Project Number 288094

eCOMPASS

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

STREP
Funded by EC, INFSO-G4(ICT for Transport) under FP7

eCOMPASS – TR – 058

User-Constrained Multi-Modal Route Planning

Julian Dibbelt, Thomas Pajor, Dorothea Wagner

June 2014

User-Constrained Multi-Modal Route Planning

JULIAN DIBBELT, THOMAS PAJOR, DOROTHEA WAGNER
Karlsruhe Institute of Technology

In the multi-modal route planning problem we are given multiple transportation networks (e. g., pedestrian, road, public transit) and ask for a best *integrated* journey between two points. The main challenge is that a seemingly optimal journey may have changes between networks that do not reflect the user's modal preferences. In fact, quickly computing reasonable multi-modal routes remains a challenging problem: Previous approaches either suffer from poor query performance or their available choices of modal preferences during query time is limited. In this work we focus on computing exact multi-modal journeys that can be restricted by specifying *arbitrary* modal sequences at query time. For example, a user can say whether he wants to only use public transit, or also prefers to use a taxi or walking at the beginning or end of the journey; or if he has no restrictions at all. By carefully adapting node contraction, a common ingredient to many speedup techniques on road networks, we are able to compute point-to-point queries on a continental network combined of cars, railroads and flights several orders of magnitude faster than Dijkstra's algorithm. Thereby, we require little space overhead and obtain fast preprocessing times.

Categories and Subject Descriptors: G.2.2 [**Graph Theory**]: Graph algorithms

General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Shortest Paths, Route Planning, Multi-Modal, Contraction

1. INTRODUCTION

Research on route planning algorithms in transportation networks has undergone a rapid development over the last years. See [Delling et al. 2009d] 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 [Goldberg 2008], it is too slow for practical applications in real-world transportation networks. They 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 [Abraham et al. 2011]. 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 [Bast 2009; Bauer et al. 2010].

Partially supported by DFG grant WA 654/16-1 and by the EU FP7/2007-2013 (DG INFSO.G4-ICT for Transport), under grant agreement no. 288094 (project eCOMPASS). An extended abstract appeared at the 14th SIAM Meeting on Algorithm Engineering & Experiments (ALENEX12).
Authors' addresses: Julian Dibbelt, Karlsruhe Institute of Technology, P.O. Box 6980, 76128 Karlsruhe, Germany, email: *dibbelt@kit.edu*; Thomas Pajor, Karlsruhe Institute of Technology, P.O. Box 6980, 76128 Karlsruhe, Germany, email: *pajor@kit.edu*; Dorothea Wagner, Karlsruhe Institute of Technology, P.O. Box 6980, 76128 Karlsruhe, Germany, email: *dorothea.wagner@kit.edu*.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 may become more important. To solve such scenarios we are interested in an integrated system that can handle multiple transportation networks with a single algorithm. Thereby it is crucial to respect a user's modal preferences: Not every mode of transport might be feasible to 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. A realistic multi-modal route-planning system must handle such constraints as a *user input* for each *query*.

Related Work. For an overview on unimodal speedup techniques, we direct the reader to [Bast 2009; Delling et al. 2009d]. Most techniques are composed of the following ingredients: Bidirectional search, goal-directed search [Goldberg and Harrelson 2005; Hart et al. 1968; Lauther 2004; Wagner et al. 2005], hierarchical techniques [Bast et al. 2010; Bast et al. 2007; Geisberger et al. 2008; Gutman 2004; Sanders and Schultes 2005], and separator-based techniques [Delling et al. 2011b; Delling et al. 2009a; Holzer et al. 2008]. Various combinations have been studied in [Bauer et al. 2010; Schulz et al. 2000].

Regarding multi-modal route planning less work exists. An elegant approach to restricting modal transfers is the label constrained shortest paths problem (LCSPP) [Mendelzon and Wood 1995]: 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 [Barrett et al. 2000; Mendelzon and Wood 1995]. An experimental study of this approach, including basic goal-directed techniques, is conducted in [Barrett et al. 2009]. In [Pajor 2009] it is concluded that augmenting preprocessing techniques for LCSPP is a challenging task.

