eCO-friendly urban Multi-modal route PlAnning Services for mobile uSers

FP7 - Information and Communication Technologies

Grant Agreement No: 288094
Collaborative Project
Project start: 1 November 2011, Duration: 36 months

D3.3.2 – New eco-aware models and solutions to robust multimodal route-planning & their empirical assessment

Workpackage:	WP3 - Algorithms for Multimodal Human Mobility
Due date of deliverable:	30 June 2013
Actual submission date:	07 July 2013
Responsible Partner:	KIT
Contributing Partners:	KIT

Nature: ☒ Report ☐ Prototype ☐ Demonstrator ☐ Other

Dissemination Level:
☒ PU: Public
☐ PP: Restricted to other programme participants (including the Commission Services)
☐ RE: Restricted to a group specified by the consortium (including the Commission Services)
☐ CO: Confidential, only for members of the consortium (including the Commission Services)

Keyword List: Multimodal, Multi-criteria, Route Planning, Shortest Path, Public Transportation, Timetable, Road Networks, Combinatorial Optimization, Algorithms, Algorithm Engineering, Experimental Algorithmics

The eCOMPASS project (www.ecompass-project.eu) is funded by the European Commission, DG CONNECT (Communications Networks, Content and Technology Directorate General), Unit H5 - Smart Cities & Sustainability, under the FP7 Programme.

The eCOMPASS Consortium

Computer Technology Institute & Press 'Diophantus' (CTI) (coordinator), Greece

Centre for Research and Technology Hellas (CERTH), Greece

Eidgenössische Technische Hochschule Zürich (ETHZ), Switzerland

Karlsruher Institut fuer Technologie (KIT), Germany

TOMTOM INTERNATIONAL BV (TOMTOM), Netherlands

PTV PLANUNG TRANSPORT VERKEHR AG. (PTV), Germany

Document history			
Version	Date	Status	Modifications made by
0.1	17.06.2013	First Draft	Julian Dibbelt, KIT
0.2	29.06.2013	Second Draft	Rousias Nikolaos, CTI
0.3	30.06.2013	Third Draft	Julian Dibbelt, KIT
1.0	02.07.2013	Sent to internal reviewers	Julian Dibbelt, KIT
1.1	07.07.2013	Reviewers' comments incorporated (sent to PQB)	Julian Dibbelt, KIT
1.2	07.07.2013	PQB's comments incorporated	Julian Dibbelt, KIT
1.3	08.07.2013	PQB's comments incorporated	Nikolaos Rousias and Christos Zaroliagis, CTI
1.4	09.07.2013	Final (approved by PQB, sent to the Project Officer)	Christos Zaroliagis, CTI

Deliverable manager

- Julian Dibbelt, KIT

List of Contributors

- Julian Dibbelt, KIT
- Nikolaos Rousias, CTI
- Thomas Pajor, KIT
- Ben Strasser, KIT
- Christos Zaroliagis, CTI

List of Evaluators

- Dionisis Kehagias, CERTH
- Sandro Montanari, ETHZ

Summary

In this deliverable we report on our progress in designing new models and algorithmic solutions to multimodal route-planning. We identify the core algorithmic challenges that arise for fully-multimodal urban networks. These are: Modelling issues and better solutions to subproblems; Preprocessing flexible enough for user-defined path constraints; Capturing the richness of "best" solutions in fully-multimodal networks. We then propose several interesting new approaches to these challenges and extensively test our algorithms on benchmark data from, e. g., Deliverable D3.2. A selection of these solutions will be integrated by project partners in WP 5 and extensively evaluated during the pilot in Berlin.

Contents

1 Introduction **5**

2 Problem Statement and Related Work **6**

3 Subproblems: New Approaches **7**
3.1 A Simple and Fast Approach to Public Transit Routing . . . 7
3.1.1 Preliminaries . . . 7
3.1.2 Basic Connection Scan Algorithm . . . 8
3.1.3 Extensions . . . 9
3.1.4 Experiments . . . 10
3.1.5 Final Remarks . . . 12

4 User-Constrained Multimodal Route Planning **13**
4.1 Preliminaries . . . 14
4.1.1 Models . . . 15
4.1.2 Path Constraints on the Sequences of Transport Modes . . . 15
4.1.3 Contraction Hierarchies (CH) . . . 16
4.2 Our Approach . . . 17
4.2.1 Contraction Hierarchies for Multimodal Networks . . . 17
4.2.2 UCCH: Contraction for User-Constrained Route Planning . . . 17
4.2.3 Improvements . . . 19
4.3 Experiments . . . 21
4.3.1 Evaluating Average Core Degree Limit . . . 22
4.3.2 Preprocessing . . . 23
4.3.3 Query Performance . . . 24
4.4 Conclusion . . . 24

5 Multi-Criteria Search: Finding (All) the Good Options **24**
5.1 Computing and Evaluating Multimodal Journeys . . . 25
5.1.1 Problem Statement . . . 25
5.1.2 Exact Algorithms . . . 29
5.1.3 Heuristics . . . 31
5.1.4 Experiments . . . 32
5.1.5 Final Remarks . . . 37
5.2 Multiobjective Route Planning . . . 39
5.2.1 Graph structure . . . 39
5.2.2 The heuristic algorithm NAMOA* . . . 40
5.2.3 Computing heuristic functions . . . 41
5.2.4 Extensions of NAMOA* . . . 41
5.2.5 Experimental Evaluation . . . 42
5.2.6 Final Remarks . . . 43

6 Conclusions and Plans for Next Periods **43**

1 Introduction

The aim of this deliverable is to document the results of Task 3.3. It describes the models and solutions developed, including extensive experimental validation thereof.

Background. The aim of Work Package WP3 is to provide novel methods for environmentally friendly routes in urban public transportation networks. In particular, the goal is to develop mathematical sound models for various (context-aware) route-planning scenarios arising in the field of urban human mobility for city residents, commuters and tourists, as well as to provide algorithmic methods for multimodal routes in urban transportation networks with respect to multiple criteria and high robustness with a strong focus on the environmental footprint of these routes. Algorithms and methods developed within this work package will be implemented and an extensive experimental evaluation regarding performance and quality will be conducted following the *Algorithm Engineering* [79] paradigm. This paradigm differs from traditional Algorithmic Design in Theoretical Computer Science in several key factors: instead of deriving asymptotic bounds of a given algorithm for worst-case inputs on abstract machine models, *typical and realistic instances* are examined on real machines to measure the practical performance of the implementation of a given algorithm. Also, the results of this experimentation are analyzed and used to guide the design of the algorithm, the improvement of which is then again experimentally verified in a continuing feedback loop.

Objectives and Scope of Deliverable D3.3. Research on route planning algorithms in transportation networks has undergone a rapid development over the last years. See [33] for an overview. Usually the network is modeled as a directed graph G. While Dijkstra's algorithm can be used to compute a best route between two nodes of G in almost linear time [51], it is too slow for practical applications in real-world transportation networks. Such applications are confronted with millions of queries per hour [55], while the considered networks consist of several million nodes and we expect almost instant results. Thus, over the years a multitude of speedup techniques for Dijkstra's algorithm were developed, all following a similar paradigm: In a *preprocessing phase* auxiliary data is computed which is then used to accelerate Dijkstra's algorithm in the *query phase*. The fastest techniques today can answer a single query within only a few memory accesses [1]. However, most of the techniques were developed with one type of transportation network in mind. In fact, the fastest techniques developed for road networks heavily rely on structural properties of these and their performance degrades significantly on other networks [11, 16].

In the real world different modes of travel are linked extensively, and realistic transportation scenarios imply frequent modal changes. Furthermore, with the increasing appearance of electric vehicles and their inherent range restrictions, the choice between taking the car and public transit will become more important. To solve such scenarios we are interested in an integrated system that can handle multiple transportation networks with a single algorithm.

The goal of Task 3.3 is the development of new models and solutions to route planning in multimodal urban transportation networks; results will be integrated by partners in WP 5 and tested in the pilot within the scope of WP 6. Within Task 3.3, we have achieved a better understanding of the algorithmic nature of route planning for multimodal urban transportation networks. The research done so far in WP 3 has led to several peer-reviewed scientific publications as well as several pending publications (c. f. Technical Reports ECOMPASS-TR-003, ECOMPASS-TR-005, ECOMPASS-TR-006, ECOMPASS-TR-011, ECOMPASS-TR-013, ECOMPASS-TR-019, ECOMPASS-TR-21, ECOMPASS-TR-022).

Outline. Section 2 gives an overview of recent state-of-the-art as well as the challenges we aim to address. Section 3 discusses new solutions to subproblems of the fully-multimodal route planning problem, while Section 4 reports on our progress towards acceleration techniques for multimodal

networks. In Section 5 we discuss how to capture all the good solutions that a fully-multimodal network offers. Section 6 concludes with future directions.

2 Problem Statement and Related Work

Online services for journey planning have become a commodity used daily by millions of commuters. The problem of efficiently computing good journeys in transportation networks presents several algorithmic challenges, and has been an active area of research in recent years. Much focus has been given to the computation of routes both in road networks [2, 25, 33, 47, 58, 82] and in scheduled-based public transit [11, 13, 19, 23, 32, 39, 71, 73, 81], but these are often considered separately. In practice, however, users want an integrated solution that can find the "best" way to get to their destination considering all available modes of transportation. Within a metropolitan area, this includes buses, trains, driving, cycling, taxis, and, of course, walking. We refer to this as the *multimodal route planning* problem.

In fact, any public transportation network necessarily has a multimodal component, since journeys require some amount of walking. Existing solutions [13, 19, 28, 32, 39] handle this by predefining transfer arcs between nearby stations, and running a search algorithm on the public transit network to find the "best" journey. Unlike in road networks, however, defining "best" is not straightforward. For example, while some people want to arrive as early as possible, others are willing to spend a little more time to avoid extra transfers. Most recent approaches to public transportation route planning therefore compute the *Pareto set* [56] of non-dominating journeys optimizing multiple criteria, which is practical even for large metropolitan areas [32, 73], when relying on very short, predefined walking transfer arcs and a small subset of interesting criteria (e. g., arrival time and number of transfers).

Challenges. Extending current solutions to a full multimodal scenario (with unrestricted walking, biking, and taxis) may seem trivial at first: One could just incorporate routing techniques for road networks [25, 47, 58] and public transportation techniques [13, 19, 28, 32, 36, 39] to solve the new subproblems. However, three types of algorithmic challenges remain.

Better Solutions to Subproblems on Subnetworks. Preprocessing-based techniques for computing point-to-point shortest paths have been very successful on road networks [33, 82], but their adaption to public transit networks [13, 44] is harder than expected [11, 17, 18]. In parts this is due to the inherent time-dependency of public transit networks as well as the fact that they are, in general, less hierarchically well-structured. While solutions to public transit route planning in metropolitan areas are now reasonably fast [32, 36] (the latter of which was developed within the scope of Task 3.3 and is presented in Section 3.1) we are still far from being able to solve queries at a global scale (unlike for road networks).

User Constraints. In a fully-multimodal network, not every fastest path is feasible: on a long train ride from, e. g., Karlsruhe to Nantes, taking a *private car* between train stations, e. g. in Paris, might look like a good idea (in terms of travel time)—but most likely the user will not have access to one. When designing preprocessing schemes for multimodal route planning, such path constraints have to be considered, preferably by considering a user's modal preferences: Not every mode of transport might be available or preferable for him at any point along the journey. In general, the user has restrictions on the sequence of transport modes. For example, some users might be willing to take a *taxi* between two train rides if it makes the journey quicker. Others prefer to use public transit at a stretch. For some users, such preferences might be subject to change over time. A realistic multimodal route-planning system must handle such constraints as a *user input* for each *query*.

Capturing the Diversity of Solutions. Multimodal urban transportation networks offer a rich set of travel possibilities. Just optimizing for the fastest journey, even if it adheres to feasibility constraints, does not capture the diversity of solutions. This is not only true because single subnetworks offer multi-criteria solutions (e.,g., in public transit you find journeys trading-off travel time and number of transfers), but also because there are trade-offs between different modes of transportation.

We conjecture that meaningful multimodal optimization needs to take more criteria into account, such as walking duration and costs. Some people are happy to walk 10 minutes to avoid an extra transfer, while others are not. In fact, some will walk half an hour to avoid using public transportation at all. Taking a taxi all the way to the airport is a good solution for some; users on a budget may prefer a cheaper solution.

3 Subproblems: New Approaches

3.1 A Simple and Fast Approach to Public Transit Routing

The problem of computing "best" journeys in public transportation networks comes in several variants [72]: The simplest, called *earliest arrival*, takes a departure time as input, and determines a journey that arrives at the destination as early as possible. If further criteria, such as the number of transfers, are important, one may consider *multi-criteria* optimization [32, 39]. Finally, a *profile query* [28, 32] computes a set of optimal journeys that depart during a period of time (such as a day). Traditionally, these problems have been solved by (variants of) Dijkstra's algorithm on an appropriate graph model. Well-known examples are the time-expanded and time-dependent models [28, 44, 72, 77]. Recently, Delling et al. [32] introduced RAPTOR. It solves the multi-criteria problem (arrival time and number of transfers) by using dynamic programming directly on the timetable, hence, no longer requires a graph or a priority queue.

In this section, we present the *Connection Scan Algorithm* (CSA). In its basic variant, it solves the earliest arrival problem, and is, like RAPTOR [32], not graph-based (cf. [28, 44, 72, 77]). However, it is not centered around *routes* (as RAPTOR), but elementary *connections*, which are the most basic building block of a timetable. CSA organizes them as one single array, which it then scans once (linearly) to compute journeys to all stops of the network. The algorithm turns out to be intriguingly simple with excellent spatial data locality. We also extend CSA to handle multi-criteria profile queries: For a full time period, it computes Pareto sets of journeys optimizing arrival time and number of transfers. We extend CSA to handle these queries very efficiently. Moreover, we do not make use of heavy preprocessing, thus, enabling dynamic scenarios including train cancellations, route changes, real-time delays, etc. Our experiments on the dense metropolitan network of London validate our approach. With CSA, we compute earliest arrival queries in under 2 ms, and multi-criteria profile queries for a full period in 221 ms—faster than previous algorithms.

Outline Section 3.1.1 sets necessary notion, and Section 3.1.2 presents our new algorithm. Section 3.1.3 extends it to multi-criteria profile queries. The experimental evaluation is available in Section 3.1.4, while Section 3.1.5 contains concluding remarks. Note that Deliverable D3.4 reports on an extension of our algorithm to robust route planning in stochastic scenarios.

3.1.1 Preliminaries

Our public transit networks are defined in terms of their aperiodic *timetable*, consisting of a set of *stops*, a set of *connections*, and a set of *footpaths*. A *stop* p corresponds to a location in the network where a passenger can enter or exit a vehicle (such as a bus stop or train station). Stops may have associated minimum change times, denoted $\tau_{\text{ch}}(p)$, which represent the minimum time required to

change vehicles at p. A *connection* c models a vehicle departing at a stop $p_{\text{dep}}(c)$ at time $\tau_{\text{dep}}(c)$ and arriving at stop $p_{\text{arr}}(c)$ at time $\tau_{\text{arr}}(c)$ without intermediate halt. Connections that are subsequently operated by the same vehicle are grouped into *trips*. We identify them by $t(c)$. We denote by c_{next} the next connection (after c) of the same trip, if available. Trips can be further grouped into *routes*. A route is a set of trips serving the exact same sequence of stops. For correctness, we require trips of the same route to not overtake each other. *Footpaths* enable walking transfers between nearby stops. Each footpath consists of two stops with an associated walking duration. Note that our footpaths are transitively closed. A *journey* is a sequence of connections and footpaths. If two subsequent connections are not part of the same trip, their arrival-departure time-difference must be at least the minimum change time of the stop. Because our footpaths are transitively closed, a journey never contains two subsequent footpaths.

In this paper we consider several well-known problems. In the *earliest arrival problem* we are given a source stop p_s, a target stop p_t, and a departure time τ. It asks for a journey that departs from p_s no earlier than τ and arrives at p_t as early as possible. The *profile problem* asks for the set of all earliest arrival journeys (from p_s to p_t) for every departure at p_s. Besides arrival time, we also consider the number of transfers as criterion: In multi-criteria scenarios one is interested in computing a *Pareto set* of nondominated journeys. Here, a journey J_1 *dominates* a journey J_2 if it is better with respect to every criterion. Nondominated journeys are also called to be *Pareto-optimal*. Finally, the *multi-criteria profile problem* requests a set of Pareto-optimal journeys (from p_s to p_t) for all departures (at p_s).

Usually, these problems have been solved by (variants of) Dijkstra's algorithm on an appropriate graph (representing the timetable). Most relevant to our work is the realistic *time-expanded model* [77]. It expands time in the sense that it creates a vertex for each *event* in the timetable (such as a vehicle departing or arriving at a stop). Then, for every connection it inserts an arc between its respective departure/arrival events, and also arcs that link subsequent connections. Arcs are always weighted by the time difference of their linked events. Special vertices may be added to respect minimum change times at stops. See [72, 77] for details.

3.1.2 Basic Connection Scan Algorithm

We now introduce the Connection Scan Algorithm (CSA), our approach to public transit route planning. We describe it for the earliest arrival problem and extend it to more complex scenarios in Sections 3.1.3. Our algorithm builds on the following property of public transit networks: We call a connection c *reachable* iff either the user is already traveling on a preceding connection of the same trip $t(c)$, or, he is standing at the connection's departure stop $p_{\text{dep}}(c)$ on time, i. e., before $\tau_{\text{dep}}(c)$. In fact, the time-expanded approach encodes this property into a graph G, and then uses Dijkstra's algorithm to obtain optimal sequences of reachable connections [77]. Unfortunately, Dijkstra's performance is affected by many priority queue operations and suboptimal memory access patterns. However, since our timetables are aperiodic, we observe that G is acyclic. Thus, its arcs may be sorted topologically, e. g., by departure time. Dijkstra's algorithm on G, actually, scans (a subsequence of) them in this order.

Instead of building a graph, our algorithm assembles the timetable's connections into a single array C, sorted by departure time. Given source stop p_s and departure time τ as input, it maintains for each stop p a label $\tau(p)$ representing the earliest arrival time at p. Labels $\tau(\cdot)$ are initialized to all-infinity, except $\tau(p_s)$, which is set to τ. The algorithm scans all connections $c \in C$ (in order), testing if c can be *reached*. If this is the case and if $\tau_{\text{arr}}(c)$ improves $\tau(p_{\text{arr}}(c))$, CSA *relaxes* c by updating $\tau(p_{\text{arr}}(c))$. After scanning the full array, the labels $\tau(\cdot)$ provably hold earliest arrival times for all stops.

