1/2$.

► **Lemma 2.** *The approximation ratio of the algorithm that greedily chooses the shortest request to serve for DARP with uniform revenues on a non-uniform metric is no greater than $1/2$.*

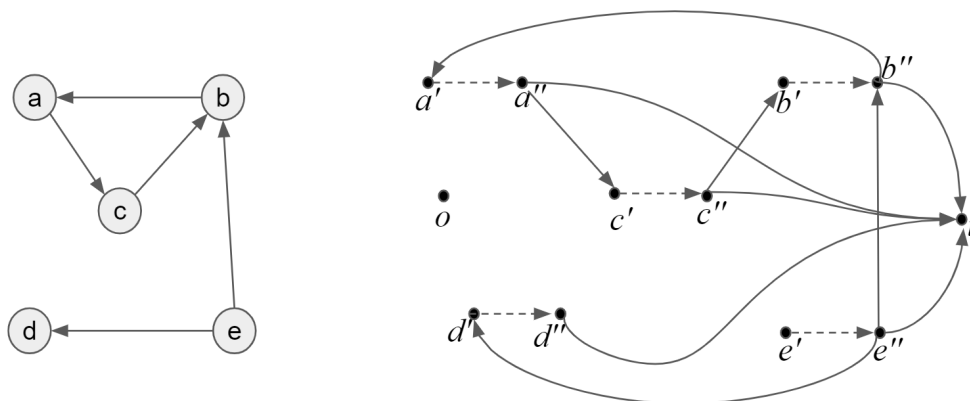
Proof. Let a , b , and c denote three points on a non-uniform metric space such that the distance between a and b and b and c is T/k for some positive even integer k such that $T \bmod k = 0$, and the distance between a and c is $T/k - \epsilon$, for some small $\epsilon > 0$. Let a be the origin. Consider an instance on this space with $k/2$ requests from a to b , $k/2$ requests from b to a , and $k/2$ requests from a to c . OPT will alternately serve the $k/2$ requests from a to b and the $k/2$ requests from b to a , i.e. as a chain of k requests. An algorithm that greedily chooses the shortest request at a time to serve will serve only the requests from a to c while spending $T/k - \epsilon$ time on an empty drive from c to a between each serve, thereby serving $k/2$ requests in total. ◀

2.1 Hardness

While it was already shown in [6] that the problem of offline RDARP on a general metric is NP-hard, we now show that even URDARP, where the metric is uniform and the requests have uniform revenue, is NP-hard by reduction from the Hamiltonian Path problem. The reduction proceeds as follows.

Given a directed Hamiltonian Path input $G = (V, E)$ where $n = |V|$, build a uniform metric space G' with $2n + 2$ points (see Figure 1): one point will be the server origin o , one will be a designated “sink” point t , and the other $2n$ points are as follows. For each node $v \in V$, create a point v' and a point v'' in G' . Create a URDARP request in G' from point v' to point v'' for each $v \in V$, which we will refer to as a *node request*. Further, for each edge $e = (u, v)$ in E of G , create a URDARP request from point u'' to point v' in G' , which we will refer to as an *edge request*. Additionally, for each $v \in V$, create an *edge request* from v'' to the designated sink point t in G' . Set $T = 2n + 1$. Finally, make the server origin a separate point that is one unit away from all other points.

► **Lemma 3.** *There is a Hamiltonian Path in G if and only if $2n$ requests can be served within time $T = 2n + 1$ in the URDARP instance.*



■ **Figure 1** An example instance G of the Hamiltonian Path problem where $n = 5$ (left), and its corresponding instance for URDARP where $T = 2n + 1$ (right). Any Hamiltonian path on a graph of n vertices has length $n - 1$, which would correspond to a sequence in the corresponding URDARP instance of $2n - 1$ requests (since a URDARP request is created for each vertex and each edge of G). But note that here there is no Hamiltonian path in G , yet the URDARP instance still has a sequence of requests of length $2n - 1$ which starts from e' . The extra edges we add from each point v'' to t in the URDARP instance prevent such false positives by ensuring that any Hamiltonian path in G will in fact correspond to a URDARP sequence of length $2n$.

Proof. Let $p = (v_1, v_2, \dots, v_n)$ be a Hamiltonian Path in G . Construct the sequence of $2n$ URDARP requests in G' by the node request from v'_1 to v''_1 , the edge request from v''_1 to v'_2 , the node request from v'_2 to v''_2 , the edge request from v''_2 to v'_3 , and so forth, through the edge request from v''_{n-1} to v'_n , the node request from v'_n to v''_n , and finally the edge request from v''_n to the designated sink t . This sequence can be executed in time $T = 2n + 1$ since it requires one unit of time for the server to drive from the origin to v'_1 and $2n$ units for the remaining drives.

Conversely, consider a URDARP sequence in G' of length $2n$. Note that by construction of G' , any sequence of URDARP requests must alternate between node requests and edge requests, where any edge to the sink is counted as an edge request (and must be a terminal request). Since destinations in G' can be partitioned into the sink, single-primed points, and double-primed points, we can thus analyze the three possibilities for the destination of the final URDARP request.

If either the sink or a single-primed point is the destination for the final URDARP request, the URDARP sequence must end with an edge request. The alternating structure ensures the URDARP sequence begins with a node request, and thus contains exactly n node requests and n edge requests. If a double-primed point is the destination for the final URDARP request, the URDARP sequence must end with a node request. The alternating structure ensures the URDARP sequence begins with an edge request, and contains exactly n edge requests and exactly n node requests. Thus, the URDARP sequence always contains n node requests. This ensures that the length n path in the original graph G includes all n vertices in the original graph G , and thus the existence of a Hamiltonian Path. ◀

Due to the above reduction procedure along with Lemma 3, we have the following theorem.

► **Theorem 4.** *The problem URDARP is NP-hard.*

For the remainder of the work we focus strictly on URDARP (uniform metric, uniform revenues).

3 Algorithms

We begin by presenting our TWOCHAIN algorithm that is a $2/3$ -approximation for URDARP (please see Algorithm 1 for details). The idea of this polynomial-time algorithm is that it simply looks for chains of requests of length at least 2 whenever a drive is required. At each time unit if there is a request that starts at the current location of the server, the server will always serve that request (continuing the chain) rather than driving away to a different request. We note that this subtlety makes the algorithm differ from the algorithm that simply chooses *any* 2-chain to serve; the approximation ratio of this latter algorithm is an open problem. In addition, the server is never “idle” in that if there are remaining requests to serve that can be served before the deadline, the server will drive to serve one of them.

While the analysis of TWOCHAIN we provide requires many detailed cases, we were surprised to discover that simpler more elegant approaches all failed in subtle ways, indicating to us the problem is more nuanced than what one expects at first blush. We believe that the interplay between requests and the possibility for numerous criss-crosses of chains of requests prevents simpler analyses. We note that our proof actually yields a guarantee of not only $2/3$ of the optimal number of requests, but instead $1/3(|OPT| + T - 1)$, where T is the time limit.

3.1 The TWOCHAIN Algorithm

■ **Algorithm 1** The TWOCHAIN algorithm.

```

1: Input: Set  $S$  of requests, time limit  $T$ , origin  $o$ 
2: Set  $t := T$ 
3: Let  $S'$  denote the subset of requests  $(a, b) \in S$  where  $b$  is the source of another request
   in  $S$ .
4: while  $t > 0$  do
5:   if there exists a request starting from  $o$  in  $S$  then
6:     Choose one such request  $(o, b)$ , with preference given to requests from  $S'$ .
7:     Serve  $(o, b)$ .
8:      $t := t - 1$ 
9:     Remove  $(o, b)$  from  $S$  (and  $S'$  if  $(o, b)$  was in  $S'$ ).
10:     $o := b$ , and update  $S'$  based on the new  $S$ .
11:   else if  $t \geq 2$  and requests remain in  $S$  then
12:     Choose one such request  $(a, b)$ , with preference given to requests from  $S'$ .
13:     Drive from  $o$  to  $a$ , then serve the request  $(a, b)$ .
14:      $t := t - 2$ 
15:     Remove  $(a, b)$  from  $S$  (and  $S'$  if  $(o, b)$  was in  $S'$ ).
16:      $o := b$ , and update  $S'$  based on the new  $S$ .
17:   else ▷ no requests remain (that we have enough time to serve)
18:      $t := t - 1$ 

```

Let S , T , and o denote the set of requests, time limit, and origin, respectively. Let $OPT(S, T, o)$ and $ALG(S, T, o)$ denote the schedules returned by OPT and TWOCHAIN, respectively, on the instance (S, T, o) and let $|OPT(S, T, o)|$ and $|ALG(S, T, o)|$ denote the number of requests served by OPT and TWOCHAIN, respectively.

We begin by showing that in the special case where the deadline is more than twice the number of requests, TWOCHAIN is optimal.

► **Lemma 5.** *If $T \geq 2|S|$ then $|OPT(S, T, o)| = |ALG(S, T, o)| = |S|$.*

Proof. By induction on $|S|$. If $|S| = 1$, then clearly TWOCHAIN can serve the request if $T \geq 2$. If $|S| \geq 2$, then within the first two time units TWOCHAIN serves at least one request. So there are at most $|S| - 1$ remaining requests to serve within $T - 2$ time. Since $T \geq 2|S|$, then by the inductive hypothesis, $T - 2 \geq 2(|S| - 1)$, so TWOCHAIN can serve the remaining requests within the remaining time. ◀

In the next lemma, we tackle the general case where the deadline T is tighter. We prove a lower bound on what TWOCHAIN earns, that will suffice for later showing it yields a $2/3$ -approximation.

► **Lemma 6.** *Let $m = |OPT(S, T, o)|$. If $T < 2|S|$, then $|ALG(S, T, o)| \geq \frac{1}{3}(m + T - 1)$.*

Proof. Since $T < 2|S|$, $S \neq \emptyset$. Let k denote the number of requests in the first chain served by TWOCHAIN and denote this chain as $(u_0, u_1), (u_1, u_2), \dots, (u_{k-1}, u_k)$. Let c denote the number of drives TWOCHAIN makes to get to the first request, that is, either $c = 0$ if there is a request starting at o and $c = 1$ if not. After TWOCHAIN serves the first chain, we are left with a smaller instance of the problem $(S_{new}, T_{new}, o_{new})$ where $S_{new} = S - \{(u_0, u_1), (u_1, u_2), \dots, (u_{k-1}, u_k)\}$, $T_{new} = T - c - k$, and $o_{new} = u_k$.

We proceed by strong induction on T . If $T = 0, 1$, or 2 , then the lemma is trivially true. If $T \geq 3$, then since $|S| > T/2$, TWOCHAIN serves at least one chain. We assume inductively that $|ALG(S_{new}, T_{new}, o_{new})| \geq \frac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1)$ and will show $|ALG(S, T, o)| \geq \frac{1}{3}(|OPT(S, T, o)| + T - 1)$.

Case 1: $k = 1$.

Case 1.1: $c = 1$. Then there is no ride starting at o and the first chain has length 1, so we know that there must be no 2-chains in S . Then all solutions require an empty drive after each service drive, so $|ALG(S, T, o)| = m = \lfloor T/2 \rfloor \geq \frac{T}{2} - \frac{1}{2}$ and hence, $m \geq \frac{1}{3}(m + T - 1)$.

Case 1.2: $c = 0$. Then there is a ride starting at o but there is no 2-chain that starts at o . Let $OPT(S, T, o)$ return the path $(o, v_1), (v_1, v_2), \dots, (v_{T-1}, v_T)$. Therefore (o, v_1) and (v_1, v_2) cannot both be rides. Then the path $(v_2, v_3), \dots, (v_{T-1}, v_T)$ has at least $m - 1$ rides from S and therefore at least $m - 2$ rides from $S_{new} = S - \{(o, u_1)\}$. So the path $(o_{new}, v_2), (v_2, v_3), \dots, (v_{T-1}, v_T)$ also has at least $m - 2$ rides from S_{new} . Thus $|OPT(S_{new}, T - 1, u_1 = o_{new})| \geq m - 2$. By induction, $|ALG(S, T, o)| \geq 1 + \frac{1}{3}(|OPT(S_{new}, T - 1, u_1)| + (T - 1) - 1) \geq 1 + \frac{1}{3}(m - 2 + T - 1 - 1) = \frac{1}{3}(m + T - 1)$.

Case 2: $k \geq 2$. There are two subcases.

Case 2.1: $T_{new} \geq 2|S_{new}|$. In this case, by Lemma 5 we have $|ALG(S_{new}, T_{new}, o_{new})| = |S_{new}| = |S| - k$. So we have:

$$|ALG(S, T, o)| = k + |ALG(S_{new}, T_{new}, o_{new})| = k + |S| - k = |S|$$

Hence, $|OPT(S, T, o)| = |S|$ as well, so recalling that $T < 2|S|$, we have, as desired:

$$|ALG(S, T, o)| = |S| = \frac{1}{3}(|S| + 2|S|) > \frac{1}{3}(|OPT(S, T, o)| + T - 1).$$

Case 2.2: $T_{new} < 2|S_{new}|$. Let the path P^* of length $T + 1 - c$ be the path that traverses $OPT(S, T, o)$ starting from o_{new} . More formally, if $c = 0$, P^* is $(o_{new}, o), (o, v_1), (v_1, v_2), \dots, (v_{T-1}, v_T)$. If $c = 1$, then since (o, v_1) is not in S , P^* is $(o_{new}, v_1), (v_1, v_2), \dots, (v_{T-1}, v_T)$.

11:8 Maximizing the Number of Rides for DARP

Let r denote the number of requests in $(u_0, u_1), (u_1, u_2), \dots, (u_{k-1}, u_k)$ that are also in $OPT(S, T, o)$ and note that $r \leq k$. So P^* has m requests from S and $m - r$ requests from S_{new} . Note that $T_{new} = T - c - k = (T + 1 - c) - (k + 1) = |P^*| - (k + 1)$.

We modify P^* to create a path P by deleting the last $k + 1$ drives from P^* . Then $|P| = T_{new}$ and P has at most $k + 1$ fewer requests from S_{new} than P^* so P has at least $m - r - (k + 1)$ requests from S_{new} . Hence, we have:

$$|OPT(S_{new}, T_{new}, o_{new})| \geq m - r - k - 1 \quad (1)$$

There are two subcases.

Case 2.2.1: If $-r + k - 1 - c \geq 0$, then we have:

$$\begin{aligned} |ALG(S, T, o)| &= k + |ALG(S_{new}, T_{new}, o_{new})| \\ &\geq k + \frac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1) \text{ by ind. hyp.} \\ &\geq k + \frac{1}{3}(m - r - k - 1 + T_{new} - 1) \text{ by eqn.(1)} \\ &\geq \frac{1}{3}(m + T - 1 - r + k - 1 - c) \\ &\geq \frac{1}{3}(m + T - 1) \end{aligned}$$

which is the desired equation.

Case 2.2.2: If $-r + k - 1 - c < 0$, then $k - r \leq c$ and there are two subcases.

Case 2.2.2.1: $k = r$. Please see the Appendix where we show that in all subcases of Case 2.2.2.1, P starts at o_{new} , has at least $m - r - k + 1$ requests from S_{new} , and has length T_{new} . Thus:

$$|OPT(S_{new}, T_{new}, o_{new})| \geq m - r - k + 1 \quad (2)$$

Then, since $c = 0$ or $c = 1$, we have:

$$\begin{aligned} |ALG(S, T, o)| &\geq k + \frac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1) \text{ by ind. hyp.} \\ &\geq k + \frac{1}{3}(m - r - k + 1 + T_{new} - 1) \text{ by eqn.(2)} \\ &\geq k + \frac{1}{3}(m - 2k + 1 + T - k - c - 1) \text{ since } k = r \\ &\geq \frac{1}{3}(m + T - c) \geq \frac{1}{3}(m + T - 1) \text{ since } c = 0 \text{ or } 1 \end{aligned}$$

So we are done with Case 2.2.2.1 and must now prove Case 2.2.2.2 to complete the proof.

Case 2.2.2.2: $k \neq r$. Recall that since $k - r \leq c$, it must be that $k = r + 1$. Please see the Appendix where we show that in all subcases of Case 2.2.2.2, P starts at o_{new} , has at least $m - r - k$ requests from S_{new} , and has length T_{new} . Thus:

$$|OPT(S_{new}, T_{new}, o_{new})| \geq m - r - k \quad (3)$$

So we have:

$$\begin{aligned} |ALG(S, T, o)| &\geq k + \frac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1) \text{ by ind. hyp.} \\ &\geq k + \frac{1}{3}(m - r - k + T - k - c - 1) \text{ by eqn.(3)} \\ &= \frac{1}{3}(m + T - 1 - r + k - c) \\ &= \frac{1}{3}(m + T - 1 - r + (r + 1) - c) \\ &\geq \frac{1}{3}(m + T - 1) \end{aligned}$$

This completes the proof. ◀

► **Theorem 7.** TWOCHAIN gives a $2/3$ approximation for URDARP.

Proof. We again proceed by considering two cases.

Case 1: $T \geq 2|S|$: Then by Lemma 5, $|ALG(S, T, o)| = |OPT(S, T, o)|$, and we are done.

Case 2: $T < 2|S|$: Then by Lemma 6, $|ALG(S, T, o)| \geq \frac{1}{3}(|OPT(S, T, o)| + T - 1)$.

As in Lemma 6, let $m = |OPT(S, T, o)|$. There are two subcases.

Case 2.1: If $m < T$, then $|ALG(S, T, o)| \geq \frac{1}{3}(m+T-1) > \frac{1}{3}(m+m-1)$. Since $|ALG(S, T, o)|$ is an integer, this implies $|ALG(S, T, o)| \geq 2m/3$.

Case 2.2: If $m = T$, then an $OPT(S, T, o)$ solution must be $(o = v_1, v_1), (v_1, v_2), \dots, (v_{m-1}, v_m)$ where every drive must be a service drive, serving a request from S .

We use the same definitions of k, r , and c as in Lemma 6 and note that $c = 0$ in this case.

Denote the first chain served by TWOCHAIN as $(o = u_0, u_1), (u_1, u_2), \dots, (u_{k-1}, u_k)$. Note that TWOCHAIN would start with a service drive right from o because in this case there is a 2-chain starting at o . If $k = T = m$ then $|ALG(S, T, o)| = |OPT(S, T, s)|$ so we are done. If $m = 1$ or $m = 2$ then, $k = m$, so we are done. If $m = 3$ then $k = 2$ or 3 , and in both cases we have $k > 2m/3$, so we are also done.

So we consider the case where $m \geq 4$ (so $k \geq 2$) and $k < m$. After TWOCHAIN serves the first chain, the server is at u_k and there is $T - k$ time remaining, so in the smaller instance of the problem, $T_{new} = T - k$, and $o_{new} = u_k$.

Since $|OPT(S, T, o)| = m$, then $|OPT(S_{new}, T + 1, u_k)| \geq m - r$, since in time $T + 1$, OPT can drive from u_k to the origin, and then follow the path of $OPT(S, T, o)$ to serve $m - r$ requests (recall that r is the number of requests in $OPT(S, T, o)$ that are also in the first chain of $ALG(S, T, o)$). So recalling that $T_{new} = T - k$, we have,

$$|OPT(S_{new}, T_{new}, u_k)| \geq m - r - k - 1 \quad (4)$$

And thus:

$$\begin{aligned} |ALG(S, T, o)| &= |ALG(S_{new}, T_{new}, u_k)| + k \\ &\geq \frac{1}{3}(|OPT(S_{new}, T_{new}, u_k)| + T_{new} - 1) + k \text{ by Lemma 6} \\ &\geq \frac{1}{3}(m - r - k - 1 + T - k - 1) + k \text{ by eqn. 4} \\ &= \frac{1}{3}(2m) + \frac{1}{3}(-r + k - 2) \geq 2m/3 \text{ unless } k = r \text{ or } k = r + 1 \end{aligned}$$

For the cases of $k = r$ and $k = r + 1$, we follow the same steps we did for these cases in the proof of Lemma 6 to modify the OPT path.

Case $k = r$: Then by Case 2.2.2.1 of the proof of Lemma 6, we have

$|OPT(S_{new}, T_{new}, o_{new})| \geq m - r - k + 1$. So:

$$\begin{aligned} |ALG(S, T, o)| &= |ALG(S_{new}, T_{new}, u_k)| + k \\ &\geq \frac{1}{3}(|OPT(S_{new}, T_{new}, u_k)| + T_{new} - 1) + k \text{ by Lemma 6} \\ &\geq \frac{1}{3}(m - r - k + 1 + T - k - 1) + k \\ &\geq \frac{1}{3}(2m - 3k) + k \text{ since } T = m \text{ and } r = k \\ &\geq 2m/3 \end{aligned}$$

Case $k = r + 1$: Then by Case 2.2.2.2 of the proof of Lemma 6, we have

$|OPT(S_{new}, T_{new}, o_{new})| \geq m - r - k$. So:

$$\begin{aligned} |ALG(S, T, o)| &\geq \frac{1}{3}(m - r - k + T - k - 1) + k \\ &\geq \frac{1}{3}(2m - 3k) + k \text{ since } T = m \text{ and } r = k - 1 \\ &\geq 2m/3 \end{aligned}$$

We have shown that for all cases, $|ALG(S, T, o)| \geq 2m/3$, so the proof is complete. ◀

11:10 Maximizing the Number of Rides for DARP

We now show that the approximation ratio of $2/3$ for TWOCHAIN is tight.

► **Theorem 8.** *The approximation ratio of TWOCHAIN for URDARP is no greater than $2/3$.*

Proof. Consider an instance with three requests in a single chain with no requests starting at the origin o . Let $T = 4$. TWOCHAIN may select the second and third requests of the chain as its first two requests. For TWOCHAIN to drive to and then serve the two requests takes three time units. It then drives and runs out of time. On the other hand, OPT starts at the first request of the chain and completes all three requests by time $T = 4$. ◀

3.2 The k -CHAIN Algorithm

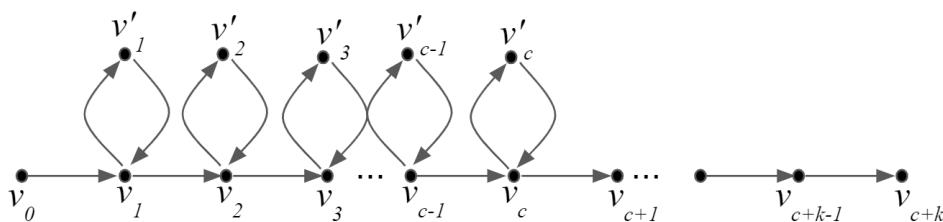
We now show that a natural generalization of TWOCHAIN, which we refer to as k -CHAIN (see Algorithm 2) yields at most a $7/9$ -approximation. This polynomial-time algorithm (which is exponential in the fixed k that is selected) proceeds analogously to TWOCHAIN, but rather than prioritizing requests that are the first in a 2-chain, instead it prioritizes requests that are the first in a k -chain. One might expect that this algorithm yields a $k/(k+1)$ -approximation but we show that, surprisingly, there exists an instance of URDARP where k -CHAIN earns at most $7/9$ times the revenue of the optimal solution.