A first efficient multi-modal speedup technique, called Access-Node Routing (ANR), has been proposed in [Delling et al. 2009b]. 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 multi-modal 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 of transport, called SDALT [Kirchler et al. 2011]. It allows 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 % [Kirchler et al. 2012].

Finally, in [Rice and Tsotras 2011] 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 three orders of magnitude on a continental road network. However, Kleene languages are rather restrictive: In a multi-modal context, they only allow excluding modes of transportation *globally*. In particular, they cannot be used to define feasible *sequences* of transportation modes.

Our Contribution. In this work we present *User-Constrained Contraction Hierarchies* (UCCH), the first multi-modal speedup technique that handles arbitrary mode-sequence constraints as input to the query—a feature unavailable from previous techniques. Unlike Access-Node Routing, 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. 2008]. 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[1] of auxiliary data, we achieve query times that are fast enough for interactive scenarios.

This work is organized as follows. Section 2 sets necessary notation, summarizes graph models we use, precisely defines the problem we are solving, and also recaps Contraction Hierarchies. Section 3 introduces our new technique. Finally, Section 4 presents experiments to evaluate our algorithm, while Section 5 concludes this work and mentions interesting open problems.

2. 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., $(u, v) \in \overleftarrow{E}$ if and only if $(v, u) \in E$. Note that we use the terms graph and network interchangeably. To distinguish 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, we require link and merge operations which generalize the summation and minimum operations from scalar values to travel time functions. Thereby, the *link* operation of two functions f_1, f_2 is defined for any departure time τ as $(f_1 \oplus f_2)(\tau) = f_1(\tau) + f_2(\tau + f_1(\tau))$, and depicts the total travel time when first evaluating f_1 (at departure time τ) and then f_2 (at departure time $\tau + f_1(\tau)$, i. e., the arrival time after "traversing" f_1). The *merge* operation $\min(f_1, f_2)(\tau)$ is defined as the element-wise minimum of f_1 and f_2, i. e., $\min(f_1(\tau), f_2(\tau))$. 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 over all choices of τ.

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 from 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 $w = \sigma_0\sigma_1 \ldots \sigma_k$ 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*

[1]MiB: $2^{20} = 1024^2$ bytes, GiB: $2^{30} = 1024^3$ bytes

$\delta \subseteq Q \times \Sigma \times Q$, 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.

2.1. Models

Following [Delling et al. 2009b], our multi-modal 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 pedestrian paths. 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 [Delling et al. 2012] which is based on the well-known realistic time-dependent model [Pyrga et al. 2008]. 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 [Delling et al. 2012] 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 [Delling et al. 2009c]. It has small size and models airport procedures realistically. Nodes and edges are labeled with `flight`.

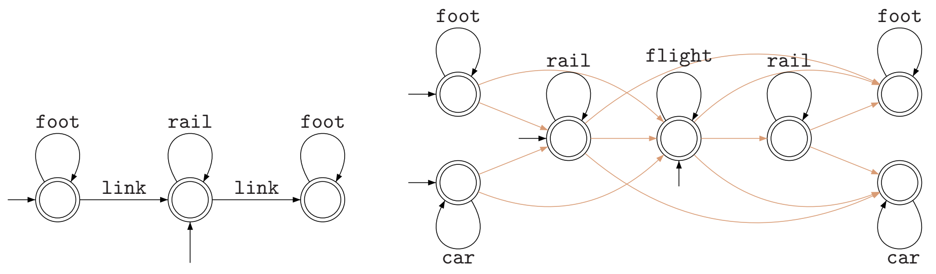
Note that the travel time functions in our networks are a special form of piecewise linear functions that can be efficiently evaluated [Pyrga et al. 2008; Delling et al. 2012]. Also all edges in our networks have the FIFO property.

Merging the Networks. To obtain an integrated *multi-modal* 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.

2.2. 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) [Barrett et al. 2000]: 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. To represent mode sequence constraints, 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, 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 Fig. 1.