Reachability, Minimum Change Times and Footpaths. To account for minimum change times in our data, we check a connection c for reachability by testing if $\tau(p_{\text{dep}}(c)) + \tau_{\text{ch}}(p_{\text{dep}}(c)) \leq \tau_{\text{dep}}(c)$ holds. Additionally, we track whether a preceding connection of the same trip $t(c)$ has been

used. We, therefore, maintain for each connection a flag, initially set to 0. Whenever the algorithm identifies a connection c as reachable, it sets the flag of c's subsequent connection c_{next} to 1. Note that for networks with $\tau_{\text{ch}}(\cdot) = 0$, trip tracking can be disabled and testing reachability simplifies to $\tau(p_{\text{dep}}(c)) \leq \tau_{\text{dep}}(c)$. To handle footpaths, each time the algorithm relaxes a connection c, it scans all outgoing footpaths of $p_{\text{arr}}(c)$.

Improvements. Clearly, connections departing before time τ can never be reached and need not be scanned. We do a binary search on C to identify the first relevant connection and start scanning from there (*start criterion*). If we are only interested in *one-to-one queries*, the algorithm may stop as soon as it scans a connection whose departure time exceeds the target stop's earliest arrival time. Also, as soon as one connection of a trip is reachable, so are all subsequent connections of the same trip (and preceding connections of the trip have already been scanned). We may, therefore, keep a flag (indicating reachability) per trip (instead of per connection). The algorithm then operates on these *trip flags* instead. Note that we store all data sequentially in memory, making the scan extremely cache-efficient. Only accesses to stop labels and trip flags are potentially costly, but the number of stops and trips is small in comparison. To further improve spatial locality, we subtract from each connection $c \in C$ the minimum change time of $p_{\text{dep}}(c)$ from $\tau_{\text{dep}}(c)$, but keep the original ordering of C. Hence, CSA requires random access only on small parts of its data, which mostly fits in low-level cache.

3.1.3 Extensions

CSA can be extended to profile queries. Given the timetable and a source stop p_s, a profile query computes for every stop p the set of all earliest arrival journeys to p for every departure from p_s, discarding dominated journeys. Such queries are useful for preprocessing techniques, but also for users with flexible departure (or arrival) time. We refer to the solution as a Pareto set of $(\tau_{\text{dep}}(p_s), \tau_{\text{arr}}(p_t))$ pairs.

In the following, we describe the *reverse* p–p_t-profile query. The forward search works analogously. Our algorithm, pCSA (p for profile), scans once over the array of connections sorted by *decreasing* departure time. For every stop it keeps a partial (tentative) profile. It maintains the property that the partial profiles are correct wrt. the subset of already scanned connections. Every stop is initialized with an empty profile, except p_t, which is set to a constant identity-profile. When scanning a connection c, pCSA *evaluates* the partial profile at the arrival stop $p_{\text{arr}}(c)$: It asks for the earliest arrival time τ^* at p_t over all journeys departing at $p_{\text{arr}}(c)$ at $\tau_{\text{arr}}(c)$ or later. It then *updates* the profile at $p_{\text{dep}}(c)$ by potentially adding the pair $(\tau_{\text{dep}}(c), \tau^*)$ to it, discarding newly dominated pairs, if necessary.

Maintaining Profiles. We describe two variants of maintaining profiles. The first, pCSA-P (P for Pareto), stores them as arrays of Pareto-optimal (τ_{dep}, τ_{arr}) pairs ordered by decreasing arrival (departure) time. Since new candidate entries are generated in order of decreasing departure time, profile updates are a constant-time operation: A candidate entry is either dominated by the last entry or is appended to the array. Profile evaluation is implemented as a linear scan over the array. This is quick in practice, since, compared to the timetable's period, connections usually have a short duration. The identity profile of p_t is handled as a special case. By slightly modifying the data structure, we obtain pCSA-C (C for constant), for which evaluation is also possible in constant time: When updating a profile, pCSA may append a candidate entry, even if it is dominated. To ensure correctness, we set the candidate's arrival time τ^* to that of the dominating entry. We then observe that, independent of the input's source or target stop, profile entries are always generated in the same order. Moreover, each connection is associated with only two such entries, one at its departure stop, relevant for updating, and, one at its arrival stop, relevant for evaluation. For each connection, we precompute *profile indices* pointing to these two entries, keeping them with the

connection. Furthermore, its associated departure time and stop may be dropped. Note that the space consumption for keeping all (even suboptimal) profile entries is bounded by the number of connections. Following [28], we also collect—in a quick preprocessing step—at each stop all arrival times (in decreasing order). Then, instead of storing arrival times in the profile entries, we keep *arrival time indices*. For our scenarios, these can be encoded using 16 (or fewer) bits. We call this technique *time indexing*, and the corresponding algorithm pCSA-CT.

Minimum Change Times and Footpaths. We incorporate minimum change times by evaluating the profile at a stop p for time τ at $\tau + \tau_{\text{ch}}(p)$. The trip bit is replaced by a trip arrival time, which represents the earliest arrival time at p_t when continuing with the trip. When scanning a connection c, we take the minimum of the trip arrival time and the evaluated profile at $p_{\text{arr}}(c)$. We update the trip arrival time and the profile at $p_{\text{dep}}(c)$, accordingly. *Footpaths* are handled as follows. Whenever a connection c is relaxed, we scan all incoming footpaths at $p_{\text{dep}}(c)$. However, this no longer guarantees that profile entries are generated by decreasing departure time, making profile updates a non-constant operation for pCSA-P. Also, we can no longer precompute profile indices for pCSA-C. Therefore, we expand footpaths into *pseudoconnections* in our data, as follows. If p_a and p_b are connected by a footpath, we look at all reachable (via the footpath) pairs of incoming connections c_{in} at p_a and outgoing connections c_{out} at p_b. We create a new pseudoconnection (from p_a to p_b, departure time $\tau_{\text{arr}}(c_{\text{in}})$, and arrival time $\tau_{\text{dep}}(c_{\text{out}})$) iff there is no other pseudoconnection with a later or equal departure time and an earlier or equal arrival time. Pseudoconnections can be identified by a simultaneous sweep over the incoming/outgoing connections of p_a and p_b. During query, we handle footpaths toward p_t as a special case of the evaluation procedure. Footpaths at p_s are handled by merging the profiles of stops that are reachable by foot from p_s.

One-to-One Queries. So far we described *all-to-one profile queries*, i. e., from all stops to the target stop p_t. If only the *one-to-one profile* between stops p_s and p_t is of interest, a well-known pruning rule [28, 72] can be applied to pCSA-P: Before inserting a new profile entry at any stop, we check whether it is dominated by the last entry in the profile at p_s. If so, the current connection cannot possibly be extended to a Pareto-optimal solution at the source, and, hence, can be pruned. However, we still have to continue scanning the full connection array.

Multi-Criteria. CSA can be extended to compute *multi-criteria* profiles, optimizing triples $(\tau_{\text{dep}}(p_s), \tau_{\text{arr}}(p_t), \# t)$ of departure time, arrival time and number of taken trips. We call this variant mcpCSA-CT. We organize these triples hierarchically by mapping arrival time $\tau_{\text{arr}}(p_t)$ onto *bags* of $(\tau_{\text{dep}}(p_s), \# t)$ pairs. Thus, we follow the general approach of pCSA-CT, but now maintain profiles as $(\tau_{\text{arr}}(p_t), \text{bag})$ pairs. Evaluating a profile, thus, returns a bag. Where pCSA-CT computes the minimum of two departure times, mcpCSA-CT *merges* two bags, i. e., it computes their union and removes dominated entries. When it scans a connection c, $\# t$ is increased by one for each entry of the evaluated bag, unless c is a pseudoconnection. It then merges the result with the bag of trip $t(c)$, and updates the profile at $p_{\text{dep}}(c)$, accordingly. Exploiting that, in practice, $\# t$ only takes small integral values, we store bags as fixed-length vectors using $\# t$ as index and departure times as values. Merging bags then corresponds to a component-wise minimum, and increasing $\# t$ to shifting the vector's values. A variant, mcpCSA-CT-SSE, uses SIMD-instructions for these operations.

3.1.4 Experiments

We ran experiments pinned to one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. We compiled our C++ code using g++ 4.7.1 with flags `-O3 -mavx`.

We consider three realistic inputs whose sizes are reported in Table 1. They are also used in [28, 44, 32], but we additionally filter them for (obvious) errors, such as duplicated trips and

Table 1: Size figures for our timetables including figures of the time-dependent (TD), colored time-dependent (TD-col), and time-expanded (TE) graph models [28, 72, 77].

Figures	London		Germany		Europe	
Stops		20 843		6 822		30 517
Trips		125 537		94 858		463 887
Connections		4 850 431		976 678		4 654 812
Routes		2 135		9 055		42 547
Footpaths		45 652		0		0
Expanded Footpaths		8 436 763		0		0
TD Vertices (Arcs)	97 k	(272 k)	114 k	(314 k)	527 k	(1 448 k)
TD-col Vertices (Arcs)	21 k	(71 k)	20 k	(86 k)	79 k	(339 k)
TE Vertices (Arcs)	9 338 k	(34 990 k)	1 809 k	(3 652 k)	8 778 k	(17 557 k)

connections with non-positive travel time. Our main instance, London, is available at [63]. It includes tube (subway), bus, tram, Dockland Light Rail (DLR) and is our only instance that also includes footpaths. However, it has no minimum change times. The German and European networks were kindly provided by HaCon [55] under a restricted license. Both have minimum change times. The German network contains long-distance, regional, and commuter trains operated by Deutsche Bahn during the winter schedule of 2001/02. The European network contains long-distance trains in Austria, Belgium, Bulgaria, Croatia, Czech Republic, France, Germany, Great Britain, Greece, Hungary, Italy, Luxembourg, Netherlands, Poland, Romania, Sweden, Slovak Republic, Slovenia, Switzerland, Turkey, and is based on the winter schedule of 1996/97. To account for overnight trains and long journeys, our (aperiodic) timetables cover one (London), two (Germany), and three (Europe) consecutive days.

We ran for every experiment 10 000 queries with source and target stops chosen uniformly at random. Departure times are chosen at random between 0:00 and 24:00 (of the first day). We report the running time and the number of label comparisons, counting an SSE operation as a single comparison. Note that we disregard comparisons in the priority queue implementation.

Earliest Arrival. In Table 2, we report performance figures for several algorithms on the London instance. Besides CSA, we ran realistic time-expanded Dijkstra (TE) with two vertices per connection [77] and footpaths [72], realistic time-dependent Dijkstra (TD), and time-dependent Dijkstra using the optimized coloring model [28] (TD-col). For CSA, we distinguish between scanned, reachable and relaxed connections. Algorithms in Table 2 are grouped into blocks.

The first considers one-to-all queries, and we see that basic CSA scans *all* connections (4.8 M), only half of which are reachable. On the other hand, TE scans about half of the graph's arcs (20 M). Still, this is a factor of four more entities due to the modeling overhead of the time-expanded graph. Regarding query time, CSA greatly benefits from its simple data structures and lack of priority queue: It is a factor of 52 faster than TE. Enabling the start criterion reduces the number of scanned connections by 40 %, which also helps query time. Using trip bits increases spatial locality and further reduces query time to 9.7 ms. We observe that just a small fraction of scanned arcs/connections actually improve stop labels. Only then CSA must consider footpaths. The second block considers one-to-one queries. Here, the number of connections scanned by CSA is significantly smaller; journeys in London rarely have long travel times. Since our London instance does not have minimum change times, we may remove trip tracking from the algorithm entirely. This yields the best query time of 1.8 ms on average. Although CSA compares significantly more labels, it outperforms Dijkstra in almost all cases (also see Table 4 for other inputs). Only for one-to-all queries on London TD-col is slightly faster than CSA.

Table 2: Figures for the earliest arrival problem on our London instance. Indicators are: • enabled, ∘ disabled, – not applicable. "Sta." refers to the start criterion. "Trp." indicates the method of trip tracking: connection flag (∘), trip flag (•), none (×). "One." indicates one-to-one queries by either using the stop criterion or pruning.

Alg.	Sta.	Trp.	One.	# Scanned Arcs/Con.	# Reachable Arcs/Con.	# Relaxed Arcs/Con.	# Scanned Footpaths	# L.Cmp. p. Stop	Time [ms]
TE	–	–	∘	20 370 117	—	5 739 046	—	977.3	876.2
TD	–	–	∘	262 080	—	115 588	—	11.9	18.9
TD-col	–	–	∘	68 183	—	21 294	—	3.2	7.3
CSA	∘	∘	∘	4 850 431	2 576 355	11 090	11 500	356.9	16.8
CSA	•	∘	∘	2 908 731	2 576 355	11 090	11 500	279.7	12.4
CSA	•	•	∘	2 908 731	2 576 355	11 090	11 500	279.7	9.7
TE	–	–	•	1 391 761	—	385 641	—	66.8	64.4
TD	–	–	•	158 840	—	68 038	—	7.2	10.9
TD-col	–	–	•	43 238	—	11 602	—	2.1	4.1
CSA	•	•	•	420 263	126 983	5 574	7 005	26.6	2.0
CSA	•	×	•	420 263	126 983	5 574	7 005	26.6	1.8

Profile and Multi-Criteria Queries. In Table 3 we report experiments for (multi-criteria) profile queries on London. Other instances are available in Table 4. We compare CSA to SPCS-col [28] (an extension of TD-col to profile queries) and rRAPTOR [32] (an extension of RAPTOR to multi-criteria profile queries). Note that in [32] rRAPTOR is evaluated on two-hours range queries, whereas we compute full profile queries. A first observation is that, regarding query time, one-to-all SPCS is outperformed by all other algorithms, even those which additionally minimize the number of transfers. Similarly to our previous experiment, CSA generally does more work than the competing algorithms, but is, again, faster due to its cache-friendlier memory access patterns. We also observe that one-to-all pCSA-C is slightly faster than pCSA-P, even with target pruning enabled, although it scans 2.7 times as many connections because of expanded footpaths. Note, however, that the figure for pCSA-C does not include the postprocessing that removes dominated journeys. Time indexing further accelerates pCSA-C, indicating that the algorithm is, indeed, memory-bound. Regarding multi-criteria profile queries, doubling the number of considered trips also doubles both CSA's label comparisons and its running time. For rRAPTOR the difference is less (only 12 %)—most work is spent in the first eight rounds. Indeed, journeys with more than eight trips are very rare. This justifies mcpCSA-CT-SSE with eight trips, which is our fastest algorithm (221 ms on average). Note that using an AVX2 processor (announced for June 2013), one will be able to process 256 bit-vectors in a single instruction. We, therefore, expect mcpCSA-CT-SSE to perform better for greater numbers of trips in the future.

3.1.5 Final Remarks

In this work, we introduced the Connection Scan framework of algorithms (CSA) for several public transit route planning problems. One of its strengths is the conceptual simplicity, allowing easy implementations. Yet, it is sufficiently flexible to handle complex scenarios, such as multi-criteria profile queries. Our experiments on the metropolitan network of London revealed that CSA is faster than existing approaches. All scenarios considered are fast enough for interactive applications. For future work, we are interested in investigating network decomposition techniques to make CSA more scalable, as well as more realistic delay models. Also, since CSA does not use a priority queue, parallel extensions seem promising. Regarding multimodal scenarios, we like to combine CSA with existing techniques developed for road networks.

Table 3: Figures for the (multi-criteria) profile problem on London. "# Tr." is the max. number of trips considered. "Arr." indicates minimizing arrival time, "Tran." transfers. "Prof." indicates profile queries. "# Jn." is the number of Pareto-optimal journeys.

Algorithm	# Tr.	Arr.	Tran.	Prof.	One.	# Jn.	# L.Cmp. p. Stop	Time [ms]
SPCS-col	–	•	◦	•	◦	98.2	477.7	1 262
SPCS-col	–	•	◦	•	•	98.2	372.5	843
pCSA-P	–	•	◦	•	◦	98.2	567.6	177
pCSA-P	–	•	◦	•	•	98.2	436.9	161
pCSA-C	–	•	◦	•	–	98.2	1 912.5	134
pCSA-CT	–	•	◦	•	–	98.2	1 912.5	104
rRAPTOR	8	•	•	•	◦	203.4	1 812.5	1 179
rRAPTOR	8	•	•	•	•	203.4	1 579.6	878
rRAPTOR	16	•	•	•	•	206.4	1 634.0	922
mcpCSA-CT	8	•	•	•	–	203.4	15 299.8	255
mcpCSA-CT-SSE	8	•	•	•	–	203.4	1 912.5	221
mcpCSA-CT-SSE	16	•	•	•	–	206.4	3 824.9	466

4 User-Constrained Multimodal Route Planning

In this section, we present UCCH, the first multimodal speedup technique that handles arbitrary mode-sequence constraints as input to the query—a feature unavailable from previous techniques. Unlike Access-Node Routing [30], it also answers local queries correctly and requires significantly less preprocessing effort. We revisit one technique, namely *node contraction*, that has proven successful in road networks in the form of Contraction Hierarchies, introduced by Geisberger et al. [46]. By ensuring that shortcuts never span multiple modes of transport, we extend Contraction Hierarchies in a sound manner. Moreover, we show how careful engineering further helps our scenario. Our experimental study shows that, unlike previous techniques, we can handle an intercontinental instance composed of cars, railways and flights with over 50 million nodes, 125 million edges, and 30 thousand stations. With only 557 MiB of auxiliary data, we achieve query times that are fast enough for interactive scenarios.

Related Work For an overview on unimodal speedup techniques, we direct the reader to [11, 33]. Most techniques are composed of the following ingredients: Bidirectional search, goal-directed search [52, 57, 62, 85], hierarchical techniques [13, 14, 46, 53, 80], and separator-based techniques [25, 26, 59]. Combinations have been studied [16, 81].

Regarding multimodal route planning less work exists. An elegant approach to restricting modal transfers is the label constrained shortest paths problem (LCSPP) [68]: Edges are labeled, and the sequence of edge labels must be element of a formal language (passed as query input) for any feasible path. A version of Dijkstra's algorithm can be used, if the language is regular [10, 68]. An experimental study of this approach, including basic goal-directed techniques, is conducted in [9]. In [74] it is concluded that augmenting preprocessing techniques for LCSPP is a challenging task.

A first efficient multimodal speedup technique, called Access-Node Routing (ANR), has been proposed in [30]. It skips the road network during queries by precomputing distances from every road node to all its relevant access points of the public transportation network. It has the fastest query times of all previous multimodal techniques which are in the order of milliseconds. However, the preprocessing phase predetermines the modal constraints that can be used for queries. Also, it cannot compute short-range queries and requires a separate algorithm to handle them correctly.

Another approach adapts ALT by precomputing different node potentials depending on the mode

Table 4: Evaluating other instances. Start criterion and trip flags are always used.