■ **Algorithm 2** The k -CHAIN algorithm.

```
1: Input: Set  $S$  of requests, time limit  $T$ , origin  $o$ 
2: Set  $t := T$ 
3: For  $i = 1 \dots k$ , for each request  $r \in S$ , add  $r$  to  $S_i$  if request  $r$  is followed by a chain of
   requests of length  $i - 1$ . So  $r \in S_i$  means  $r$  is the start of a chain of length  $i$ . Note that
   if  $r \in S_i$  then we also have  $r \in S_j$  for all  $j < i$ .
4: while  $t > 0$  do
5:   Let  $j$  be the highest value for which  $S_j$  has a request starting at  $o$ .
6:   if  $j > 0$  then
7:     Choose one such request  $(o, b)$  from  $S_j$  and serve it.
8:      $t := t - 1$ 
9:     Remove  $(o, b)$  from  $S$  and update the sets  $S_i$  for  $i = 1 \dots k$  as needed.
10:     $o := b$ 
11:   else if  $t \geq 2$  and requests remain in  $S$  then
12:     Let  $j$  be the highest value for which  $S_j$  is non-empty.
13:     Choose one request  $(a, b)$  from  $S_j$ .
14:     Drive from  $o$  to  $a$ , then serve the request  $(a, b)$ .
15:      $t := t - 2$ 
16:     Remove  $(a, b)$  from  $S$  and update the sets  $S_i$  for  $i = 1 \dots k$  as needed.
17:      $o := b$ 
18:   else                                     ▷ no requests remain (that we have enough time to serve)
19:      $t := t - 1$ 
```

► **Theorem 9.** *The k -CHAIN algorithm yields at most a $7/9$ -approximation.*

Proof. In the input instance (see Figure 2) there is a chain of $c + k$ requests, for two positive integers c and k , and the origin, o , is at the start of this chain. Denote these $c + k$ requests as $(v_0, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_{c-1}, v_c), \dots, (v_{c+k-1}, v_{c+k})$, so $o = v_0$. In addition, for each



■ **Figure 2** An instance showing that the k -CHAIN algorithm has approximation ratio at most $7/9$.

point v_i , for $i = 1, 2, \dots, c$, there is another pair of requests: one that leaves from v_i to a point not on the chain, call it v'_i , and another that leaves from v'_i and returns to v_i , forming a total of c loops each of length 2.

Let $T = 3c$. Then $\text{OPT}(S, T, o) = \text{OPT}(S, 3c, v_0) = 3c$ since OPT can serve all the loops “on the way” as it proceeds across from v_1 to v_{c+k} . I.e., the optimal schedule is

$$(v_0, v_1), (v_1, v'_1), (v'_1, v_1), (v_1, v_2), (v_2, v'_2), (v'_2, v_2), (v_2, v_3), \dots, (v_{c+k-1}, v_{c+k}).$$

On the other hand, Algorithm 2, which prioritizes chains of length k , may choose one request at a time from the “spine” of this input instance, and end up serving all the requests along the straight path first, rather than serving the loops along the way. In this event at time $c+k$ it must then go back and serve as many loops (chains of length 2) as it can in the remaining $3c - (c+k) = 2c - k$ units of time, expending one unit of time on an empty drive to the next loop after serving each loop. Hence:

$$|\text{ALG}(S, T, o)| = c + k + \lfloor \frac{2}{3}(2c - k) \rfloor$$

And note that

$$\lim_{c \rightarrow \infty} \frac{|\text{ALG}(S, T, o)|}{|\text{OPT}(S, T, o)|} = \lim_{c \rightarrow \infty} \frac{c + k + \lfloor \frac{2}{3}(2c - k) \rfloor}{3c} = \frac{7}{9}. \quad \blacktriangleleft$$

3.3 The LONGEST-CHAIN-FIRST Algorithm

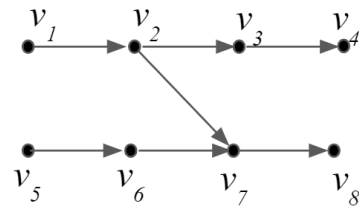
We now provide a brief discussion of the greedy algorithm that serves the longest chain of requests first, removing these requests from the instance, then serves the longest chain among the remaining requests and removes these, and continues this way until time runs out. We refer to this algorithm as the LONGEST-CHAIN-FIRST (LCF) algorithm.

Implementation of this algorithm requires a solution to the longest trail problem, where a trail is defined as a path with no repeated edges, i.e., a chain of DARP requests. Although the longest trail problem is NP-hard [10, 11], a standard poly-time algorithm that simply requires a topological sort on the vertices of the acyclic graph as a pre-processing step can be employed for finding the longest trail in acyclic graphs. We use the term *request-graph* to refer to the directed multigraph where each request is represented by an edge in the graph and each vertex in the graph is the source or destination of a request. So if we consider the space of inputs where the request-graphs are acyclic, we can employ the poly-time algorithm for finding the longest trail in an acyclic graph to implement the greedy LCF algorithm.

It turns out that even when restricting to acyclic graphs, uniform revenues and a uniform metric space, the LCF algorithm yields an approximation ratio of at most $5/6$.

► **Theorem 10.** *The approximation ratio of the LCF algorithm for URDARP on acyclic request-graphs is at most $5/6$.*

11:12 Maximizing the Number of Rides for DARP



■ **Figure 3** An instance showing that the LCF algorithm has an approximation ratio of at most $5/6$.

Proof. Please refer to Figure 3. The instance depicts a request graph for which $T = 8$ and the origin is one unit away from the source of all requests. An optimal solution is to serve the top 3-chain followed by the bottom 3-chain for a total revenue of 6. The LCF algorithm may instead start with (v_1, v_2) , but then take (v_2, v_7) , finishing with (v_7, v_8) . LCF would then require an empty drive to a remaining 2-chain, but after serving the 2-chain, there would be no time left to drive to and serve any more requests, so LCF earns a revenue of only 5. ◀

We expect that in future work we will be able to prove that LCF does indeed yield a $5/6$ approximation for URDARP on acyclic request graphs.

References

- 1 Giorgio Ausiello, Vincenzo Bonifaci, and Luigi Laura. The online prize-collecting traveling salesman problem. *Information Processing Letters*, 107(6):199–204, 2008.
- 2 Baruch Awerbuch, Yossi Azar, Avrim Blum, and Santosh Vempala. New approximation guarantees for minimum-weight k -trees and prize-collecting salesmen. *SIAM Journal on Computing*, 28(1):254–262, 1998.
- 3 Egon Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.
- 4 Daniel Bienstock, Michel X Goemans, David Simchi-Levi, and David Williamson. A note on the prize collecting traveling salesman problem. *Mathematical programming*, 59(1-3):413–420, 1993.
- 5 Avrim Blum, Shuchi Chawla, David R Karger, Terran Lane, Adam Meyerson, and Maria Minkoff. Approximation algorithms for orienteering and discounted-reward TSP. *SIAM Journal on Computing*, 37(2):653–670, 2007.
- 6 Ananya Christman, Christine Chung, Nicholas Jaczko, Marina Milan, Anna Vasilchenko, and Scott Westvold. Revenue Maximization in Online Dial-A-Ride. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59, pages 1:1–1:15, Dagstuhl, Germany, 2017. doi:10.4230/OASIcs.ATMOS.2017.1.
- 7 Ananya Christman, William Forcier, and Aayam Poudel. From theory to practice: maximizing revenues for on-line dial-a-ride. *Journal of Combinatorial Optimization*, 35(2):512–529, 2018.
- 8 Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*, 153(1):29–46, September 2007. doi:10.1007/s10479-007-0170-8.
- 9 Yves Molenbruch, Kris Braekers, and An Caris. Typology and literature review for dial-a-ride problems. *Annals of Operations Research*, 259(1):295–325, 2017.
- 10 Christos H Papadimitriou and Umesh V Vazirani. On two geometric problems related to the travelling salesman problem. *Journal of Algorithms*, 5(2):231–246, 1984.
- 11 stackexchange.com. Is the longest trail problem easier than the longest path problem? <https://cstheory.stackexchange.com/questions/20682/is-the-longest-trail-problem-easier-than-the-longest-path-problem>. Accessed: 2019-02-19.

A Appendix

We first show that in all subcases of Case 2.2.2.1 of Lemma 6, P starts at o_{new} , has at least $m - r - k + 1$ requests from S_{new} , and has length T_{new} .

Case 2.2.2.1: If $k = r$ then every request (u_{i-1}, u_i) is in P^* , and in particular both (u_{k-2}, u_{k-1}) and (u_{k-1}, u_k) are in P^* . Either (u_{k-1}, u_k) is the last drive of P^* or it is not.

Case 2.2.2.1.1: (u_{k-1}, u_k) is the last drive of P^* . Then there is a drive (u_{k-1}, y) immediately following (u_{k-2}, u_{k-1}) in P^* . There are three subcases.

Case 2.2.2.1.1.1: $(u_{k-1}, y) = (u_{k-1}, u_k)$ Then we delete the last $k + 1$ drives from P^* to make P . Since the $k + 1$ drives include (u_{k-2}, u_{k-1}) and (u_{k-1}, u_k) , which are not in S_{new} , then P loses at most $k - 1$ requests from S_{new} and is a total $k + 1$ shorter than P^* . So P has at least $m - r - (k - 1) = m - r - k + 1$ requests from S_{new} and has length $|P^*| - (k + 1) = T_{new}$.

Case 2.2.2.1.1.2: $(u_{k-1}, y) \neq (u_{k-1}, u_k)$ and $(u_{k-1}, y) \notin S_{new}$. We make P from P^* as follows. We first make \hat{P} from P^* by deleting (u_{k-1}, u_k) and replacing (u_{k-2}, u_{k-1}) and (u_{k-1}, y) by the shortcut (u_{k-2}, y) . So \hat{P} has at least $m - r$ requests from S_{new} (since none of the deleted requests are requests from S_{new}) and length two shorter than P^* . Then we make P from \hat{P} by deleting the last $(k - 1)$ drives from \hat{P} . So P has at least $m - r - (k - 1)$ requests from S_{new} and length T_{new} .

Case 2.2.2.1.1.3: $(u_{k-1}, y) \neq (u_{k-1}, u_k)$ and $(u_{k-1}, y) \in S_{new}$. Note that u_{k-1} is not the source of a request in S'_{new} (those requests in S_{new} that start a 2-chain, as defined in line 3 of Algorithm 1) since if it were, TWOCHAIN would have chosen that request instead of (u_{k-1}, u_k) as the next request. So y cannot be the source of a request in S_{new} . Let (y, z) be the next drive in P^* after (u_{k-1}, y) , and we know (y, z) is not in S_{new} . Then we make P from P^* as follows. We first make \hat{P} from P^* by deleting (u_{k-1}, u_k) and replacing (u_{k-2}, u_{k-1}) , (u_{k-1}, y) and (y, z) by the shortcut (u_{k-2}, z) . Then \hat{P} has at least $m - r - 1$ requests from S_{new} (since the only request from S_{new} \hat{P} lost was (u_{k-1}, y)) and length three shorter than P^* . We then make P from \hat{P} by deleting the last $k - 2$ drives from \hat{P} . So P has at least $m - r - 1 - (k - 2) = m - r - k + 1$ requests from S_{new} and length $|P^*| - 3 - (k - 2) = T_{new}$.

Case 2.2.2.1.2: (u_{k-1}, u_k) is not the last drive of P^* so there is a drive (u_k, x) in P^* . Note that (u_k, x) cannot be in S_{new} since if it were, then TWOCHAIN would have served it after (u_{k-1}, u_k) . There are several subcases.

Case 2.2.2.1.2.1: (u_{k-2}, u_{k-1}) is the last drive of P^* and $(u_{k-2}, u_{k-1}) = (u_k, x)$. Then we make P by deleting the last $k + 1$ drives from P^* . Then P loses at most $k - 1$ requests from S_{new} since at least 2 of the $k + 1$ drives (namely, (u_{k-2}, u_{k-1}) and (u_{k-1}, u_k)) are not in S_{new} . So P has at least $m - r - (k - 1)$ requests from S_{new} and length T_{new} .

Case 2.2.2.1.2.2: (u_{k-2}, u_{k-1}) is the last drive of P^* and $(u_{k-2}, u_{k-1}) \neq (u_k, x)$. Then we make P from P^* as follows. We first make \hat{P} from P^* by deleting (u_{k-2}, u_{k-1}) and replacing (u_{k-1}, u_k) and (u_k, x) by the shortcut (u_{k-1}, x) . So \hat{P} has at least $m - r$ requests from S_{new} (since none of (u_{k-2}, u_{k-1}) , (u_{k-1}, u_k) , and (u_k, x) are in S_{new}) and has length two shorter than P^* . We then make P from \hat{P} by deleting the last $k - 1$ drives from \hat{P} . So P has at least $m - r - (k - 1)$ requests from S_{new} and length $|\hat{P}| - (k - 1) = |P^*| - 2 - k + 1 = T_{new}$.

- Case 2.2.2.1.2.3:** (u_{k-2}, u_{k-1}) is not the last drive of P^* so there is a drive (u_{k-1}, y) in P^* and $(u_{k-1}, y) = (u_{k-1}, u_k)$. Then we make \hat{P} from P^* by replacing (u_{k-2}, u_{k-1}) , (u_{k-1}, u_k) and (u_k, x) by the shortcut (u_{k-2}, x) . So \hat{P} has at least $m - r$ requests from S_{new} and length two shorter than P^* . We then make P from \hat{P} by deleting the last $k - 1$ drives. So P has at least $m - r - (k - 1)$ requests from S_{new} and length T_{new} .
- Case 2.2.2.1.2.4:** (u_{k-2}, u_{k-1}) is not the last drive of P^* so there is a drive (u_{k-1}, y) in P^* and $(u_{k-1}, u_k) \neq (u_{k-1}, y)$ and $(u_{k-1}, y) \notin S_{new}$. Then we make \hat{P} from P^* by replacing (u_{k-1}, u_k) and (u_k, x) by (u_{k-1}, x) and replacing (u_{k-2}, u_{k-1}) and (u_{k-1}, y) by (u_{k-2}, y) . So \hat{P} has at least $m - r$ requests from S_{new} (since none of (u_{k-1}, u_k) , (u_k, x) , (u_{k-2}, u_{k-1}) and (u_{k-1}, y) are in S_{new}) and length two shorter than P^* . Then we make P from \hat{P} by deleting the last $k - 1$ drives. So P has at least $m - r - (k - 1)$ requests from S_{new} and length T_{new} .
- Case 2.2.2.1.2.5:** (u_{k-2}, u_{k-1}) is not the last drive of P^* so there is a drive (u_{k-1}, y) in P^* and $(u_{k-1}, u_k) \neq (u_k, y)$ and $(u_{k-1}, y) \in S_{new}$. Note that u_{k-1} is not the beginning of a request in S'_{new} , (since if it were, TWOCHAIN would have chosen that request instead of (u_{k-1}, u_k) as the next request. Thus y cannot be the start of a request in S_{new} . Either (u_{k-1}, y) is at the end of P^* or let (y, z) denote the next drive in P^* after (u_{k-1}, y) and observe that (y, z) is not in S_{new} . There are three subcases.
- Case 2.2.2.1.2.5.1:** (u_{k-1}, y) is the last drive of P^* . Then we make \hat{P} from P^* by replacing (u_{k-1}, u_k) and (u_k, x) by (u_{k-1}, x) and deleting (u_{k-2}, u_{k-1}) and (u_{k-1}, y) . So \hat{P} has at least $m - r - 1$ requests from S_{new} (since the only S_{new} request \hat{P} lost was (u_{k-1}, y)) and length three shorter than P^* . Then we make P from \hat{P} by deleting the last $k - 2$ drives from \hat{P} . So P has at least $m - r - 1 - (k - 2) = m - r - k + 1$ requests from S_{new} and length $T_{new} = |P^*| - 3 - (k - 2)$ shorter than P^* .
- Case 2.2.2.1.2.5.2:** (u_{k-1}, y) is not the last drive of P^* so there is a drive (y, z) and $(y, z) = (u_{k-1}, u_k)$. Then to make \hat{P} from P^* we replace (u_{k-2}, u_{k-1}) , (u_{k-1}, y) , (y, z) , (u_k, x) by the shortcut (u_{k-2}, x) . So \hat{P} has at least $m - r - 1$ S_{new} drives (since the only S_{new} drive \hat{P} lost was (u_{k-1}, y)) and length three shorter than P^* . Then we make P from \hat{P} by deleting the last $k - 2$ drives from \hat{P} . So P has at least $m - r - 1 - (k - 2) = m - r - k + 1$ S_{new} drives and length $T_{new} = |P^*| - 3 - (k - 2)$ shorter than P^* .
- Case 2.2.2.1.2.5.3:** (u_{k-1}, y) is not the last drive of P^* so there is a drive (y, z) and $(y, z) \neq (u_{k-1}, u_k)$. Then we make \hat{P} from P^* by replacing (u_{k-1}, u_k) , (u_k, x) by (u_{k-1}, x) and replacing (u_{k-2}, u_{k-1}) , (u_{k-1}, y) and (y, z) by (u_{k-2}, z) . So \hat{P} has at least $m - r - 1$ S_{new} drives (since the only request from S_{new} \hat{P} lost was (u_{k-1}, y)) and length three shorter than P^* . Then we make P from \hat{P} by deleting the last $k - 2$ drives from \hat{P} . So P has at least $m - r - 1 - (k - 2) = m - r - k + 1$ requests from S_{new} and length $T_{new} = |P^*| - 3 - (k - 2)$ shorter than P^* .

This concludes all the subcases of Case 2.2.2.1 of Lemma 6. We now show that in all subcases of Case 2.2.2.2 of Lemma 6, P starts at o_{new} and has at least $m - r - k$ requests from S_{new} .

- Case 2.2.2.2:** $k \neq r$ so it must be that $k = r + 1$. So all but one (u_{i-1}, u_i) from P is in P^* , thus at least one of (u_{k-2}, u_{k-1}) and (u_{k-1}, u_k) is in P^* .
- Case 2.2.2.2.1:** (u_{k-1}, u_k) is in P^* . There are two subcases.
- Case 2.2.2.2.1.1:** (u_{k-1}, u_k) is at the end of P^* . Then to make P from P^* we delete the last $k + 1$ drives (which include (u_{k-1}, u_k)). So we deleted at most k requests from S_{new} from P^* (since (u_{k-1}, u_k) is not in S_{new}). So P has at least $m - r - k$ requests from S_{new} and length T_{new} .

Case 2.2.2.2.1.2: (u_{k-1}, u_k) is not at the end of P^* . Then there is a next drive (u_k, x) in P^* . So we first make \hat{P} from P^* by replacing (u_{k-1}, u_k) and (u_k, x) by the shortcut (u_{k-1}, x) . So \hat{P} has length one shorter than P^* . Note that (u_k, x) cannot be in S_{new} since otherwise TWOCHAIN would have continued with a request after (u_{k-1}, u_k) . So \hat{P} has at least $m - r$ requests from S_{new} . Now we make P by deleting the last k drives from \hat{P} . So P has at least $m - r - k$ requests from S_{new} and length $T_{new} = |P^*| - 1 - k$.

Case 2.2.2.2.2: (u_{k-1}, u_k) is not in P^* , and therefore (u_{k-2}, u_{k-1}) is in P^* . There are several subcases.

Case 2.2.2.2.2.1: (u_{k-2}, u_{k-1}) is at the end of P^* . Then to make P from P^* we delete the last $k + 1$ drives (which include (u_{k-2}, u_{k-1})). So we deleted at most k requests from S_{new} from P^* (since (u_{k-2}, u_{k-1}) is not in S_{new}). So P has at least $m - r - k$ requests from S_{new} and length T_{new} .

Case 2.2.2.2.2.2: (u_{k-2}, u_{k-1}) is not at the end of P^* and has a next drive (u_{k-1}, y) that is not in S_{new} . We first make \hat{P} from P^* by replacing (u_{k-2}, u_{k-1}) and (u_{k-1}, y) by the shortcut (u_{k-2}, y) . So \hat{P} has at least $m - r$ requests from S_{new} and is one shorter than P^* . We then make P from \hat{P} by deleting the last k drives from \hat{P} . So P has at least $m - r - k$ requests from S_{new} and length T_{new} .

Case 2.2.2.2.2.3: (u_{k-2}, u_{k-1}) is not at the end of P^* and has a next drive (u_{k-1}, y) that is in S_{new} and (u_{k-1}, y) is at the end of P^* . We first make \hat{P} from P^* by deleting (u_{k-2}, u_{k-1}) and (u_{k-1}, y) . So \hat{P} has at least $m - r - 1$ requests from S_{new} (since the only request from S_{new} \hat{P} lost was (u_{k-1}, y)) and length two shorter than P^* . We then make P from \hat{P} by deleting the last $k - 1$ drives from \hat{P} . So P has at least $m - r - 1 - (k - 1) = m - r - k$ requests from S_{new} and length $T_{new} = |P^*| - 2 - (k - 1)$.

Case 2.2.2.2.3.4: (u_{k-2}, u_{k-1}) is not at the end of P^* so there is a next drive (u_{k-1}, y) that is in S_{new} and (u_{k-1}, y) is not at the end of P^* so there is a next drive (y, z) in P^* . Then by the same reasoning as in subcase 2.2.2.1.2.5, we have that (y, z) is not in S_{new} . To make \hat{P} from P^* we replace (u_{k-2}, u_{k-1}) , (u_{k-1}, y) and (y, z) by the shortcut (u_{k-2}, z) . So \hat{P} has at least $m - r - 1$ requests from S_{new} (since the only request from S_{new} \hat{P} lost was (u_{k-1}, y)) and length two shorter than P^* . We then make P from \hat{P} by deleting the last $k - 1$ drives from \hat{P} . So P has at least $m - r - 1 - (k - 1) = m - r - k$ requests from S_{new} and length $T_{new} = |P^*| - 2 - (k - 1)$.

This concludes all the subcases of Case 2.2.2.2 of Lemma 6.

A Graph- and Monoid-Based Framework for Price-Sensitive Routing in Local Public Transportation Networks

Ricardo Euler 

Zuse Institute Berlin, Germany
euler@zib.de

Ralf Borndörfer

Zuse Institute Berlin, Germany
borndorfer@zib.de

Abstract

We present a novel framework to mathematically describe the fare systems of local public transit companies. The model allows the computation of a provably cheapest itinerary even if prices depend on a number of parameters and non-linear conditions. Our approach is based on a *ticket graph* model to represent tickets and their relation to each other. Transitions between tickets are modeled via transition functions over partially ordered monoids and a set of symbols representing special properties of fares (e.g. surcharges). Shortest path algorithms rely on the subpath optimality property. This property is usually lost when dealing with complicated fare systems. We restore it by relaxing domination rules for tickets depending on the structure of the ticket graph. An exemplary model for the fare system of Mitteldeutsche Verkehrsbetriebe (MDV) is provided. By integrating our framework in the multi-criteria RAPTOR algorithm we provide a price-sensitive algorithm for the earliest arrival problem and assess its performance on data obtained from MDV. We discuss three preprocessing techniques that improve run times enough to make the algorithm applicable for real-time queries.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Applied computing → Transportation

Keywords and phrases shortest path, public transit, optimization, price-sensitive, raptor, fare, operations research

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.12

Funding Our research was supported by the Federal Ministry of Transport and Digital Infrastructure (BMVI) under the project no. 19E17001C.

Acknowledgements We thank Niels Lindner and Pedro Maristany de las Casas for many fruitful discussions on the subject as well as MDV and InfraDialog GmbH for providing the data for this study.

1 Introduction

Recent progress in the field of routing algorithms for public transportation has led to several very fast algorithms [2]. Usually, these algorithms determine the best itinerary with respect to travel time in mere milliseconds. This led to the desire to optimize additional criteria such as the number of transfers or the reliability of the connection. For most users of public transportation systems the price of a journey is one of the most important criteria for assessing its quality. Unfortunately, public transportation fare systems are notoriously complex and therefore algorithmically hard to deal with. The ticket price can depend on a variety of variables such as the set of fare zones, the distance traveled, the number of stops visited, surcharges for night buses or ferries, etc. To reduce this complexity, previous research



© Ricardo Euler and Ralf Borndörfer;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 12; pp. 12:1–12:15

OpenAccess Series in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

has usually focused only on specific aspects such as zone- or distance-based prices and/or dealt with them heuristically. In this study, we present a novel and flexible framework to model the price structures of (regional) public transportation companies that is able to take all of the aforementioned criteria into account. The framework can be used to compute price-optimal journeys by applying typical multi-criteria shortest path algorithms such as RAPTOR or Dijkstra. When comparing itineraries by price, the subpath optimality principle is usually lost. An example could be taking a detour into a new fare zone to avoid paying a special connections surcharge (e.g. for using ferries). It is possible that, at a later point in the journey, the surcharge has to be paid anyway (e.g. the target station can only be reached via a ferry). In that case, the detour was a suboptimal decision. To avoid this problem, we base dominance relations between labels not on price, but on paths in a directed *ticket graph* modeling the relations between tickets. Transitions between different tickets are modeled as directed arcs and usually depend on a number of additional *fare attributes* such as fare zones or the distance traveled. We model these fare attributes as (positive) partially ordered monoids.