(a) foot-and-rail. (b) hierarchical.

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

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 Dijkstra's algorithm works [Barrett et al. 2000].

2.3. Contraction Hierarchies (CH)

Our algorithm is based on Contraction Hierarchies [Geisberger et al. 2008]. 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 $\mathrm{rank}(v)$. 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 $\mathrm{len}(u, w) \leq \mathrm{len}(u, v) + \mathrm{len}(v, w)$, 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 $\mathrm{rank}(u) \leq \mathrm{rank}(v)$, and the backward search only visits edges where $\mathrm{rank}(u) \geq \mathrm{rank}(v)$. 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 $\mathrm{dist}(u) + \mathrm{len}(e) < \mathrm{dist}(v)$ holds ($\mathrm{dist}(v)$ denotes the tentative distance at v). If this is the case, we may prune the search at v. See [Geisberger et al. 2008] for details.

Partial Hierarchy. If the preprocessing is aborted prematurely, i. e., before all nodes are contracted, we obtain a partial contraction hierarchy (PCH). Let $\mathrm{rank}(v) = \infty$ if and only if v is never contracted, then the same query algorithm as for Contraction Hierarchies is applicable and yields correct results [Bauer et al. 2010]. The induced subgraph of all uncontracted nodes is called 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 [Abraham et al. 2010]. 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.

3. 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 multi-modal graph $\mathbf{G}$ without careful consideration either yields incorrect results to LCSPP-MS or finalizes 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 3.3.

3.1. Contraction Hierarchies for Multi-Modal Networks

Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a multi-modal network. Recall that $\mathbf{G}$ 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 $\mathbf{G}$ 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 precisely, when a path $P = (e_1, \dots, e_k)$ is composed into a single shortcut edge e', its labels need to be *concatenated* into a super label $\mathrm{lbl}(e') = \mathrm{lbl}(e_1) \cdots \mathrm{lbl}(e_k)$. In particular, if there are subsequent edges e_i, e_j in P where $\mathrm{lbl}(e_i) \neq \mathrm{lbl}(e_j)$, 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 2.3). Note that the partial time-dependent nature of $\mathbf{G}$ 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 $\mathrm{lbl}(e') = \mathrm{lbl}(e_1) \cdots \mathrm{lbl}(e_k)$. 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 $\mathrm{lbl}(v)$, we run a multi-modal 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 $\mathrm{lbl}(e')$. Only if for all these profile-searches $\mathrm{dist}(w) \leq \mathrm{len}(e')$ 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 [Geisberger et al. 2010] and Timetable CH [Geisberger 2010]. As a result we expect a significant slowdown over unimodal CH on road networks. But most notably, SDCH requires a predetermined automaton $\mathcal{A}$ during preprocessing.

3.2. 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 a simpler witness search. We first turn toward preprocessing before we go into detail about the query algorithm.

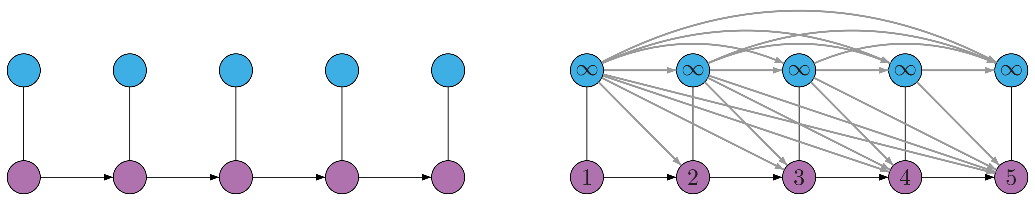
Preprocessing. The main reason behind the disadvantages discussed in Section 3.1 is the computation of shortcuts that span over boundaries of different modal networks. Instead, let Σ be the alphabet of labels of a multi-modal graph $\mathbf{G}$. We now process each subnetwork independently. We compute—in 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 consider the traditional 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, represented by e'.