Algorithm	# Tr.	Arr.	Tran.	Prof.	One.	Germany # Jn.	Germany # L.Cmp. p. Stop	Germany Time [ms]	Europe # Jn.	Europe # L.Cmp. p. Stop	Europe Time [ms]
TE	–	•	○	○	○	1.0	317.0	117.1	0.9	288.6	624.1
TD-col	–	•	○	○	○	1.0	11.9	3.5	0.9	10.0	21.6
CSA	–	•	○	○	○	1.0	228.7	3.4	0.9	209.5	19.5
TE	–	•	○	○	•	1.0	29.8	11.7	0.9	56.3	129.9
TD-col	–	•	○	○	•	1.0	6.8	2.0	0.9	5.3	11.5
CSA	–	•	○	○	•	1.0	40.8	0.8	0.9	74.2	8.3
pCSA-CT	–	•	○	•	–	20.2	429.5	4.9	11.4	457.6	46.2
rRAPTOR	8	•	•	•	○	29.4	752.1	161.3	17.2	377.5	421.8
rRAPTOR	8	•	•	•	•	29.4	640.1	123.0	17.2	340.8	344.9
mcpCSA-CT-SSE	8	•	•	•	–	29.4	429.5	17.9	17.2	457.6	98.2

of transport, called SDALT [61]. It has fast preprocessing, but both preprocessing space and query times are high. Also, it cannot handle arbitrary modal restrictions as query input. By combining SDALT with a label-correcting algorithm, the query time can be improved by up to 50 % [60].

Finally, in [78] a technique based on contraction is presented that handles arbitrary Kleene languages as user input. The authors use them to exclude certain road categories. They report speedups of 3 orders of magnitude on a continental road network. However, Kleene languages are rather restrictive: In a multimodal context, they only allow excluding modes of transportation *globally*. In particular, they cannot be used to define feasible *sequences* of transportation modes.

Outline This work is organized as follows. Section 4.1 sets necessary notation, summarizes graph models we use, precisely defines the problem we are solving, and also recaps Contraction Hierarchies. Section 4.2 introduces our new technique. Finally, Section 4.3 presents experiments to evaluate our algorithm, while Section 4.4 concludes this work and mentions interesting open problems.

4.1 Preliminaries

Throughout this work $G = (V, E)$ is a *directed graph* where V is the set of *nodes* and $E \subseteq V \times V$ the set of *edges*. For an edge $(u, v) \in E$, we call u the *tail* and v the *head* of the edge. The *degree* of a node $u \in V$ is defined as the number of edges $e \in E$ where u is either head or tail of e. The *reverse* graph $\overleftarrow{G} = (V, \overleftarrow{E})$ of G is obtained from G by flipping all edges, i. e., $(u, v) \in E$ if and only if $(v, u) \in \overleftarrow{E}$. Note that we use the terms graph and network interchangeably. To distinct between different modes of transport, our graphs are *labeled* by node labels $\mathrm{lbl} : V \to \Sigma$ and edge labels $\mathrm{lbl} : E \to \Sigma$. Often Σ is called the *alphabet* and contains the available modes of transport in G, for example, `road`, `rail`, `flight`. All edges in our graphs are *weighted* by periodic time-dependent travel time functions $f : \Pi \to \mathbb{N}_0$ where Π depicts a set of time points (think of it as the seconds of a day). If f is constant over Π, we call f *time-independent*. Respecting periodicity in a meaningful way, we say that a function f has the *FIFO property* if for all $\tau_1, \tau_2 \in \Pi$ with $\tau_1 \leq \tau_2$ it holds that $f(\tau_1) \leq f(\tau_2) + (\tau_2 - \tau_1)$. In other words, waiting never pays off. Moreover, the *link* operation of two functions f_1, f_2 is defined as $f_1 \oplus f_2 = f_1 + (\mathrm{id} + f_1) \circ f_2$, and the *merge* operation $\min(f_1, f_2)$ is defined as the element-wise minimum of f_1 and f_2. Note that to depict the travel time function $f(\tau)$ of an edge $e \in E$, we sometimes write $\mathrm{len}(e, \tau)$, or just $\mathrm{len}(e)$ if it is clear from the context that $\mathrm{len}(e, \tau)$ is constant.

In time-dependent graphs there are two types of queries relevant to this work: A *time-query* has

as input $s \in V$ and a departure time τ. It computes a shortest path tree to every node $u \in V$ when departing at s at time τ. In contrast, a *profile-query* computes a shortest path graph from s to all $u \in V$, consisting of shortest paths for all departure times $\tau \in \Pi$.

Whenever appropriate, we use some notion of formal languages. A finite sequence $w = \sigma_0\sigma_1 \dots \sigma_k$ of symbols $\sigma_i \in \Sigma$ is called a *word*. A not necessarily finite set of words L is called formal *language* (over Σ). A nondeterministic finite *automaton* (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, S, F)$ characterized by the set Q of *states*, the *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, and sets $S \subseteq Q$ of *initial states* and $F \subseteq Q$ of *final states*. A language L is called *regular* if and only if there is a finite automaton $\mathcal{A}_L$ such that $\mathcal{A}_L$ accepts L.

4.1.1 Models

Following [30], our multimodal graphs are composed of different models for each mode of transportation. We briefly introduce each model and explain how they are combined.

In the *road network*, nodes model intersections and edges depict street segments. We either label edges by `car` for roads or `foot` for pedestrians. Our road networks are weighted by the average travel time of the street segment. For pedestrians we assume a walking speed of 4.5 kph. Note that our road networks are time-independent.

Regarding the *railway network*, we use the coloring model [28] which is based on the well-known realistic time-dependent model [77]. It consists of station nodes connected to route nodes. Trains are modeled between route nodes via time-dependent edges. Different trains use the same route node as long as they are not conflicting. In the coloring model conflicting trains are computed explicitly which yields significantly smaller graphs compared to the original realistic time-dependent model (without dropping correctness). Moreover, to enable transfers between trains, some station nodes are interconnected by time-independent foot paths. See [28] for details. We label nodes and edges with `rail`. Note that we also use this model for bus networks.

Finally, to model *flight networks*, we use the time-dependent phase II model [31]. It has small size and models airport procedures realistically. Nodes and edges are labeled with `flight`.

Note that the travel time functions in our networks are a special form of piecewise linear functions that can be efficiently evaluated [77, 27]. Also all edges in our networks have the FIFO property.

Merging the Networks To obtain an integrated *multimodal* network $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, we merge the node and edge sets of each individual network. Detailed data on transfers between modes of transport was not available to us. Thus, we heuristically add link edges labeled `link`. More precisely, we link each station node in the railway network to its geographically closest node of the road network. We also link each airport node of the flight network to their closest nodes in the road and rail networks. Thereby we only link nodes that are no more than distance δ apart, a parameter chosen for each instance. The time to traverse a link edge is computed from its geographical length and a walking speed of 4.5 kph.

4.1.2 Path Constraints on the Sequences of Transport Modes

Since the naïve approach of using Dijkstra's algorithm on the combined network $\mathbf{G}$ does not incorporate modal constraints, we consider the Label Constrained Shortest Path Problem (LCSPP) [10]: Each edge $e \in \mathbf{E}$ has a label $\mathrm{lbl}(e)$ assigned to it. The goal is to compute a shortest s-t-path P where the word $w(P)$ formed by concatenating the edge labels along P is element of a language L, a query input.

Modeling sequence constraints is done by specifying L. For our case, regular languages of the following form suffice. The alphabet Σ consists of the available transport modes. In the corresponding NFA $\mathcal{A}_L$, states depict one or more transport modes. To model traveling within one transport mode, we require $(q, \sigma, q) \in \delta$ for those transport modes $\sigma \in \Sigma$ that q represents. Moreover, to allow transfers between different modes of transport, states $q, q' \in Q$, $q \neq q'$ are connected by `link` labels,

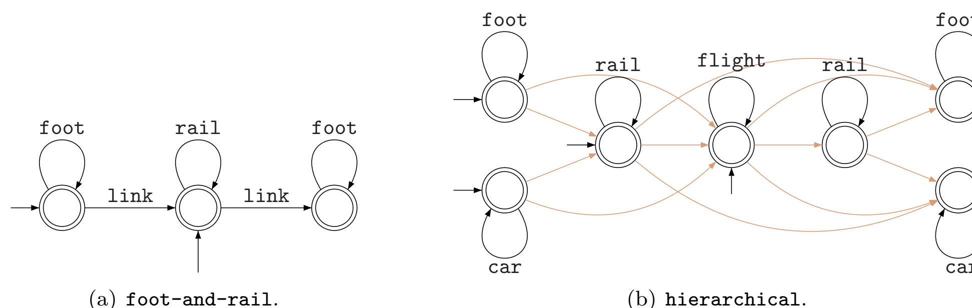


Figure 1: Two example automata. In the right figure, light edges are labeled as `link`.

i. e., $(q, \texttt{link}, q') \in \delta$. Finally, states are marked as initial/final if its modes of transport can be used at the beginning/end of the journey. Example automata are shown in Figure 1.

We refer to this variant of LCSPP as LCSPP-MS (as in Modal Sequences). In general, LCSPP is solvable in polynomial time, if L is context-free. In our case, a generalization of Dijsktra's algorithm works [10].

4.1.3 Contraction Hierarchies (CH)

Our algorithm is based on Contraction Hierarchies [46]. Preprocessing works by heuristically ordering the nodes of the graph by an *importance* value (a linear combination of edge expansion, number of contracted neighbors, among others). Then, all nodes are contracted in order of ascending importance. To contract a node $v \in V$, it is removed from G, and shortcuts are added between its neighbors to preserve distances between the remaining nodes. The index at which v has been removed is denoted by $r(v)$. To determine if a shortcut (u, w) is added, a local search from u is run (without looking at v), until w is settled. If $\text{len}(u, w) \le \text{len}(u, v) + \text{len}(v, w)$, the shortcut (u, w) is not added, and the corresponding shorter path is called a *witness*.

The CH query is a bidirectional Dijkstra search operating on G, augmented by the shortcuts computed during preprocessing. Both searches (forward and backward) go "upward" in the hierarchy: The forward search only visits edges (u, v) where $r(u) \le r(v)$, and the backward search only visits edges where $r(u) \ge r(v)$. Nodes where both searches meet represent candidate shortest paths with combined length μ. The algorithm minimizes μ, and a search can stop as soon as the minimum key of its priority queue exceeds μ. Furthermore, we make use of *stall-on-demand*: When a node v is scanned in either query, we check for all its incident edges $e = (u, v)$ of the *opposite* direction if $\text{dist}(u) + \text{len}(e) < \text{dist}(v)$ holds ($\text{dist}(v)$ denotes the tentative distance at v). If this is the case, we may prune the search at v. See [46] for details.

Partial Hierarchy If the preprocessing is aborted prematurely, i. e., before all nodes are contracted, we obtain a partial hierarchy (PCH). Let $r(v) = \infty$ if and only if v is never contracted, then the same query algorithm as for Contraction Hierarchies is applicable and yields correct results. We call the induced subgraph of all uncontracted nodes the *core*, and the remaining (contracted) subgraph the *component*. Note that both core and component can contain shortcuts not present in the original graph.

Performance Both preprocessing and query performance of CH depend on the number of shortcuts added. It works well if the network has a pronounced hierarchy, i. e., far journeys eventually converge to a "freeway subnetwork" which is of a small fraction in size compared to the total graph [3]. Note that if computing a complete hierarchy produces too many shortcuts, one can always abort early

and compute a partial hierarchy. A possible stopping criterion is the *average node degree* on the core that is approached during the contraction process.

4.2 Our Approach

We now introduce our basic approach and show how CH can be used to compute shortest path with restrictions on sequences of transport modes. We first argue that applying CH on the combined multimodal graph **G** without careful consideration either yields incorrect results to LCSPP-MS or predetermines the automaton $\mathcal{A}$ during preprocessing. We then introduce UCCH: A practical adaption of Contraction Hierarchies to LCSPP-MS that enables arbitrary modal sequence constraints as query input. Further improvements that help accelerating both preprocessing and queries are presented in Section 4.2.3.

4.2.1 Contraction Hierarchies for Multimodal Networks

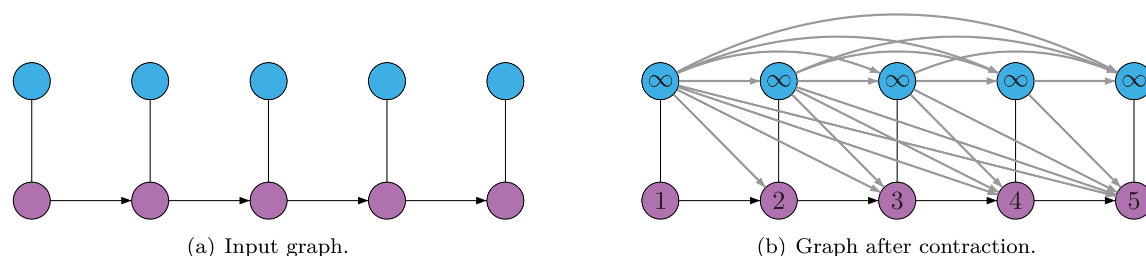
Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a multimodal network. Recall that **G** is a combination of time-independent and time-dependent networks (for example, of road and rail), hence, contains edges having both constants and travel time functions associated with them. Applying CH to **G** already requires some engineering effort: Shortcuts may represent paths containing edges of different type. In order to compute the shortcuts' travel time functions, these edges have to be linked, resulting in *inhomogeneous* functions that slow down both preprocessing and query performance. More significantly, when a path $P = (e_1, \ldots, e_k)$ is composed into a single shortcut edge e', its labels need to be concatenated into a super label $\text{lbl}(e') = \text{lbl}(e_1) \cdots \text{lbl}(e_k)$. In particular, if there are subsequent edges e_i, e_j in P where $\text{lbl}(e_i) \neq \text{lbl}(e_j)$, the shortcut induces a modal transfer. Running a query where this particular mode change is prohibited potentially yields incorrect results: The shortcut must not be used but the label constrained path (i. e. the one without this transfer) may have been discarded during preprocessing by the witness search (see Section 4.1.3). Note that the partial time-dependent nature of **G** further complicates things. A shortcut $e' = (u, v)$ needs to represent the travel time profile from u to v, that is, the underlying path P depends on the time of day. As a consequence, the super label of e' is time-dependent as well.

If the automaton $\mathcal{A}$ is known during preprocessing, we can modify CH preprocessing to yield correct query results with respect to $\mathcal{A}$. While contracting node $v \in \mathbf{G}$ and thereby considering to add a shortcut $e' = (u, w)$, we look at its super label $\text{lbl}(e') = (\text{lbl}_1, \ldots, \text{lbl}_k)$. To determine if e' has to be inserted, we run multiple witness searches as follows: For each state $q \in \mathcal{A}$ where q represents $\text{lbl}(v)$, we run a multimodal profile-search from u (ignoring v). We run it with q as initial state and all those states $q' \in \mathcal{A}$ as final state, where q' is reachable from q in $\mathcal{A}$ by applying $\text{lbl}(e')$. Only if for all these profile-searches $\text{dist}(w) \leq \text{len}(e')$ holds, the shortcut e' is not required: For every relevant transition sequence of the automaton, there is a shorter path in the graph. Note that shortcuts $e' = (u, w)$ may be required even if an edge from u to w already existed before contraction. This results in parallel edges for different subsequences of the constraint automaton.

This approach which we call *State-Dependent CH* (SDCH) has some disadvantages, however. First, witness search is slow and less effective than in the unimodal scenario, resulting in many more shortcuts. This hurts preprocessing and query performance. Adding to it the more complicated data structures required for inhomogeneous travel time functions and arbitrary label sequences, SDCH combines challenges of both Flexible CH [45] and Timetable CH [44]. As a result we expect a significant slowdown over unimodal CH on road networks. But most notably, SDCH predetermines the automaton $\mathcal{A}$ during preprocessing.

4.2.2 UCCH: Contraction for User-Constrained Route Planning

We now introduce User-Constrained Contraction Hierarchies (UCCH). Unlike SDCH, it can handle arbitrary sequence constraint automata during query and has an easier witness search. We first turn



(a) Input graph. (b) Graph after contraction.

Figure 2: Contracting only route nodes in the realistic time-dependent rail model [77]. The bottom row of nodes are station nodes, while the top row are route nodes contracted in the order depicted by their labels. Grey edges represent added shortcuts. Note that these shortcuts are required as they incorporate different transfer times (for boarding and exiting vehicles at different stations).

towards preprocessing and then we go into the details of the query algorithm.

Preprocessing The main reason behind the disadvantages discussed in Section 4.2.1 is the computation of shortcuts that span over boundaries of different modal networks. Instead, let Σ be the alphabet of labels of a multimodal graph $\mathbf{G}$. We now process each subnetwork independently. We compute—in no particular order—a partial Contraction Hierarchy restricted to the subgraph $G_{\text{lbl}} = (V_{\text{lbl}}, E_{\text{lbl}})$ (for every $\text{lbl} \in \Sigma$). Here, G_{lbl} is exactly the original graph of the particular transportation mode (before merging). We keep the contraction order with the exception of *transfer nodes*: Nodes which are incident to at least one edge labeled `link` in $\mathbf{G}$. We fix the rank of all such nodes v to infinity, i. e., they are never contracted. Note that all other nodes have only incident edges labeled by lbl in $\mathbf{G}$. As a result, shortcuts only span edges within one modal network. Hence, we neither obtain inhomogeneous travel time functions nor "mixed" super labels. We set the label of each shortcut edge e' to $\text{lbl}(e)$, where e is an arbitrary edge along the path, e' represents.

To determine if a shortcut $e' = (u, w)$ is required (when contracting a node v), we restrict the witness search to the modal subnetwork G_{lbl} of v. Restricting the search space of witness searches does not yield incorrect query results: Only *too many* shortcuts might be inserted, but no required shortcuts are *omitted*. In fact, this is a common technique to accelerate CH preprocessing [46]. Note that broadening the witness search beyond network boundaries is prohibitive in our case: It may find a shorter u-v-path using parts of other modal networks. However, such a path is not necessarily a witness if one of these other modes is forbidden during the query. Thus, we must not take it into account to determine if e' can be dropped.

Our preprocessing results in a partial hierarchy for each modal network of $\mathbf{G}$. Its transfer nodes are not contracted, thus, stay at the top of the hierarchy. Recall that we call the subgraph induced by all nodes v with $r(v) = \infty$ the *core*. Because of the added shortcuts, the shortest path between every pair of core nodes is also fully contained in the core. As a result, we achieve independence from the automaton $\mathcal{A}$ during preprocessing.

A Practical Variant Contraction is independent for every modal network of $\mathbf{G}$: We can use any combination of partial, full or no contraction. Our *practical variant* only contracts time-independent modal networks, that is, the road networks. Contracting the time-dependent networks is much less effective. Recall that we do not contract station nodes as they have incident link edges. Applying contraction only on the non-station nodes, however, yields too many shortcuts (see Figure 2 and [44]). It also hides information encoded in the timetable model (such as railway lines), further complicating query algorithms [18].