1.1 Related Literature

For an exhaustive overview of shortest path algorithms in road and public transportation networks please refer to [2]. It has previously been observed that shortest path problems can be generalized to ordered monoids [10] and semirings [8]. In the study of public transit routing, prices are taken into account to varying degrees. Müller-Hannemann and Schnee study fare systems that entail distance- and relation-based prices [9], i.e., fare systems that in Germany are usually associated with long-distance public transportation. They approximate fares by assigning a fixed price to every edge. Their approach, however, does not take into account fares based on fare zones and short-distance city tickets. Both of these are usually more prominent in local public transportation. Delling et al. [6][5] use their RAPTOR algorithm to compute itineraries that touch the smallest number of fare zones. The idea of restoring subpath optimality by relaxing rules for label domination is discussed in a different context in [3]. The ticket graph approach bears similarity to the finite automata used to find language-constrained shortest paths (Barrett et al [1]). It is different, however, in that it serves to evaluate paths instead of restricting the set of feasible paths. Furthermore, our approach also covers fares based on numerical attributes that are not expressed as part of a formal languages. We originally presented the idea of using a ticket graph in [4, German language].

1.2 Our Contribution

In this paper, we make two novel contributions. The first is a framework based on graphs and monoids to mathematically model public transportation fare systems. The aspects of fare systems that can be modeled include (but are not limited to): zone-based fares, distance-based fares, surcharges for special vehicles or daytime, and discounted short-distance tickets that do not allow transfers. Our second contribution is a formal definition of label domination rules for public transportation tickets while retaining the subpath optimality property. We prove that these rules do in fact yield lowest-price itineraries.

1.3 Overview

In Chapter 2, we present a framework for public transportation fare systems and show how it can be used to model several aspects of fare systems. The algorithmic treatment of fares is laid out in Chapter 3. Chapter 4 discusses how our framework can be used in the

context of the RAPTOR algorithm. We propose three different speed-up techniques and an evaluation of the framework's performance for the network of MDV, a public transit company in Saxony, Germany. Chapter 5 concludes the paper with some final remarks. The proofs of all propositions can be found in the appendix.

2 A Formal Framework for Fare Systems

We are given a (directed) routing graph $G = (V, A)$, in which arcs represent either public transport connections, footpaths or transfers between lines and/or modes of transportation. Every path p in G is associated with a ticket t and every ticket has a corresponding price $\pi(t) \in \mathbb{Q}^+$. In order to efficiently compute cheapest paths, we need a model that is able to locally describe the development of a path's ticket once arcs are added to it as well as a provably correct way of comparing those tickets. This is done in the following way: We denote the set of all available tickets by T and define a *ticket graph* $\mathcal{F} = (T, E)$ where an edge $e \in E$ models how the ticket changes when following an arc in the routing graph. Each edge in E carries a Boolean function determining the conditions under which the path transitions to a different ticket. For example, when visiting the fifth stop on a path, a short-distance ticket t_0 could be lost and a more expensive ticket t_1 would be applicable. The edge (t_0, t_1) would then carry a condition on the number of stops visited.

The transition along an edge $e \in E$ usually depends on multiple factors, e.g., the set of fare zones visited thus far, the distance traveled (in meters), surcharges for special trains, etc. These factors are picked up when relaxing an arc in the routing graph. We generalize them in two kinds of mathematical objects: abstract symbols from a symbol set S and elements of a partially ordered positive monoid $(H, +, \leq)$.

► **Definition 1** (Partially ordered monoid). *Let $(H, +)$ be a monoid and let \leq be a partial order on H that is translation-invariant with respect to the monoid operation $+$, i.e., $h_1 \leq h_2 \Rightarrow h_1 + x \leq h_2 + x \forall h_1, h_2, x \in H$. We call $(H, +, \leq)$ a partially ordered monoid. We call $(H, +, \leq)$ positive, if $e \leq h \forall h \in H$ where e is the neutral element of $(H, +)$.*

A common example for fare systems is the power set 2^Z of a set of fare zones Z combined with k numerical fare attributes in \mathbb{Q}^k . In this case, $H = 2^Z \times \mathbb{Q}^k$. For $h_1 = (z_1, r_1)$, $h_2 = (z_2, r_2) \in H$, we define $h_1 + h_2 = (z_1 \cup z_2, r_1 + r_2)$ and $h_1 \leq h_2$ if and only if $z_1 \subseteq z_2$ and $r_1 \leq r_2$. Translation invariance in $(H, +, \leq)$ is inherited from the translation invariance in $(2^Z, \cup, \subseteq)$ and $(\mathbb{Q}^k, +, \leq)$.

Every arc $a \in A$ of G is annotated with a *fare attribute* $\mathcal{A}(a) = (\mathcal{A}^s(a), \mathcal{A}^h(a)) \in S \times H$ that is picked up when relaxing the arc. Every vertex $v \in V$ is annotated with a *start state* $\mathcal{S}(v) = (\mathcal{S}^t(v), \mathcal{S}^h(v)) \in T \times H$ giving an initial ticket and element from the monoid.

Using this notation we can define the *fare state* of a path.

► **Definition 2** (Fare State). *We call an element $f \in T \times H$ a fare state. The set of all fare states is denoted by $F := T \times H$. We use f^t and f^h to denote its components.*

A fare state contains all information necessary for calculating the price of a journey (the ticket associated with the path) as well as all information necessary to decide domination between paths. We now want to enable the tracking of the fare states along paths in G . To do so, we formalize the notion of the *fare transition function* on arcs in the ticket graph. The definition is intentionally left rather general to capture a large number of possible ticketing conditions.

12:4 Price-Sensitive Routing in PT Networks

► **Definition 3** (Fare Transition Function). *We call a function $Tr : E \times S \times H \rightarrow \{0, 1\}$ a fare transition function for the ticket graph \mathcal{F} if*

$$\forall e = (t_1, t_2) \in E : \forall s \in S : \forall h \in H : \sum_{e \in \delta^+(t_1)} Tr(e, s, h) \in \{0, 1\}.$$

Hence, a fare transition function allows only one well-defined transition from any fare state $f \in F$. We use the notion of fare transition functions to define the update of tickets when relaxing an arc of the routing graph.

► **Definition 4** (Ticket Update Function). *We introduce a ticket update function $Up : F \times A \rightarrow F$ in the following way. Let $f = (f^t, f^h) \in F, a \in A$. Then, we define $\tilde{f} = Up(f)$ by*

$$\tilde{f}^h := f^h + \mathcal{A}^h(a) \tag{1}$$

$$\tilde{f}^t := \begin{cases} \text{head}(e) & \text{if } \exists e \in \delta^+(f^t) \text{ with } Tr(e, \mathcal{A}^s(a), \tilde{f}^h) = 1 \\ f^t & \text{otherwise.} \end{cases} \tag{2}$$

We now have a tool at our disposal to track the tickets along a path in G in the ticket graph \mathcal{F} . Each path in G can be associated with a sequence of fare states in F .

► **Definition 5** (Path-Induced Fare Sequence). *We call a sequence of fare states (f_1, \dots, f_n) path-induced if there is a path $p = (v_1, \dots, v_n)$ with the following properties:*

1. $f_1 = \mathcal{S}(v_1)$
2. $f_i = Up(f_{i-1}, (v_{i-1}, v_i)) \forall i = 2, \dots, n$

We call the fare state $f(p) := f_n$ the fare state of the path p .

Combining all the above definitions we arrive at the notion of *conditional fare networks* which can precisely capture a fare system.

► **Definition 6** (Conditional Fare Network). *Let $G = (V, A)$ be a routing graph and let the following be given:*

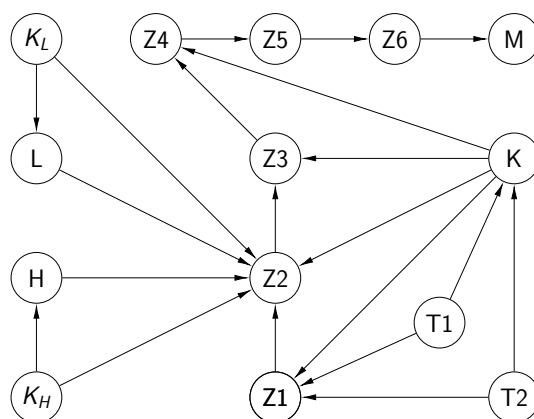
1. *the space of fare attributes $S \times H$ as product of a set of symbols S and a partially ordered, positive monoid $(H, +, \leq)$,*
2. *a set of tickets T ,*
3. *a cycle-free ticket graph $\mathcal{F} = (T, E)$ with transition function $Tr : E \times S \times H \rightarrow \{0, 1\}$,*
4. *arc attributes $\mathcal{A}(a) \in S \times H \quad \forall a \in A$,*
5. *start fare states $\mathcal{S}(v) \in T \times H \quad \forall v \in V$ and*
6. *a price function $\pi : T \rightarrow \mathbb{Q}_+$ that is monotonously non-decreasing along directed paths in \mathcal{F} , i.e., if there is a directed $t_1 - t_2$ -path in \mathcal{F} for $t_1, t_2 \in T$, then $\pi(t_1) \leq \pi(t_2)$.*

We call the six-tuple $(\mathcal{F}, \mathcal{A}, \mathcal{S}, \mathcal{S}, Tr, \pi)$ a conditional fare network \mathcal{N} of G .

Note that cycle-freeness in \mathcal{F} and the monotonicity condition on π as well as the positivity of H ensure that no price-decreasing cycles exist in G . We consider those assumptions natural enough that most reasonable price system should satisfy them.

Soon, we will see that dominance relations between paths need to be based on their fare states instead of their price. Thus, we can drop the monotonicity condition on π while still retaining optimality. In this case, however, price-based target pruning (confer Section 4.2), which proved essential in ensuring acceptable performance for our shortest path search, cannot be applied.

Having introduced conditional fare networks, we now define the price-sensitive earliest arrival problem.



■ **Figure 1** Ticket graph for MDV. There is a total number of fourteen different tickets.

► **Definition 7** (Price-Sensitive Earliest Arrival Problem). *Let a public transportation network be given as a graph $G = (V, A)$ and let $(\mathcal{F}, \mathcal{A}, \mathcal{S}, S, Tr, \pi)$ be a conditional fare network of G . Let for all $a \in A$ a time-dependent FIFO travel time function $c(a) : I \rightarrow I$ be given, where I is the set of time points. Finally, let $P_{s,t}$ be the set of all s, t -paths in G for some $s, t \in V$. Then, the price-sensitive earliest arrival problem (PSEAP) is defined as finding a Pareto-set of s, t -paths $P_{s,t}^* \subset P_{s,t}$ in G , such that*

$$\forall p^* \in P_{s,t}^* \nexists p \in P_{s,t} : \pi(p) \leq \pi(p^*) \wedge c(p) \leq c(p^*) \wedge (\pi(p) < \pi(p^*) \vee c(p) < c(p^*)). \quad (3)$$

In our following theoretical discussion, we will ignore the arrival time aspect of PSEAP and focus only on the fare framework. The correctness results carry over to the full version of PSEAP.

► **Example 8** (The Fare System of MDV). The fare systems of MDV divides its operational area into 67 fare zones Z . A ticket Z_i is applicable if the itinerary touches i zones from Z . Once a total of seven fare zones have been touched a maximum price M is reached that does not change. Two fare zones, Halle and Leipzig, are cities and are more expensive than other zones. They offer special tickets H and L . They, however, count as normal zones for all itineraries that pass through multiple fare zones. Several smaller cities are part of larger fare zones, but allow for discounted tickets (T_1 and T_2) when traveling in the city only. They do not count as fare zones of their own. For Halle and Leipzig there are discounted tickets for short trips (K_H and K_L), which do not allow for transfers and are valid for a maximum number of four stations on the itinerary. Discounted tickets exist also for other zones (K), but these are cheaper and depend on the length of the itinerary (4 km maximum). Figure 1 depicts the associated ticket graph \mathcal{F} . The monoid $(H, +_H, \leq_H)$ is defined by $H := (2^Z, \mathbb{N}^2, \{0, 1\})$, where Z captures the fare zones, \mathbb{N}^2 the distance traveled in meters and the number of stations and $\{0, 1\}$ the existence of transfers. Addition $+_H$ and partial order \leq_H are induced from the respective operations in 2^Z and \mathbb{N}^2 and the logical or \vee on $\{0, 1\}$. Note that there is no connection between, e.g., Z_2 and Z_4 . This is due to the fact that we can touch only one fare zone at any station. Also, there are no discounted trips that cover more than three fare zones and hence there is no arc between K and Z_5 . This highlights that the structure of the routing graph influences the structure of the ticket graph.

A list of all fare transition functions can be found in the appendix.

Transfer Penalties, Footpaths and Surcharges

Footpaths are modeled as arcs with the arc attribute (S_0, e) , where $S_0 \in S$ is a symbol that cannot activate a ticket transition and $e \in H$ is the neutral element of the monoid. The fare attribute is set to e so as to not modify the current fare state. The transition from a footpath to a public transportation vehicle requires some care. Assume we walk from station v_0 to v_1 along arc $a_0 = (v_0, v_1)$ to take a vehicle along $a_1 = (v_1, v_2)$ to reach v_2 . Some fare systems use the number of stations a path touches to calculate prices. Here, this number would obviously be two. Counting a station when relaxing a_0 is a mistake if the optimal journey would be to continue on foot. Counting both v_1 and v_2 when relaxing a_1 is also wrong since this would overcount the number of stations for every itinerary that reaches v_1 via a vehicle. Hence, the graph model needs to be extended by splitting up stations into vertices for every route and a vertex that is connected to footpaths. These vertices are then connected via transfer arcs and boarding arcs. We can also have arc attributes different from (S_0, e) on transfer arcs. This allows us to make the applicable ticket dependent on the number of transfers. Arc attributes on arcs representing boarding can be used to model surcharges for the route boarded. For more details on how to build these expanded graphs, we refer to [7].

Neutral Zones

Some fare systems that are based on fare zones contain neutral zones. Stations in a neutral zone can be counted as part of either of its neighboring zones, whichever is cheapest for the costumer. This is meant to mitigate sharp price increases at fare zone borders. MDV uses them as well as several other German railway companies (e.g. Verkehrsverbund Bremen/Niedersachsen GmbH). Neutral zones can be incorporated by label duplication: Assume a neutral zone neighbors n fare zones. We associate each arc a whose $head(a)$ represents a station in the neutral zone with n fare attributes, one for each fare zone it could possibly be part of. When settling the vertex in a shortest path search, the current fare state is updated once for each fare attribute thereby creating n new labels.

3 The Fare Framework in Routing Algorithms

Classical shortest path algorithms rely on dynamic programming and the subpath optimality condition [3]. That is, every subpath of an optimal s, t -path is in itself an optimal path. When comparing paths in G naively by means of the price function π , the subpath optimality condition is usually violated. Think about taking a local detour to avoid a fare zone: Later on, travelers may be forced to cross the zone due to the infrastructure, turning the locally dominant detour into a suboptimal choice. On the other hand, a locally dominated subpath might still lead to an optimal s, t -path. This type of problem persists in our framework: the transition between tickets depends on the fare attributes already collected, but also on the structure of the reachable ticket graph and the transition functions of reachable fare arcs. Example 9 highlights that problems can already arise even with simple examples of ticket graphs.

► **Example 9** (Label Dominance in Figure 2). Consider the routing graph **(a)** together with the conditional fare network **(b)**. Examining the paths $p_1 = (v_1, v_2, v_4)$ and $p_2 = (v_1, v_3, v_4)$, we find their respective fare states are $f(p_1) = (B, 1)$ and $f(p_2) = (D, 2)$. Extending them by v_5 to p'_1 and p'_2 yields $f(p_1) = (C, 3)$ and $f(p_2) = (E, 4)$. Comparing fare states by price would indicate that p_1 could be cut off at v_4 since $\pi(B) > \pi(D)$. This is a suboptimal choice

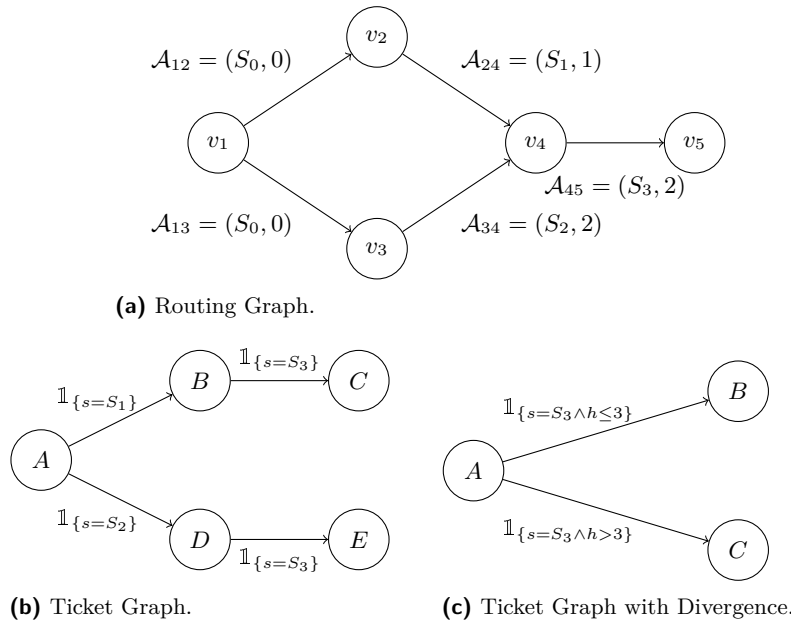


Figure 2 Example of a routing graph (a) with two possible conditional fare networks (b) and (c). For both, the underlying partially ordered monoid is $(\mathbb{R}, +, \leq)$, the symbol set is $S = \{S_0, S_1, S_2, S_3\}$ and the start state for all vertices v_i with $i = 1, \dots, 5$ is $\mathcal{S}(v_i) = (A, 0)$. We give prices for the tickets as $\pi(A) = 0, \pi(B) = 2, \pi(C) = 3, \pi(D) = 1$ and $\pi(E) = 5$. Transition functions are displayed as indicator functions on fare arcs. Using the ticket graph (b), the upper v_1, v_5 -path yields ticket C, while the lower path yields ticket E. Using ticket graph (c), the upper path yields ticket B, the lower path yields ticket C.

as p'_1 dominates p'_2 since $\pi(C) < \pi(E)$. Hence, price cannot be used as dominance criterion for fare states. A natural alternative would be to use the partial order defined by paths in the ticket graph, instead. A ticket t_1 then dominates a ticket t_2 if there is a t_1, t_2 -path. This would render the tickets B and D and the tickets C and E mutually incomparable. The idea, however, comes with problems of its own. To see this, consider now the conditional fare network (c). At v_4 , we have $f(p_1) = (A, 1)$ and $f(p_2) = (A, 2)$ and hence both paths are equivalent and it would be sensible to keep only one of them based on the relation between $f^h(p_1)$ and $f^h(p_2)$. By relaxing (v_4, v_5) , we obtain $f(p'_1) = (B, 3)$ and $f(p'_2) = (C, 4)$, which are incomparable, i.e., the fare states of p'_1 and p'_2 diverged from comparable to incomparable. Consequently, any dominance ruling cutting off either p_1 or p_2 would be defective.

To mitigate these and similar problems, we might assume a general incomparability of fare states. This comes down to enumerating all s, t -paths and simply comparing them by price. However, in a sensibly designed fare system it is usually clear which ticket is better and taking a cheaper subpath should usually not turn out more expensive overall. In the remainder of this chapter, we propose a more tailored approach. It generally bases domination rules on path relationships but adds exceptions to cover cases in which it is not safe to do so.

3.1 Dominance for Fare States

We want to define a comparison operator for fare states that restores subpath optimality while not relaxing dominance too strongly.

To do so, we partition the ticket set T into three disjoint *comparability groups*: C_F (full comparability), C_P (partial comparability), C_N (no comparability). Based on the partition $C = (C_F, C_P, C_N)$, we define a comparison operator for fare states.

► **Definition 10** (Comparability of Fare States). *Let $f_1 = (t_1, h_1)$, $f_2 = (t_2, h_2)$ be fare states. We say $f_1 \leq_C f_2$ if and only if $t_1 \notin C_N$, $h_1 \leq h_2$ and*

$$t_1 = t_2 \quad \text{if } t_1 \in C_P \quad (4)$$

$$\exists t_1, t_2\text{-path in } \mathcal{F} \quad \text{if } t_1 \in C_F. \quad (5)$$

If and only if $f_1 \leq_C f_2$ and either $h_1 < h_2$ or $t_1 \neq t_2$, we say that f_1 is strictly lesser than f_2 , i.e., $f_1 <_C f_2$.

We denote by $P_{s,t}^f$ the set of all paths Pareto-optimal with respect to \leq_C , i.e.,

$$p^* \in P_{s,t}^f \Rightarrow \nexists s, t\text{-path } p : f(p) <_C f(p^*). \quad (6)$$

Note that $P_{s,t}^f$ is not equal to $P_{s,t}^*$. Proposition 16 shows that it is in fact a superset of the set of all price-optimal paths, which we denote by $P_{s,t}^\pi$. We call paths in $P_{s,t}^f$ *state-optimal* and paths in $P_{s,t}^\pi$ *price-optimal*.

The partition C has to be defined in a way that monotonicity of the update function along all arcs $a \in A$ is not violated¹, i.e.,

$$\forall f_1, f_2 \in F : f_1 \leq_C f_2 \implies \forall a \in A : Up(f_1, a) \leq_C Up(f_2, a). \quad (7)$$

This condition is enough to ensure that a weaker form of subpath optimality holds.

► **Proposition 11** (Weak Subpath Optimality). *Let $G = (V, A)$ be a routing network and $\mathcal{N} = (\mathcal{F}, \mathcal{A}, \mathcal{S}, \mathcal{S}, Tr, \pi)$ be its conditional fare network. Let $p^* \in P_{s,t}^f$ be a state-optimal s, t -path in G for some $s, t \in V$. Then, there is a path $p' = (s = v_0, v_1, \dots, v_{n-1}, v_n = t) \in P_{s,t}^f$ with $f_{p^*} = f_{p'}$, such that every subpath $p'' = (v_0, \dots, v_l), l < n$ of p' is a state-optimal v_0, v_l -path.*

Note that Proposition 11 doesn't imply that every subpath of a state-optimal path is state-optimal. It, however, implies that we can discard all state-optimal paths without this property since a path with equal fare state still remains in $P_{s,t}^f$. Hence, all classical algorithms relying on subpath-optimality can still be applied. Note also that Equation 7 need not hold for all $f_1, f_2 \in F$ but only for those that might occur on paths in G .

3.2 Comparability Partitions

In choosing C_F , C_P and C_N , there is some degree of freedom. We want C_F to be as big and C_N as small as possible while still fulfilling Equation 7. It is clear that the choice does not only depend on \mathcal{F} and Tr but also on the structure of G and its arc attributes \mathcal{A} . Choosing the partition depending on G and \mathcal{A} would require some preprocessing of G . We propose a solution that depends only on \mathcal{F} and Tr and needs less recomputation when changes in the network occur. To deal with this dependency, we use the notion of a vertex's *reach*.