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 [Geisberger et al. 2008]. 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 $\text{rank}(v) = \infty$ 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 Fig. 2 and [Geisberger 2010]). It also hides information encoded in the timetable model (such as railway lines), further complicating query algorithms [Berger et al. 2009].

Query. Our query algorithm combines the concept of a multi-modal 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 $\mathcal{A}$



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

Fig. 2. Contracting only route nodes in the realistic time-dependent rail model [Pyrga et al. 2008]. 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).

(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 $\mathrm{rank}(w) \geq \mathrm{rank}(v)$. Similarly, the backward search only looks at edges (v, w) where $\mathrm{rank}(v) \geq \mathrm{rank}(w)$. 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 [Delling et al. 2009d]. 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 [Delling et al. 2009b], 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 [Delling and Nannicini 2008; Batz et al. 2010], 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.

3.3. 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 [Delling et al. 2011a]. 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 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 Fig. 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 queue for the first time on a branch of

Table I. Comparing size figures of our input instances. The column "Col." indicates whether we use the coloring approach (see Section 2.1) to model the railway subnetwork. The bottom two instances are taken from [Delling et al. 2009b].

	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

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 up to 1.1 GiB 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 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. (Section 4.5 of the experimental evaluation will show detailed figures.)

4. EXPERIMENTS

We conducted our experiments on 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. We use our own custom implementations for most data structures. In particular, we represent graphs as adjacency arrays, and as a priority queue we use a 4-ary heap. All runs are sequential for comparison.

Inputs. We assemble a total of six multi-modal networks where two are imported from [Delling et al. 2009b]. Their size figures are reported in Table I. For `ny-road-rail`, we combine New York's foot network with the public transit network operated by MTA [Metropolitan Transportation Authority of the State of New York 1966]. 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

Table II. 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 and partial CH (PCH).

	Preprocessing				Query			
Algorithm	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

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` are from [Delling et al. 2009b]. The data of the Western European and North American road networks (thus Germany and New York) was kindly provided to us by PTV AG [PTV AG – Planung Transport Verkehr 1979] for scientific use. The timetable data of New York is publicly available through General Transit Feeds [General Transit Feed 2010], while the data of the German and European railway networks was kindly provided by HaCon [HaCon - Ingenieurgesellschaft mbH 1984]. Unlike the data from HaCon, the New York timetable did not contain any foot path data for short transfers between nearby stops (as typically defined by the operator). Thus, we generated artificial foot paths with a known heuristic [Delling et al. 2012].

Our instances vary in the fractional size of their public transit subnetwork with respect to the total network size. We call the fraction of linked nodes in a subgraph *density* (see last column of Table I). 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). Note that we take the figures for ANR from [Delling et al. 2009b]. Since they used a different machine, we scale the running time figures by comparing the running time of Dijkstra's algorithm on our machine to theirs. Also note that for comparison we do not use the improved coloring model (see Section 2.1) on these two instances.

We use the following automata as query input. The `foot-and-rail` automaton allows either 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. Note that completely disallowing walking is not reasonable. Instead, taking the predefined (by the timetable) transfer foot paths within the `rail` (`flight`) model is always allowed within the `rail` (`flight`) state. Finally, the `everything` automaton allows arbitrary modal sequences in any order. See Fig. 1 for transition graphs of `foot-and-rail` and `hierarchical`.

Table III. Preprocessing figures for UCCH and Access-Node Routing on the road subnetwork. Figures for the latter are taken from [Delling et al. 2009b], which were obtained on a different machine. We thus scale the preprocessing time with respect to running time figures compared to Dijkstra.

	UCCH						ANR	
	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

Methodology. We evaluate both preprocessing and query performance. The contraction order is always computed according to the aggressive variant from [Geisberger et al. 2008]. 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 [Geisberger et al. 2008].

4.1. 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 by dividing the number of edges by the number of nodes in our graph data structure. Note that we use *edge compression* [Delling 2009]: 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 II shows preprocessing and query figures on `de-road-rail`. For queries we use the `foot` automaton, which 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 is too large, and preprocessing does not finish within reasonable time. Interestingly, 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 II 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

Table IV. Query performance of UCCH compared to plain multi-modal Dijkstra and Access-Node Routing. Figures for the latter are taken from [Delling et al. 2009b], which were obtained on a different machine. We thus scale the running time with respect to Dijkstra.