Query Our query algorithm combines the concept of a multimodal Dijkstra algorithm with unimodal CH. Let $s, t \in \mathbf{V}$ be source and target nodes and $\mathcal{A}$ some finite automaton with respect to LCSPP-MS. Our query algorithm works as follows. First, we initialize distance values for all pairs of $(v, q) \in \mathbf{V} \times \mathcal{A}$ with infinity. We now run a bidirectional Dijkstra search from s and t. Each search runs independently and maintains priority queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$ of tuples (v, q) where $v \in \mathbf{V}$ and $q \in \mathcal{A}$. We explain the algorithm for the forward search; the backward search works analogously. The queue $\overrightarrow{Q}$ is ordered by distance and initialized with (s, q) for all initial states q in $\mathcal{A}$ (the backward queue is initialized with respect to final states). Whenever we extract a tuple (v, q) from Q, we scan all edges $e = (v, w)$ in $\mathbf{G}$. For each edge, we look at all states q' in $\mathcal{A}$ that can be reached from q by $\mathrm{lbl}(e)$. For every such pair (w, q') we check whether its distance is improved, and update the queue if necessary. To use the preprocessed data, we consider the graph $\mathbf{G}$, augmented by all shortcuts computed during preprocessing. We run the aforementioned algorithm, but when scanning edges from a node v, the forward search only looks at edges (v, w) where $r(w) \geq r(v)$. Similarly, the backward search only looks at edges (v, w) where $r(v) \geq r(w)$. Note that by these means we automatically search inside the core whenever we reach the top of the hierarchy. Thereby we never reinitialize any data structures when entering the core like it is typically the case for core-based algorithms, e. g., Core-ALT [33]. The stopping criterion carries over from basic CH: A search stops as soon as its minimum key in the priority queue exceeds the best tentative distance value μ. We also use stall-on-demand, however, only on the component.

Intuitively, the search can be interpreted as follows. We simultaneously search upward in those hierarchies of the modal networks that are either marked as initial or as final in the automaton $\mathcal{A}$. As soon as we hit the top of the hierarchy, the search operates on the common core. Because we always find correct shortest paths between core nodes in *any* modal network, our algorithm supports *arbitrary* automata (with respect to LCSPP-MS) as query input. Note that our algorithm implicitly computes *local* queries which use only one of the networks. It makes the use of a separate algorithm for local queries, as in [30], unnecessary.

Handling Time-Dependency Some of the networks in $\mathbf{G}$ are time-dependent. Weights of time-dependent edges (u, v) are evaluated for a departure time τ. However, running a reverse search on a time-dependent network is non-trivial, since the arrival time at the target node is not known in advance. Several approaches, such as using the lower-bound graph for the reverse search, exist [29, 15], but they complicate the query algorithm. Recall that in our practical variant we do not contract any of the time-dependent networks, hence, no time-dependent edges are contained in the component. This makes backward search on the component easy for us. We discuss search on the core in the next section.

4.2.3 Improvements

We now present improvements to our algorithm, some of which also apply to CH.

Average Node Degree Recall that whenever we contract a modal network, we never contract transfer nodes, even if they were of low importance in the context of that network. As a result, the number of added shortcuts may increase significantly. Thus, we stop the contraction process as soon as the *average node degree* in the core exceeds a value α. By varying α, we trade off the number of core nodes and the number of core edges: Higher values of α produce a smaller core but with more shortcut edges. We evaluate a good value of α experimentally.

Edge Ordering Due to the higher average node degree compared to unimodal CH, the search algorithm has to look at more edges. Thus, we improve performance of iterating over incident edges of a node v by *reordering* them locally at v: We first arrange all outgoing edges, followed by all bidirected edges, and finally, all incoming edges. By these means, the forward respective

backward search only needs to look at their relevant subsets of edges at v. The same optimization is applied to the stalling routine. Preliminary experiments revealed that edge reordering improves query performance up to 21 %.

Node Ordering To improve the cache hit rate for the query algorithm, we also reorder nodes such that adjacent nodes are stored consecutively with high probability. We use a DFS-like algorithm to determine the ordering [24]. Because most of the time is spent on the core, we also move core nodes to the front. This improves query performance up to a factor of 2.

Core Pruning Recall that a search stops as soon as its minimum key from the priority queue exceeds the best tentative distance value μ. This is conservative, but necessary for CH (and UCCH) to be correct. However, UCCH spends a large fraction of the search inside the core. We can easily expand road and transfer edges both forward and backward, but because of the conservative stopping criterion, many core nodes are settled twice. To reduce this amount, we do not scan edges of core nodes v, where v has been settled by both searches and did not improve μ. A path through v is provably not optimal. This increases performance by up to 47 %. Another alternative is not applying bidirectional search on the core at all. The forward search continues regularly, while the backward search does not scan edges incident to core nodes. This approach turns out most effective with a performance increase by a factor of 2.

State Pruning Recall that our query algorithm maintains distances for *pairs* (v, q) where $v \in \mathbf{V}$ and $q \in \mathcal{A}$. Thus, whenever we scan an edge $(u, v) \in \mathbf{E}$ resulting in some state $q \in \mathcal{A}$, we update the distance value of (v, q) only if it is improved, and discard (or prune it) it otherwise. However, we can even make use of a stronger *state pruning* rule: Let q_i and q_j be two states in $\mathcal{A}$. We say that q_i *dominates* q_j if and only if the language $L_{\mathcal{A}}(q_j)$ accepted by $\mathcal{A}$ with modified initial state q_j is a subset of the language $L_{\mathcal{A}}(q_i)$ accepted by $\mathcal{A}$ with modified initial state q_i. In other words, any feasible mode sequence beginning with q_j is also feasible when starting at q_i. As a consequence, when we are about to update a pair (v, q_j), we can additionally prune (v, q_j) if there exists a state q_i that dominates q_j and where (v, q_i) has smaller distance: Any shortest path from v is provably not using (v, q_j). As an example, consider the first automaton in Figure 1. Let its states be denoted by $\{q_0, q_1, q_2\}$, from left to right. Here, q_0 dominates q_2 with respect to our definition: Any foot path beginning at state q_2 is also a feasible (foot) path beginning at state q_0. Therefore, any pair (v, q_2) can be pruned if (v, q_0) has better distance than (v, q_2). State pruning improves performance by $\approx 10\,\%$.

State-Independent Search in Component Automata are used to model sequence constraints, however, by definition their state may only change when traversing link edges. In particular, when searching inside the component, there is never a state transition (recall that all link edges are inside the core). Thus, we use the automaton only on the core. We start with a regular unimodal CH-query. Whenever we are about to insert a core node v into the priority for the first time on a branch of the shortest path tree, we create labels (v, q) for all initial/final states q (regarding forward/backward search). Because the amount of settled component nodes on average is small compared to the total search space, we do not observe a performance gain. However, on large instances with complicated query automata we save several gigabytes of RAM during query by keeping only one distance value for each component node. Recall that component nodes constitute the major fraction of the graph.

Parallelization In general, the multimodal graph $\mathbf{G}$ is composed of more than one contractable modal subnetwork, for instance foot and car. In this case, we have to run the aforementioned unimodal CH-query on every component individually. Because these queries are independent from each other, we are able to parallelize them easily. In a first phase, we allocate one thread for every contracted network which then runs the unimodal CH-query on its respective component until it

Table 5: Comparing size figures of our input instances. The column "col." indicates whether we use the coloring approach (see Section 4.1.1) to model the railway subnetwork. The bottom two instances are taken from [30].

	Public Transportation			Road		
network	stations	connections	col.	nodes	edges	density
`ny-road-rail`	16 897	2 054 896	•	579 849	1 527 594	1 : 56
`de-road-rail`	6 822	489 801	•	5 055 680	12 378 224	1 : 749
`europe-road-rail`	30 517	1 621 111	•	30 202 516	72 586 158	1 : 1 133
`wo-road-rail-flight`	31 689	1 649 371	•	50 139 663	124 625 598	1 : 1 846
`de-road-rail(long)`	498	16 450	◦	5 055 680	12 378 224	1 : 10 711
`wo-road-flight`	1 172	28 260	◦	50 139 663	124 625 598	1 : 139 277

hits the core. In the second phase, we synchronize the threads, and continue the search on the core sequentially. Note that we only need to run the first phase on those components that are represented by an initial or final state in the input automaton $\mathcal{A}$.

Combining all improvements yields a speedup of up to factor 4.9.

4.3 Experiments

We conducted our experiments on one core of an Intel Xeon E5430 processor running SUSE Linux 11.1. It is clocked at 2.66 GHz, has 32 GiB of RAM and 12 MiB of L2 cache. The program was compiled with GCC 4.5, using optimization level 3. Our implementation is written in C++ using the STL and Boost at some points. As a priority queue we use a 4-ary heap.

Inputs We assemble a total of six multimodal networks where two are imported from [30]. Their size figures are reported in Table 5. For `ny-road-rail`, we combine New York's foot network with the public transit network operated by MTA [69]. We link bus and subway stops to road intersections that are no more than 500 m apart. The `de-road-rail` network combines the pedestrian and railway networks of Germany. The railway network is extracted from the timetable of the winter period 2000/01. It includes short and long distance trains, and we link stations using a radius of 500 m. The `europe-road-rail` network combines the road (as in car) and railway networks of Western Europe. The railway network is extracted from the timetable of the winter period 1996/97 and stations are linked within 5 km. The `wo-road-rail-flight` network is a combination of the road networks of North America and Western Europe with the railway network of Western Europe and the flight network of Star Alliance and One World. The flight networks are extracted from the winter timetable 2008. As radius we use 5 km.

Both `de-road-rail(long)` and `wo-road-flight` stem from [30]. The data of the Western European and North American road networks (thus Germany and New York) was kindly provided to us by PTV AG [76] for scientific use. The timetable data of New York is publicly available through General Transit Feeds [48], while the data of the German and European railway networks was kindly provided by HaCon [54]. Unfortunately, the New York timetable did not contain any foot path data for transfers. Thus, we generated artificial foot paths with a known heuristic [28].

Our instances have varying fractional size of their public transit parts. We call the fraction of linked nodes in a subgraph *density* (see last column of Table 5). Our densest network is `ny-road-rail`, which also has the highest number of connections. On the other hand, `de-road-rail(long)` and `wo-road-flight` are rather sparse. However, we include them to compare our algorithm to Access Node Routing (ANR). Also note that for this reason we do not use the improved coloring model (see Section 4.1.1) on these two instances.

We use the following automata as query input. The `foot-and-rail` automaton allows either

Table 6: Comparing preprocessing performance of UCCH on `de-road-rail` with varying average core degree limit. For queries we use the `foot` automaton. We also report numbers for unconstrained unimodal CH.

	Preprocessing				Query			
	avg core-degree	core-nodes	shortcut-edges	time [min]	settled nodes	relaxed edges	touched edges	time [ms]
UCCH	10	30 908	42.3 %	6	15 531	27 506	155 776	5.85
	15	16 003	43.1 %	7	8 090	16 844	121 631	3.11
	20	12 239	43.7 %	9	6 240	14 425	124 201	2.82
	25	10 635	44.2 %	10	5 465	13 687	135 151	2.80
	30	9 742	44.7 %	12	5 049	13 486	148 735	2.96
	35	9 171	45.1 %	14	4 794	13 598	163 376	3.15
	40	8 788	45.4 %	15	4 628	13 787	179 483	3.38
PCH	13	10 635	41.7 %	6	5 567	11 402	71 860	1.93
PCH	15	6 750	41.8 %	7	3 636	7 970	53 655	1.37
CH	—	0	41.8 %	9	677	1 290	11 434	0.25

walking, or walking, taking the railway network and walking again. Similarly, the `car-and-rail` automaton uses the road network instead of walking, while the `car-and-flight` automaton uses the flight network instead of the railway network. The `hierarchical` automaton is our most complicated automaton. It hierarchically combines road, railways and flights (in this order). All modal sequences are possible, except of going up in the hierarchy after once stepping down. For example, if one takes a train after a flight, it is impossible to take another flight. Finally, the `everything` automaton allows arbitrary modal sequences in any order. See Figure 1 for transition graphs of `foot-and-rail` and `hierarchical`.

Methodology We evaluate both preprocessing and query performance. The contraction order is always computed according to the aggressive variant from [46]. We report the time and the amount of computed auxiliary data. Queries are generated with source, target nodes and departure times uniformly picked at random. For Dijkstra we run 1 000 queries, while for UCCH we run a superset of 100 000 queries. We report the average number of: (1) extracted nodes in the implicit product graph from the priority queue (settled nodes), (2) priority queue update operations (relaxed edges), (3) touched edges, (4) the average query time, and (5) the speedup over Dijkstra. Note that we only report the time to compute the length of the shortest path. Unpacking of shortcuts can be done efficiently in less than a millisecond [46].

4.3.1 Evaluating Average Core Degree Limit

The first experiment evaluates preprocessing and query performance with varying average core degree. We abort contraction as soon as the average node degree in the core exceeds a limit α. In our implementation we compute the average node degree as follows. We divide the number of edges by the number of nodes in our graph data structure. Note that we use *edge compression* [22]: Whenever there are edges $e = (u, v)$ and $e' = (v, u)$ where $\text{len}(e) = \text{len}(e')$, we combine both edges in a single entry at u and v. As a result, the number we report may be smaller than the true average degree (at most by a factor of 2) which is, however, irrelevant for the result of this experiment.

Table 6 shows preprocessing and query figures on `de-road-rail`. We use an automaton that does not use public transit edges. With higher values of α more nodes are contracted, resulting in higher preprocessing time and more shortcuts (we report them as a fraction of the input's size). At the same time, less nodes (but with higher degree) remain in the core. Setting $\alpha = \infty$ is infeasible. The amount of shortcuts explodes, and preprocessing does not finish within reasonable time. Interestingly,

Table 7: Preprocessing figures for UCCH and Access-Node Routing on the road subnetwork. Figures for the latter are taken from [30]. We scale the preprocessing time with respect to running time figures compared to Dijkstra.

	UCCH						Access-Node	
	avg core-	core nodes		shortcuts	shortcuts	time	space	time
network	degree	total	ratio	percent	[MiB]	[min]	[MiB]	[min]
ny-road-rail	8	11 057	1:52	48.3 %	8	< 1	—	—
de-road-rail	25	10 635	1:475	44.2 %	63	10	—	—
europe-road-rail	25	39 665	1:761	39.0 %	324	38	—	—
wo-road-rail-flight	30	38 610	1:1 298	39.1 %	558	87	—	—
de-road-rail(long)	35	996	1:5 075	42.3 %	60	10	504	26
wo-road-flight	35	727	1:68 967	38.0 %	542	78	14 050	184

the query time decreases (with smaller core size) up to $\alpha \approx 25$ and then increases again. Though we settle less nodes, the increase in shortcuts results in more touched edges during query, that is, edges the algorithm has to iterate when settling a node. We conclude that for `de-road-rail` the trade-off between number of core nodes and added shortcut edges is optimal for $\alpha = 25$. Hence, we use this value in subsequent experiments. Accordingly, we determine α for all instances.

Comparison to Unimodal CH In Table 6 we also compare UCCH to CH when run on the unimodal road network. Computing a full hierarchy results in queries that are faster by a factor of 11.2. Since UCCH does not compute a full hierarchy by design, we evaluate two partial CH hierarchies: The first stops when the core reaches a size of 10 635—equivalent to the optimal core size of UCCH. We observe a query performance almost comparable to UCCH (slightly faster by 45 %). The second partial hierarchy stops with a core size of 6 750 which is equal to the number of transfer nodes in the network (i. e., the smallest possible core size on this instance for UCCH). Here, CH is a factor of 2 faster than UCCH. Recall that UCCH must not contract transfer nodes. In road networks these are usually unimportant: Long-range queries do not pass many railway stations or bus stops in general, which explains that UCCH's hierarchy is less pronounced. However, for *multimodal* queries transfer nodes are indeed very important, as they constitute the interchange points between different networks. To enable arbitrary automata during query, we overestimate their importance by not contracting them at all, which is reflected by the (relatively small) difference in performance compared to CH.

4.3.2 Preprocessing

Table 7 shows preprocessing figures for UCCH on all our instances. Besides the average degree we evaluate the core in terms of total and fractional number of core nodes, and the amount of added shortcuts. Added shortcuts are reported as percentage of all road edges and in total MiB. We observe that the preprocessing effort correlates with the graph size. On the small `ny-road-rail` instance it takes less than a minute and produces 8 MiB of data. On our largest instance, `wo-road-rail-flight`, we need 1.5 hours and produce 558 MiB of data. Because the size of the core depends on the size of the public transportation network, we obtain a much higher ratio of core nodes on `ny-road-rail` (1 : 52) than we do, for example, on `wo-road-rail-flight` (1 : 1 298).

Comparing the preprocessing effort of UCCH to scaled figures of ANR, we observe that UCCH is more than twice as fast and produces significantly less amount of data: on `de-road-rail(long)` by a factor of 8.4, while on `wo-road-flight`, ANR requires 14 GiB of space. Here, UCCH only uses 542 MiB, a factor of 26. Concluding, UCCH outperforms ANR in terms of preprocessing space and time.

Table 8: Query performance of UCCH compared to plain multimodal Dijkstra and Access-Node Routing. Figures for the latter are taken from [30]. We scale the running time with respect to Dijkstra.

network	automaton	Dijkstra		Access-Node			UCCH		
		settled nodes	time [ms]	settled nodes	time [ms]	speed-up	settled nodes	time [ms]	speed-up
`ny-road-rail`	`foot-and-rail`	404 816	226	—	—	—	25 525	13.61	17
`de-road-rail`	`foot-and-rail`	2 611 054	2 005	—	—	—	18 275	12.78	157
`europe-road-rail`	`car-and-rail`	30 021 567	23 993	—	—	—	90 579	53.78	446
`wo-road-rail-flight`	`car-and-flight`	36 053 717	33 692	—	—	—	42 056	26.72	1 260
`wo-road-rail-flight`	`hierarchical`	36 124 105	35 261	—	—	—	126 072	70.52	500
`wo-road-rail-flight`	`everything`	25 267 202	23 972	—	—	—	71 389	50.77	472
`de-road-rail(long)`	`foot-and-rail`	2 735 426	2 075	13 524	3.45	602	12 509	3.13	663
`wo-road-flight`	`car-and-flight`	36 582 904	33 862	4 200	1.07	31 551	1 647	0.67	50 540

4.3.3 Query Performance

In this experiment we evaluate the query performance of UCCH and compare it to Dijkstra and ANR (where applicable). Figures are presented in Table 8. We observe that we achieve speedups of several orders of magnitude over Dijkstra, depending on the instance. Generally, UCCH's speedup over Dijkstra correlates with the ratio of core nodes after preprocessing (thus, indirectly with the density of transfer nodes): the sparser our networks are interconnected, the higher the speedups we achieve. On our densest network, `ny-road-rail`, we have a speedup of 17, while on `wo-road-flight` we achieve query times of less than a millisecond—a speedup of over 50 540. Note that most of the time is spent inside the core (particularly, in the public transit network), which we do not accelerate. Comparing UCCH to ANR, we observe that query times are in the same order of magnitude, UCCH being slightly faster. Note that we achieve these running times with significantly less preprocessing effort.