► **Definition 12** (Reach). *Let $\mathcal{F} = (T, E)$ be a directed graph. We define the reach $R(t)$ of a vertex $t \in T$ as the subgraph induced by all vertices reachable from t , i.e.,*

$$R(t) := \mathcal{F}[\{\tilde{t} \in T : \exists t, \tilde{t}\text{-path}\}]. \quad (8)$$

¹ Shortest path algorithms on graphs with weights from partially ordered monoids require the monoid operation to be translation-invariant with respect to the partial order. Since the fare states and arc attributes do not belong to the same structure, the notion of translation invariance is relaxed to a monotonicity formulation.

To simplify our notation, we introduce operators that represent path relations. If there is a path in \mathcal{F} between $t_1, t_2 \in T, t_1 \neq t_2$, we write $t_1 \longrightarrow t_2$. We write $t_1 \Longrightarrow t_2$ if either $t_1 \longrightarrow t_2$ or $t_1 = t_2$.

► **Definition 13** (No-overtaking Property). *Let \mathcal{F}^* be a vertex-induced subgraph of \mathcal{F} . We say it has the no-overtaking property if for all $e = (t_1, t_2) \in \mathcal{F}^*$, $(s, h) \in S \times H$ and $\tilde{t}_1 \in T$ with $t_1 \Longrightarrow \tilde{t}_1 \Longrightarrow t_2$ the following holds:*

$$Tr(e, s, h) = 1 \Rightarrow \forall \tilde{h} \geq h \in H : \exists (\tilde{t}_1, \tilde{t}_2) \in E : Tr(\tilde{t}_1, \tilde{t}_2, s, \tilde{h}) = 1 \text{ and } t_2 \Longrightarrow \tilde{t}_2. \quad (9)$$

The no-overtaking property bears some resemblance to the FIFO (first-in, first-out) property: A worse fare state, i.e., either worse fare attributes from H or a worse ticket, cannot give rise to a better fare state when relaxing the same arc in the routing graph. Note that the condition is necessary not only for the neighbors of t but for $R(t)$.

Subgraphs with the no-overtaking property allow for the strictest domination rules. We use them as comparability group C_F .

► **Definition 14** (Comparability Partition). *Let $G = (V, A)$ be a routing network and $\mathcal{N} = (\mathcal{F}, \mathcal{A}, S, S, Tr, \pi)$ be its conditional fare network. We define*

$$C_F := \{t \in T : R(t) \text{ traceable and has the no-overtaking property}\} \quad (10)$$

$$C_P := \{t \in T \setminus C_F : \forall e \in R(t) : \forall s \in S : \forall h_1, h_2 \in H : Tr(e, s, h_1) = Tr(e, s, h_2)\} \quad (11)$$

$$C_N := \{t \in T \setminus (C_F \cup C_P)\}. \quad (12)$$

To be in the set C_F , the reach of a ticket has to be traceable, i.e., contain a Hamiltonian path. This condition is needed to avoid the divergence seen in Example 9. If a ticket has non-traceable reach it is placed in C_P . Transition functions here must be independent of H . This also ensures that comparable fare states do not diverge in an incomparable state. All remaining tickets are added to C_N . Fare states containing tickets from C_N cannot be compared at all.

The comparison operator defined by this comparability partition fulfills Equation 7.

► **Proposition 15** (Monotonicity of the Comparability Partition). *The partial order defined by Definitions 10 and 14 fulfills the monotonicity condition*

$$\forall f_1, f_2 \in F : f_1 \leq_C f_2 \implies \forall a \in A : Up(f_1, a) \leq_C Up(f_2, a). \quad (13)$$

Propositions 15 and 11 enable us to apply dynamic programming shortest path algorithms to the PSEAP using the comparability partition from Definition 14. However, we obtain only the set of state-optimal paths. It remains to show that this set contains the cheapest price itinerary.

► **Proposition 16** (Correctness). *Let $\pi^* := \min_{P_{s,t}} \pi(p)$. Then, there is at least one s, t -path p^* with $\pi^* = \pi(p^*)$ and $p^* \in P_{s,t}^f := \{\tilde{p} \text{ } s, t\text{-path} : \nexists p : f(p) <_C f(\tilde{p})\}$.*

► **Example 17** (Dominance Rules for MDV Fares). In the graph in Figure 1 all nodes have traceable reach. It is also easy to see that the no-overtaking property holds for all tickets as well and hence $C_F = \{T_1, T_2, Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, M, H, L, K_L, K_H, K\}$, $C_P = \emptyset$ and $C_N = \emptyset$. Note that changing the transition condition from Z_1 to Z_2 from $|h^z| > 1$ to $|h^z| = 1$ breaks the no-overtaking property in cases where more than one fare zone is covered by a station. Since this is never the case, we can replace the inequality by an equality while maintaining optimality.

4 Price-Sensitive RAPTOR

In this last section, we will discuss how to integrate conditional fare networks into the RAPTOR-algorithm. Finally, we introduce some speed-up techniques and evaluate their performance on real-world instances obtained from MDV. We will change notation slightly and use $P_{s,t}^*$ to refer to the Pareto-set optimized for transfers, arrival time and price and $P_{s,t}^f$ to refer to the Pareto-set optimized for transfers, arrival time and fare state.

4.1 Applying the Framework to RAPTOR

We use our framework to implement a price-sensitive version of the multi-criteria RAPTOR algorithm (McRAPTOR) [6]. That is, we solved the earliest arrival problem with price as an additional optimization criterion. RAPTOR implicitly optimizes also for the number of transfers. To facilitate discussion, we presented the framework in a graph-based context. However, RAPTOR does not use a graph model but works directly on the timetable. It operates in rounds $k = 1, \dots, n$. In each round, a set of marked routes is visited and labels are propagated along them. Labels are updated in this process by reading the new arrival time directly from the route's arrival time array. At each station, it is checked whether the new label improves upon the current optimal label. If so, that local label is updated and the station is marked as a starting point for the next round. The standard RAPTOR version labels all stations solely with arrival times.

Adapting our findings to work with RAPTOR is straightforward: Each pair of adjacent stations (v_i, v_{i+1}) on a trip can be interpreted as an arc. Hence, we store for every trip not only an array of arrival times but also an array of all fare attributes. A label $l_i = (t_i, f_i)$ (at station v_i) consists of an arrival time t_i and a fare state f_i . When traversing along a trip from station v_i to v_{i+1} , the arrival time is updated and the fare state is set to $f_{i+1} = Up(f_i, (v_i, v_{i+1}))$. Dominance of labels is checked according to the theory developed above while also taking arrival times into account. Update steps that were associated with arcs modeling transfers are performed whenever the algorithm hops on a new route. This requires storing an additional array with fare attributes at every station to represent transfer costs for every trip at the station. Note that, in most cases, it is enough to store one array for each route instead of trip, as fare attributes seldom vary among the trips of a route. Since walking is free, fare states do not need to be updated in the footpath stage. Depending on the data set, it might thus be possible to save money by walking a long distance in the middle of an itinerary. This kind of journey can be excluded during postprocessing.

4.2 Speed-up Techniques

Price-based Target Pruning

In RAPTOR as well as Dijkstra's algorithm, it is possible to prune labels that are worse than the labels that have already been found at the target station. Naturally, the same speed-up technique is also possible for our algorithm. Moreover, we need not use \leq_C to compare fare states. Since the labels at the target station are never updated and the price function π is non-decreasing, a path already more expensive than the incumbent cheapest s, t -path cannot be price-optimal. Hence, we can prune all labels with a fare state f with $\pi(f^t) \geq \pi^*$, with π^* being the incumbent price at the target station.

Bounded McRAPTOR

By design of fare systems the cheapest path is often among the fastest as detours are penalized by increases in both price and travel time. Therefore, it seems beneficial for a multi-criteria search to compute a minimal travel time itinerary early on by running a standard RAPTOR query. The labels obtained in this first stage can then be used to prune the multi-criteria search. Let t_k be the optimal arrival time at the target station in round k of the first stage (computed with RAPTOR). During round k of McRAPTOR, we prune every label that has an arrival time t with $t > t_k + \epsilon$. Note that this possibly cuts off optimal paths from $P_{s,t}^f$ (and $P_{s,t}^*$) if ϵ is chosen to be small. This technique, alongside an even tighter pruning scheme, has been introduced in [5].

Problem Specific Speed-ups

Certain dimensions in H might only be relevant for some tickets in T . For example, many short-distance tickets depend on the number of stations visited while this number is irrelevant for all other tickets that can be reached from that ticket. We can therefore alter the comparison operator \leq_C for those tickets to ignore the number of stations. Hence, more labels become comparable which results in a smaller Pareto-set $P_{s,t}^{pss}$ with $P_{s,t}^* \subseteq P_{s,t}^{pss} \subseteq P_{s,t}^f$.

4.3 Computational Results

We implemented the McRAPTOR algorithm in C++17 compiled with gcc 8.1.0 and `-O3` optimization. All tests were conducted on Dell Poweredge M620 machines with 64 GB of RAM. While the general structure of the MDV price system is captured in our model, our computations deviate from the prices charged by MDV in the following three cases: A list of relations that is, contrary to the general rules, not eligible for the short-distance discount is not taken into account. Neutral zones are as of yet not implemented. Instead, we add each neutral zone to one of its surrounding zones. Stations and fare zones that a route passes through without stopping are not represented in the available data and therefore cannot be taken into account.

From MDV's² GTFS³ feed, we extracted the timetable of the 22 May 2019. It contains 4,371 stations, 18,215 trips, 5347 routes and 845 footpaths. The original footpath set was not transitively closed⁴. After computing the transitive closure, there were 1,029 footpaths. 40 unconnected stations as well as 960 duplicate trips were removed from the data. We then chose a test set of 5,000 queries uniformly at random from the set of connected stations.

In Table 1, we depict results for six different versions of RAPTOR. A standard RAPTOR query (*RAPTOR*) and a McRAPTOR query optimizing for fare zones (*zones*) were included for comparison. All other versions optimize for price and travel time using the framework presented above. Version *fare* uses price-based target pruning as presented in Section 4.2. Employing standard target pruning (using \leq_C for comparison) instead yielded extremely poor results in exploratory computations and is therefore not included in this study (One query found 1000 paths of which only four were in $P_{s,t}^*$). Version *pss* additionally uses

² The MDV GTFS timetable is licensed under CC BY 4.0 and is publicly available under <https://www.mdv.de/informationen/downloads/>.

³ Fare data is not part of the feed and was obtained separately via InfraDialog GmbH.

⁴ RAPTOR requires a transitively closed footpath set [6].

■ **Table 1** Computation results for 5000 queries in the MDV network. All values depicted are averages. 30 queries for which no itinerary was found were excluded. The column *pareto* the size of the Pareto-set as computed by RAPTOR, while *sols* reports the size of $P_{s,t}^*$. We also report the number of scanned routes (*scan*). For *bound60* and *bound30*, the results of the actual multi-criteria query are reported (second phase). *Total* combines the results for the first and second phase.

	Criteria				Technique			Query				Total	
	<i>transfers</i>	<i>time</i>	<i>zones</i>	<i>fare</i>	<i>TP</i>	<i>EB</i>	<i>PSS</i>	<i>pareto</i>	<i>sols</i>	<i>time[ms]</i>	<i>scan</i>	<i>time[ms]</i>	<i>scan</i>
RAPTOR	•	•	◦	◦	◦	◦	◦	1.58	–	6	11032	–	–
zones	•	•	•	◦	◦	◦	◦	49.53	–	5066	19563	–	–
fare	•	•	◦	•	•	◦	◦	6.01	2.66	48084	16457	–	–
pss	•	•	◦	•	•	•	◦	3.5	2.66	406	14842	–	–
bound60	•	•	◦	•	•	•	•	2.51	2.21	274	14322	280	25354
bound30	•	•	◦	•	•	•	•	2.34	2.14	260	14063	266	25085

problem-specific speed-ups and *bound60* (*bound30*) computes bounds to cut off all paths that are more than 60 (30) minutes slower than those computed in the first phase. The field *pareto* reports the average number of solutions computed by the RAPTOR/McRaptor query. When optimizing for price and arrival time, this set is bigger than the size of $P_{s,t}^*$, which is reported as *sol*. Note that these sets are smaller for *bound60* and *bound30* since they compute restricted Pareto-sets. When taking fare zones into account (*zones*), the size of the Pareto-set increased from 1.58 (*RAPTOR*) to a staggering 49.53. This is reflected in a high run time of more than 5 s. The price-based target pruning used in *fare* results in a significantly smaller Pareto-set $P_{s,t}^*$ (6.01) on average. The average size of $P_{s,t}^*$ is 2.66. This indicates that most of the itineraries computed by *zones* are not interesting for a typical customer and the fare framework can be leveraged to avoid their computation. However, the framework is computationally more involved and requires optimization of two additional criteria (distance and visited stations), which leads to run times of 48 s on average. In *pss*, fare zones are not compared anymore after reaching ticket M . Distance and the number of visited stations is only considered for the tickets K , K_L and K_H , T_1 and T_2 . This results in an average run time of 406 ms which is even considerably faster than the *zones* queries. Computing bounds in a first phase reduces run times further to 280 and 266 ms, respectively, while not reducing the number of itineraries found too much.

5 Conclusion

We presented a novel framework for modeling fare systems of public transportation companies. It is independent of the shortest path algorithm used and can be used to solve price-sensitive earliest arrival queries in real-world networks. Our test case forces the implicit optimization for distance, number of stations visited and fare zones, which resulted in slow run times and large Pareto-sets. Both can be greatly improved by utilizing insights into the price structure to tighten dominance rules, which lead to the framework faring even better than a purely fare zone-based McRAPTOR. The speed-up is, however, dependent on the ticket transition rules and thus, performance might vary significantly depending on the fare system in question.

References

- 1 Chris Barrett, Riko Jacob, and Madhav Marathe. Formal-Language-Constrained Path Problems. *SIAM J. Comput.*, 30(3):809–837, May 2000. doi:10.1137/S0097539798337716.
- 2 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.
- 3 Annabell Berger and Matthias Müller-Hannemann. Subpath-Optimality of Multi-Criteria Shortest Paths in Time- and Event-Dependent Networks. Technical report, Institute of Computer Science, Martin-Luther-Universität Halle-Wittenberg, 2009. URL: <http://wcms.uzi.uni-halle.de/download.php?down=10850&elem=2163494>.
- 4 Ralf Borndörfer, Ricardo Euler, Marika Karbstein, and Fabian Mett. Ein mathematisches Modell zur Beschreibung von Preissystemen im öV. Technical Report 18-47, ZIB, Takustr. 7, 14195 Berlin, 2018. URL: <urn:nbn:de:0297-zib-70564>.
- 5 Daniel Delling, Julian Dibbelt, and Thomas Pajor. Fast and Exact Public Transit Routing with Restricted Pareto Sets. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 54–65, 2019. doi:10.1137/1.9781611975499.5.
- 6 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. *Transportation Science*, 49(3):591–604, 2015. doi:10.1287/trsc.2014.0534.
- 7 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 International Conference on Experimental Algorithms, WEA'08*, pages 347–361, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-68552-4_26.
- 8 Mehryar Mohri. Semiring Frameworks and Algorithms for Shortest-distance Problems. *J. Autom. Lang. Comb.*, 7(3):321–350, January 2002. URL: <http://dl.acm.org/citation.cfm?id=639508.639512>.
- 9 Matthias Müller-Hannemann and Mathias Schnee. Paying less for train connections with MOTIS. In *Proceedings of the 5th Workshop on Algorithmic Methods and Models for Optimization of Railways*, volume 2 of *OpenAccess Series in Informatics*, page 657, January 2005. doi:10.4230/OASIcs.ATMOS.2005.657.
- 10 U. Zimmermann. *Linear and combinatorial optimization in ordered algebraic structures*, volume 10 of *Annals of discrete mathematics*. North-Holland, 1981.

A Fare Transition Functions for MDV

In the following, we provide the fare transition function associated with the fare transition arcs in Figure 1. We refer to the dimensions of the partially ordered monoid $(H, +_h, \leq_h)$ as $H := H^z \times H^s \times H^d \times H^t$, where $H^z = 2^Z$, $H^s = \mathbb{N}$, $H^d = \mathbb{N}$ and $H^t = \{0, 1\}$. The set H^z represents all possible combinations of fare zones, H^s and H^d represent distances measured in the number of stations and in meters, respectively, and H^t , whether a trip contains a transfer. The symbol set is $S = \{H, L, T_1, T_2, S_0\}$, where H and L are associated with all stations in Halle and Leipzig, respectively. The symbol T_1 is associated with stations in cities that allow the T_1 price, the symbol T_2 with stations in cities that allow the T_2 price. All remaining stations have the symbol S_0 . Let $s \in S$ and $h = (h_z, h_s, h_d, h_t) \in H$. Then, the fare transition function is defined by

$$\begin{aligned}
 Tr(Z_i, Z_{i+1}, s, h) &= 1 \Leftrightarrow |h_z| > i && \forall i \in 1, \dots, 6 \text{ with } Z_7 := M \\
 Tr(T_j, Z_1, s, h) &= 1 \Leftrightarrow \\
 & s \neq T_j \wedge (h_t = 1 \vee h_d > 4) && \forall j = 1, 2 \\
 Tr(T_j, K, s, h) &= 1 \Leftrightarrow \\
 & s \neq T_j \wedge (h_t = 1 \wedge h_d \leq 4) && \forall j = 1, 2 \\
 Tr(K, Z_i, s, h) &= 1 \Leftrightarrow |h_z| = i \wedge (h_t = 1 \vee h_d > 4) && \forall i = 1, 2, 3, 4 \\
 Tr(L, Z_2, s, h) &= 1 \Leftrightarrow s \neq L \\
 Tr(H, Z_2, s, h) &= 1 \Leftrightarrow s \neq H \\
 Tr(K_L, Z_2, s, h) &= 1 \Leftrightarrow s \neq L \wedge (h_t = 1 \vee h_s > 4) \\
 Tr(K_H, Z_2, s, h) &= 1 \Leftrightarrow s \neq H \wedge (h_t = 1 \vee h_s > 4) \\
 Tr(K_L, L, s, h) &= 1 \Leftrightarrow s = L \wedge (h_t = 1 \vee h_s > 4) \\
 Tr(K_H, H, s, h) &= 1 \Leftrightarrow s = H \wedge (h_t = 1 \vee h_s > 4).
 \end{aligned}$$

B Proof of Proposition 11

Proof. Let $p^* = (s = v_0, v_1, \dots, v_{n-1}, v_n = t) \in P_{s,t}^f$ be a state-optimal s, t -path and let (f_0, \dots, f_n) be the fare sequence associated with it. Assume there is another s, t -path $\tilde{p} = (s = u_0, u_1, \dots, u_{l-1}, u_l = t)$ with fare states $(f_0, \tilde{f}_1, \dots, \tilde{f}_l)$. Let k be the largest integer such that $v_{n-k} = u_{l-k}$, i.e. the paths (v_{n-k}, \dots, v_n) and (u_{l-k}, \dots, u_l) are equal. Now assume $\tilde{f}_{l-k-1} <_C f_{n-k-1}$. By definition, $f_{n-k} = Up(f_{n-k-1}, (v_{n-k-1}, v_{n-k}))$ and $\tilde{f}_{l-k} = Up(\tilde{f}_{l-k-1}, (u_{l-k-1}, u_{l-k}))$. We apply Equation 7 to obtain $\tilde{f}_{l-k} \leq_C f_{n-k}$. By repeating the process for $i \in \{k-1, \dots, 0\}$, we find $\tilde{f}_l \leq_C f_n$. Since p^* was state-optimal, it follows that $\tilde{f}_l =_C f_n$ and consequently \tilde{p} is also state-optimal. Since the number of paths in G is finite, we can repeat this procedure to find the path p' . ◀

C Proof of Proposition 15

Proof. Let $a \in A$ and $f_1, f_2 \in F$ such that $f_1 \leq_C f_2$. We write $\tilde{f}_1^t := Up(f_1, a)$ and $\tilde{f}_2^t := Up(f_2, a)$. Clearly, $f_1^h \leq f_2^h$ implies $\tilde{f}_1^h \leq \tilde{f}_2^h$. First, assume that $f_1^t \in C_P$, hence $f_1^t = f_2^t$. By applying the definition of C_P , we obtain

$$Tr(f_1^t, \mathcal{A}^s(a), f_1^h + \mathcal{A}^h(a)) = Tr(f_2^t, \mathcal{A}^s(a), f_2^h + \mathcal{A}^h(a)).$$

Thus, $\tilde{f}_1^t = \tilde{f}_2^t$. Note that the definitions of C_P and C_F imply that $\tilde{f}_1^t \in C_P \cup C_F$ since $\tilde{f}_1^t \in R(f_1^t)$ and hence $\tilde{f}_1^t \leq_C \tilde{f}_2^t$. Now, assume $f_1^t \in C_F$. Note that $R(f_1^t) \subset C_F$. Hence, if $\tilde{f}_1^t = \tilde{f}_2^t$, we obtain $\tilde{f}_1^t \leq_C \tilde{f}_2^t$.

If instead $\tilde{f}_1^t \neq \tilde{f}_2^t$ we need to show that $\tilde{f}_1^t \rightarrow \tilde{f}_2^t$. Note that $f_2^t \in R(f_1^t)$ and $R(f_1^t)$ is traceable. Hence, f_2^t, \tilde{f}_2^t and \tilde{f}_1^t are on a directed path. Since \mathcal{F} is acyclic and $f_2^t \Rightarrow \tilde{f}_2^t$ holds there are four cases to consider:

1. $\tilde{f}_1^t \Rightarrow f_2^t \Rightarrow \tilde{f}_2^t$, which implies $\tilde{f}_1^t \Rightarrow \tilde{f}_2^t$;
2. $f_2^t \Rightarrow \tilde{f}_1^t \Rightarrow \tilde{f}_2^t$, which implies $\tilde{f}_1^t \Rightarrow \tilde{f}_2^t$;
3. $f_2^t \Rightarrow \tilde{f}_2^t \rightarrow \tilde{f}_1^t$ and $f_1^t = \tilde{f}_1^t$, which creates the cycle $f_1^t \Rightarrow f_2^t \Rightarrow \tilde{f}_2^t \rightarrow \tilde{f}_1^t = f_1^t$ in \mathcal{F} and is hence contradictory;

4. $f_2^t \rightrightarrows \tilde{f}_2^t \rightarrow \tilde{f}_1^t$ and $f_1^t \rightarrow \tilde{f}_1^t$. Note that $f_1^t \rightrightarrows f_2^t \rightarrow \tilde{f}_1^t$ and also that $Tr(f_1^t, \tilde{f}_1^t, \mathcal{A}^s(a), f_1^h + \mathcal{A}^h(a)) = 1$. Hence, we can apply the no-overtaking property for the edge (f_1^t, \tilde{f}_1^t) , the ticket f_2^t and the fare attribute $(\mathcal{A}^s(a), f_2^h + \mathcal{A}^h(a))$. Since $f_2^h + \mathcal{A}^h(a) > f_1^h + \mathcal{A}^h(a)$, this yields the existence of an edge $(f_2^t, f^t) \in E$ with $Tr(f_2^t, f^t, \mathcal{A}^s(a), f_2^h + \mathcal{A}^h(a)) = 1$ and $\tilde{f}_1^t \rightrightarrows f^t$. Note that (f_2^t, f^t) is not necessarily the only outgoing edge at f_2^t but by definition of Tr there is only outgoing edge at f_2^t with transition function Tr equal to one for the fare attribute $(\mathcal{A}^s(a), f_2^h + \mathcal{A}^h(a))$. Since also $Tr(f_2^t, \tilde{f}_2^t, \mathcal{A}^s(a), f_2^h + \mathcal{A}^h(a)) = 1$, it follows that $\tilde{f}_2^t = f^t$. This creates the cycle $\tilde{f}_2^t \rightarrow \tilde{f}_1^t \rightarrow \tilde{f}_2^t$, which also yields a contradiction.
- Hence, we conclude that in fact $\tilde{f}_1^t \rightarrow \tilde{f}_2^t$ and therefore, concluding the proof, $\tilde{f}_1 \leq_C \tilde{f}_2$. ◀

D Proof of Proposition 16

Proof. Consider $p \in P_{s,t}^\pi := \{\tilde{p} \text{ s,t-path} : \pi(f^t(\tilde{p})) = \pi^*\} \neq \emptyset$. If there is a path $p' \in P_{s,t}^f$ with $f^t(p') = f^t(p)$, we are done. If not, there is a path $p' \in P_{s,t}^f$ with $f^t(p') \rightarrow f^t(p)$. This implies $\pi(f^t(p')) \leq \pi(f^t(p))$ and hence that a path of the same price as p is present in $P_{s,t}^f$. ◀

Mode Personalization in Trip-Based Transit Routing

Vassilissa Lehoux

NAVER LABS Europe, Meylan, France

https://europe.naverlabs.com/people_user/vassilissa-lehoux/

firstname.lastname@naverlabs.com

Darko Drakulic

NAVER LABS Europe, Meylan, France

https://europe.naverlabs.com/people_user/darko-drakulic/

firstname.lastname@naverlabs.com

Abstract

We study the problem of finding bi-criteria Pareto optimal journeys in public transit networks. We extend the Trip-Based Public Transit Routing (TB) approach [18] to allow for users to select modes of interest at query time. As a first step, we modify the preprocessing of the TB method for it to be correct for any set of selected modes. Then, we change the bi-criteria earliest arrival time queries, and propose a similar algorithm for latest departure time queries, that can handle the definition of the allowed mode set at query time. Experiments are run on 3 networks of different sizes to evaluate the cost of allowing for mode personalization. They show that although preprocessing times are increased, query times are similar when all modes are allowed and lower when some part of the network is removed by mode selection.