		Dijkstra		ANR			UCCH		
Network	Automaton	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

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 *multi-modal* 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.

4.2. Preprocessing

Table III 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 Access-Node Routing (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, on `wo-road-flight` by a factor of 26. Here, ANR requires 14 GiB of space, whereas UCCH only uses 542 MiB. Concluding, UCCH outperforms ANR in terms of preprocessing space and time.

4.3. Query Performance

In this experiment we evaluate the query performance of UCCH and compare it to Dijkstra and ANR (where figures are available). Results are presented in Table IV. 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

Table V. Evaluating the impact of integrating modal sequence constraints on the paths.

		Mode Used in % Paths				# Modal Changes		Stretch		
Network	Automaton	Foot	Car	Rail	Flight	Avg.	Max.	Avg.	Max.	Ident. [%]
`ny-road-rail`	`everything`	100	—	57	—	4.1	22	—	—	—
`ny-road-rail`	`foot-and-rail`	100	—	57	—	1.1	2	1.07	2.83	64
`de-road-rail`	`everything`	100	—	100	—	6.8	24	—	—	—
`de-road-rail`	`foot-and-rail`	100	—	100	—	2.0	2	1.08	2.94	87
`europe-road-rail`	`everything`	—	100	41	—	1.2	10	—	—	—
`europe-road-rail`	`car-and-rail`	—	100	41	—	0.8	2	1.03	1.46	92
`wo-road-rail-flight`	`everything`	—	100	13	85	2.2	12	—	—	—
`wo-road-rail-flight`	`hierarchical`	—	100	9	85	1.8	4	1.08	2.25	89
`wo-road-rail-flight`	`car-and-flight`	—	100	—	85	1.7	2	1.06	2.34	84

50,540. To further highlight how the density of the network affects the speedup, Fig. 3 plots the speedup of UCCH on each instance subject to its density. Note that most of the time is spent inside the core (particularly, in the public transit network), which we do not accelerate. Section 4.6 contains a detailed query time distribution analysis. 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. Detailed path properties

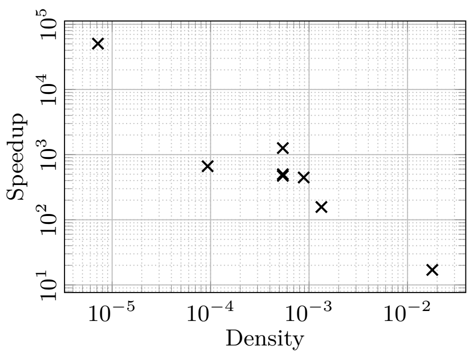


Fig. 3. Evaluating the speedup of UCCH from Table IV subject to the density of the input from Table I.

Table V reports the impact of integrating modal sequence constraints on the paths output by the algorithm. It does so by evaluating three main figures: The percentage of the total number of paths that utilize a certain transportation mode (foot, car, rail with transfers, and flight with transfers), the average and maximum number of interchanges between transportation modes along the journeys, and the average and maximum factor by which the travel time increases when mode sequence constraints are enabled. Note that for the latter, we only count paths that actually differ from the unconstrained one, additionally reporting the amount of paths where mode sequence constraints have no impact (Ident.). Each instance in Table V is evaluated on both an appropriate constrained automaton as well as the `everything` automaton, which essentially corresponds to running unrestricted queries.