4.4 Conclusion

In this work we introduced UCCH: The first, fast multimodal speedup technique that handles arbitrary modal sequence constraints at *query time*—a problem considered challenging before. Besides not determining the modal constraints during preprocessing, its advantages are small space overhead, fast preprocessing time and the ability to implicitly handle local queries without the need for a separate algorithm. Its preprocessing can handle huge networks of intercontinental size with many more stations and airports than those of previous multimodal techniques. For future work we are interested in augmenting our approach to more general scenarios such as profile or multi-criteria queries. We also like to further accelerate search on the uncontracted core—especially on the rail networks. Moreover, we are interested to improve the contraction order. In particular, we like to use ideas from [30] to enable contraction of some transfer nodes in order to achieve better results, especially on more densely interlinked networks.

5 Multi-Criteria Search: Finding (All) the Good Options

As mentioned before, meaningful multimodal optimization needs to take more criteria into account, such as walking duration and costs. Some people are happy to walk 10 minutes to avoid an extra transfer, while others are not. In fact, some will walk half an hour to avoid using public transportation at all. Taking a taxi all the way to the airport is a good solution for some; users on a budget may prefer a cheaper solution. Not only do these additional criteria significantly increase

the Pareto set [34, 45], but some of the resulting journeys tend to look unreasonable, as Figure 5 in Section 5.1.1 illustrates.

As a result, recent research efforts tend to avoid multicriteria search altogether [12], looking for reasonable routes by other means. A natural approach is to work with a weighted combination of all criteria, transforming the search into a single-criterion problem [4, 7, 70, 86]. When extended to find the k-shortest paths [20, 42], this method can even take user preferences into account. Unfortunately, linear combination may produce undesired results [21] (see Section 5.1.1 for an example). To avoid such issues, another line of multimodal single-criterion research considers the computation of label-constrained quickest journeys [10, 68]. The idea is to label edges according to the mode of transportation and require paths to obey a user-defined pattern (often given as regular expressions), typically enforcing a hierarchy of modes [20, 86] (such as "no car travel between trains"). The main advantage of this strategy is that preprocessing techniques developed for road networks carry over [9, 30, 37, 60, 61]. This approach, however, can hide interesting journeys (for example, taking a taxi between train stations in Paris may be an option). In fact, this exposes a fundamental conceptual problem with label-constrained optimization: It essentially relies on the user to know his options before planning the journey.

5.1 Computing and Evaluating Multimodal Journeys

Given the limitations of current approaches, we revisit the problem of finding multicriteria multimodal journeys on a metropolitan scale. Instead of optimizing each mode of transportation independently [40], we argue in Section 5.1.1 that most users optimize three criteria: travel time, convenience, and costs. While this produces a large Pareto set, we propose using fuzzy logic [41, 88] to filter it in a principled way to a modest-sized set of representative journeys. This postprocessing step is not only quick, but can also be user-dependent, incorporating personal preferences. As Section 5.1.2 shows, recent algorithmic developments [32, 37, 47] allow us to answer exact queries optimizing time and convenience in less than two seconds within a large metropolitan area, for the simpler scenario of walking, cycling, and public transit. Unfortunately, this is not enough for interactive applications, and becomes much slower when additional criteria, such as costs, are incorporated. We therefore also propose (in Section 5.1.3) heuristics (still multicriteria) that are significantly faster, and closely match the top journeys in the Pareto set. Section 5.1.4 presents a thorough experimental evaluation of all algorithms in terms of both solution quality and performance, and shows that our approach can be fast enough for interactive applications. Moreover, since it does not rely on heavy preprocessing, it can be used in fully dynamic scenarios.

5.1.1 Problem Statement

We want to find journeys in a network built from several *partial networks*. The first is a *public transportation network* representing all available schedule-based means of transportation, such as trains, buses, rail, or ferries. We can specify this network in terms of its timetable, which is defined as follows. A *stop* is a location in the network (such as a train platform or a bus stop) at which a user can board or leave a particular vehicle. A *route* is a fixed sequence of stops for which there is scheduled service during the day; a typical example is a bus or subway line. A route is served by one or more distinct *trips* during the day; each trip is associated with a unique vehicle, with fixed (scheduled) arrival and departure times for every stop in the route. Each stop may also keep a *minimum change time*, which must be obeyed when changing trips.

Besides the public transportation network, we also take as input several *unrestricted networks*, with no associated timetable. Walking, cycling, and driving are modeled as distinct unrestricted networks, each represented as a directed graph $G = (V, A)$. Each vertex $v \in V$ represents an intersection and has associated coordinates (latitude and longitude). Each arc $(v, w) \in A$ represents a (directed) road segment and has an associated *duration* $\mathrm{dur}(v, w)$, which corresponds to the (constant) time to traverse it.

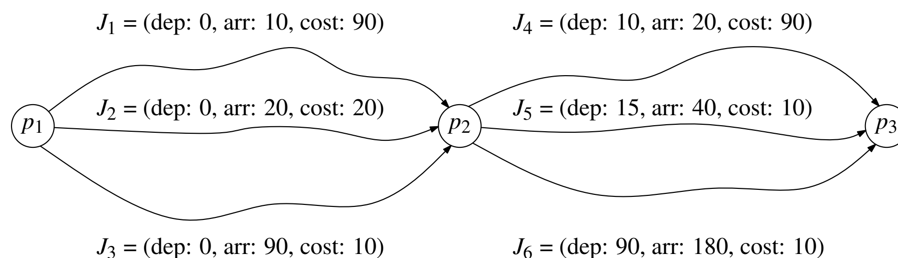


Figure 3: Problem of linear combination search in time-dependent multimodal networks.

The *integrated transportation network* is the union of these partial networks with appropriate *link vertices*, i. e., vertices (or stops) in different networks are identified with one another to allow for changes in modes of transportation. Note that, unlike previous work [13, 28, 32, 39, 77], we do not necessarily require explicit *footpaths* in the public transportation networks (to walk between nearby stops). For pure public transport optimization, adding these footpaths is often done by the operator of the network or by heuristics [28].

A query takes as input a *source location* s, a *target location* t, and a *departure time* τ, and it produces *journeys* that leave s no earlier than τ and arrive at t. A *journey* is a valid path in the integrated transportation network that obeys all timetable constraints.

Criteria. We still have to define *which* journeys the query should return. We argue that users optimize three natural criteria in multimodal networks: arrival time, costs, and "convenience". For our first (simplified) scenario (with public transit, cycling, and walking, but no taxi), we work with three criteria. Besides arrival time, we use number of trips and walking duration as proxies for convenience. We add cost for the scenario that includes taxi.

Given this setup, a first natural problem we need to solve is the *full multicriteria problem*, which must return a full (maximal) Pareto set of journeys. We say that a journey J_1 *dominates* J_2 if J_1 is strictly better than J_2 according to at least one criterion and no worse according to all other criteria. A *Pareto set* is a set of pairwise nondominating journeys [73, 56]. If two journeys have equal values in all criteria, we only keep one.

Shortcomings of Existing Approaches As discussed above, a different approach to multimodal journey planning is computing a weighted combination of all criteria under consideration and then running a single-criterion search. However, especially for time-dependent problems (such as ours), the weighted combination of travel time with other criteria may yield bad journeys. For example, preferring cheaper subpaths might make us miss the last bus home, forcing us to take an expensive taxi.

Figure 3 shows a more concrete example. The Pareto-optimal set from stop p_1 to p_2 when departing at time $\tau = 0$ contains three journeys. Concatenating $J_1 + J_4$ yields an arrival time of 20 and a cost of 180, $J_1 + J_5$ yields 40 and 100, and $J_3 + J_6$ yields 180 and 20. Note that J_2 is never used. Now, for the weighted linear optimization of $\alpha \cdot (\text{arr} - \text{dep}) + (1 - \alpha) \cdot \text{cost}$ (with $\alpha \in [0, 1]$), one might expect we can obtain these journeys for different values of α. However, for $\alpha \in [0, 0.125]$ we get $J_3 + J_6$, for $\alpha \in (0.125, 0.875]$ we get $J_2 + J_6$, and for $\alpha \in (0.875, 1]$ we obtain $J_1 + J_4$. So, we do not find $J_1 + J_5$, which provides a reasonable tradeoff between arrival time and costs. Even worse, for most values of α we get a journey that is not part of the Pareto set.

Fuzzy Dominance. Solving the full multicriteria problem, however, can lead to solution sets that are too large for most users. Moreover, many solutions provide undesirable tradeoffs, such as journeys that arrive much later to save a few seconds of walking (or walk much longer to save a few

seconds in arrival time). Intuitively, most criteria are diffuse to the user, and only large enough differences are significant. Pareto optimality fails to capture this.

To formalize the notion of significance, we propose to *score* the journeys in the Pareto set in a post-processing step using concepts from fuzzy logic [88] (and fuzzy set theory [87]). Loosely speaking, fuzzy logic generalizes Boolean logic to handle (continuous) degrees of truth. For example, the statement "60 and 61 seconds of walking are equal" is false in classical logic, but might be considered "almost true" in fuzzy logic. Formally, a *fuzzy set* is a tuple $\mathcal{S} = (\mathcal{U}, \mu)$, where $\mathcal{U}$ is a set and $\mu\colon \mathcal{U} \to [0,1]$ a *membership function* that defines "how much" each element in $\mathcal{U}$ is contained in $\mathcal{S}$. Mostly, we use μ to refer to $\mathcal{S}$. Our application requires fuzzy relational operators $\mu_<$, $\mu_=$, and $\mu_>$. For any $x, y \in \mathbb{R}$, they are evaluated by $\mu_<(x-y)$, $\mu_>(y-x)$, and $\mu_=(x-y)$. We use the well-known [88] exponential membership functions for the operators: $\mu_=(x) := \exp(\frac{\ln(\chi)}{\varepsilon^2}x^2)$, where $0 < \chi < 1$ and $\varepsilon > 0$ control the degree of fuzziness. The other two operators are derived by $\mu_<(x) := 1 - \mu_=(x)$ if $x < 0$ (0 otherwise) and $\mu_> := 1 - \mu_=(x)$ if $x > 0$ (0 otherwise). A *triangular norm* (short: *t-norm*) $T\colon [0,1]^2 \to [0,1]$ is a commutative, associative, and monotone (i. e., $a \le b, x \le y \Rightarrow T(a,x) \le T(b,y)$) binary operator to which 1 is the neutral element. If $x, y \in [0,1]$ are truth values, $T(x,y)$ is interpreted as a fuzzy conjunction (*and*) of x and y. Given a t-norm T, the *complementary conorm* (or *s-norm*) of T is defined as $S(x,y) := 1 - T(1-x, 1-y)$, which we interpret as a fuzzy disjunction (*or*). Note that the neutral element of S is 0. Two well-known pairs of t- and s-norms are $(\min(x,y), \max(x,y))$, called *minimum/maximum norms*, and $(xy, x+y-xy)$, called *product norm/probabilistic sum*.

We now recap the concept of fuzzy dominance in multicriteria optimization, which is introduced by Farina and Amato [41]. Given journeys J_1 and J_2 with M optimization criteria, we denote by $n_b(J_1, J_2)$ the (fuzzy) number of criteria in which J_1 is better than J_2. More formally, $n_b(J_1, J_2) := \sum_{i=1}^{M} \mu^i_<(\kappa^i(J_1), \kappa^i(J_2))$, where $\kappa^i(J)$ evaluates the i-th criterion of J and $\mu^i_<$ is the i-th fuzzy less-than operator. (Note that each criterion may use different fuzzy operators.) Analogously, we define $n_e(J_1, J_2)$ for equality and $n_w(J_1, J_2)$ for greater-than. By definition, $n_b + n_e + n_w = M$. Hence the Pareto dominance can be generalized to obtain a *degree of domination* $d(J_1, J_2) \in [0,1]$, defined as $(2n_b + n_e - M)/n_b$ if $n_b > (M - n_e)/2$ (and 0 otherwise). Here, $d(J_1, J_2) = 0$ means that J_1 does not dominate J_2, while a value of 1 indicates that J_1 Pareto-dominates J_2. Otherwise, we say J_1 *fuzzy-dominates* J_2 by degree $d(J_1, J_2)$. Figure 4 shows contour lines for values of d between 0 and 1 when using the maximum norm and two exemplary criteria: arrival time and walking duration (with fuzziness parameters set as in Section 5.1.4). In the figure we fix the criteria of J_1 to $(0,0)$. The area right-above each contour line t then contains all journeys J_2 (with respective values for their criteria) which are dominated by J_1 with degree at least t. For example, a journey is still dominated by J_1 with degree 0.4 if it has 10 minutes less walking while arriving 5 minutes later.

Now, given a (Pareto) set $\mathcal{J}$ of n journeys $J_1, \dots, J_n$, we define a *score function* $\mathrm{sc}\colon \mathcal{J} \to [0,1]$ that computes the degree of domination by the whole set for each J_i. More precisely, $\mathrm{sc}(J) := 1 - S(J_1, \dots, J_n)$. Note that if we set S to be the maximum norm, the score is based on the (one) journey that dominates J most. On the other hand, with the probabilistic sum the score may be based on several fuzzily dominating journeys.

We finally use the score to order the journeys by significance. One may then decide to only show the k journeys with highest score to the user.

Fuzzy Dominance Example Figure 5 shows a (quite representative) location-to-location query from William Road (near Warren Street Station) to Caxton Street (near Westminster Abbey) on our London instance using public transit, walking, and taxi with optimization criteria arrival time, number of transfers, walking duration, and cost (in pounds). The departure time is 4:27 pm. The left figure shows all nondominating journeys of the full Pareto set (there are 65 in total), while the right figure shows the three journeys with highest score from the (same) Pareto set, when our fuzzy dominance approach is used (cf. Section 5.1.1). This example clearly demonstrates that we obtain too many nondominating solutions (left figure), a known problem for multicriteria search. But not

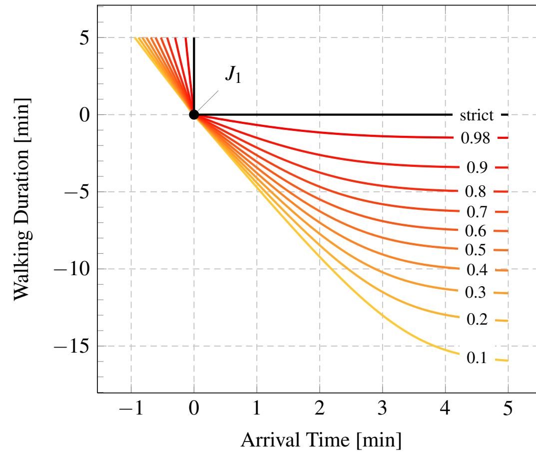


Figure 4: Contour lines of the fuzzy dominance function $d(J_1, J_2) = t$ for different values t and a fixed journey $J_1 = (0, 0)$ when considering two exemplary criteria: arrival time and walking duration. The thick black line marks the classical Pareto-dominance ($t = 1$).

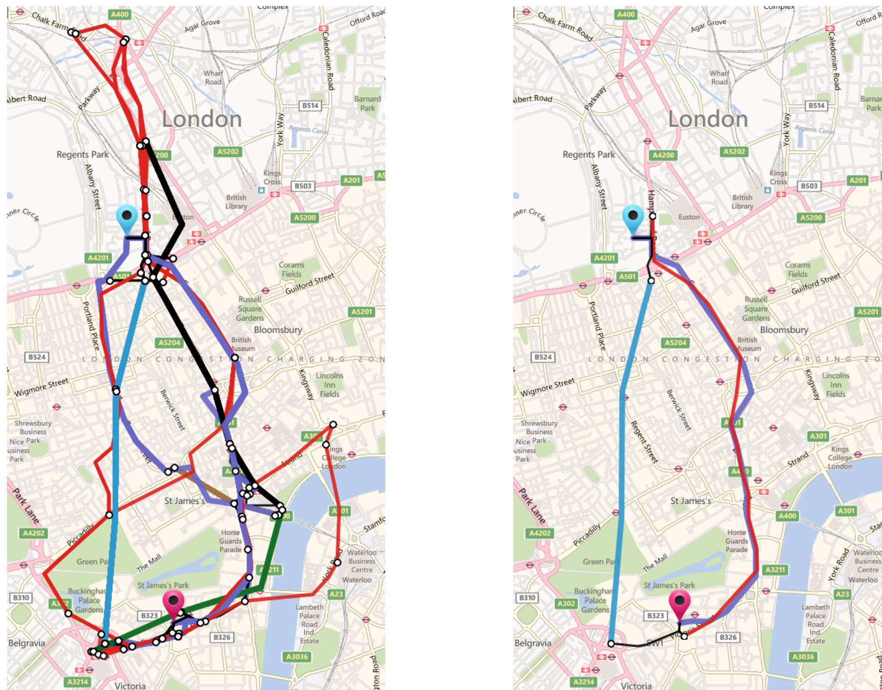


Figure 5: Exemplary multicriteria multimodal query on London with criteria arrival time, number of transfers, walking duration, and cost. The left figure shows the full Pareto set (65 journeys), while the right figure shows the three journeys with highest score (cf. Section 5.1.1). Each dot represents a transfer and included transportation modes are walking (thin black), taxi (thick purple), buses (thin red), and tube (other thick colors).

only is the number of solutions too high for presentation to a user, in fact, most of the journeys are not meaningful. Some of them take considerable detours (for example north of the source location), just to save some (insignificant) amount of walking. In contrast, our scoring approach by fuzzy domination (right figure) is able to identify the significant solutions in the Pareto set, resulting in three meaningful journeys: One taking taxi the full way (purple), one taking the subway (blue) which is faster at the cost of more walking (black), and one taking the bus (red) which takes longer but with significantly less total walking (4 min instead of 14 min).

5.1.2 Exact Algorithms

This section considers exact algorithms for the multicriteria multimodal problem. Sections 5.1.2 and 5.1.2 propose two solutions, each building on a different algorithm for multicriteria optimization on public transportation networks (MLC [77] and RAPTOR [32]). Section 5.1.2 then describes an acceleration technique that applies to both. To simplify the discussion (and notation), we first describe the algorithms in terms of our simplest scenario, considering only the (timetable-based) public transit network and the (unrestricted) walking network. Section 5.1.2 explains how to handle cycling and taxis, which are unrestricted but have special properties.

Multi-label-correcting Algorithm Traditional solutions to the multicriteria problem on public transportation networks typically model the timetable as a graph [18, 28, 44, 72]. A particularly effective approach is to use the *time-dependent route model* [72]. For each stop p, we create a single *stop vertex* linked by time-independent *transfer edges* to multiple *route vertices*, one for each route serving p. We also add *route edges* between route vertices associated to consecutive stops within the same route. To model the trips along a route, the cost of a route edge is given by a piecewise linear function reflecting the traversal time (including waiting for the next departure).