2012 ACM Subject Classification Mathematics of computing → Graph theory

Keywords and phrases Public transit, Route planning, Personalization

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.13

1 Introduction

In public transit networks, part of the information is available in the form of timetables that give the schedules at given stations of different public transportation modes such as buses, trains or tramways. Transfers between those modes of transportation are possible by walking between the stations, if they are not too far away from one another.

Finding paths in such public transit networks is highly relevant in practice, as millions of users use routing applications such as Naver Map¹, Citymapper² or Google Maps³ to plan their trips daily. In the recent years, research has been very active for this problem [3] and many dedicated techniques, such as Transfer Patterns [2], RAPTOR [5] or CSA [6] have been developed to make this routing efficient.

However, defining criteria and constraints to optimize those paths according to user preferences is a complicated task. Many criteria can be considered, such as earliest arrival time (given a start time or a start time range), latest departure time (given an arrival time or an arrival time range), number of transfers, travel cost, total transfer duration, total waiting time, etc. In multicriteria optimization, a solution is said to be *dominated* in the Pareto sense if there is another solution that is strictly better on one criteria and at least as good on the others. It is frequent to look for either the Pareto set, that is all the

¹ <https://map.naver.com/>

² <https://citymapper.com/>

³ <https://www.google.com/maps>



© Vassilissa Lehoux and Darko Drakulic;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 13; pp. 13:1–13:15

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

non-dominated solutions in the Pareto sense, or the Pareto front, that is the image of the Pareto set in the criteria space. For minimum total cost path problems, considering two or more criteria often makes the problem of finding all the optimal solutions intractable, with possibly exponential size Pareto sets [11], although the number of optimal solutions in the Pareto sense can be manageable in practice for some problems [15]. In this work, we consider the following polynomial bi-criteria problems: minimizing the arrival time and the number of transfers and maximizing departure time and minimizing the number of transfers. We are interested in finding the Pareto front, rather than the Pareto set, and we want to be able to compute one solution corresponding to each element of the Pareto front for minimum number of transfers and either minimum arrival time or latest departure time. Note that in that case, the maximum number of solutions returned is bounded by the number of public transport trips (as we cannot make more transfers) and is hence polynomial in the size of the instance. Providing sets of solutions rather than a single solution enables users to make their own compromises between the number of transfers and the travel time of the trip, according to their preferences. We choose not to compute the complete Pareto set, but only one solution for each element in the Pareto front to ensure polynomial time construction of the set of solutions. Note that the actual Pareto set can be much larger in practice. Indeed, in multimodal networks, having several solutions with the same value in the criteria space can be frequent for the considered criteria, for instance solutions sharing the same final (resp. initial) trip when optimizing earliest arrival time (resp. latest departure time) and number of transfers. In a theoretical network, it can be of exponential size.

As a second step toward more personalized solutions, we consider an additional constraint: at query time, the user can exclude some modes of transportation from the network. For example, a user want to avoid buses, because he/she thinks that they are not reliable enough.

Indeed, the type of transit modes is an important vector of choice between itineraries. In addition to the speed to reach destination, it impacts the price, the comfort, and of course, some modes are more or less appreciated by the user depending on his/her preferences. In many cases, the mode type information is available in the transit data. For instance, the General Transit Feed Standard format [10], very often used to describe public transit information, proposes this information as mandatory. In an itinerary planning application or website, the user will often be able to choose the modes that he/she wants to enable or disable in the interface as option for the search.

Several methods have been designed for mode related personalization. In [12], the authors consider a graph based approach with a time-dependent model of the timetables [16]. They propose personalized mode sequences described by a regular language and solve the associated *regular language constrained shortest path problem* using a combination of the D_{RegLC} [1] and ALT [9] algorithms called State-Dependent ALT. Preprocessing time is light (less than a minute on Île-De-France transportation network), but the languages involved must be defined at the preprocessing step. The User-Constrained Contraction Hierarchy [7] on the other hand, doesn't have this restriction and the mode sequence can be defined at query time. It is also based on a time-depend model for the transit modes and uses Contraction Hierarchy [8] on each mode's network to speed up the search. As a consequence, preprocessing time can be important on large networks (42 min for a small Europe network with 30K stations).

If we want to select the enabled scheduled modes for a query, rather than defining specific mode sequences, dynamic programming approaches such as CSA or RAPTOR might be used with very few modifications. The Connection Scan Algorithm [6] is based on a sorted connections array that contains all the trip segments between two consecutive stops. You can pass from one connection to the next if they are one after another in the same trip or if

you can leave the first connection and reach the next on time to take it (for instance by a walking transfer). This algorithm could be modified for mode personalization by pruning at query time the search space by taking only connections corresponding to allowed scheduled modes. Similarly, the RAPTOR algorithm [5] works directly with timetable information. It uses a round-based approach where in each round, trips are taken from lines passing at stops reached at the preceding iteration. In this context, some trips of disabled scheduled modes could also be avoided at query time by saving and checking the mode of each line. This approach can be found in [17] where a wider set of mode sequences is considered. Another approach can be found in [4], where the authors use the number of buses as an additional criterion for computing a subset of the Pareto set. They can hence obtain optimal solutions with no buses (as well as solutions with several bus trips).

In this work, we are interested in extending the Trip-Based Public Transit Routing approach [18] (TB) in order to be able to personalize the set of scheduled modes used at query time. TB is a round-based approach, iterating on the maximum number of transfers allowed in a solution, that relies on a different graph model: the nodes of the graph are the trips, while the arcs represent possible transfers. A preprocessing phase computes a non-minimal arc set T such that for any value in the Pareto front, there exists a solution S with this value such that all the transfers of S belong to T . Search phase is then breadth-first search like and builds one solution for each element of the Pareto front for minimum arrival time and minimum number of transfers. Note that the author uses a slightly modified definition of Pareto dominance to call the set built *Pareto set*, but here we choose to keep the standard definition.

2 Preliminaries

As we extend the TB algorithm, we give in this section a brief description of this method and discuss some of the claims made by the author in his article.

2.1 Notations

We will use notations similar to that of [18] to describe the public transit network. A sequence of stops $\vec{p}(t) = \langle p_t^1, p_t^2, \dots \rangle$ is associated with each trip t . The schedule of t is defined by the arrival and departure times of t at the stops of its sequence. We denote by $\tau_{arr}(t, i)$ (resp. $\tau_{dep}(t, i)$) the arrival time (resp. departure time) of t at the i^{th} stop of $\vec{p}(t)$. Trips are grouped into lines, that do not exactly represent the routes of the public transport network. First, all the trips of a line L have exactly the same sequence of stops, denoted $\vec{p}(L) = \langle p_L^1, p_L^2, \dots \rangle$. Second, all the trips of a line are completely ordered following comparison relations \preceq and \prec defined for two trips having the same sequence:

$$\begin{cases} t \preceq u \iff \forall i \in [0, |\vec{p}(t)|), & \tau_{arr}(t, i) \leq \tau_{arr}(u, i) \\ t \prec u \iff t \preceq u \text{ and } \exists i \in [0, |\vec{p}(t)|), & \tau_{arr}(t, i) < \tau_{arr}(u, i) \end{cases}$$

L_t denotes the line of trip t . For a given stop s , we define $\mathbf{L}(s)$ as the set of all pairs (L, i) with L a line and i an index in the sequence of L such that $s = p_L^i$. A displacement between the i^{th} stop of t and the j^{th} stop of t using trip t is denoted $p_t^i \rightarrow p_t^j$ and similarly, a walking transfer between trip t at the i^{th} station and trip u at the j^{th} station is denoted $p_t^i \rightarrow p_u^j$. Walking transfer times are defined for any pair of stops (p, q) , $p \neq q$ that are close enough of one another and the associated duration is $\Delta\tau_{fp}(p, q)$. When transferring between two trips at a given station ($p_t^i = p_u^j = p$), a minimum change time $\Delta\tau_{fp}(p, p)$ can be defined, to represent the time needed to move within this station.

2.2 Preprocessing

The aim of the preprocessing of the TB algorithm is to build a set T of transfers between trips such that any transfer $p_t^i \rightarrow p_u^j$ of T is *feasible*, that is $\tau_{arr}(t, i) + \Delta\tau_{fp}(p_t^i, p_u^j) \leq \tau_{dep}(u, j)$, and for any element of the Pareto front, there exists an optimal solution with this element as value, such that all its transfers $p_t^i \rightarrow p_u^j$ belong to T . Note that the algorithm does not require the set T to be minimal, that is to be a set of minimum cardinality with this property. It needs only to be *correct*, that is to contain only feasible transfers and to contain all the transfers of at least one optimal solution per element in the Pareto front.

In order to compute a correct set of transfers, the preprocessing considers the transfers from each trip separately, which allows for trivial parallelization of the algorithm. For a given trip t , starting from the last stop of $\vec{p}(t)$ and taking the sequence in reverse order, we consider for each stop p_t^i with $i > 1$ all the reachable stops q from p_t^i (i.e. such that $\Delta\tau_{fp}(p_t^i, q)$ is defined). We set the earliest arrival and change times at each of those stops at $\tau_{arr}(t, i) + \Delta\tau_{fp}(p_t^i, q)$ and we look for transfers to all the possible lines passing by q . For each of those lines, we find the earliest trip u such that the transfer $p_t^i \rightarrow p_u^j$ is feasible (with $p_u^j = q$ and $j < |\vec{p}(u)|$). We remove U-turn transfers as described in [18]. Then, in order to reduce the transfer set, we try and update or set earliest arrival and earliest change times at the stops later in the sequence of u or at the stops reachable from those stops. If any improvement occurs or a new stop is reached, the transfer is kept. If not, it will be removed from the set of transfers. Note that if two transfers are equivalent (in term of reachable stops and arrival and change times at those stops), only the first one generated will be kept. Since we are looking at the stop sequence $\vec{p}(t)$ in reverse order, the later transfers are added to the set before equivalent earlier transfers that will be checked later in the process.

2.3 Query phase

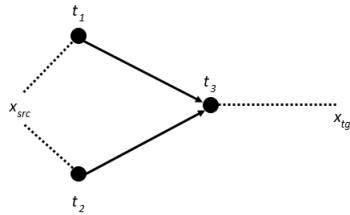
The TB algorithm deals with earliest arrival time queries, where given a start time, the objective is to find the Pareto front for earliest arrival time and number of transfers.

At initialization, origin trip segments (trips segments whose boarding stop can be reached by a walking displacement from the origin point) and destination trip segments (trips segments from the unboarding stop of which the destination point can be reached) are computed. Origin trips are added to a queue. Note that instead of considering only departure from stops, it is possible to consider a departure from any location on the transportation network by allowing to compute shortest paths in the walking network to or from the closest stops.

The query phase is then a breadth-first search like procedure in the graph whose nodes are the trips and whose arcs are the possible transfers. At each iteration n , all the trip segments of queue Q_n are processed in order. If one is a destination trip, current earliest arrival time at destination is updated. It can be used to prune the search, by not adding transfers to the queue if they cannot be part of a solution that improves on this arrival time. For a given trip segment of Q_n , all possible transfers from it are performed, increasing the number of transfers by one in the partial solutions computed. When a transfer reaches a trip segment that is not marked, the trip segment is added to the queue for the next iteration and the trip itself and all the corresponding later trip segments of the same line are marked as processed. Iterations continue until maximum transfer number has been reached or current arrival time at destination cannot be improved (which means that the queue of trip segments is empty). The earliest arrival time itinerary to destination obtained at iteration k (when it exists) is hence an earliest arrival time itinerary with at most k transfers. The Pareto front for earliest arrival time and number of transfers is hence generated during the search phase.

2.3.1 Construction of the solution

Witt [18] claims to optimize latest departure time as a secondary criterion to break ties, but the construction proposed does not ensure this property. To build solutions, he suggests to store for each trip segment a pointer to its origin trip segment when the trip is added to the queue, in order to rebuild the sequence of trips and to compute optimal transfer between the trips as a postprocessing. Consider the very simple example shown in Figure 1.



■ **Figure 1** Small network with 3 trips and 2 transfers.

The network consists in 2 lines, one with trips t_1 and t_2 such that $t_1 \prec t_2$, one with trip t_3 . Both trips t_1 and t_2 can transfer from their i^{th} stop to the same index j of trip t_3 . When preprocessing those transfers, both are kept in the set as they update the arrival times of the stops of t_3 . In the search phase, starting from source x_{src} , trip t_1 is the earliest trip of the line that can be reached and is hence added to the queue. Then transfer to t_3 is done to reach destination x_{tgt} . Hence, the sequence of trips is t_1 and then t_3 , while with latest departure time as a secondary criterion, t_2 and then t_3 should have been returned. As there is no sorting on departure trips, a similar example can be built with trip t_2 belonging to another line if the search starts with t_1 .

Note that in both cases, the transfers in the solution with latest departure time for the given earliest arrival time and number of transfers were in the computed set of transfers.

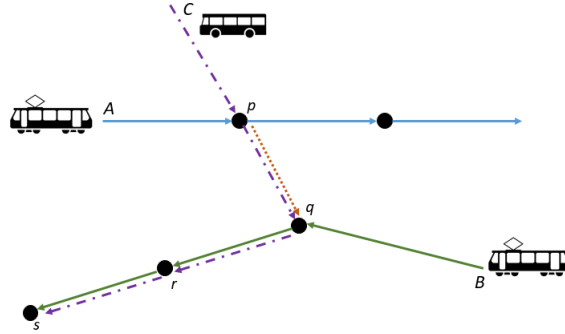
In order to actually find the maximum departure time for a given number of transfers and a given arrival time, it is possible to consider several strategies, such as using profile queries.

Profile queries are given a time range as input. In our case, for every possible start time (resp. arrival time) in that range, you want to compute optimal values of the Pareto front for earliest arrival time (resp. latest departure time) and number of transfers. In order to speed profile queries, Witt proposes for earliest arrival time profile queries to start by the end of the time range and to iterate backward on this time interval. The idea is to keep the labels of the preceding time step as journey starting later never dominates earlier ones.

Consider an earliest arrival time solution S with value (τ_{arr}, k) in the Pareto front obtained for start time τ . In order to find the latest departure time for which there exists a solution with value (τ_{arr}, k) , it is possible to make a profile query with departure time range $[\tau, \tau_{arr}]$. Going backward, we compute the Pareto front for each instant of the range. Unless origin and destination are equal, for a departure time $t = \tau_{arr}$, value (τ_{arr}, k) doesn't belong to the Pareto front, but it will belong to it for a departure time of τ . Hence, decreasing the minimum departure time value from τ_{arr} to τ , (τ_{arr}, k) will belong to the Pareto set at a certain iteration. The latest departure time associated with (τ_{arr}, k) is the first instant when value (τ_{arr}, k) belongs to the Pareto set.

3 Scheduled mode selection

The preprocessing phase of the TB algorithm removes transfers from the search phase without modifying the Pareto front when all the modes are available. Consider the example of Figure 2.



■ **Figure 2** Example of transfer removed by the TB algorithm's preprocessing pruning phase.

Suppose that we have only those 3 lines in a public transportation network. Lines *A* and *B* are tramway lines (represented with solid lines) and the line *C* is a bus line (represented with a dashed line). A walking transfer (dotted line) is possible between the stop *p* on line *A* and the stop *q* on line *B*. Suppose that exploring the transfers from a trip from line *A* at stop *p*, we first look at the transfer from that trip to line *C* at stop *p*. We set the arrival and change times of the stops *q*, *r* and *s* and move to the next transfer. We now try and update stops' arrival times by transferring from the current trip of line *A* to *B*. The earliest trip of line *B* does not improve arrival or change times at stops *r* or *s*. It is hence pruned.

Now, consider that you want to allow only tramways for a given query. If buses cannot be taken, the removed transfer could have found itself in an optimal solution, for instance starting with the trip of *A* and arriving at *s* on line *B*.

A possible solution, to avoid losing correctness, could be to have several instances of the routing service, each corresponding to a selection of transit modes. Each server would run on preprocessed data for the corresponding subnetwork. To consider all the possible combinations of modes, you will need $2^{|M|} - 1$ servers, if *M* is the set of modes of the network.

A second option, that we will use as a base line, is to use the complete set of transfers in the search phase, without any pruning based on arrival times. In [18], the author reports query times about 3 times as slow with this graph compared to the pruned one.

In this article, we propose a method for computing a reduced transfer set that is correct for any set of modes asked by the user at query time, which enables to answer that type of queries with a single server. This preprocessing step is explained in Section 3.1. Then, with such set as input, the search phase requires only a few modifications that will be explained in Section 4.

3.1 Preprocessing with mode selection

The aim of the preprocessing would now be to return a set of transfers *T* that is correct when enabling any subset μ of the set of all transit modes *M*.

In order to achieve this aim, we propose to change the pruning phase to be sure that for each value of the Pareto front and for any $\mu \subseteq M$ there is at least one solution with this value that uses only transfers of *T*. As before, we preprocess the transfers of each trip

separately. We denote $m_t \in M$ the mode associated with the line of trip t . As we are computing transfers from trip t , mode m_t needs to be allowed for the transfer to potentially belong to an optimal solution. So for a transfer to a line l of mode m to be in an optimal solution, at least mode m_t and mode m need to be in the subset μ of allowed modes.

For a given trip t , we propose to compute the contribution to the set of transfers T that will be used in the search phase in the following way. At each stop of the network, we try and update a minimum arrival time and minimum change time for any given subset μ of M such that $\mu = \{m, m_t\}$ with $m \in M$. Hence, at most $2|M|$ values are recorded for each stop. When transferring to a trip t of mode $m \in M \setminus \{m_t\}$, we can use the same procedure as before to update arrival and change times when using only modes m and m_t . When transferring to another trip of mode m_t , the arrival and change times for all the subsets of M are updated simultaneously, as mode m_t is necessarily allowed when transferring from t . We denote by $\tau_A(q, m)$ (resp. $\tau_C(q, m)$) the minimum arrival time (resp. minimum change time) found so far during the execution of the procedure when transferring from t for subset $\mu = \{m, m_t\}$ of allowed modes. The procedure is presented in Algorithm 1.

■ **Algorithm 1** Pruning.

Input: Timetable data, footpath data, transfer set T

Output: Reduced transfer set T

for trip t **do**

$\tau_A(\cdot, \cdot) \leftarrow \infty$ ▷ Earliest arrival time at stops for a given mode subset

$\tau_C(\cdot, \cdot) \leftarrow \infty$ ▷ Earliest change time at stops for a given mode subset

for $i \leftarrow |\vec{p}(t)| - 1, \dots, 1$ **do**

 UPDATE($p_t^i, m_t, m_t, \tau_{arr}(t, i), \tau_{arr}(t, i) + \tau_{fp}(p_t^i, p_t^i)$)

for each stop $q \neq p_t^i$ such that $\Delta\tau_{fp}(p_t^i, q)$ is defined **do**

 UPDATE($q, m_t, m_t, \tau_{arr}(t, i) + \tau_{fp}(p_t^i, q), \tau_{arr}(t, i) + \tau_{fp}(p_t^i, q)$)

for each transfer $p_t^i \rightarrow p_u^j \in T$ **do**

$keep \leftarrow \text{false}$

for each stop p_u^k on trip u with $k > j$ **do**

$keep \leftarrow keep \vee \tau_{arr}(u, k) < \tau_A(p_u^k, m_u)$

$keep \leftarrow keep \vee \tau_{arr}(u, k) + \tau_{fp}(p_u^k, p_u^k) < \tau_C(p_u^k, m_u)$

 UPDATE($q, m_t, m_u, \tau_{arr}(u, k), \tau_{arr}(u, k) + \tau_{fp}(p_u^k, p_u^k)$)

for each stop $q \neq p_u^k$ such that $\Delta\tau_{fp}(p_u^k, q)$ is defined **do**

$\rho \leftarrow \tau_{arr}(u, k) + \Delta\tau_{fp}(p_u^k, q)$

$keep \leftarrow keep \vee (\rho < \tau_A(q, m_u)) \vee (\rho < \tau_C(q, m_u))$

 UPDATE(q, m_t, m_u, ρ, ρ)

if $\neg keep$ **then**

$T \leftarrow T \setminus \{p_t^i \rightarrow p_u^j\}$ ▷ No improvement: remove the transfer

procedure UPDATE(q, m_t, m_u, e, c)

Input: stop q , mode m_t , mode m_u , arrival time e , change time c

$\tau_A(q, m_u) \leftarrow \min(\tau_A(q, m_u), e)$

$\tau_C(q, m_u) \leftarrow \min(\tau_C(q, m_u), c)$

if $m_t = m_u$ **then** ▷ Mode m_t is allowed since we are transferring from it

for each $m \in M \setminus \{m_t\}$ **do**

$\tau_A(q, m) \leftarrow \min(\tau_A(q, m_t), \tau_A(q, m))$

$\tau_C(q, m) \leftarrow \min(\tau_C(q, m_t), \tau_C(q, m))$

► **Proposition 1.** *Algorithm 1 computes a correct set of transfers for earliest arrival time and minimum number of transfers, for any subset μ of M .*

Proof. First, note that, when transferring from trip t , as m_t belongs to all the considered subsets of M , it is sufficient to update the value of the *keep* variable before the UPDATE procedure, even if $m_u = m_t$. Hence, for a given starting trip t , and an index i in its sequence of stops, *keep* will be set to TRUE whenever a transfer $p_t^i \rightarrow p_u^j$ improves the arrival time or the change time at a given stop for the subset $\mu = \{m, m_t\}$ of M . Hence if transfer $p_t^i \rightarrow p_u^j$ belongs to an optimal solution for $\mu = \{m, m_t\}$, at least one equivalent transfer is added to the set of transfers returned at the end of the procedure.