We observe that on `ny-road-rail` 57 % of the paths utilize the rail network, regardless whether we constrain paths by the `foot-and-rail` automaton. However, 36 % of the paths are indeed different, and enabling constraints reduces the average number of modal interchanges by a factor of almost four with only a 7 % increase in travel time. Figures for `de-road-rail` are similar: All paths use the rail network, and enabling constraints reduces the number of modal interchanges by a factor of almost 3.5 with only little increase in travel time. On our sparser long-distance networks the effects are less pronounced. For example, on `wo-road-rail-flight`, we see that 89 % of the paths

Table VI. Detailed analysis of the impact on query performance by our improvements (cf. Section 3.3). We show figures for reordering nodes (rn), reordering edges (re), improved bidirectional search (bi), only forward search on the core (fo), state-independent search on component (si), and state-pruning (sp)

Network	Automaton	Improvement	Settled Nodes	Time [ms]	Speed-Up
europe-road-rail	car	none	48 488	69.93	—
		rn	48 488	35.11	2.00
		rn,re	48 488	29.38	2.38
		rn,re,bi	31 628	20.02	3.49
		rn,re,fo	24 297	14.57	4.80
wo-road-rail-flight	car	none	35 539	54.42	—
		rn	35 539	27.93	1.95
		rn,re	35 539	23.18	2.35
		rn,re,bi	29 695	19.84	2.74
		rn,re,fo	17 862	11.50	4.73
europe-road-rail	car-and-rail	rn,re,fo	95 095	57.23	—
		rn,re,fo,si	95 024	60.12	0.95
		rn,re,fo,sp	89 770	51.72	1.11
		rn,re,fo,si,sp	89 699	54.45	1.05
wo-road-rail-flight	car-and-rail	rn,re,fo	72 997	46.73	—
		rn,re,fo,si	72 895	49.09	0.95
		rn,re,fo,sp	69 627	42.35	1.10
		rn,re,fo,si,sp	69 525	44.51	1.05

already follow a hierarchical use of transportation modes, and the difference in the number of modal interchanges decreases only by 0.4. However, while this difference may seem small, we argue that model constraints are nevertheless important, since our experiment shows that in 11 % of the cases the (unconstrained) path violates the modal constraints, which may render it completely infeasible to the user.

4.5. Improvements

In Table VI we report figures for the improvements to UCCH described in Section 3.3. The table is divided into two parts. The upper part addresses unimodal improvements that are also applicable to (partial) CH. Therefore, we evaluate them using the car automaton. For our two biggest networks, we provide the number of settled nodes and the query time for several combinations of improvements. The first row (none) reports results for the basic version of UCCH. The other rows use: Reordered nodes (rn), reordered edges (re), improved bi-directional search on the core (bi), and uni-directional search on the core (fo), that is, no backward search is performed on the core. Combining these techniques, we obtain a speedup of up to a factor of 4.8.

The lower part of Table VI is dedicated to improvements for UCCH which we evaluate using the car-and-rail automaton. We provide numbers for state-independent search on the component (si) and state-pruning (sp). Note that these figures already include the previous improvements. Interestingly, using state-independent search results in slightly worse query times of about 5 %. However, we reduce the memory footprint of the algorithm by a significant amount since we store distance values only once per component node. Maintaining distance labels on the implicit product graph requires between 6.9 MiB and 1341.2 MiB on our instances. When (si) is enabled, these numbers are reduced to 2.4 MiB and 192.1 MiB, respectively. This is an improvement of up to factor 7.

Note that from the number of settled nodes we can deduce which of the improvements impact cache efficiency and which impact the search space.

Table VII. In-depth analysis of UCCH's query time. We report the distribution of query time among the particular subnetworks and compare it to Dijkstra.

			Dijkstra		UCCH		
Network	Automaton	Subgraph	Settled Nodes	Time [ms]	Settled Nodes	Time [ms]	Speed-Up
`ny-road-rail`	`foot-and-rail`	road-comp.	—	—	203	≈ 0.0	—
		road-core	389 578	215.5	9 944	4.8	45
		rail	15 238	10.5	15 238	8.8	1.2
`de-road-rail`	`foot-and-rail`	road-comp.	—	—	188	≈ 0.0	—
		road-core	2 599 251	1 988.4	6 314	5.0	397
		rail	11 803	16.6	11 803	7.8	2.1
`europe-road-rail`	`car-and-rail`	road-comp.	—	—	213	≈ 0.0	—
		road-core	29 973 817	23 933.3	43 017	24.4	982
		rail	47 750	59.7	47 750	29.4	2.0
`wo-road-rail-flight`	`hierarchical`	road-comp.	—	—	301	≈ 0.0	—
		road-core	36 047 522	35 169.3	49 944	30.6	1 149
		rail	75 682	89.9	75 682	39.2	2.3
		flight	902	1.8	902	0.7	2.6