A journey in the public transportation network corresponds to a path in this graph. The *multi-label-correcting* (MLC) [72] algorithm uses this to find full Pareto sets for arbitrary criteria that can be modeled as edge costs. MLC extends Dijkstra's algorithm [38] by operating on labels that have multiple values, one per criterion. Each vertex v maintains a *bag* $B(v)$ of nondominated labels. In each iteration, MLC extracts from a priority queue the minimum (in lexicographic order) unprocessed label $L(u)$. For each arc (u, v) out of the associated vertex u, MLC creates a new label $L(v)$ (by extending $L(u)$ in the natural way) and inserts it into $B(v)$; newly-dominated labels (possibly including $L(v)$ itself) are discarded, and the priority queue is updated if needed. MLC can be sped up with target pruning and by avoiding unnecessary domination checks [39].

To solve the multimodal problem, we extend MLC: It suffices to augment its input graph to include the walking network. We combine the original graphs by merging (public transportation) stops and (walking) intersections that share the same location (and keeping all edges). These *link vertices* are then used to switch between modes of transportation. The MLC query remains essentially unchanged, and still processes labels in lexicographic order. Although labels can now be associated to vertices in different networks, they can all share the same priority queue.

Round-based Algorithm A drawback of MLC (even restricted to public transportation networks) is that it can be quite slow: Unlike Dijkstra's algorithm, MLC may scan the same vertex multiple times (the exact number depends on the criteria being optimized), and domination checks make each such scan quite costly. Delling et al. [32] have recently introduced RAPTOR (*Round bAsed Public Transit Optimized Router*) as a faster alternative. The simplest version of the algorithm optimizes two criteria: arrival time and number of transfers. Unlike MLC, which searches a graph, RAPTOR uses dynamic programming to operate directly on the timetable. It works in rounds, with round i processing all relevant journeys with exactly $i - 1$ transfers. It maintains one label per round i and stop p representing the best known arrival time at p for up to i trips. During round i, the algorithm processes each *route* once. It reads arrival times from round $i - 1$ to determine relevant

trips (on the route) and updates the labels of round i at every stop along the way. Once all routes are processed, the algorithm considers potential transfers to nearby (predefined) stops in a second phase. Simpler data structures and better locality make RAPTOR an order of magnitude faster than MLC. Delling et al. [32] have also proposed McRAPTOR, which extends RAPTOR to handle more criteria (besides arrival times and number of transfers). It maintains a *bag* (set) of labels with each stop and round.

Even with multiple modes of transport available, one trip always consists of a single mode. This motivates adapting the round-based paradigm to our scenario. We propose MCR (*multimodal multi-criteria RAPTOR*), which extends McRAPTOR to handle multimodal queries. As in McRAPTOR, each round has two phases: the first processes trips in the public transportation network, while the second considers arbitrary paths in the unrestricted networks. We use a standard McRAPTOR round for the first phase (on the timetable network) and MLC for the second (on the walking network). Labels generated by one phase are naturally used as input to the other. During the second phase, MLC extends bags instead of individual labels. To ensure that each label is processed at most once, we keep track of which labels (in a bag) have already been extended. The initialization routine (before the first round) runs Dijkstra's algorithm on the walking network from the source s to determine the fastest walking path to each stop in the public transportation network (and to t), thus creating the initial labels used by MCR. During round i, the McRAPTOR subroutine reads labels from round $i-1$ and writes to round i. In contrast, the MLC subroutine may read and write labels of the same round if walking is not regarded as a trip.

Contracting the Unrestricted Networks As our experiments will show, the bottleneck of the multimodal algorithms is processing the walking network $G = (V, A)$. We improve performance using a quick preprocessing technique [37]. For any journey involving public transportation, walking between trips always begins and ends at the restricted set $K \subset V$ of link vertices. During queries, we must only be able to compute the pairwise distances between these vertices. We therefore use preprocessing to compute a smaller *core graph* [81] that preserves these distances. More precisely, we start from the original graph and iteratively *contract* [47] each vertex in $V \setminus K$ in the order given by a rank function r. Each contraction step (temporarily) removes a vertex and adds shortcuts between its uncontracted neighbors to maintain shortest path distances (if necessary). It is usually advantageous to first contract vertices with relatively small degrees that are evenly distributed across the network [47]. We stop contraction when the average degree in the core graph reaches some threshold (we use 12 in our experiments) [37].

To run a faster multimodal s–t query, we use essentially the same algorithm as before (based on either MLC or RAPTOR), but replacing the full walking network with the (smaller) core graph. Since the source s and the target t may not be in the core, we handle them during initialization. It works on the graph $G^+ = (V, A \cup A^+)$ containing all original arcs A as well as all shortcuts A^+ added during the contraction process. We run upward searches (only following arcs (u, v) such that $r(u) > r(w)$) in G^+ from s (scanning forward arcs) and t (scanning reverse arcs); they reach all potential entry and exit points of the core, but arcs within the core are not processed [37]. These core vertices (and their respective distances) are used as input to MCR's (or MLC's) standard initialization, which can operate on the core from this point on.

The main loop works as before, with one minor adjustment. Whenever MLC extracts a label $L(v)$ for a scanned core vertex v, we check if it has been reached by the reverse search during initialization. If so, we create a temporary label $L'(t)$ by extending $L(v)$ with the (already computed) walking path to t and add it to $B(t)$ if needed. MCR is adjusted similarly, with bags instead of labels.

Beyond Walking We now consider other unrestricted networks (besides walking). In particular, our experiments include a bicycle rental scheme, which can be seen as a hybrid network: It does not have a fixed schedule (and is thus unrestricted), but bicycles can only be picked up and dropped off at designated *cycling stations*. Picking a bike from its station counts as a trip. To handle cycling

within MCR, we consider it during the first stage of each round (together with RAPTOR and before walking). Because bicycles have no schedule, we process them independently (from RAPTOR) by running MLC on the bicycle network. To do so, we initialize MLC with labels from round $i-1$ for all relevant bicycle stations and, during the algorithm, we update labels of (the current) round i.

We consider a taxi ride to be a trip as well, since we board a vehicle. Moreover, we also optimize a separate criterion reflecting the (monetary) *cost* of taxi rides. If taxis were not penalized in any way, an all-taxi journey would almost always dominate all other alternatives (even sensible ones), since it is fast and has no walking. Our round-based algorithms handle taxis as they do walking, except that in the taxi stage labels are read from round $i-1$ and written into round i. Note that we link the taxi network to public transit stops as well as bicycle stations and that—unlike with rental bicycles—we also allow taking a taxi as the first and/or last leg of any location-to-location query. Dealing with personal cars or bicycles is simpler. Assuming that they are only available for the first or last legs of the journey, we must only consider them during initialization. Initialization can also handle other special cases, such as allowing rented bicycles to be ridden to the destination (to be returned later).

Note that contraction can be used for cycling and driving. For every unrestricted network (walking, cycling, driving), we keep the link vertices (stops and bicycle stations) in one common core and contract (up to) all other nodes. As before, queries start with upward searches in each relevant unrestricted network.

5.1.3 Heuristics

Even with all accelerations, the exact algorithms proposed in Section 5.1.2 are not fast enough for interactive applications. This section proposes quick heuristics aimed at finding a set of journeys that is similar to the exact solution, which we take as ground truth. We consider three approaches: weakening the dominance rules, restricting the amount of walking, and reducing the number of criteria. We also discuss how to measure the quality of the heuristic solutions we find.

Weak Dominance. The first strategy we consider is to weaken the domination rules during the algorithm, reducing the number of labels pushed through the network. We test four implementations of this strategy. The first, MCR-hf, uses fuzzy dominance (instead of strict dominance) when comparing labels during the algorithm: For labels L_1 and L_2, we compute the fuzzy dominance value $d(L_1, L_2)$ (cf. Section 5.1.1) and dominate L_2 if d exceeds a given threshold (we use 0.9). The second, MCR-hb(κ), uses strict dominance, but discretizes criterion κ: before comparing labels L_1 and L_2, we first round $\kappa(L_1)$ and $\kappa(L_2)$ to predefined discrete values (*buckets*); this can be extended to use buckets for several criteria. The third heuristic, MCR-hs(κ), uses strict dominance but adds a slack of x units to κ. More precisely, L_1 already dominates L_2 if $\kappa(L_1) \leq \kappa(L_2) + x$ and L_1 is at least as good L_2 in all other criteria. The last heuristic, MCR-ht, weakens the domination rule by trading off two or more criteria. More concretely, consider the case in which walking (walk) and arrival time (arr) are criteria. Then, L_1 already dominates L_2 if $\text{arr}(L_1) \leq \text{arr}(L_2) + a \cdot (\text{walk}(L_1) - \text{walk}(L_2))$, $\text{walk}(L_1) \leq \text{walk}(L_2) + a \cdot (\text{arr}(L_1) - \text{arr}(L_2))$, and L_1 is at least as good as L_2 in all other criteria, for a tradeoff parameter a (we use $a = 0.3$).

Restricting Walking. Consider our simple scenario of walking and public transit. Intuitively, most journeys start with a walk to a nearby stop, followed by one or more trips (with short transfers) within the public transit system, and finally a short walk from the final stop to the actual destination. This motivates a second class of heuristics, MCR-tx. It still runs three-criterion search (walking, arrival, and trips), but limits walking transfers between stops to x minutes; in this case we precompute these transfers. MCR-tx-ry also limits walking in the beginning and end to y minutes. Note that existing solutions often use such restrictions [13].

Fewer Criteria. The last strategy we study is reducing the number of criteria considered during the algorithm. As already mentioned, this is a common approach in practice. We propose MR-x, which still works in rounds, but optimizes only the number of trips and arrival times explicitly (as criteria). To account for walking duration, we count every x minutes of a walking segment (transfer) as a trip; the first x minutes are free. With this approach, we can run plain Dijkstra to compute transfers, since link vertices no longer need to keep bags. The round index to which labels are written then depends on the walking duration (of the current segment) of the considered label. A special case is $x = \infty$, where a transfer is never a trip. Another variant is to always count a transfer as a single trip, regardless of duration; we abuse notation and call this variant MR-0. We also consider MR-∞-tx: Walking duration is not an explicit criterion and transfers do not count as trips, but are limited to x minutes.

For scenarios that include cost as a criterion (for taxis), we consider variants of the MCR-hb and MCR-hf heuristics. In both cases, we drop walking as an independent criterion, leaving only arrival time, number of trips, and costs to optimize. We account for walking by making it a (cheap) component of the costs.

Quality Evaluation To measure the quality of a heuristic, we compare the set of journeys it produces to the *ground truth*, which we define as the solution found by MCR. To do so, we first compute the score of each journey with respect to the Pareto set that contains it (cf. Section 5.1.1). Then, for a given parameter k, we measure the similarity between the top k scored journeys returned by the heuristics and the top k scored journeys in the ground truth. Note that the score depends only on the algorithm itself and does not assume knowledge of the ground truth, which is consistent with a real-world deployment.

To compare two sets of k journeys, we run a greedy maximum matching algorithm. First, we compute a $k \times k$ matrix where entry (i, j) represents the similarity between the i-th journey in the first set and the j-th in the second. To measure the similarity, we make use of the same fuzzy relational operators we use for scoring. More precisely, given two journeys J_1 and J_2, the similarity with respect to the i-th criterion is given by $c^i := \mu^i_=(\kappa^i(J_1) - \kappa^i(J_2))$, where κ^i is the value of this criterion and $\mu^i_=$ is the corresponding fuzzy equality relation. Then, we define the similarity between J_1 and J_2 as $T(c^1, c^2, \dots, c^M)$, where T is an arbitrary t-norm. We always select T to be consistent with the s-norm that we use to compute the score values.

After computing the pairwise similarities, we greedily select the unmatched pairs with highest similarity (by picking the highest entry in the matrix that does not share a row or column with a previously picked entry). The similarity of the whole matching is the average similarity of its pairs, weighted by the fuzzy score of the reference journey. This means that matching the highest-scored reference journey is more important than matching the k-th one.

5.1.4 Experiments

This section presents an extensive evaluation of the methods introduced in this paper. All algorithms from Sections 5.1.2 and 5.1.3 were implemented in C++ and compiled with `g++` 4.6.2 (64 bits, flag `-O3`). We ran our experiments on one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM.

Input and Methodology. We focus on the transportation network of London (England); results for other instances (available in Section 5.1.4) are similar. We use the timetable information made available by Transport for London (TfL) [63, 83], from which we extracted a Tuesday in the periodic summer schedule of 2011. The data includes subway (tube), buses, tram, ferries, and light rail (DLR), as well as bicycle station locations. To model the underlying road network, we use data provided by PTV AG [76] from 2006, which explicitly indicates whether each road segment is open for driving, cycling and/or walking. We set the walking speed to 5 km/h and the cycling speed to 12 km/h,

Table 9: Performance and solution quality on journeys considering walking, cycling, and public transit. Bullets (•) indicate the criteria taken into account by the algorithm.

Algorithm	Arr.	Trp.	Wlk.	Rnd.	Scans / Ent.	Comp. / Ent.	Jn.	Time [ms]	Quality-3 Avg.	Quality-3 Sd.	Quality-6 Avg.	Quality-6 Sd.
MCR-full	•	•	•	13.8	13.8	168.2	29.1	4 634.0	100 %	0 %	100 %	0 %
MCR	•	•	•	13.8	3.4	158.7	29.1	1 438.7	100 %	0 %	100 %	0 %
MLC	•	•	•	—	10.6	1 246.7	29.1	4 543.0	100 %	0 %	100 %	0 %
MCR-hf	•	•	•	15.6	2.9	14.3	10.9	699.4	89 %	15 %	89 %	11 %
MCR-hb	•	•	•	10.2	2.1	12.7	9.0	456.7	91 %	12 %	91 %	10 %
MCR-hs	•	•	•	14.7	2.6	11.1	8.6	466.1	67 %	28 %	69 %	23 %
MCR-ht	•	•	•	10.5	2.0	6.4	8.6	373.6	84 %	22 %	82 %	20 %
MCR-t10	•	•	•	13.8	2.7	132.7	29.0	1 467.6	97 %	10 %	95 %	10 %
MCR-t10-r15	•	•	•	10.7	1.7	73.3	13.2	885.0	38 %	40 %	30 %	31 %
MCR-t5	•	•	•	13.8	2.7	126.6	28.9	891.9	93 %	16 %	92 %	15 %
MR-∞	•	•	○	7.6	1.4	4.8	4.5	44.4	63 %	28 %	63 %	24 %
MR-0	•	•	○	13.7	2.1	6.9	5.4	61.5	63 %	28 %	63 %	24 %
MR-10	•	•	○	20.0	1.1	4.8	4.3	39.4	51 %	33 %	45 %	29 %
MR-∞-t10	•	•	○	7.6	1.1	4.8	4.5	22.2	63 %	28 %	62 %	24 %

and we assume driving at free-flow speeds. We do not consider turn costs, which are not defined in the data. The resulting combined network has 564 cycle stations and about 20 k stops, 5 M departure events, and 259 k vertices in the walking network. Exact numbers are given in Table 12 of Section 5.1.4.

Recall that we specify the fuzziness of each criterion by a pair (χ, ε), roughly meaning that the corresponding Gaussian (centered at $x = 0$) has value χ for $x = \varepsilon$. We set these pairs to $(0.8, 5)$ for walking, $(0.8, 1)$ for arrival time, $(0.1, 1)$ for trips, and $(0.8, 5)$ for costs (given in pounds; times are in minutes). Note that the number of trips is sharper than the other criteria. Later in this section we show that our approach is robust to small variations in these parameters, but they can be tuned to account for user-dependent preferences. If not indicated otherwise, our experiments consider the minimum/maximum norms by default. We run *location-to-location* queries, with sources, targets, and departure times picked uniformly at random (from the walking network and during the day, respectively).

Algorithms Evaluation. For our first experiment, we use walking, cycling, and the public transportation network and consider three criteria: arrival time, number of trips, and walking duration. We ran 1 000 queries for each algorithm. Table 9 summarizes the results (Section 5.1.4 has additional statistics). For each algorithm, the table first shows which criteria are explicitly taken into account. The next five columns show the average values observed for the number of rounds, scans per entity (stop/vertex), label comparisons per entity, journeys found, and running time (in milliseconds). The last four columns evaluate the quality of the top 3 and 6 journeys found by our heuristics, as explained in Section 5.1.3. Note that we show both averages and standard deviations.

The methods in Table 9 are grouped in blocks. Those in the first block compute the full Pareto set considering all three criteria (arrival time, number of trips, and walking). MCR, our reference algorithm, is round-based and uses contraction in the unrestricted networks. As anticipated, it is faster (by a factor of about three) than MCR-full (which does not use the core) and MLC (which uses the core but is not round-based). Accordingly, all heuristics we test are round-based and use the core.

The second block contains heuristics that accelerate MCR by weakening the domination rules,

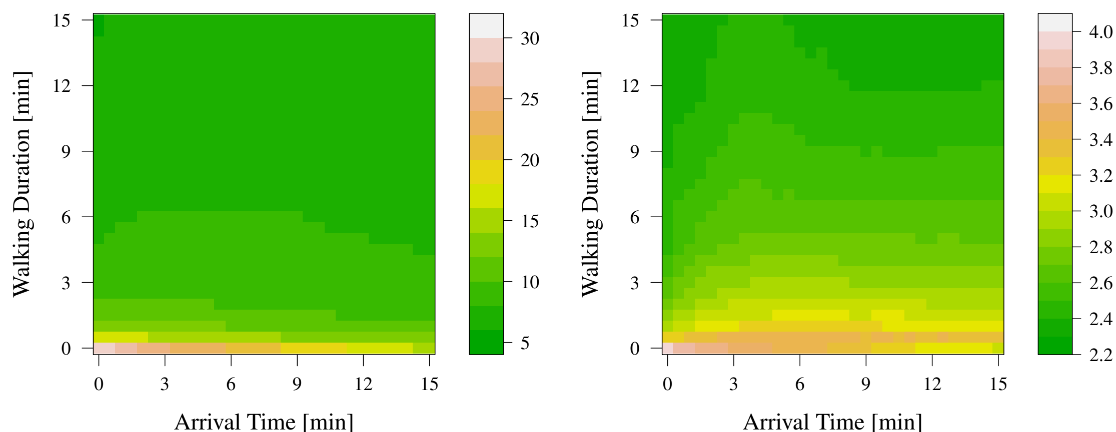


Figure 6: Number of Pareto optimal journeys with score higher than 0.1 for varying fuzziness. We consider both the maximum norm (left) and probabilistic sum (right). The x axis varies the fuzziness in the arrival time, while the y axis considers the walking duration. The intensity (color) of the corresponding entry indicates the average number of journeys in the filtered output.

causing more labels to be pruned (and losing optimality guarantees). As explained in Section 5.1.3, MCR-hf uses fuzzy dominance during the algorithm, MCR-hb uses walking *buckets* (discretizing walking by steps of 5 minutes for domination), MCR-hs uses a slack of 5 minutes on the walking criterion when evaluating domination, and MCR-ht considers a tradeoff parameter of $a = 0.3$ between walking and arrival time. All heuristics are faster than pure MCR, and MCR-hb gives the best quality at a reasonable running time.