Now, consider an arbitrary subset μ of M and an optimal value (τ_{arr}, k) of the Pareto front with $1 \leq k$ and a solution s from the Pareto set of value (τ_{arr}, k) . We represent this solution by the trip segment sequence that composes it:

$$s = \langle p_{t_1}^{j_1} \rightarrow p_{t_1}^{i_1}, p_{t_2}^{j_2} \rightarrow p_{t_2}^{i_2} \dots, p_{t_{k+1}}^{j_{k+1}} \rightarrow p_{t_{k+1}}^{i_{k+1}} \rangle$$

Consider the last transfer $p_{t_k}^{i_k} \rightarrow p_{t_{k+1}}^{j_{k+1}}$ of s . Since s is an optimal solution for μ , it is not possible to arrive sooner at stop $p_{t_{k+1}}^{j_{k+1}}$ from trip segment $p_{t_k}^{j_k} \rightarrow p_{t_k}^{i_k}$. Hence, either $p_{t_k}^{i_k} \rightarrow p_{t_{k+1}}^{j_{k+1}}$ is in T or there is a transfer from t_k at $p_{t_k}^{i'_k}$, $i_k \leq i'_k$ in T leading to a trip with the same arrival time at $p_{t_{k+1}}^{j_{k+1}}$ for the subset $\{m_{t_k}, m_{t_{k+1}}\}$ of μ . Let $p_{t_k}^{i'_k} \rightarrow p_{t_{k+1}}^{j_{k+1}}$ be the transfer that is actually in T . Trip t'_{k+1} is either of mode m_{t_k} or of mode $m_{t_{k+1}}$ which both belong to μ .

Now consider the previous transfer in s , $p_{t_{k-1}}^{i_{k-1}} \rightarrow p_{t_k}^{j_k}$. Either this transfer is in T or, as $j_k \leq j'_k$, there exist another transfer from t_{k-1} in T that has a change time at least as early at stop $p_{t_k}^{j'_k}$. We denote $p_{t_{k-1}}^{j'_{k-1}} \rightarrow p_{t_k}^{i'_k}$, with $j_{k-1} \leq j'_{k-1}$ the transfer actually in T . The transfer from t'_k at stop $p_{t_k}^{j'_k}$ to trip segment $p_{t_{k+1}}^{j_{k+1}} \rightarrow p_{t_{k+1}}^{i_{k+1}}$ with $p_{t_{k+1}}^{i_{k+1}} = p_{t_{k+1}}^{i_{k+1}}$ is feasible as trip t'_k has a change time at least as early as t_k at that stop. Hence, either it is in T or there is a transfer with at least as good an arrival time at $p_{t_{k+1}}^{i_{k+1}}$ for the subset of modes $\{m_{t'_k}, m_{t_{k+1}}\} \subseteq \mu$ that is in T . Hence, going backward in the transfer of s , we can build a solution using a subset of the modes of s , with the same number of transfers, all its transfers being in T and the arrival time at $p_{t_{k+1}}^{i_{k+1}}$ being identical. This solution has therefore the same value as s . ◀

► **Lemma 2.** *Algorithm 1 computes a correct set of transfers for latest departure time and minimum number of transfers for any subset μ of M .*

Proof. As before, we need to prove that for any element (τ_{dep}, k) of the Pareto front, there exists a solution those transfers are in T such that the value of s is (τ_{dep}, k) . Consider an instance $I_{dep} = (\tau, \mu, x_{org}, x_{tgt})$ of the latest departure time problem with τ the latest departure time, $\mu \subseteq M$ the allowed modes, x_{org} the origin and x_{dest} the destination. Let (τ_{dep}, k) be an optimal value of the Pareto front and τ_{arr} the earliest arrival time when starting at τ_{dep} and using at most k transfers. Let s be an optimal solution of the latest departure time problem for I_{dep} with value (τ_{dep}, k) and arrival time τ_{arr} .

Consider the following earliest arrival time problem and an instance $I_{arr} = (\tau_{dep}, \mu, x_{org}, x_{tgt})$ with the same origin, the same destination and a minimum departure time equals to τ_{dep} .

Suppose that a solution s' with k transfers arrives at τ_{arr} and leaves at $\tau > \tau_{dep}$. Then, it dominates s for the latest departure time problem and instance I_{dep} which is not possible as s is optimal. So no solution with k transfers can improve other s .

It remains the possibility of an optimal solution s'' with value (τ_{arr}, k') with $k' < k$ for the earliest arrival time problem with instance I_{arr} and departure time $\tau \geq \tau_{dep}$. Suppose

first that $\tau > \tau_{dep}$. In that case, solution s' dominates s for the latest departure time problem and instance I_{dep} , which is not possible as s is optimal. Hence $\tau = \tau_{dep}$. In that case, we have a contradiction as (τ_{dep}, k) is optimal for I_{dep} and would be dominated by (τ_{dep}, k') .

So all the optimal solutions with k transfers of the earliest arrival time problem for I_{arr} start at τ_{dep} and arrive at τ_{arr} . From Proposition 1, T contains all the transfers of at least one of those solutions, that we denote s_{arr} . s_{arr} is also optimal for the latest departure time problem and instance I_{dep} , which completes the proof. ◀

4 Earliest arrival time and latest departure time queries

Algorithm 2 Latest departure time query.

Input: Transfer set T , origin x_{src} , destination x_{tgt} , latest arrival time τ , mode selection μ
Output: Pareto front J

$J \leftarrow \emptyset$ ▷ Pareto front
 $\mathcal{L} \leftarrow \emptyset$ ▷ Target lines
 $Q_n \leftarrow \emptyset$ for all $n = 1, 2, \dots$ ▷ Queue of trips for each iteration
 $R(t) \leftarrow 0$ for all trips t ▷ Maximum index at which a trip is unboarded during the search

for each stop q such that $\Delta\tau_{fp}(x_{src}, q)$ is defined **do**
 $\Delta\tau \leftarrow 0$ if $x_{src} = q$, else $\Delta\tau_{fp}(x_{src}, q)$
for each $(L, i) \in \mathbf{L}(q)$ such that $m_L \in \mu$ **do**
 $\mathcal{L} \leftarrow \mathcal{L} \cup \{(L, i, \Delta\tau_{fp}(x_{src}, q))\}$

for each stop q such that $\Delta\tau_{fp}(q, x_{tgt})$ is defined **do**
 $\Delta\tau \leftarrow 0$ if $x_{tgt} = q$, else $\Delta\tau_{fp}(q, x_{tgt})$
for each $(L, i) \in \mathbf{L}(q)$ such that $m_L \in \mu$ **do**
 $t \leftarrow$ latest trip of L such that $\tau_{arr}(t, i) + \Delta\tau \leq \tau$
 BW_ENQUEUE($t, i, 0$)

$n \leftarrow 0$
 $\tau_{max} \leftarrow 0$
while $Q_n \neq \emptyset$ **do**
for each $p_t^b \rightarrow p_t^e \in Q_n$ **do**
for each $(L, i, \Delta\tau) \in \mathcal{L}$ with $b \leq i < e$ and $\tau_{dep}(t, i) - \Delta\tau > \tau_{max}$ **do**
 $\tau_{max} \leftarrow \tau_{dep}(t, i) - \Delta\tau$
 $J \leftarrow J \cup \{(\tau_{max}, n)\}$ and remove dominated entries
for each $p_u^i \rightarrow p_t^j \in T$ with $b \leq j < e$ and $m_t \in \mu$ **do**
if $\tau_{dep}(u, i - 1) < \tau_{max}$ **then**
 BW_ENQUEUE($u, i, n + 1$)
 $n = n + 1$

return J

procedure BW_ENQUEUE(trip t , index i , nb transfers n)
if $R_n(t) < i$ **then**
 $Q_n \leftarrow Q_n \cup \{p_t^{R_n(t)} \rightarrow p_t^i\}$
for each trip u such that $L_t = L_u$ and $u \preceq t$ **do**
 $R(u) \leftarrow \max(R(u), i)$

In order to adapt earliest arrival time queries from [18] to transit mode selection, only a few modifications are necessary. First, only add to the queue by the ENQUEUE procedure trips that belong to the selected set of modes μ . Then, when considering transfers from a given mode, only scan transfers to modes that belong to μ . The set of transfers being correct for any value of μ with preprocessing of Section 3.1, the search will compute the Pareto front.

13:10 Mode Personalization in Trip-Based Transit Routing

■ **Table 1** Data sets used for the experiments.

	stops	trips	lines	foot paths	connections	Modes
TCL	4583	70614	578	87834	1425044	Bus, subway, tram, funicular
IDFM	42404	351908	1869	1061959	7803633	Bus, subway, rail, tram, funicular
Korea	180948	446741	31708	4195659	22346975	Bus, subway, rail, tram

In order to deal with latest departure time queries, we propose the following modifications of the base algorithm. Basically, the search is a backward search in the graph of trips and transfers. When we add a trip t to the queue, we hence mark the maximum index $R(t)$ at which t is unboarded rather than the minimum index at which it is taken (see procedure `BW_ENQUEUE` of Algorithm 2). When a trip segment of the queue is processed, we first check if we can improve on the latest departure time found so far and update the Pareto front accordingly. Then transfers are scanned, and the origin trip segment of the transfer is potentially added to the queue. To take into account mode selection, the same modifications as before are necessary, as we check the modes of the trip segments before adding them to the queue. The algorithm can be found in Algorithm 2.

5 Experiments

The experiments are run on a 64 2.7 GHz CPU Intel(R) Xeon(R) CPU E5-4650 server with 20 M of L3 cache and 504 GB of RAM. We perform our tests on three data sets. The first, the TCL [13] (Transports en Commun de Lyon) data set, is made available by the Grand Lyon metropolitan area for research purpose. The second covers the Ile-De-France area and is provided by Île de France Mobilités [14] with permissive license. We denote it IDFM. Note that although it has been used in previous publications, it might be different to the one cited due to regular updates. In [12], for instance, the size of the IDF network is closer to that of the TCL data set. The last is a proprietary data set for whole Korea. For those data sets, we use a mixture of the provided footpaths (if any) and generated footpaths. Closure of the footpaths is not required by the TB algorithm or our adaptation (as opposed to RAPTOR [5] or CSA [6]) but users will often accept to walk, for limited distances, between stations. Hence, for each stop, we include footpaths to all the stops reachable within a distance of 600 m, using a walking speed of 3.6 kph. Data set information is summarized in Table 1.

■ **Table 2** Comparison of preprocessing steps between MS, STD and NP versions.

Data Set	TCL	IDFM	Korea
Preproc. MS (s)	18	521	1326
Preproc. STD (s)	6	141	381
Preproc. NP (s)	4	56	69
Preproc. MS/STD	3.0	3.7	3.5
# transfers MS (in million)	11.7	110.7	259
# transfers STD (in million)	10.9	103.6	245
# transfers NS (in million)	136	1984	3479
# transfers per trip MS	166	315	580
# transfers per trip STD	155	295	571
# transfers per trip NS	1952	5651	7800

Table 2 compares the results of the preprocessing obtained with the standard version (STD) and with the modified version that allows correct results for a selection of modes (MS). As an additional base line, we also run our experiment with a version that performs no pruning, but implements the mode selection in the search phase (NP). As it uses the complete set of transfers, this baseline will give correct solution sets.

As expected, the number of transfers in T is only slightly increased by the modifications of the preprocessing. On the other hand, the duration of the preprocessing is significantly increased. This is probably explained by the interconnection of the different networks: a large part of the lines will be able to connect to most of the different modes and hence, earliest arrival times are updated for nearly all trips and modes.

In order to test the effect of our modifications on query times, we generate 500 random origin-destination pairs for each data set. We compare in Table 3 our 3 implementations (with and without mode selection, and mode selection without pruning). Note that our execution times include the computation of one solution for each element in the Pareto front. The average and maximum number of solutions for the different test sets can be found in Appendix A in Table 6. As expected from Witt [18], the version without pruning is much slower (more than 3 times slower on Korea and IDFM) due to the larger number of transfers in the search phase.

■ **Table 3** Comparison of query times between MS, STD and NP versions.

Data set	Algorithm	EAT (ms)	profile 1H (ms)	profile day (ms)
TCL	MS	12	56	366
TCL	STD	20	55	304
TCL	NP	34	123	702
IDFM	MS	57	157	848
IDFM	STD	57	173	857
IDFM	NP	382	1007	6432
Korea	MS	46	236	1922
Korea	STD	51	239	1940
Korea	NP	148	940	8042

For queries with sets of allowed modes, we try and remove different scheduled modes and look at the influence on query times. Note that for the 3 data sets, the network contains a majority of bus trips. Table 4 compares our results with the no pruning base line. We also provide the results of [12] in Table 5 as an example of integration in the time-expanded model: although the data set used is not exactly identical to ours, the algorithm also builds solutions (only the earliest arrival time one) and results are provided for several mode selections. Note that for [12], the query times are similar for the different mode selections, but with the TB modifications that we propose, we see that removing parts of the public transit network improves the query time. It is expected as the transfers to disabled modes will not be performed during the search, effectively reducing the number of trip segments processed (see Table 7 in Appendix A). This property also holds for the version without pruning, but although forbidding some modes reduces the execution times, the improvement brought by the pruning is still clear, especially on more time consuming profile instances.

13:12 Mode Personalization in Trip-Based Transit Routing

■ **Table 4** Comparisons of query times for several selections of modes.

Data set	Forbidden modes	MS EAT (ms)	NP EAT (ms)	MS profile 1H (ms)	NP profile 1H (ms)	MS profile day (ms)	NP profile day (ms)
TCL	None	12	34	56	123	366	702
TCL	Bus	5	10	16	26	96	128
TCL	Bus, train	5	10	15	29	96	126
TCL	Subway	10	25	41	195	277	528
TCL	Subway, tram, train	10	21	37	88	223	445
IDFM	None	57	382	173	1007	857	6432
IDFM	Bus	17	72	34	126	114	647
IDFM	Bus, tram, train	8	25	18	56	56	257
IDFM	Subway	50	318	135	1621	698	5165
IDFM	Subway, tram, train	51	319	144	816	680	4392
Korea	None	46	148	236	940	1922	8042
Korea	Bus	28	56	47	82	185	340
Korea	Bus, tram, train	25	48	44	76	172	323
Korea	Subway	39	112	227	892	1746	7313
Korea	Subway, tram, train	37	116	216	923	1759	7698

■ **Table 5** Query times of SDALT from [12] for several selections of modes.

Data set	Forbidden modes	EAT (ms)
IDFM - SDALT	None	186
IDFM - SDALT	Bus, train	175
IDFM - SDALT	Subway, tram, train	216

6 Conclusion

In this article, we present an extension of the Trip-Based Public Transit Routing algorithm [18]. It enables the user to select any subset of the possible scheduled modes at query time as the enabled modes for the query. The preprocessing time is increased by the modification, but we show that it guarantees that the Pareto front is returned by the algorithm, and that, similarly to the standard version, it significantly improves the query times. We also prove that the computed transfer set is still correct for latest departure time queries, that we propose as an extension. Query times are not much impacted when all the modes are allowed, and removing any scheduled mode from the list of the enabled modes reduces the computation time significantly, making those personalized queries faster than the regular ones.

A perspective of this work could concern the adaptation of the Trip-Based Public Transit Routing using condensed search trees [19] to mode selection. In his article, Witt propose to a speed-up technique based on the idea of Transfer Patterns [2]. A specific search graph is precomputed for each origin from one-to-all all day profile queries. The result of those queries is of course dependent of the allowed lines and hence the obtained search graphs cannot be used directly to compute optimal queries for all possible mode selections. As the preprocessing time is important even for the standard version (231 hours for Germany on 64 threads), trade-off between correctness and preprocessing execution times might be needed for enabling mode selection at query time.

References

- 1 Christopher L. Barrett, Riko Jacob, and Madhav Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000. doi:10.1137/S0097539798337716.
- 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 *Proceedings of the 18th Annual European Conference on Algorithms: Part I, ESA'10*, pages 290–301, Berlin, Heidelberg, 2010. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=1888935.1888969>.
- 3 Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Algorithm Engineering: Selected Results and Surveys*, chapter Route Planning in Transportation Networks, pages 19–80. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49487-6_2.
- 4 Daniel Delling, Julian Dibbelt, and Thomas Pajor. Fast and exact public transit routing with restricted pareto sets. In Stephen Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 54–65, 2019. doi:10.1137/1.9781611975499.5.
- 5 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. In *Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 130–140, 2012. doi:10.1137/1.9781611972924.13.
- 6 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, *Experimental Algorithms. SEA 2013*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-38527-8_6.
- 7 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-constrained multi-modal route planning. In SIAM, editor, *Proceedings of the 14th Workshop on Algorithm Engineering and Experiments (ALENEX'12)*, pages 118–129, 2012. doi:10.1137/1.9781611972924.12.
- 8 R. Geisberger, P. Sanders, D. Schultes, and D. 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'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-68552-4_24.
- 9 Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '05*, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=1070432.1070455>.
- 10 General Transit Feed Standard (GTFS). <https://developers.google.com/transit/gtfs/reference/>.
- 11 Pierre Hansen. Bicriterion path problems. In Günter Fandel and Tomas Gal, editors, *Multiple Criteria Decision Making Theory and Application. Proceedings of the Third Conference Hagen/Königswinter, West Germany, August 20-24, 1979*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127, Berlin, Heidelberg, 1980. Springer. doi:10.1007/978-3-642-48782-8_9.
- 12 Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo. Efficient computation of shortest paths in time-dependent multi-modal networks. *Journal of Experimental Algorithmics*, 19:1–29, January 2014. doi:10.1145/2670126.
- 13 Data Grand Lyon. <https://data.grandlyon.com/>.
- 14 Île De France Mobilités. Open data. URL: <https://opendata.stif.info>.
- 15 Matthias Müller-Hannemann and Karsten Weihe. Pareto shortest paths is often feasible in practice. In Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela, editors, *Algorithm Engineering. WAE 2001*, pages 185–197, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-44688-5_15.

- 16 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008. doi:10.1145/1227161.1227166.
- 17 Luis Ulloa, Vassilissa Lehoux, and Frédéric Rouland. Trip planning within a multimodal urban mobility. *IET Intelligent Transport Systems*, 12(2):87–92, 2018. doi:10.1049/iet-its.2016.0265.
- 18 Sascha Witt. Trip-based public transit routing. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015*, volume 9294 of *Lecture Notes in Computer Science*, pages 1025–1036, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-48350-3_85.
- 19 Sascha Witt. Trip-based public transit routing using condensed search trees. In Marc Goerigk and Renato Werneck, editors, *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'16)*, number Article No. 10, pages 10:1–10:12, 2016. doi:10.4230/OASIcs.ATMOS.2016.10.

A Additional numerical results

In this section, we add figures about the earliest arrival time and profile queries. In Table 6, the mean and maximum number of solutions for earliest arrival time and profile queries on the different networks are compared. In Table 7, the mean total number of elements in the queues are displayed for the Korean network for each set of forbidden modes used during the experiments.

■ **Table 6** Comparisons of mean and max number of solutions for several selections of modes.

Data set	Forbidden modes	mean - EAT	max - EAT	mean - profile 1H	max - profile 1H	mean - profile day	max - profile day
TCL	None	2.82	7	28.36	117	268.75	1152
TCL	Bus	1.17	5	8.27	89	87.95	975
TCL	Bus, tram, train	1.17	5	8.25	82	87.91	953
TCL	Subway	2.48	6	21.82	87	205.12	778
TCL	Subway, tram, train	2.42	6	19.84	63	176.03	598
IDFM	None	2.15	6	5.78	57	25.2	419
IDFM	Bus	1.12	4	2.1	60	8.8	403
IDFM	Bus, tram, train	1.05	4	2.31	60	13.28	403
IDFM	Subway	2.12	6	9.41	32	14.56	269
IDFM	Subway, tram, train	4.06	10	16.09	41	98.2	337
Korea	None	2.94	10	32.78	80	317.84	791
Korea	Bus	1.32	5	11.72	44	105.11	320
Korea	Bus, tram, train	1.32	5	11.71	44	105.28	324
Korea	Subway	2.59	7	26.44	62	224.5	695
Korea	Subway, tram, train	2.58	7	28.17	63	273.1	695


■ **Table 7** Comparisons of mean queue sizes for several selections of modes on the Korean network.

Data set	Forbidden modes	MS	NP	MS	NP	MS	NP
		EAT (k)	EAT (k)	profile 1H (k)	profile 1H (k)	profile day (k)	profile day (k)
Korea	None	35.6	87.5	183.2	373.8	1881.9	3536.2
Korea	Bus	9.7	11.3	13.8	17.0	71.6	76.2
Korea	Bus, tram, train	8.5	9.4	13.5	14.5	65.4	68.3
Korea	Subway	30.7	78.2	169.6	358.0	1668.1	3213.3
Korea	Subway, tram, train	28.6	72.8	166.0	350.6	1650.5	3193.8


An Asymptotically Optimal Approximation Algorithm for the Travelling Car Renter Problem

Lehilton L. C. Pedrosa 

Institute of Computing, University of Campinas, SP, Brazil
lehilton@ic.unicamp.br

Greis Y. O. Quesquén 

Institute of Computing, University of Campinas, SP, Brazil
greis.quesquen@students.ic.unicamp.br

Rafael C. S. Schouery 

Institute of Computing, University of Campinas, SP, Brazil
rafael@ic.unicamp.br

Abstract

In the classical Travelling Salesman Problem (TSP), one wants to find a route that visits a set of n cities, such that the total travelled distance is minimum. An often considered generalization is the Travelling Car Renter Problem (CaRS), in which the route is travelled by renting a set of cars and the cost to travel between two given cities depends on the car that is used. The car renter may choose to swap vehicles at any city, but must pay a fee to return the car to its pickup location. This problem appears in logistics and urban transportation when the vehicles can be provided by multiple companies, such as in the tourism sector. In this paper, we consider the case in which the return fee is some fixed number $g \geq 0$, which we call the Uniform CaRS (UCaRS). We show that, already for this version, there is no $o(\log n)$ -approximation algorithm unless $P = NP$. The main contribution is an $O(\log n)$ -approximation algorithm for the problem, which is based on the randomized rounding of an exponentially large LP-relaxation.

2012 ACM Subject Classification Theory of computation \rightarrow Approximation algorithms analysis; Theory of computation \rightarrow Routing and network design problems

Keywords and phrases Approximation Algorithm, Travelling Car Renter Problem, LP-rounding, Separation Problem

Digital Object Identifier 10.4230/OASICS.ATMOS.2019.14

Funding Supported by grant #2015/11937-9, São Paulo Research Foundation (FAPESP), and grants #425340/2016-3, #313026/2017-3, #308689/2017-8, #425806/2018-9, #422829/2018-8, National Council for Scientific and Technological Development (CNPq).

1 Introduction

Transportation is a key aspect of modern society, as it connects people, businesses and services. Related problems are spread in many areas, such as logistics [18], supply-chain [34] and, in particular, urban transportation [12, 27, 2]. To the latter, the transportation decisions are determinant to, e.g., the diffusion of the population with the growth of cities and urban areas, the movement of people who commute from or to school and work, and easy access of tourists and visitors to events or leisure activities. Vehicles comprise the most used means in urban transportation, which can be divided into two main traffic modes: public transportation and private cars. The first is usually considered to be a cheap form of transport and helps cities to reduce traffic congestion and the overall level of pollution. The second mode is more flexible and convenient but has a relatively higher price and consumption [23].



© Lehilton L. C. Pedrosa, Greis Y. O. Quesquén, and Rafael C. S. Schouery;
licensed under Creative Commons License CC-BY

19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2019).