4.6. In-Depth Analysis of Query Performance

Table VII reports in-depth figures for the UCCH query including all (reasonable) improvements from the previous section. We see that a large fraction of the query is spent on the public transportation part of the multi-modal network: Up to $65\,\%$ of the settled nodes and also up to $65\,\%$ of query time. Recall that we do not further accelerate the search on the core. Interestingly, UCCH is slightly faster (up to a factor of 2.6) on the timetable subnetworks when compared to Dijkstra. UCCH settles fewer nodes in total, which helps cache performance on the public transit part. When we compare the time spent on the road network (component and core) of `de-road-rail` with the figures of Table II (where we use the same instance but with the smaller `foot` automaton), we observe that the `foot-and-rail` automaton yields a factor 1.8 slowdown. The reason is that the `foot-and-rail` automaton actually has two "foot-states" (cf. Fig. 1) and, thus, has to do twice the work on the road subnetwork. Note that the number 1.8 (instead of exactly 2) stems from the fact that we apply state pruning.

5. CONCLUSION

In this work we introduced UCCH: The first, fast multi-modal 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 multi-modal techniques. For future work we are interested in augmenting our approach to more general scenarios. For example, the computation of multi-modal profile queries would produce journeys whose departure time follows the timetable more closely. Moreover, we are interested in constraining the amount of total walking time or optimizing it in a multi-criteria setting. We would also like to further accelerate search on the uncontracted core—especially on the rail networks. Finally, we are interested to improve the contraction order. In particular, we would like to use ideas from [Delling et al. 2009b] to enable contraction of some transfer nodes in order to achieve better results, especially on more densely interlinked networks.

ACKNOWLEDGMENTS

We thank Daniel Delling for interesting discussions on multi-modal route planning, and Geisberger et al. for providing us their CH code.

REFERENCES

ABRAHAM, I., DELLING, D., GOLDBERG, A. V., AND WERNECK, R. F. 2011. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Lecture Notes in Computer Science Series, vol. 6630. Springer, 230–241.

ABRAHAM, I., FIAT, A., GOLDBERG, A. V., AND WERNECK, R. F. 2010. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*. SIAM, 782–793.

BARRETT, C., BISSET, K., HOLZER, M., KONJEVOD, G., MARATHE, M. V., AND WAGNER, D. 2009. Engineering Label-Constrained Shortest-Path Algorithms. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. DIMACS Book Series, vol. 74. American Mathematical Society, 309–319.

BARRETT, C., JACOB, R., AND MARATHE, M. V. 2000. Formal-Language-Constrained Path Problems. *SIAM Journal on Computing 30,* 3, 809–837.

BAST, H. 2009. Car or Public Transport – Two Worlds. In *Efficient Algorithms*. Lecture Notes in Computer Science Series, vol. 5760. Springer, 355–367.

BAST, H., CARLSSON, E., EIGENWILLIG, A., GEISBERGER, R., HARRELSON, C., RAYCHEV, V., AND VIGER, F. 2010. Fast Routing in Very Large Public Transportation Networks using Transfer Patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. Lecture Notes in Computer Science Series, vol. 6346. Springer, 290–301.

BAST, H., FUNKE, S., MATIJEVIC, D., SANDERS, P., AND SCHULTES, D. 2007. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 46–59.

BATZ, G. V., GEISBERGER, R., NEUBAUER, S., AND SANDERS, P. 2010. Time-Dependent Contraction Hierarchies and Approximation. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Lecture Notes in Computer Science Series, vol. 6049. Springer, 166–177.