The third block has algorithms with restrictions on walking duration. Limiting transfers to 10 minutes (as MCR-t10 does) has almost no effect on solution quality (which is expected in a well-designed public transportation network). Moreover, adding precomputed footpaths of 10 minutes is not faster than using the core for unlimited walking (as MCR does). Additionally limiting the walking range from s or t (MCR-t10-r15) improves speed, but the quality becomes unacceptably low: The algorithm misses good journeys (including all-walk) quite often. If instead we allow even more restricted transfers (with MCR-t5), we get a similar speedup with much better quality (comparable to MCR-hb).

The MR-x algorithms (fourth block) reduce the number of criteria considered by combining trips and walking. The fastest variant is MR-∞-t10, which drops walking duration as a criterion but limits the amount of walking at transfers to 10 minutes, making it essentially the same as RAPTOR, with a different initialization. As expected, however, quality is much lower than for MCR-tx, confirming that considering the walking duration explicitly during the algorithm is important to obtain a full range of solutions. MR-10 attempts to improve quality by transforming long walks into extra trips, but is not particularly successful.

Summing up, MCR-hb should be the preferred choice for high-quality solutions, while MR-∞-t10 can support interactive queries with reasonable quality.

Fuzzy Parameters Evaluation. We also evaluated the impact of the fuzzy parameters on the number of journeys we obtain. We again use London with walking, public transit, and cycling as input. Figure 6 shows the number of journeys given a score higher than 0.1 (by the fuzzy ranking routine) when we vary ε (the level of fuzziness) for two criteria, walking and arrival time. We set $\chi = 0.8$, as in our main experiments. To not overload the figure, we keep the fuzziness of the

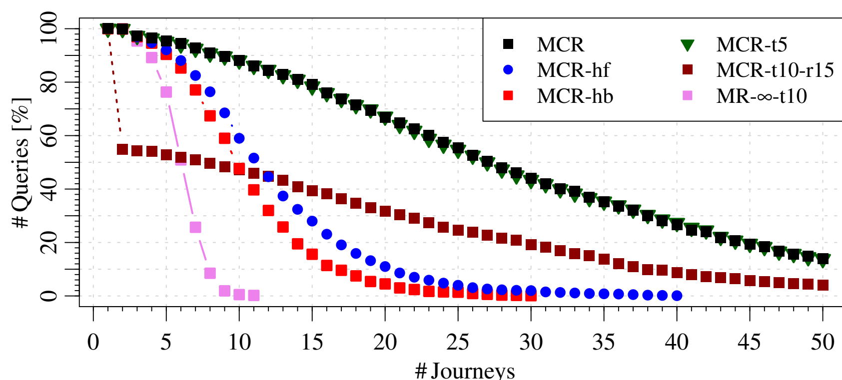


Figure 7: Evaluating the number of journeys returned by some of our algorithms: For a given n (on the abscissa), we report the percentage of 1 000 random queries that compute n or more journeys.

third criterion (number of trips) constant.

A comparison between the plots shows that, for the same set of parameters, probabilistic sum is significantly stricter than the maximum norm, and reduces the number of journeys much more drastically (for a fixed threshold). Qualitatively, however, they behave similarly. Under both norms, making the walking criterion fuzzier is more effective at identifying unwanted journeys. A couple of minutes of fuzziness in the walking criterion is enough to significantly reduce the number of journeys above the threshold. Adding fuzziness only to the arrival time has much more limited effect on the results.

Quality of the Heuristics. We here further investigate the quality of our heuristics. We use London with walking, public transit, and cycling as input. Figure 7 reports the size of the Pareto set (the input to scoring) for various algorithms, while Figure 8 shows how well the the top k heuristic journeys match the ground truth, for varying k. We observe that exact MCR (even if restricted to 5-minute transfers) does indeed produce many journeys, supporting the notion of ranking them afterwards (by score). A good heuristic, such as MCR-hb, computes much fewer journeys, but they match the top MCR journeys quite well. An interesting observation is that the quality of the heuristic hardly depends on the number of journeys we try to match.

Full Multimodal Problem. Our final experiment considers the full multimodal problem, including taxis. We add *cost* as fourth criterion (at 2.40 pounds per taxi trip plus 60 pence per minute). We do not consider the cost of public transit, since it is significantly cheaper. Table 10 presents the average performance of some of our algorithms over 1 000 random queries in London. The first block includes algorithms that optimize all four criteria (arrival time, walking duration, number of trips, and costs). While exact MCR is impractical, fuzzy domination (MCR-hf) makes the problem tractable with little loss in quality. Using 5-minute buckets for walking and 5-pound buckets for costs (MCR-hb) is even faster, though queries still take more than two seconds. The second block shows that we can reduce running times by dropping walking duration as a criterion (we incorporate it into the cost function at 3 pence per minute, instead), with almost no loss in solution quality. This is still not fast enough, though. Using 5-pound buckets (MCR-hb) reduces the average query time to about 1 second, with reasonable quality.

Detailed Performance Table 11 presents a more detailed analysis of the main experiment in Section 5.1.4 (without taxis). For each algorithm, it shows the effort (number of scans per vertex and/or stop, as well as running times in milliseconds) spent in each of the networks (public transit,

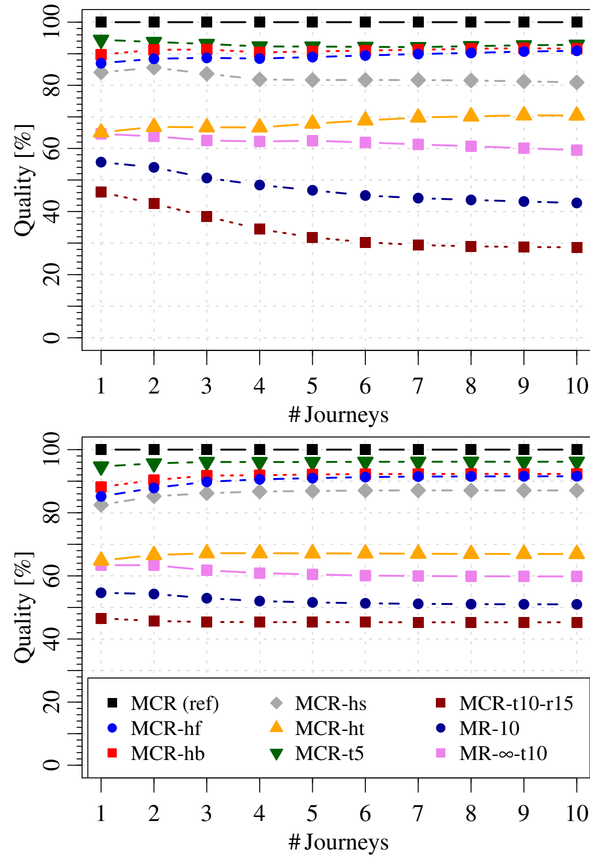


Figure 8: Evaluating the solution quality by matching the top k journeys in the solution with the top k of the reference algorithm (MCR). The scores and similarity values are obtained by using the minimum/maximum norms (left) and the product norm/probabilistic sum (right). The legend of the right plot also applies to the left.

Table 10: Performance on our London instance when taking taxi into account.

Algorithm	Arr.	Trp.	Wlk.	Cost	Rnd.	Scans / Ent.	Comp. / Ent.	Jn.	Time [ms]	Quality-3 Avg.	Quality-3 Sd.	Quality-6 Avg.	Quality-6 Sd.
MCR	•	•	•	•	16.3	3.1	369 606.0	1 666.0	1 960 234.0	100 %	0 %	100 %	0 %
MCR-hf	•	•	•	•	17.1	2.1	137.1	35.2	6 451.6	92 %	12 %	92 %	6 %
MCR-hb	•	•	•	•	9.9	1.3	86.8	27.6	2 807.7	96 %	8 %	92 %	6 %
MCR	•	•	○	•	14.6	2.4	7 901.4	250.9	25 945.8	98 %	6 %	97 %	5 %
MCR-hf	•	•	○	•	12.0	1.4	33.6	17.6	2 246.3	87 %	12 %	74 %	12 %
MCR-hb	•	•	○	•	9.0	1.0	20.0	11.6	996.4	86 %	12 %	74 %	12 %

Table 11: Detailed performance analysis of our algorithms. The total running time includes additional overhead, such as for initialization.

Algorithm	Arr.	Trp.	Wlk.	Public Transit Scans / Stop	Public Transit Time [ms]	Walking Scans / Vert.	Walking Time [ms]	Cycling Scans / Vert.	Cycling Time [ms]	Total Scans / Ent.	Total Time [ms]
MCR-full	●	●	●	32.1	350.6	9.6	3 030.9	43.6	1 203.1	13.8	4 634.0
MCR	●	●	●	32.1	341.4	1.2	889.3	1.7	159.2	3.4	1 438.7
MLC	●	●	●	119.3	—	2.6	—	2.1	—	10.6	4 543.0
MCR-hf	●	●	●	28.1	157.7	1.0	483.9	0.7	25.6	2.9	699.4
MCR-hb	●	●	●	21.1	115.2	0.7	297.4	0.5	19.7	2.1	456.7
MCR-hs	●	●	●	25.1	97.3	0.9	322.2	0.6	16.8	2.6	466.1
MCR-ht	●	●	●	20.2	86.8	0.7	246.4	0.5	17.4	2.0	373.6
MCR-t5	●	●	●	31.5	318.4	0.5	348.6	1.7	157.2	2.7	891.9
MCR-t10	●	●	●	31.6	326.2	0.5	913.7	1.7	158.5	2.7	1 467.6
MCR-t10-r15	●	●	●	20.0	207.5	0.3	554.0	1.2	103.6	1.7	885.0
MR-∞	●	●	○	14.2	10.0	0.5	31.0	0.3	1.8	1.4	44.4
MR-0	●	●	○	21.4	13.9	0.7	42.5	0.4	2.4	2.1	61.5
MR-10	●	●	○	9.7	6.3	0.5	30.5	0.2	1.3	1.1	39.4
MR-∞-t10	●	●	○	14.4	9.4	0.2	9.5	0.3	1.6	1.2	22.2

walking, and cycling) and in total. The table shows that all round-based algorithms except MR-∞-t10 spend significantly more time processing the unrestricted networks (walking and cycling) than dealing with public transportation. This was to be expected: not only are the unrestricted networks bigger (they have more vertices), but also they must be processed with a (slower) Dijkstra-based algorithm (as in MLC, rather than RAPTOR). This is the reason for the good performance of the MR-∞-t10 heuristic.

Additional Inputs In addition to London, we tested inputs representing other large metropolitan areas (New York, Los Angeles, and Chicago). We built the public transit network from publicly available General Transit Feeds (GTFS) [49], restricting ourselves to the timetable for August 10, 2011 (a Wednesday). The walking network data is still given by PTV [76], and these instances do not include bicycles. Detailed statistics for all instances are presented in Table 12.

Table 13 compares the performance of our algorithms on these inputs. For reference, we also consider a simplified version of the London network, without bicycles. For each input, we show the average values (over 1 000 queries) for number of journeys found, running time, and quality (considering the top 6 journeys). The results are consistent with those obtained for the full London network, showing that our preferred choice of heuristics also holds here. MCR-hb is always the best choice in terms of solution quality (among methods with reasonable speedups), while MR-∞-t10 is preferred if query times should be as low as possible.

5.1.5 Final Remarks

We have studied multicriteria journey planning in multimodal networks. We argued that users optimize three criteria: arrival time, costs, and convenience. Although the corresponding full Pareto set is large and has many unnatural journeys, fuzzy set theory can extract the relevant journeys and rank them. Since exact algorithms are too slow, we have introduced several heuristics that closely match the best journeys in the Pareto set. Our experiments show that our approach enables efficient realistic multimodal journey planning in large metropolitan areas. A natural avenue for

Table 12: Size figures for our input instances. We link every stop and cycle station with the walking/taxi network.

Figure	London	New York	Los Angeles	Chicago
Public Transit				
Stops	20 843	17 894	15 003	12 137
Routes	2 184	1 393	1 099	710
Trips	133 011	45 299	16 376	20 303
Daily Departure Events	4 991 125	1 825 129	931 846	1 194 571
Vertices (Route Model)	99 230	66 124	81 657	47 561
Edges (Route Model)	260 583	193 159	214 369	118 452
Walking				
Vertices	258 840	255 808	224 053	70 440
Vertices in Core	27 840	25 808	21 053	16 440
Edges	1 433 814	1 586 782	1 395 185	586 979
Footpaths $\leq$ 5 min	150 948	219 040	83 844	122 450
Footpaths $\leq$ 10 min	518 174	670 702	271 444	426 818
Cycling				
Cycle Stations	564	—	—	—
Vertices	23 311	—	—	—
Vertices in Core	1 311	—	—	—
Edges	130 971	—	—	—
Taxi				
Vertices	259 122	—	—	—
Vertices in Core	27 122	—	—	—
Edges	1 339 487	—	—	—

Table 13: Evaluating the performance of MCR and MR with different heuristics on other instances. The quality is determined identically to Table 9 (cf. Section 5.1.4).

				New York			**Los Angeles**			**Chicago**		
Algorithm	Arr.	Trp.	Wlk.	Jn.	Time [ms]	Qual. Avg.	Jn.	Time [ms]	Qual. Avg.	Jn.	Time [ms]	Qual. Avg.
MCR	•	•	•	25.5	1 703.0	100 %	16.7	644.6	100 %	22.1	532.8	100 %
MCR-hf	•	•	•	8.6	611.0	91 %	8.9	445.0	88 %	8.3	241.3	72 %
MCR-hb	•	•	•	7.2	413.8	94 %	7.6	295.8	93 %	7.1	160.8	92 %
MCR-hs	•	•	•	6.7	414.0	84 %	7.4	310.7	62 %	6.6	158.8	58 %
MCR-ht	•	•	•	6.6	300.9	80 %	6.7	228.4	69 %	6.2	113.9	79 %
MCR-t5	•	•	•	25.6	695.5	69 %	16.6	262.7	93 %	21.9	277.7	95 %
MCR-t10	•	•	•	25.3	1 401.4	85 %	16.8	424.5	96 %	22.0	578.8	98 %
MCR-t10-r15	•	•	•	5.4	677.9	10 %	3.9	202.0	13 %	9.6	372.7	28 %
MR-∞	•	•	○	3.4	26.3	65 %	3.6	21.5	51 %	3.3	12.3	63 %
MR-0	•	•	○	3.8	37.6	65 %	4.3	28.5	52 %	3.7	15.6	63 %
MR-10	•	•	○	6.0	26.1	41 %	6.1	26.6	42 %	5.1	13.9	50 %
MR-∞-t10	•	•	○	3.6	10.6	60 %	3.6	11.0	51 %	3.3	7.1	63 %

future research is accelerating our approach further to enable interactive queries with an even richer set of criteria in dynamic scenarios, handling delay and traffic information. The ultimate goal is to compute multicriteria multimodal journeys on a global scale in real time.

5.2 Multiobjective Route Planning

In route planning the goal is to find a shortest path between a source node s and a target node t. In a weighted directed graph typically a single weight or cost is used. Sometimes however, it is not enough to find the shortest path in terms of a single criterion (e.g distance), but also with regard to other criteria, such as travel time or energy cost. As a result, the weight function depends on multiple criteria. For example less travel time can mean more energy cost, but aligned with the objectives of eCOMPASS we are interested in computing environmentally friendly routes.

A core routine for multimodal and multiobjective route planning is to compute multiobjective shortest paths. This particular problem appears in applications such as QoS routing in communication networks, transport optimization and route planning. Even though numerous efficient algorithms exist for the single criterion shortest path problem, the multicriteria part of the problem is much harder. In fact, it is NP-complete. In order to deal with multimodal route planning problems, motivated by a great demand in practical applications to achieve efficiency and optimality, the NAMOA* algorithm was introduced in [66]. We are able to propose a new implementation of NAMOA*, enhanced with further heuristic optimizations on a new graph structure, Packed-Memory Graph (PMG) [65], that is especially suited for large-scale networks.

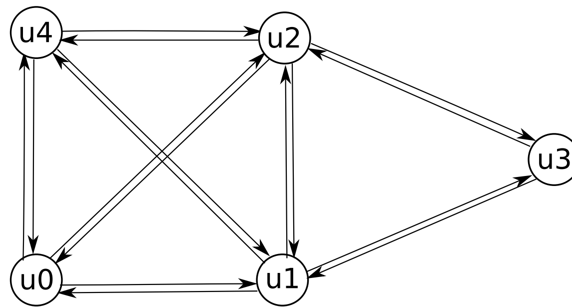


Figure 9: An example graph.

5.2.1 Graph structure

Packed-Memory Graph. This is a highly optimized graph structure which provides dynamic memory management of the graph and provides the user the ability to control the storing scheme of nodes and edges in memory for optimization purposes. It supports optimal scanning of consecutive nodes and edges and can incorporate dynamic changes in the graph layout in a matter of μs.

The PMG structure consists of an array for storing the nodes, in an arbitrary order, and two arrays for storing the edges, one considering the edges as outgoing from the nodes and one considering them as incoming to the nodes. The storing order of the edges follows the order of their base node in the node array. All three arrays reserve more memory than they need for their elements. The extra memory cells are evenly distributed throughout the arrays forming holes in them, which can then be used to efficiently add new elements. An illustration of the PMG structure, for the example graph of Figure 9, is shown in Figure 10.

The PMG structure includes the following operations. First, it provides *internal node reordering*. This means that nodes are stored in consecutive memory addresses, when they are given in an arbitrary order. As a result the edge arrays have to be reordered as well. This order can be changed at any time in an online manner. This function helps to increase the locality of references and

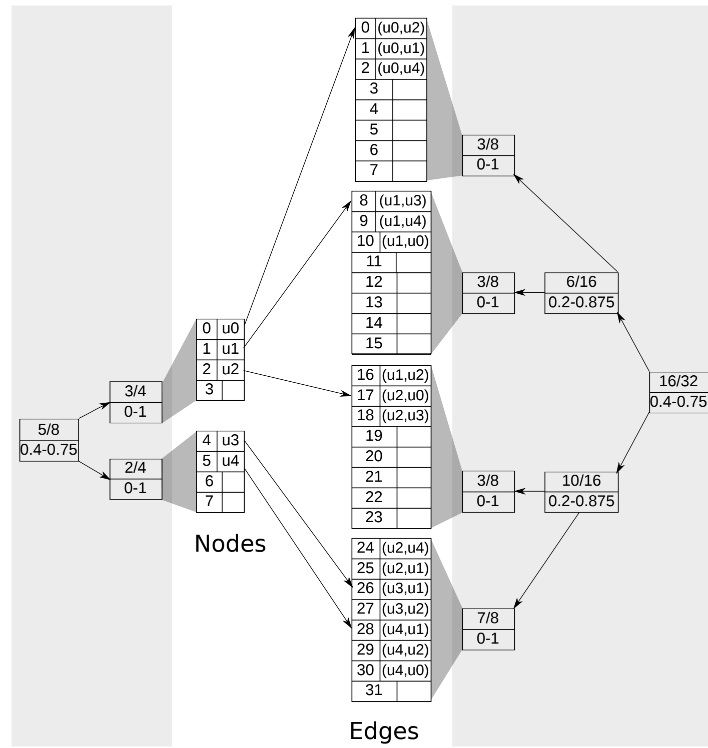


Figure 10: Packed-Memory Graph representation

read/write operations will cause as few memory misses as possible, when an algorithm needs to configure the ordering of the nodes.