Editors: Valentina Cacchiani and Alberto Marchetti-Spaccamela; Article No. 14; pp. 14:1–14:15

OpenAccess Series in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A special trend in urban transportation, especially for private cars, is the offer of individual transport services, such as sharing or rental. Car sharing allows customers to reserve, access and use vehicles remotely [7], and vehicles are rented on an hourly basis by members looking for a high level of convenience and a low level of commitment [1]. On the other hand, car rental may be less straightforward but allows rents on a day, week, month or even on a yearly basis. Independent of the form, in most cases, the vehicle rental problems are discussed from a company's point of view [31], whose goal is to maximize the total profit or the number of customers served. Some works, however, consider problems from a client's perspective [17, 15, 5, 26], who has a set of available rental services and wants to choose which services to hire to minimize its total cost.

A problem that aims at minimizing costs associated with the service's user appears in the tourism industry [5]. Goldberg et al. [17] considered the Travelling Car Renter Problem (CaRS), which can be described as follows. A tourist wants to visit a set of cities and, to this, there are many rental services which offer a set of available cars. The cost to rent each car depends on the route taken by this car from the pickup point to its return location. Moreover, for each time the tourist rents a car, there is an additional fee, which corresponds to the cost to return the car to its pickup location. The cost of both the route and the return fee may depend on the selected car. The objective is to find a closed route that visits all the cities and comes back to the original place, such that the cost of all rented cars is minimum. The classical Travelling Salesman Problem (TSP) is a particular case of this problem. Indeed, TSP corresponds to instances of CaRS in which there is only one available car, and the return fee is zero. Since TSP is NP-hard, CaRS is NP-hard as well.

In this paper, we consider the version of CaRS in which the return fee is fixed regardless of the return point, and is the same for every car, which we call the *Uniform CaRS* (UCaRS). Formally, an instance of UCaRS is composed of set of cities $V = \{1, 2, \dots, n\}$, a set of cars $C = \{1, 2, \dots, r\}$, a return fee $g \geq 0$ and, for each car $i \in C$, an associated cost function $d_i : V \times V \rightarrow \mathbb{R}_{\geq 0}$. A solution is a sequence of walks P_1, P_2, \dots, P_s on V , associated with indexes $e_1, e_2, \dots, e_s \in C$, respectively, such that $P = (P_1 P_2 \dots P_s)$ is a closed route, and each city of V appears in P at least once. Denote by $E(H)$ the set of edges of a graph H . The objective of the problem is to find a solution which minimizes the sum

$$\sum_{i=0}^s \left(\sum_{(u,v) \in E(P_i)} d_{e_i}(u,v) + g \right),$$

which is the cost of edges in each walk P_i corresponding to car e_i plus the return fees.

1.1 Our contribution and summary of techniques

In this paper, we study UCaRS from the perspective of approximation algorithms. An algorithm for a minimization problem is an $\alpha(n)$ -approximation algorithm if it runs in polynomial time and, for every instance I of size n of the problem, it returns a solution with value at most $\alpha(n) \cdot \text{OPT}(I)$, where $\text{OPT}(I)$ the value of an optimal solution for I . An $\alpha(n)$ -approximation algorithm is said to be asymptotically optimal if any approximation algorithm has factor $\Omega(\alpha(n))$ unless $P = NP$.

We assume that for each car $i \in C$, the function d_i is metric, i.e., each function d_i satisfies the following assumptions:

1. (*symmetry*) for every $u, v \in V$, $d_i(u, v) = d_i(v, u)$, and
2. (*triangle inequality*) for every $u, v, w \in V$, $d_i(u, v) \leq d_i(u, w) + d_i(w, v)$.

Intuitively, the symmetry means that going from a city u to a city v using a car i costs the same as going from city v to city u using the same car. Also, the triangle inequality means that, to go from u to v , taking a direct route is always one of the most economical choices. If no restrictions on the cost functions are made, then it is very unlikely that the problem would have any approximation, since this version of UCARS generalizes the non-metric TSP, which cannot be approximated by any computable function unless $P = NP$ [32].

For TSP, one requires that a solution P is a Hamiltonian cycle, i.e., it is a closed walk which visits every city of V exactly once. For most applications, this assumption is without loss of generality, since the triangle inequality implies that any closed walk corresponds to a cycle spanning the same cities of no larger cost. For UCARS, however, we observe that, if a solution were required to visit each city exactly once, then the problem would not admit any approximation algorithm, even under the assumption that every cost function d_i is metric. Indeed, we show that if such a variant has an $\alpha(n)$ -approximation algorithm for any computable function $\alpha(n)$, then one can solve the Hamiltonian Cycle Problem (HamC) in polynomial time. Since HamC is NP-hard, this would imply $P = NP$.

As for hardness results, we show that UCARS is as hard to approximate as the Set Cover Problem (SC), which implies that UCARS has no approximation algorithm with factor $o(\log n)$, unless $P = NP$ [20, 9]. Then, we note that there is a natural correspondence between instances of UCARS and instances of Group TSP (G-TSP), which is a generalization of TSP whose instance is comprised of a family of groups of cities, and whose goal is to find a route that visits at least one city of each group. Namely, we show that UCARS can be reduced to G-TSP preserving the approximation factor, implying an approximation factor of $O(\log^2 n \log r)$ for UCARS by results from Garg et al. [13] and Fakcharoenphol et al. [10].

Our main contribution is a better approximation algorithm, which achieves a factor of $O(\log n)$. This algorithm is asymptotically optimal because of the proved lower bound. The reduction to G-TSP gives only a polylogarithmic factor, thus our algorithm takes a different approach. We observe that a solution of UCARS can be seen as a set of walks which cover the set of cities in V . Recall that in SC, given a family of subsets of V , each with a given weight, one wants to select some subsets such that the whole set V is covered, and the total weight of selected subsets is minimized. This suggests that an instance of UCARS could be reduced to an instance of SC, where the subsets correspond to possible sets of walks whose weights are the cost of visiting the corresponding cities plus the return fee.

This reduction does not work as is, however, for two main reasons. First, there are exponentially many distinct walks, and thus it does not run in polynomial time. Second, a solution for the SC instance does not take into account connectivity, and thus the union of the walks attained from the SC solution may not form a closed walk. Yet, we use this reduction to obtain a linear programming (LP) formulation which gives a lower bound on the optimal value. While this formulation is still exponential, we show how to obtain a good solution to its relaxation in polynomial time. Then, in a first phase, we obtain a set of walks that visit every city, by using a randomized algorithm which rounds the LP solution, and whose expected cost is at an $O(\log n)$ factor of the optimal value. The set of walks might be disconnected, thus, in a second phase, we show how to combine the walks into a single closed walk of not much larger cost.

The most involved and technical part of our algorithm is finding a feasible solution for the LP formulation with bounded value. To this, we note that the number of variables is exponential, but there are only polynomially many constraints. Then, an optimal solution can be obtained if the corresponding dual separation problem can be solved in polynomial time. This separation problem, for its turn, is NP-hard, and even hard to approximate by any constant. Thus, we show that a restriction of the original LP can still be used to obtain a lower bound, and that, for this restriction, we can compute an approximate solution.

1.2 Related works

CaRS was introduced by Silva et al. [30, 17], who proposed a memetic algorithm and a hybridization between GRASP and VND heuristics. Following, several other heuristic methods have been proposed for the problem, such as ant-colony optimization [29], transgenetic algorithms [31, 16], and evolutionary algorithms [11, 5, 8, 26, 6]. Exact algorithms based on mathematical formulations have also been proposed, which include mixed integer programming formulations [24, 5, 26, 25], and models based on the Quadratic Assignment Problem (QAP) or on network-flow formulations [14]. To the best of our knowledge, no approximation algorithms for CaRS or one of its variants have been discussed before this work.

It is folklore that the so-called Double-MST is a 2-approximation for the metric TSP. This algorithm calculates a Minimum Spanning Tree (MST) of the input graph G , then duplicates all its edges to complete a Eulerian graph, i.e., a graph for which there is a route visiting each edge exactly once. By making shortcuts, one obtains a Hamiltonian cycle whose cost no more than twice the cost of the MST.

SC is a classical NP-hard problem, and it is NP-hard to approximate SC by a factor $(1 - \varepsilon) \log n$, where n is the size of the ground set, and $\varepsilon > 0$ is an arbitrary constant [20, 9]. For this problem, a simple greedy algorithm is an $O(\log n)$ approximation [4]. It is well known that the same factor can also be obtained by a probabilistic algorithm which rounds its LP relaxation by independently selecting each set according to the corresponding variable [33].

G-TSP is a generalization of both TSP and SC, and thus the SC hardness holds for G-TSP as well. In fact, even if the cost function corresponds to the Euclidean distance, it is hard to approximate G-TSP by any constant [28]. A closely related problem is the Group Steiner Problem (G-ST). In this problem, an instance is composed of an edge-weighted graph on n vertices and m groups of vertices. The objective is to find a minimum weight tree that contains at least one vertex from every group. For this problem, it is unlikely that a $O(\log^{2-\varepsilon} m)$ -approximation algorithm exist, for any $\varepsilon > 0$ [21]. The best-known algorithm has a factor of $O(\log^2 n \log m)$ [13, 3].

1.3 Paper organization

The remainder of the paper is organized as follows. In Section 2, we discuss the inapproximability of UCaRS and give the reduction from UCaRS to G-TSP. In Section 3, we present a covering integer programming formulation which gives a lower bound for the problem. Then, we show how to use the LP relaxation to derive a solution for the problem whose cost is at most at a factor $O(\log n)$ of the optimal value. This is done by a randomized LP rounding algorithm which takes as the input a pre-computed LP solution. In Section 4, we show how to obtain such a solution. First, we describe the separation problem corresponding to the LP formulation and observe that it is NP-hard. Then, we consider a restriction of the formulation for which the corresponding separation problem can be approximated and show that the value of this restriction is not too far from the optimal value. In Section 5, we discuss the used techniques and possible extensions to similar problems.

2 Inapproximability

As in the case of TSP, if the cost functions of an instance of UCaRS are arbitrary, then the problem cannot be approximated by any function computable in polynomial time unless $P = NP$. Next, we observe that, for the version of the problem which requires a solution to be a Hamiltonian cycle, this hardness holds even if we assume that each cost function d_i is symmetric and satisfies the triangle inequality.

► **Theorem 1.** *Let $\alpha(n)$ be a function computable in polynomial time with $\alpha(n) \geq 1$. If $P \neq NP$, then there is no $\alpha(n)$ -approximation algorithm for the problem of, given an instance of UCARS, finding a solution P which is a Hamiltonian cycle with minimum cost.*

Thus, from now on, we allow routes which visit the same city more than once. Even if every cost function is metric and the solution is not required to be a Hamiltonian cycle, next lemma states that UCARS is SC-hard, i.e., there is a reduction from the unweighted version of SC to UCARS. Recall that an instance of unweighted SC is composed of subsets S_1, S_2, \dots, S_m of a ground set E . The objective is to find a set of indices e_1, e_2, \dots, e_s such that $S_{e_1} \cup S_{e_2} \cup \dots \cup S_{e_s} = E$ and s is minimum.

► **Theorem 2.** *If there is an α -approximation for UCARS, then there is an α -approximation for unweighted SC.*

Using the hardness result for SC [9], this immediately implies the following.

► **Corollary 3.** *If $P \neq NP$, there is no $(1 - \varepsilon) \log n$ -approximation for UCARS, for any $\varepsilon > 0$.*

On the other hand, one can reduce an instance of UCARS to an instance of G-TSP, such that the optimal value of the first is at a constant factor of the optimal value of the second, and vice-versa. An instance of G-TSP is formed by a graph G , a cost function on the edges d' , and sets $S_1, S_2, \dots, S_m \subseteq V(G)$. A solution is a closed walk P such that $S_i \cap V(P) \neq \emptyset$ for $1 \leq i \leq m$. The objective is to find a solution which minimizes the sum $\sum_{(u,v) \in E(P)} d'(u,v)$.

► **Lemma 4.** *If G-TSP admits an α -approximation, then UCARS admits an α -approximation.*

Since an $\alpha(n)$ -approximation for G-ST implies a $2\alpha(n)$ -approximation for G-TSP, using the $O(\log^2 n \log m)$ -approximation for G-ST by Garg et al. [13, 10], we obtain the following.

► **Theorem 5.** *There exists an $O(\log^2 n \log r)$ -approximation for UCARS.*

3 An LP rounding algorithm

In this section, we describe the LP rounding algorithm with factor $O(\log n)$. We start with an integer programming formulation which gives a lower bound for the optimal value.

3.1 A covering formulation

The main idea for the algorithm comes from the observation that a solution consists of a set of walks that visit the whole set of cities. Each walk corresponds to a pair of a car i and a subset of cities S , which we call a *component*. The cost of each component corresponds to the total weight of its edges plus the return fee. For the sake of simplicity, instead of considering every possible walk for each subset of cities, we consider each subset only once for each car. Thus, instead of looking for a set of segments that forms a closed walk, we are only interested in finding a set of connected components that cover the set of cities. The cost of each representative connected component corresponds to the weight of a minimum spanning tree of S according to d_i plus the return fee g . Note that any minimum walk which visits S has cost at most twice that of the minimum spanning tree. This is sufficient for our purposes since we are only interested in an asymptotic factor $O(\log n)$.

In the following, we denote by \mathcal{S} the set of all pairs (i, S) with $S \subseteq V$ and $i \in C$. For each $(i, S) \in \mathcal{S}$, we denote by $\text{MST}_i(S)$ the cost of a minimum spanning tree of S with respect to d_i . The cost of (i, S) is defined as $\text{cost}(i, S) = \text{MST}_i(S) + g$. In the following integer

14:6 Optimal Approximation Algorithm for the Travelling Car Renter Problem

linear program, for each $(i, S) \in \mathcal{S}$, there is a binary variable $x_{(i,S)}$ which indicates whether the component (i, S) belongs to the solution.

$$\begin{aligned} & \text{minimize} && \sum_{(i,S) \in \mathcal{S}} x_{(i,S)} \text{cost}(i, S) \\ & \text{subject to} && \sum_{(i,S) \in \mathcal{S}: v \in S} x_{(i,S)} \geq 1, \quad \forall v \in V, \\ & && x_{(i,S)} \in \{0, 1\}, \quad \forall (i, S) \in \mathcal{S}. \end{aligned} \tag{IP}$$

In the following, given an integer linear program or a linear program (Q) , denote by $\text{OPT}(Q)$ the optimal value of (Q) . Also, we denote by OPT the value of an optimal solution for the instance of UCaRS. We observe that a solution for the UCaRS instance induces a feasible solution of (IP), thus $\text{OPT}(\text{IP})$ is a lower bound for the optimal value.

► **Lemma 6.** *Consider an instance of UCaRS and the corresponding formulation (IP). Let OPT be the value of an optimal solution for this instance. Then, $\text{OPT}(\text{IP}) \leq \text{OPT}$.*

Let (P) be the linear relaxation of (IP). Notice that (P) has an exponential number of variables and therefore we do not know how to solve this problem directly. However, we can obtain an approximate solution with only a polynomial number of non-zero variables, according to the following lemma, which is a central result for the algorithm. The proof is given in Section 4.

► **Lemma 7.** *There is an algorithm that, in polynomial time, finds a feasible solution for (P) whose value is at most $c \cdot \text{OPT}(\text{IP})$, for some constant c .*

3.2 Rounding the fractional solution

Next, we assume that we are given a solution x of (P), and the objective is to find a solution for the instance of UCaRS by rounding the value of variables $x_{(i,S)}$ for each $(i, S) \in \mathcal{S}$. The rounding algorithm can be broken into three phases. In the first phase, we find a set of components which cover V . In the second phase, we complement the set of components to obtain a connected cover. In the third phase, we turn the set of components into a route that visits each city at least once.

Covering phase

Note that for each $(i, S) \in \mathcal{S}$, we can assume that $0 \leq x_{(i,S)} \leq 1$. Thus, the value of $x_{(i,S)}$ can be interpreted as a probability that component (i, S) is used in an integral solution. We start with an empty set of components \mathcal{C} and execute the following iteration: include each component (i, S) in \mathcal{C} independently with probability $x_{(i,S)}$. The expected cost of the included components in this iteration is at most the value of x , but this does not guarantee that every city is covered by some component of \mathcal{C} . If we repeat this process a sufficient number of times, then the probability of a city remaining uncovered tends to zero. After $\log n$ iterations, the probability that a city is uncovered is small, then for each uncovered city v , we include a component corresponding to a singleton $\{v\}$ and an arbitrary car, say, the first one. The steps for the covering phase are summarized in Figure 1.

The following lemma states that indeed the obtained set covers all the cities. The proof follows directly from the algorithm.

► **Lemma 8.** *Let \mathcal{C} be the set of components returned by the covering phase of the algorithm. Then, for every $v \in V$, there is a component $(i, S) \in \mathcal{C}$ such that $v \in S$.*

1. Compute an (P) solution x using Lemma 7.
2. Let $\mathcal{C} \leftarrow \emptyset$ and repeat $\lceil \log n \rceil$ times:
 - For each (i, S) include (i, S) in \mathcal{C} with probability $x_{(i,S)}$.
3. For each $v \in V$:
 - If $v \notin S$ for all $(i, S) \in \mathcal{C}$, then add $(1, \{v\})$ to \mathcal{C} .
4. Return \mathcal{C} .

■ **Figure 1** First phase of the rounding algorithm: covering.

Lemma 10 estimates the expected cost of \mathcal{C} . First, using standard analysis, we bound the probability that a city is not covered by one component included in the random step.

► **Lemma 9.** *Let \mathcal{C}' be the set of components included in any iteration of step 2 of the covering phase of the algorithm. Also, let V' be the set of cities v for which there exists $(i, S) \in \mathcal{C}'$ with $v \in S$. Then, $\Pr(v \notin V') \leq 1/n$ for every $v \in V$.*

► **Lemma 10.** *Let \mathcal{C} be the set of components returned by the covering phase. Then*

$$\mathbb{E} \left[\sum_{(i,S) \in \mathcal{C}} \text{cost}(i, S) \right] \leq O(\log n) \cdot \text{OPT}.$$

Proof. Consider set \mathcal{C}_ℓ of components (i, S) included in the iteration ℓ of step 2. Since each component (i, S) is included with probability $x_{(i,S)}$, the expected cost of these components is

$$\begin{aligned} \mathbb{E} \left[\sum_{(i,S) \in \mathcal{C}_\ell} \text{cost}(i, S) \right] &= \sum_{(i,S) \in \mathcal{C}} \Pr((i, S) \in \mathcal{C}_\ell) \text{cost}(i, S) \\ &= \sum_{(i,S) \in \mathcal{C}} x_{(i,S)} \text{cost}(i, S) \leq c \cdot \text{OPT}(\text{IP}), \end{aligned}$$

where the inequality comes from the fact that, from Lemma 7, the objective value for x is at most $c \cdot \text{OPT}(\text{IP})$, for some constant c . Thus, the expected cost of the set of components \mathcal{C}' drawn in step 2 can be bounded as

$$\mathbb{E} \left[\sum_{(i,S) \in \mathcal{C}'} \text{cost}(i, S) \right] \leq \sum_{\ell=1}^{\lceil \log n \rceil} \mathbb{E} \left[\sum_{(i,S) \in \mathcal{C}_\ell} \text{cost}(i, S) \right] \leq \lceil \log n \rceil \cdot c \cdot \text{OPT}(\text{IP}).$$

Let V' be the set of vertices covered by some component of \mathcal{C}' . For each $v \in V \setminus V'$, a new component of the form $(1, \{v\})$ is added in step 3 of the first phase of the algorithm. Using Lemma 9, this happens with probability at most $1/n$. Let \mathcal{C}_+ be the set of such components and observe that $\mathcal{C}_+ = \mathcal{C} \setminus \mathcal{C}'$. Also, note that for any component $(i, S) \in \mathcal{C}_+$, a minimum spanning tree of S contains no edges, and thus $\text{cost}(i, S) = g$. It follows that

$$\mathbb{E} \left[\sum_{(i,S) \in \mathcal{C}_+} \text{cost}(i, S) \right] = \sum_{v \in V} \Pr(v \notin V') \text{cost}(1, \{v\}) \leq \sum_{v \in V} \frac{1}{n} g = g.$$

Observe that any feasible solution of (IP) has at least one component, thus $g \leq \text{OPT}(\text{IP})$. From Lemma 6, we know that $\text{OPT}(\text{IP}) \leq \text{OPT}$. Combining everything, the costs of all components in \mathcal{C} add up to $O(\log n) \cdot \text{OPT}$. ◀

1. Let $\mathcal{D} \leftarrow \emptyset$.
2. While $\mathcal{C} \cup \mathcal{D}$ induces a disconnected graph H :
 - Find vertices v and v' in distinct components of H which minimize $d_{\min}(v, v')$.
 - Add $(j, \{v, v'\})$ to \mathcal{D} , where j is such that $d_j(v, v') = d_{\min}(v, v')$.
3. Return \mathcal{D} .

■ **Figure 2** Second phase of the rounding algorithm: connection.

Connection phase

After the first phase, the set of components \mathcal{C} covers all cities V , but they are possibly disconnected. So we want to connect \mathcal{C} by selecting additional components, \mathcal{D} , each of which connect two components of \mathcal{C} . Since each such an additional component corresponds to an edge which connects two cities, v, v' , we may simply select the car i with the smallest cost $d_i(v, v')$. Thus, we consider an edge-weight function d_{\min} such that for every pair of cities $v, v' \in V$, we define $d_{\min}(v, v') = \min\{d_i(v, v') : i \in \mathcal{C}\}$.

We execute the following. Start with an empty set \mathcal{D} . Then, while the graph H induced by the minimum spanning trees of $\mathcal{C} \cup \mathcal{D}$ is disconnected, find vertices v and v' such that v and v' are in distinct connected components of H , and $d_{\min}(v, v')$ is minimum. Include component $(j, \{v, v'\})$ in \mathcal{D} , where j is such that $d_j(v, v') = d_{\min}(v, v')$. After the last iteration, the minimum spanning trees of $\mathcal{C} \cup \mathcal{D}$ induce a connected graph H which contains all vertices. The steps for the connection phase are summarized in Figure 2.

To bound the cost of components in \mathcal{D} , we need the following auxiliary lemma.

► **Lemma 11.** *Let T be a minimum spanning tree of V with respect to d_{\min} . Then*

$$\sum_{(v, v') \in E(T)} d_{\min}(v, v') \leq OPT.$$

Proof. Consider an optimal solution $P = P_1, P_2, \dots, P_s$ of UCARS. For each $(v, v') \in E(P)$, let $\phi(v, v') \in \mathcal{C}$ be the car associated with edge (v, v') . Let T be a minimum spanning tree of P . Since P spans every vertex of V , T is also a spanning tree of V . We get

$$\sum_{(v, v') \in E(T)} d_{\min}(v, v') \leq \sum_{(v, v') \in E(T)} d_{\phi(v, v')}(v, v') \leq OPT,$$

where the last inequality holds because T is a minimum spanning tree of P . ◀

The edge cost of components of \mathcal{D} is bounded by the next lemma.

► **Lemma 12.** *Let \mathcal{D} be the set of components returned by the connection phase of the algorithm. Then*

$$|\mathcal{D}| \leq |\mathcal{C}| \quad \text{and} \quad \sum_{(i, S) \in \mathcal{D}} MST_i(S) \leq OPT.$$

Routing phase

Given the set of components \mathcal{C} and \mathcal{D} , we construct a closed walk which visits all cities by considering each component at a time. Let $r \in V$ be an arbitrary vertex and start a trivial closed walk P composed only of r . Now, while there are components of $\mathcal{C} \cup \mathcal{D}$ which have not been considered yet, find one such a component (i, S) such that $S \cap V(P) \neq \emptyset$ and let

1. Let $r \in V$ and make $P \leftarrow (r)$.
2. Make $\mathcal{N} \leftarrow \mathcal{C} \cup \mathcal{D}$.
3. While $\mathcal{N} \neq \emptyset$:
 - Find $(i, S) \in \mathcal{N}$ such that $S \cap V(P) \neq \emptyset$, and let $v \in S \cap V(P)$.
 - Construct a cycle P' of $S \setminus V(P) \cup \{v\}$.
 - For each edge $(u, u') \in E(P')$, make $\phi(u, u') \leftarrow i$.
 - Insert P' into P .
 - Make $\mathcal{N} \leftarrow \mathcal{N} \setminus \{(i, S)\}$.
4. Return P, ϕ .