The PMG structure can scan S consecutive nodes or edges in $O(S)$ time and $O(S/B)$ memory transfers, where B is the size of the block transferred between the memory layers. Hence, during Dijkstra's algorithm, it can access all outgoing edges of a node very efficiently. It is shown in [65] that the time and memory of the update operations are polylogarithmic in the size of the graph. Further details about PMG can be found in [65].

5.2.2 The heuristic algorithm NAMOA*

Our implementation for finding all Pareto-optimal solutions in the multiobjective shortest path problem is based on NAMOA* algorithm, which incorporates the A^* search technique with the multiobjective Dijkstra's algorithm along with several optimizations. This approach is strict, has very good performance and requires little preprocessing. The s-t query is similar to the multiobjective Dijkstra's algorithm with some extensions. The main difference is that the priority of a label $(\ell_1, \ell_2, ..., \ell_k)$ in the queue is modified according to a heuristic function $h_t : V \rightarrow \mathbb{R}^k$. This function gives a lower bound estimate $h_t(u) = (w_1, w_2, ..., w_k)$ for the cost $c_i(u, t)$ of a shortest u-t path with respect to criterion i, that is, $w_i \leq c_i(u, t), \forall u \in V$ and $1 \leq i \leq k$. By adding this heuristic function to the priority of each generated label of a node, the search is pulled faster towards the target. The tighter the lower bound is, the faster the target is reached.

In order for the A^* search extension to have as large an effect as possible, more modifications have been introduced to the core multiobjective Dijkstra's algorithm [66]. First, the list of labels on a node u are split into two sets, $G_{op}(u)$ and $G_{cl}(u)$ where the first contains the labels that are also present in the queue, and the second contains the rest. This way, when discarding a label from the list that is also present in the queue, it is discarded from the queue as well.

Moreover, as soon as the target node is reached through a non-dominated path P_{st}, this path gets recorded and then removed from the search space. On each iteration, when a label representing a path P_{su} is extracted from the queue, a check takes place; if the label representing P_{su} is dominated by the label representing P_{st}, then there can be no path to t consisting of P_{su} that is not dominated. Therefore, the label representing P_{su} is discarded, and the search is pruned at this point. It is clear that the fastest a first non-dominated path to t is discovered, the earliest the search will be pruned. Hence, the heuristic function that pulls the search towards the target is a very important factor affecting the overall performance of the algorithm.

5.2.3 Computing heuristic functions

TC heuristic. Tung and Chew in [84] have proposed the following heuristic. Let $h_t(u) = (w_1, w_2, ..., w_k)$ be the heuristic function of a node u during a search towards a target node t. The heuristic function consists of the shortest distances from u to t with respect to only one criterion at a time. For each criterion i, a single-criterion shortest path tree is grown from t on the reverse graph $\overleftarrow{G} = (V, \overleftarrow{E})$, $\overleftarrow{E} = \{(v, u)|(u, v) \in E\}$ and each shortest path distance $c_i^*(u, t)$ is recorded for criterion i, $\forall u \in V$ and $1 \leq i \leq k$. Then, the heuristic function becomes $h_t(u) = (c_1^*(u, t), c_2^*(u, t), ..., c_k^*(u, t))$. Clearly, this is a lower bound for any generated distance label of node u using the NAMOA* algorithm.

Bounded calculation for the TC heuristic. The TC heuristic builds a full reverse single-criterion shortest path tree for each criterion of the problem. Even though the single-criterion search is efficient, this process is executed during the query, which clearly must be as fast as possible. Whereas this process is executed during the query, we have to reduce the search space, in order to decrease the running time. To achieve this, an improvement on the TC heuristic was presented in [64]. For simplicity, we shall describe the main idea using $h = 2$ criteria. The approach can easily be extended to multiple criteria.

Let $c_1^*(u, t)$ be the shortest path cost from u to t with respect to the first criterion. The cost of this path using the second criterion is denoted as $c_2^{'}(u, t)$, which clearly may not be optimal. Accordingly, $c_2^*(u, t)$ is defined as the cost of the shortest path from u to t with respect to the second criterion, and the cost of this path under the first criterion is denoted as $c_1^{'}(u, t)$. It has been shown in [66] that NAMOA* does not consider paths whose costs are dominated by $(c_1^{'}(u, t), c_2^{'}(u, t))$, since this can never lead to non-dominated solutions. The approach in [64] called TC-bounded heuristic, for the computation of $h_t(u)$ consists of the following steps:

1. A reverse single-criterion shortest path tree is grown from t using the first criterion. During the growth of the tree, for each node, the shortest path distance towards t is assigned as the first criterion heuristic for this node. The search is stopped as soon as it reaches s with cost $c_1^*(s, t)$. The cost of P_{st} under the second criterion is recorded as $c_2^{'}(s, t)$ and the search is paused at this point.

2. A reverse single-criterion shortest path tree is grown from t using the second criterion. In the same manner as in the first step, each node u gets assigned its lower bound for the second criterion. The search is stopped as soon as the minimum cost in the queue is greater than $c_2^{'}(s, t)$. As it can be observed, node s is settled before quitting the search, and gets assigned the shortest path $c_2^*(s, t)$, with $c_2^*(s, t) \leq c_2^{'}(s, t)$. The cost of this path under the first criterion is recorded as $c_1^{'}(s, t)$.

3. Now, the first search continues from the same point it was paused in Step 1. The search stops as soon as the minimum cost in the queue is greater than $c_1^{'}(s, t)$.

5.2.4 Extensions of NAMOA*

We have extended the NAMOA* algorithm by enhancing it with several implementation optimizations that greatly improve memory accesses of any functions used. In particular, we used three

optimizations which are the following:

1. We do not keep $G_{op}(u)$ and $G_{cl}(u)$ as different entities on each node. Instead, we have combined them into one list of labels, and have extended the actual labels to contain a flag determining whether a particular label is in the queue or not. A a result, all labels of a node, either open or closed, reside on consecutive memory addresses, yielding less cache misses.

2. We keep a pointer on each label, pointing to the predecessor node that generated it. In this way, by following the pointers to the predecessor of a node, we can construct a predecessor graph and all non-dominated routes.

3. We have made the following observation. Any label L residing on a node u during an iteration of the algorithm represents a currently non-dominated path P_{su}. This path might have been the prefix of more non-dominated paths P_{sv} towards a node v. The paths P_{sv} are a concatenation of P_{su} and some path P_{uv}. In case P_{su} becomes dominated by another path P'_{su}, then all paths P_{sv} will be dominated by P'_{sv} consisting of P'_{su} and the original P_{uv}. Hence, when discarding a dominated label L from the list of a node u, we search forward for all subsequent labels of other nodes v generated by L and discard them both from the lists and from the queue.

5.2.5 Experimental Evaluation

We present an extensive experimental evaluation of our NAMOA* implementation. The experiments were conducted on an Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz with a cache size of 6144 Kb and 8 GB of RAM. Our approach was implemented in C++ and compiled by GCC version 4.6.3 with optimization level 3.

To assess the performance of our graph structure and algorithmic implementations, we performed a series of experiments on the road networks with two criteria. The first criterion is the actual distance and the second criterion is the travel time between two nodes. The travel time is not always relative to the actual distance, since different roads have different speed limits. The road networks for our experiments, were taken from the DIMACS homepage, consist of the cities of New York, London, Berlin and the state of Florida. The provided graphs are strongly connected and undirected, so every edge is considered as bidirectional.

We have measured the running times for bounded TC heuristic, which has the best running times in comparison with other heuristics, such as Great Circle distance heuristic and plain TC heuristic. The difference from the plain TC heuristic is the initial computation of the heuristics, not the actual running time of NAMOA*. The running times reported here are the mean values of 10 query repetitions. Table I shows the average running times for all queries on the road maps of New York, London, Berlin and the state of Florida respectively. The time is measured in seconds and values that are omitted are running times that exceed the one-hour limit. It is apparent that the PMG NAMOA* is fast enough to be used in multimodal route planning problems.

Maps	Nodes (n)	Edges (m)	TC Bounded Heuristic (sec)	Pareto Paths
Berlin	101,402	253,078	0.3435	38
London	188,333	429,724	0.4915	37
New York	264,346	733,846	5.2700	166
Florida	1,070.376	2,712.798	15.7226	283

Table 14: Performance of PMG NAMOA* in the road maps of New York, London, Berlin and the state of Florida using the bounded TC heuristic. The execution times are given in seconds. The Pareto optimal path is measured in nodes.

5.2.6 Final Remarks

We have presented a first implementation of an exact method for a core problem in multiobjective and multimodal route planning. The running times of the experiments are very satisfactory and indicate that this algorithm can be used successfully for multimodal route planning. In the next period, we plan to enhance the algorithm with further heuristic improvements.

6 Conclusions and Plans for Next Periods

This deliverable presented the main algorithmic achievements obtained during Task 3.3. It started by discussing the main challenges that we have identified for journey planning in multimodal networks. At the core of these challenges lies the fact that in multimodal transportation networks there is much more choice and a multitude of "best" solutions (where "best" relates to hidden user preferences and is hard to capture as a single metric; in fact, lack of knowledge of what is "best" might well be the reason for users to adopt a journey planning service in the first place). Unlike in road networks it does not suffice anymore to only preserve unique shortest paths during preprocessing.

This requires preprocessing techniques, a necessary ingredient to fast query performance, to be flexible with respect to user constraints that are specified only at query time. Also, query algorithms are needed that are able to compute all (or some of the many) good journeys through the network. In this deliverable, we have presented several such approaches that enable efficient realistic multimodal journey planning in large metropolitan areas. The fastest of these allow us to compute complex queries that return diverse and relevant sets of journey options in below one seconds or less. Less complex but still very realistic scenarios can be solved in even a few milliseconds by our algorithms.

A natural direction for future research is accelerating our approach further to enable interactive queries with an even richer set of criteria. Based on realistic consumption data provided by the transit operators, we would like to finally include eco-friendliness as a forth optimization criterion. Seeing all the good solutions for traveling through the urban area, the wealth of choice, should significantly help users to choose eco-friendly routes. For future work, we are interested in investigating network decomposition techniques to make our approach more scalable. We are also interested in including dynamic scenarios, better handling delay and traffic information. Of course, the ultimate goal is to compute multi-criteria multimodal journeys on a global scale in real time.

References

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In Pardalos and Rebennack [75], pages 230–241.

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In Leah Epstein and Paolo Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

[3] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In Moses Charikar, editor, *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 782–793. SIAM, 2010.

[4] Georgia Aifadopoulou, Athanasios Ziliaskopoulos, and Evangelia Chrisohoou. Multiobjective Optimum Path Algorithm for Passenger Pretrip Planning in Multimodal Transportation Networks. *Journal of the Transportation Research Board*, 2032(1):26–34, December 2007. 10.3141/2032-04.

[5] Susanne Albers, Helmut Alt, and Stefan Näher, editors. *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*. Springer, 2009.

[6] *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012.

[7] Leonid Antsfeld and Toby Walsh. Finding Multi-criteria Optimal Paths in Multi-modal Public Transportation Networks using the Transit Algorithm. In *Proceedings of the 19th ITS World Congress*, 2012.

[8] *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, OpenAccess Series in Informatics (OASIcs), 2009.

[9] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Engineering Label-Constrained Shortest-Path Algorithms. In Demetrescu et al. [35], pages 309–319.

[10] Chris Barrett, Riko Jacob, and Madhav V. Marathe. Formal-Language-Constrained Path Problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[11] Hannah Bast. Car or Public Transport – Two Worlds. In Albers et al. [5], pages 355–367.

[12] Hannah Bast. Next-Generation Route Planning: Multi-Modal, Real-Time, Personalized, 2012. Talk given at ISMP.

[13] 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 Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[14] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.

[15] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Festa [43], pages 166–177.

[16] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[17] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 57(1):38–52, January 2011.

[18] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In ATMOS'09 [8].

[19] Annabell Berger, Martin Grimmer, and Matthias Müller–Hannemann. Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In Festa [43], pages 35–46.

[20] Maurizio Bielli, Azedine Boulmakoul, and Hicham Mouncif. Object modeling and path computation for multimodal travel systems. *European Journal of Operational Research*, 175(3):1705–1730, 2006.

[21] David W. Corne, Kalyanmoy Deb, Peter J. Fleming, and Joshua D. Knowles. The Good of the Many Outweighs the Good of the One: Evolutionary Multi- Objective Optimization. *Connections*, 1(1):9–13, 2003.

[22] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

[23] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, May 2011.

[24] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 921–931. IEEE Computer Society, 2011. Best Paper Award - Algorithms Track.

[25] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In Pardalos and Rebennack [75], pages 376–387.

[26] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In Demetrescu et al. [35], pages 73–92.

[27] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–12. IEEE Computer Society, 2010.

[28] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. *ACM Journal of Experimental Algorithmics*, 17(1), July 2012.

[29] Daniel Delling and Giacomo Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.

[30] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating Multi-Modal Route Planning by Access-Nodes. In Amos Fiat and Peter Sanders, editors, *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*, volume 5757 of *Lecture Notes in Computer Science*, pages 587–598. Springer, September 2009.

[31] Daniel Delling, Thomas Pajor, Dorothea Wagner, and Christos Zaroliagis. Efficient Route Planning in Flight Networks. In ATMOS'09 [8].

[32] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In ALENEX'12 [6], pages 130–140.

[33] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[34] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 125–136. Springer, June 2009.

[35] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[36] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013.

[37] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-Constrained Multi-Modal Route Planning. In ALENEX'12 [6], pages 118–129.

[38] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[39] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In McGeoch [67], pages 347–361.

[40] Andrew Ensor and Felipe Lillo. Partial order approach to compute shortest paths in multimodal networks. Technical report, http://arxiv.org/abs/1112.3366v1, 2011.

[41] Marco Farina and Paolo Amato. A Fuzzy Definition of "Optimality" for Many-Criteria Optimization Problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 34(3):315–326, 2004.

[42] Qing-Wen Feng, Xiaofei Zheng, and Cairong Yan. Study and Practice of an Improving Multi-path Search Algorithm in a City Public Transportation Network. In *Advanced Electrical and Electronics Engineering*, volume 87 of *Lecture Notes in Electrical Engineering*, pages 87–96. Springer, 2011.

[43] Paola Festa, editor. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*. Springer, May 2010.

[44] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In Festa [43], pages 71–82.

[45] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.

[46] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch [67], pages 319–333.

[47] 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.

[48] General Transit Feed. `https://developers.google.com/transit/gtfs/`, 2010.

[49] General Transit Feed Specification. `https://developers.google.com/transit/gtfs/`, 2012.

[50] Frank Geraets, Leo G. Kroon, Anita Schöbel, Dorothea Wagner, and Christos Zaroliagis. *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*. Springer, 2007.

[51] Andrew V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.

[52] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

[53] Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

[54] HaCon - Ingenieurgesellschaft mbH. `http://www.hacon.de`, 1984.

[55] HaCon website. `http://www.hacon.de/hafas/`, 2013.

[56] Pierre Hansen. Bricriteria Path Problems. In Günter Fandel and T. Gal, editors, *Multiple Criteria Decision Making – Theory and Application –*, pages 109–127. Springer, 1979.

[57] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[58] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [35], pages 41–72.

[59] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[60] Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo. A Label Correcting Algorithm for the Shortest Path Problem on a Multi-Modal Route Network. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *Lecture Notes in Computer Science*. Springer, 2012.

[61] Dominik Kirchler, Leo Liberti, Thomas Pajor, and Roberto Wolfler Calvo. UniALT for Regular Language Constraint Shortest Paths on a Multi-Modal Transportation Network. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–75, 2011.

[62] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[63] London Data Store. `http://data.london.gov.uk/`, 2011.

[64] E Machuca and L Mandow. Multiobjective Heuristic Search in Road Maps. *Expert Systems with Applications*, 39(7):6435–6445, 2012.

[65] Georgia Mali, Panagiotis Michail, Andreas Paraskevopoulos, and Christos Zaroliagis. A New Dynamic Graph Structure for Large-Scale Transportation Networks. In *Algorithms and Complexity*, pages 312–323. Springer, 2013.

[66] Lawrence Mandow and José Luis Pérez De La Cruz. Multiobjective A* Search with Consistent Heuristics. *Journal of the ACM (JACM)*, 57(5):27, 2010.

[67] Catherine C. McGeoch, editor. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*. Springer, June 2008.

[68] Alberto O. Mendelzon and Peter T. Wood. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.

[69] Metropolitan Transportation Authority of the State of New York. `http://www.mta.info/`, 1966.

[70] Paola Modesti and Anna Sciomachen. A utility measure for finding multiobjective shortest paths in urban multimodal transportation networks. *European Journal of Operational Research*, 111(3):495–508, 1998.

[71] Matthias Müller–Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization* [50], pages 246–263.

[72] Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization* [50], pages 67–90.

[73] Matthias Müller–Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

[74] Thomas Pajor. Multi-Modal Route Planning. Master's thesis, Universität Karlsruhe (TH), March 2009.

[75] Panos M. Pardalos and Steffen Rebennack, editors. *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*. Springer, 2011.

[76] PTV AG – Planung Transport Verkehr. `http://www.ptv.de`, 1979.

[77] 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.

[78] Michael Rice and Vassilis Tsotras. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. In *Proceedings of the 37th International Conference on Very Large Databases (VLDB 2011)*, pages 69–80, 2011.

[79] Peter Sanders. Algorithm Engineering – An Attempt at a Definition. In Albers et al. [5], pages 321–340.

[80] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[81] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

[82] Christian Sommer. Shortest-Path Queries in Static Networks, 2012. Submitted. Preprint available at `http://www.sommer.jp/spq-survey.htm`.

[83] Transport for London. `http://www.tfl.gov.uk/`, 2000.

[84] Chi Tung Tung and Kim Lin Chew. A Multicriteria Pareto-Optimal Path Algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992.

[85] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10(1.3):1–30, 2005.

[86] Haicong Yu and Feng Lu. Advanced multi-modal routing approach for pedestrians. In *2nd International Conference on Consumer Electronics, Communications and Networks*, pages 2349–2352, 2012.

[87] Lotfi A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965.

[88] Lotfi A. Zadeh. Fuzzy Logic. *IEEE Computer*, 21(4):83–93, 1988.