■ **Figure 3** Third phase of the rounding algorithm: routing.

$v \in S \cap V(P)$. Then, from a minimum spanning tree T' of S with respect to d_i , construct a cycle P' of $S \setminus V(P) \cup \{v\}$. Observe that, by doubling the edges of T' and then taking shortcuts, one can build such a cycle with cost at most $2 \text{MST}_i(S)$. The edges of P' are labelled with the index of car i , and the walk P is extended by including P' .

Note that P visits all vertices and can be decomposed into a sequence of maximal walks with the same label. Therefore, it induces a feasible solution for the instance of UCaRS. The steps for the routing phase are summarized in Figure 3.

The cost of the returned solution corresponds to travelled edges and to the return fees. The edge cost is bounded as follows.

► **Lemma 13.** *Let P, ϕ be the solution returned by the routing phase of the algorithm. Then,*

$$\sum_{(v,v') \in E(P)} d_{\phi(v,v')}(v, v') \leq 2 \cdot \sum_{(i,S) \in \mathcal{C} \cup \mathcal{D}} \text{MST}_i(S).$$

Proof. Recall that to each component $(i, S) \in \mathcal{C} \cup \mathcal{D}$, one constructs a cycle P' by doubling the edges of minimum spanning tree of S and taking shortcuts, then the cycle P' has cost at most $2 \text{MST}_i(S)$. Now observe that $E(P)$ is the union of the edges of cycles P' and each such a cycle P' corresponds to a distinct component $(i, S) \in \mathcal{C} \cup \mathcal{D}$. ◀

Finally, we can show the main result.

► **Theorem 14.** *There exists a randomized $O(\log n)$ -approximation algorithm for UCaRS.*

Proof. By Lemma 7, the solution x of (IP) is computed in polynomial time. Since each of the three phases of the LP rounding algorithm also runs in polynomial time, the whole algorithm is polynomial.

Using Lemma 13, the expected edge cost of edges is

$$\begin{aligned} \mathbb{E} \left[\sum_{(v,v') \in E(P)} d_{\phi(v,v')}(v, v') \right] &\leq 2 \cdot \mathbb{E} \left[\sum_{(i,S) \in \mathcal{C}} \text{MST}_i(S) \right] + \mathbb{E} \left[\sum_{(i,S) \in \mathcal{D}} \text{MST}_i(S) \right] \\ &\leq O(\log n) \cdot \text{OPT} + \text{OPT} = O(\log n) \cdot \text{OPT}, \end{aligned}$$

where the last inequality follows from Lemmas 10 and 12.

Now it remains to bound the expected cost of the return fees. There is a return fee of cost g for each vertex in the walk P where there is a switching car, thus it is sufficient to count the number of vertices in P whose adjacent edges have different labels. In each iteration of step 3 of the routing phase of the algorithm, the labels of every edge in the inserted subwalk P' are the same, thus the number of swaps in P increases by at most 2. Since this step is executed for $|\mathcal{C}| + |\mathcal{D}|$ iterations, the total expected return fee is at most

14:10 Optimal Approximation Algorithm for the Travelling Car Renter Problem

$$\mathbb{E}[2 \cdot (|\mathcal{C}| + |\mathcal{D}|) \cdot g] \leq \mathbb{E}[4 \cdot |\mathcal{C}| \cdot g] \leq \mathbb{E}\left[4 \cdot \sum_{(i,S) \in \mathcal{C}} \text{cost}(i, S)\right] \leq O(\log n) \cdot \text{OPT},$$

where the first inequality comes from Lemma 12, the second inequality is due to $g \leq \text{cost}(i, S)$, and the last inequality comes from Lemma 10.

Adding up the expected costs of edges and the return fees, we conclude that the expected cost of the returned solution cost is at most $O(\log n) \cdot \text{OPT}$, and the theorem follows. ◀

4 Approximating the LP-relaxation

The objective of this section is to find a feasible solution of (P) and prove Lemma 7. The main ingredient will be solving a relaxed version of the separation problem and using the Ellipsoid method. For the sake of completeness, we begin by briefly reviewing this method and identifying the corresponding separation problem.

4.1 The separation problem

Remember that (P) has an exponential number of variables. A strategy to tackle this difficulty is solving the separation problem corresponding to the dual formulation. The idea is that, if the number of variables in the dual is bounded by a polynomial, then one can use the Ellipsoid method. Informally, this method consists of iteratively picking a candidate solution y and querying an *oracle* for the separation problem. Given vector y , the separation problem is the task of either deciding that y is feasible, or finding a violated constraint. If y is not feasible, another solution y is picked, until a feasible optimal solution is found

► **Theorem 15** ([19]). *Let (Q) be a non-empty linear program. Then an optimal solution can be found querying an oracle for its separation problem only a polynomial number of times.*

The dual formulation of (P) is given next.

$$\begin{aligned} & \text{maximize} && \sum_{v \in V} y_v \\ & \text{subject to} && \sum_{v \in S} y_v - \text{MST}_i(S) \leq g \quad \forall (i, S) \in \mathcal{S} \\ & && y_v \geq 0 \quad \forall v \in V. \end{aligned} \tag{D}$$

For a set $A \subseteq V$, we define $y(A) = \sum_{v \in A} y_v$. Note that the separation problem consists of finding $(i, S) \in \mathcal{S}$ for which the constraint $y(S) - \text{MST}_i(S) \leq g$ is violated, or showing that there is no such constraint. Since the number of cars is polynomial, we can solve the problem separately for each $i \in C$. For some $i \in C$, it is enough to find $S \subseteq V$ such that $y(S) - \text{MST}_i(S)$ is maximum and verify that this value is greater than g . Unfortunately, this problem corresponds to the Net Worth Maximization Problem (NWMP), which has no constant-factor approximation algorithm unless $P = NP$ [22].

4.2 Cost-restricted components

To be able to use the Ellipsoid method, we will define a slightly different linear program, but whose separation problem is easier. If one considers only sets S for which $\text{MST}_i(S)$ is small, i.e., less than g , then maximizing $y(S)$ is a good approximation for the optimization version of the separation problem. Since, a priori, $\text{MST}_i(S)$ is greater than g , we will consider a relaxation of (D) which contains only constraints where $\text{MST}_i(S) \leq g$.

Let $\mathcal{S}_{\leq g}$ be the subset of components $(i, S) \in \mathcal{S}$ such that $\text{MST}_i(S) \leq g$, and define $(\mathbf{P}_{\leq g})$ to be the restriction of (\mathbf{P}) which contains only variables corresponding to $(i, S) \in \mathcal{S}_{\leq g}$. We note that the value of this restriction is at a constant factor of the value of (\mathbf{IP}) .

► **Lemma 16.** $\text{OPT}(\mathbf{P}_{\leq g}) \leq 4 \cdot \text{OPT}(\mathbf{IP})$.

Proof. We build a feasible solution for $(\mathbf{P}_{\leq g})$ from a feasible solution for (\mathbf{IP}) . For each $U \subseteq \mathcal{S}$, define $\text{cost}(U) = \sum_{(i,S) \in U} \text{cost}(i, S)$.

Note that a solution x of \mathbf{IP} corresponds to a set $U^* \subseteq \mathcal{S}$ such that $\text{cost}(U^*) = \text{OPT}(\mathbf{IP})$. We build an integral solution of $(\mathbf{P}_{\leq g})$ corresponding to a set $U \subseteq \mathcal{S}_{\leq g}$. If $g = 0$, we add component $(1, \{v\})$ to U for $v \in V$, thus $\text{cost}(U) = 0$, and we are done. Thus, assume $g > 0$.

Consider a component $(i, S) \in U^*$. We can partition S into parts S' such that $\text{MST}_i(S') \leq g$. First, let T be a minimum spanning tree of S with respect to d_i . Then, obtain a closed walk P by doubling the edges of T and finding an Eulerian walk whose edge weight is at most $2 \cdot \text{MST}_i(S)$. Finally, break P into disjoint maximal subwalks whose edges add up to at most g . To do this, greedily find a maximal prefix P' of P whose edges weights do not add up more than g , and remove $V(P')$ from P . Observe that the total weight of the subwalks is at most $2 \cdot \text{MST}_i(S)$, and that the number of subwalks is at most $\lceil (2 \cdot \text{MST}_i(S))/g \rceil$. Therefore, the component (i, S) can be replaced by a set of components whose total cost is $2 \cdot \text{MST}_i(S) + \lceil (2 \cdot \text{MST}_i(S))/g \rceil g \leq 4 \cdot \text{MST}_i(S) + g \leq 4 \cdot \text{cost}(i, S)$.

Repeating this procedure for each component of U^* , we obtain a set U of components (i, S) such that $\text{MST}_i(S) \leq g$ and $\text{cost}(U) \leq 4 \cdot \text{cost}(U^*)$. Since every vertex is in some component, U induces a feasible solution for $(\mathbf{P}_{\leq g})$, and the lemma holds. ◀

4.3 Constructing an approximate solution

Let $(\mathbf{D}_{\leq g})$ be the dual formulation of $(\mathbf{P}_{\leq g})$. Note that $(\mathbf{D}_{\leq g})$ is a relaxation of (\mathbf{D}) which contains only constraints corresponding to $(i, S) \in \mathcal{S}_{\leq g}$. Instead of solving $(\mathbf{D}_{\leq g})$, we will consider a different linear program in which we replace constraints $\sum_{v \in S} y_v - \text{MST}_i(S) \leq g$ by constraints $\sum_{v \in S} y_v \leq g$. The resulting linear program is described below.

$$\begin{aligned} & \text{maximize} && \sum_{v \in V} y_v \\ & \text{subject to} && \sum_{v \in S} y_v \leq g, \quad \forall (i, S) \in \mathcal{S}_{\leq g}, \\ & && y_v \geq 0, \quad \forall v \in V. \end{aligned} \tag{D'_{\leq g}}$$

Now, the separation problem of $(\mathbf{D}'_{\leq g})$ corresponds to, given a vector y and an index i , find a set $S \subseteq V$ such that $y(S)$ is maximum and $\text{MST}_i(S) \leq g$. This problem corresponds to the Budget Steiner Tree Problem (BSTP), which is, again, an NP-Hard maximization problem. However, unlike NWMP, it admits a constant-factor approximation algorithm with factor $5 + \varepsilon$, for every $\varepsilon > 0$ [22]. Thus, instead of solving $(\mathbf{D}'_{\leq g})$ directly, we will obtain a solution to a relaxation.

To do this, we execute the Ellipsoid method, but stop if we cannot decide whether the candidate solution is infeasible for $(\mathbf{D}'_{\leq g})$. Thus, it may return an infeasible solution whose value might be larger than $\text{OPT}(\mathbf{D}'_{\leq g})$. More precisely, we execute the following algorithm. First, we initialize $\mathcal{A} = \emptyset$ and start the Ellipsoid method. Suppose that we are given a candidate solution y . For each car i , we execute the $(5 + \varepsilon)$ -approximation algorithm for the BSTP whose input is y and the distance function is d_i , and find a set $A_i \subseteq V$ with $\text{MST}_i(A) \leq g$. Let j be the car for which $y(A_j)$ is maximum. There are two cases to consider:

14:12 Optimal Approximation Algorithm for the Travelling Car Renter Problem

- a) If $y(A_j) > g$, then we add the constraint $y(A_j) \leq g$, which is a constraint of $(D'_{\leq g})$, because $(j, A_j) \in \mathcal{S}_{\leq g}$. We make $\mathcal{A} = \mathcal{A} \cup \{(j, A_j)\}$ and the execution of Ellipsoid method resumes with the set of constraints corresponding to \mathcal{A} .
- b) If $y(A_j) \leq g$, then we stop and return $(y, \mathcal{A}, (j, A_j))$.

The previous algorithm returns a set \mathcal{A} of polynomial size since we stopped before the end of the Ellipsoid algorithm. Let (D'_A) be the relaxation of $(D'_{\leq g})$ where only constraints corresponding to components in \mathcal{A} are added. Similarly, let (D_A) be the relaxation of $(D_{\leq g})$ where only the constraints corresponding to components in \mathcal{A} are added.

Next lemmas relate the values of (D_A) and (D'_A) and of D'_A and $D'_{\leq g}$.

► **Lemma 17.** $OPT(D_A) \leq 2 \cdot OPT(D'_A)$.

Proof. Let y be an optimal solution for (D_A) and $(i, S) \in \mathcal{A}$. Since y is feasible for (D_A) , we have $y(S) \leq g + MST_i(S)$, and, since $(i, S) \in \mathcal{S}_{\leq g}$, we have $MST_i(S) \leq g$. Then,

$$\frac{y(S)}{2} \leq \frac{MST(S) + g}{2} \leq \frac{g + g}{2} = g.$$

Since the choice of (i, S) is arbitrary, this inequality holds for every $(i, S) \in \mathcal{A}$, thus $\frac{y}{2}$ is feasible for (D'_A) . Therefore the value of the solution $\frac{y}{2}$ is not larger than the value of an optimal solution and we conclude that $OPT(D_A) = 2 \cdot \frac{y}{2}(V) \leq 2 \cdot OPT(D'_A)$. ◀

► **Lemma 18.** $OPT(D'_A) \leq (5 + \varepsilon) \cdot OPT(D'_{\leq g})$.

4.4 Proof of Lemma 7

► **Lemma 7.** *There is an algorithm that, in polynomial time, finds a feasible solution for (P) whose value is at most $c \cdot OPT(IP)$, for some constant c .*

Proof. Note that dual formulation of (D_A) corresponds to a linear program (P_A) which is similar to (P), but contains only variables corresponding to components in \mathcal{A} .

We execute the following algorithm. First, execute the modified Ellipsoid method described in Subsection 4.3 and obtain a set \mathcal{A} . Next, construct the linear program (P_A) . Then, solve (P_A) with any polynomial-time algorithm (e.g., Ellipsoid method), and obtain a solution x , indexed in \mathcal{A} . Finally, return the extension of x in which, for each $(i, S) \notin \mathcal{A}$, we represent $x_{(i,S)} = 0$ implicitly.

Notice that the above algorithm runs in polynomial time since \mathcal{A} has polynomial size and therefore (P_A) has a polynomial size too. Also, note that x is a feasible solution for (P) since (P_A) is a restriction of (P).

It remains to bound the value of an optimal solution for (P_A) . Using lemmas 17 and 18, we have $OPT(P_A) = OPT(D_A) \leq 2 \cdot OPT(D'_A) \leq 2 \cdot (5 + \varepsilon) \cdot OPT(D'_{\leq g})$. Now observe that $(D'_{\leq g})$ is a restriction of $(D_{\leq g})$, thus $OPT(P_A) \leq 2 \cdot (5 + \varepsilon) \cdot OPT(D_{\leq g}) = 2 \cdot (5 + \varepsilon) \cdot OPT(P_{\leq g})$. Since, by Lemma 16, we have $OPT(P_{\leq g}) \leq 4 \cdot OPT(IP)$, the lemma follows. ◀

5 Concluding remarks

While our $O(\log n)$ -approximation is probabilistic, it can be derandomized by techniques of conditional probabilities. Also, this paper focus on the asymptotic analysis, so changes to the algorithm may improve the hidden constant which multiplies $\log n$. Notice that our algorithm achieves the same asymptotic guarantee for the variant of the problem in which, instead of a closed walk, asks for a spanning tree composed of multiples subtrees.

We believe that the techniques of our algorithm can generalize to other variants of CaRS or similar problems. For example, one might consider the version in which the return fee depends on the car, that is, there is a return fee g_i for each car $i \in C$. A possible direction is considering the more general version of CaRS, in which, for each car i , the return fee is a given distance function f_i which depends on the return and pickup locations.

References

- 1 Fleura Bardhi and Giana M. Eckhardt. Access-Based Consumption: The Case of Car Sharing. *Journal of Consumer Research*, 39(4):881–898, December 2012. doi:10.1086/666376.
- 2 John Black. *Urban Transport Planning*. Routledge, May 2018. doi:10.4324/9781351068604.
- 3 Chandra Chekuri, Guy Even, and Guy Kortsarz. A greedy approximation algorithm for the group Steiner problem. *Discrete Applied Mathematics*, 154(1):15–34, January 2006. doi:10.1016/j.dam.2005.07.010.
- 4 V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. doi:10.1287/moor.4.3.233.
- 5 André Renato Villela da Silva and Luiz Satoru Ochi. An efficient hybrid algorithm for the Traveling Car Renter Problem. *Expert Systems with Applications*, 64:132–140, December 2016. doi:10.1016/j.eswa.2016.07.038.
- 6 André Renato Villela da Silva and Luiz Satoru Ochi. Um Algoritmo Evolutivo para o Problema do Caixeiro Alugador. In *Proceeding Series of the Brazilian Society of Applied and Computational Mathematics*, volume 5, 2017. doi:10.5540/03.2017.005.01.0460.
- 7 Kenan Degirmenci and Michael H. Breitner. Carsharing: A literature review and a perspective for information systems research. In Lars Beckmann, editor, *Multikonferenz Wirtschaftsinformatik (MKWI)*, Paderborn, Germany, February 2014. URL: <https://eprints.qut.edu.au/105684/>.
- 8 Sávio S. Dias, Luiz Satoru Ochi, Victor M. C. Machado, Luidi Gelabert Simonetti, and André Renato Villela da Silva. Uma Heurística Baseada em ILS Para o Problema do Caixeiro Alugador. In *Anais do XLVIII SBPO Simpósio Brasileiro de Pesquisa Operacional*, pages 1625–1636, 2016.
- 9 Irit Dinur and David Steurer. Analytical Approach to Parallel Repetition. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 624–633, New York, NY, USA, 2014. ACM. doi:10.1145/2591796.2591884.
- 10 Jittat Fakcharoenphol, Satish Rao, Satish Rao, and Kunal Talwar. A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 448–455, New York, NY, USA, 2003. ACM. doi:10.1145/780542.780608.
- 11 Denis Felipe, Elizabeth Ferreira Gouvêa Goldberg, and Marco César Goldberg. Scientific algorithms for the Car Renter Salesman Problem. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 873–879. IEEE, July 2014. doi:10.1109/cec.2014.6900556.
- 12 David Foot. *Operational Urban Models*. Routledge, October 2017. doi:10.4324/97813515105307.
- 13 Naveen Garg, Goran Konjevod, and R. Ravi. A Polylogarithmic Approximation Algorithm for the Group Steiner Tree Problem. *Journal of Algorithms*, 37(1):66–84, October 2000. doi:10.1006/jagm.2000.1096.
- 14 Marco César Goldberg, Elizabeth Ferreira Gouvêa Goldberg, Henrique P. L. Luna, Matheus da S. Menezes, and Lucas Corrales. Integer programming models and linearizations for the traveling car renter problem. *Optimization Letters*, 12(4):743–761, April 2017. doi:10.1007/s11590-017-1138-5.
- 15 Marco César Goldberg, Elizabeth Ferreira Gouvêa Goldberg, Matheus da S. Menezes, and Henrique P. L. Luna. Quota traveling car renter problem: Model and evolutionary algorithm. *Information Sciences*, 367-368:232–245, November 2016. doi:10.1016/j.ins.2016.05.027.

- 16 Marco César Goldberg, Elizabeth Ferreira Gouvêa Goldberg, Paulo Henrique Asconavieta da Silva, Matheus da S. Menezes, and Henrique P. L. Luna. A transgenetic algorithm applied to the Traveling Car Renter Problem. *Expert Systems with Applications*, 40(16):6298–6310, November 2013. doi:10.1016/j.eswa.2013.05.072.
- 17 Marco César Goldberg, Paulo Henrique Asconavieta da Silva, and Elizabeth Ferreira Gouvêa Goldberg. Memetic algorithm for the Traveling Car Renter Problem: An experimental investigation. *Memetic Computing*, 4(2):89–108, September 2011. doi:10.1007/s12293-011-0070-y.
- 18 M. Grazia Speranza. Trends in transportation and logistics. *European Journal of Operational Research*, 264(3):830–836, February 2018. doi:10.1016/j.ejor.2016.08.032.
- 19 Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988. doi:10.1007/978-3-642-97881-4.
- 20 Sudipto Guha and Samir Khuller. Greedy Strikes Back: Improved Facility Location Algorithms. *Journal of Algorithms*, 31(1):228–248, April 1999. doi:10.1006/jagm.1998.0993.
- 21 Eran Halperin and Robert Krauthgamer. Polylogarithmic Inapproximability. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 585–594, New York, NY, USA, 2003. ACM. doi:10.1145/780542.780628.
- 22 David S. Johnson, Maria Minkoff, and Steven Phillips. The Prize Collecting Steiner Tree Problem: Theory and Practice. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 760–769, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=338219.338637>.
- 23 Xin Luan, Lin Cheng, Yang Zhou, and Fang Tang. Strategies of Car-Sharing Promotion in Real Market. In *2018 3rd IEEE International Conference on Intelligent Transportation Engineering (ICITE)*, pages 159–163. IEEE, September 2018. doi:10.1109/icite.2018.8492652.
- 24 Matheus da S. Menezes. *O problema do Caixeiro alugador com coleta de prêmios: Um estudo algorítmico*. PhD thesis, Universidade Federal do Rio Grande do Norte, 2014.
- 25 Brenner Humberto Ojeda Rios. *Hibridização de meta-heurísticas com métodos baseados em programação linear para o problema do caixeiro alugador*. Master's thesis, Universidade Federal do Rio Grande do Norte, 2017.
- 26 Brenner Humberto Ojeda Rios, Elizabeth Ferreira Gouvêa Goldberg, and Greis Yvet Oropeza Quesquen. A hybrid metaheuristic using a corrected formulation for the Traveling Car Renter Salesman Problem. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 2308–2314. IEEE, June 2017. doi:10.1109/cec.2017.7969584.
- 27 Philipp Rode, Graham Floater, Nikolas Thomopoulos, James Docherty, Peter Schwinger, Anjali Mahendra, and Wanli Fang. Accessibility in Cities: Transport and Urban Form. In *Disrupting Mobility*, pages 239–273. Springer International Publishing, 2017. doi:10.1007/978-3-319-51602-8_15.
- 28 Shmuel Safra and Oded Schwartz. On the complexity of approximating tsp with neighborhoods and related problems. *Computational Complexity*, 14(4):281–307, March 2006. doi:10.1007/s00037-005-0200-3.
- 29 Paulo Henrique Asconavieta da Silva. *O problema do caixeiro viajante alugador: Um estudo algorítmico*. PhD thesis, Universidade Federal do Rio Grande do Norte, 2011.
- 30 Paulo Henrique Asconavieta da Silva, Marco César Goldberg, and Elizabeth Ferreira Gouvêa Goldberg. The car renter salesman problem: An algorithmic study. In *Anais do XLII SBPO Simpósio Brasileiro de Pesquisa Operacional*, 2010.
- 31 Paulo Henrique Asconavieta da Silva, Marco César Goldberg, and Elizabeth Ferreira Gouvêa Goldberg. Evolutionary algorithm for the Car Renter Salesman. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 593–600. IEEE, June 2011. doi:10.1109/cec.2011.5949673.
- 32 Vijay V. Vazirani. *Approximation Algorithms*. Springer, Berlin, 2001. doi:10.1007/978-3-662-04565-7.

- 33 David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, Cambridge, 2011. doi:10.1017/CB09780511921735.
- 34 Henk Zijm and Matthias Klumpp. Logistics and Supply Chain Management: Developments and Trends. In *Logistics and Supply Chain Innovation*, pages 1–20. Springer International Publishing, August 2015. doi:10.1007/978-3-319-22288-2_1.