BAUER, R., DELLING, D., SANDERS, P., SCHIEFERDECKER, D., SCHULTES, D., AND WAGNER, D. 2010. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics 15,* 2.3, 1–31. Special Section devoted to WEA'08.

BERGER, A., DELLING, D., GEBHARDT, A., AND MÜLLER–HANNEMANN, M. 2009. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. OpenAccess Series in Informatics (OASIcs).

DELLING, D. 2009. Engineering and Augmenting Route Planning Algorithms. Ph.D. thesis, Universität Karlsruhe (TH), Fakultät für Informatik.

DELLING, D., GOLDBERG, A. V., NOWATZYK, A., AND WERNECK, R. F. 2011a. PHAST: Hardware-Accelerated Shortest Path Trees. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 921–931. Best Paper Award - Algorithms Track.

DELLING, D., GOLDBERG, A. V., PAJOR, T., AND WERNECK, R. F. 2011b. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Lecture Notes in Computer Science Series, vol. 6630. Springer, 376–387.

DELLING, D., HOLZER, M., MÜLLER, K., SCHULZ, F., AND WAGNER, D. 2009a. High-Performance Multi-Level Routing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. DIMACS Book Series, vol. 74. American Mathematical Society, 73–92.

DELLING, D., KATZ, B., AND PAJOR, T. 2012. Parallel Computation of Best Connections in Public Transportation Networks. *ACM Journal of Experimental Algorithmics 17,* 4, 4.1–4.26.

DELLING, D. AND NANNICINI, G. 2008. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*. Lecture Notes in Computer Science Series, vol. 5369. Springer, 813–824.

DELLING, D., PAJOR, T., AND WAGNER, D. 2009b. Accelerating Multi-Modal Route Planning by Access-Nodes. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*. Lecture Notes in Computer Science Series, vol. 5757. Springer, 587–598.

DELLING, D., PAJOR, T., WAGNER, D., AND ZAROLIAGIS, C. 2009c. Efficient Route Planning in Flight Networks. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. OpenAccess Series in Informatics (OASIcs).

DELLING, D., SANDERS, P., SCHULTES, D., AND WAGNER, D. 2009d. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*. Lecture Notes in Computer Science Series, vol. 5515. Springer, 117–139.

GEISBERGER, R. 2010. Contraction of Timetable Networks with Realistic Transfers. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Lecture Notes in Computer Science Series, vol. 6049. Springer, 71–82.

GEISBERGER, R., KOBITZSCH, M., AND SANDERS, P. 2010. Route Planning with Flexible Objective Functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*. SIAM, 124–137.

GEISBERGER, R., SANDERS, P., SCHULTES, D., AND DELLING, D. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Lecture Notes in Computer Science Series, vol. 5038. Springer, 319–333.

GENERAL TRANSIT FEED. 2010. `https://developers.google.com/transit/gtfs/`.

GOLDBERG, A. V. 2008. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing 37*, 1637–1655.

GOLDBERG, A. V. AND HARRELSON, C. 2005. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*. SIAM, 156–165.

GUTMAN, R. J. 2004. 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)*. SIAM, 100–111.

HACON - INGENIEURGESELLSCHAFT MBH. 1984. `http://www.hacon.de`.

HART, P. E., NILSSON, N., AND RAPHAEL, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 100–107.

HOLZER, M., SCHULZ, F., AND WAGNER, D. 2008. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics 13,* 2.5, 1–26.

KIRCHLER, D., LIBERTI, L., AND CALVO, R. W. 2012. 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)*. Lecture Notes in Computer Science Series, vol. 7276. Springer.

KIRCHLER, D., LIBERTI, L., PAJOR, T., AND CALVO, R. W. 2011. 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)*. OpenAccess Series in Informatics (OASIcs) Series, vol. 20. 64–75.

LAUTHER, U. 2004. 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*. Vol. 22. IfGI prints, 219–230.

MENDELZON, A. O. AND WOOD, P. T. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing 24,* 6, 1235–1258.

METROPOLITAN TRANSPORTATION AUTHORITY OF THE STATE OF NEW YORK. 1966. `http://www.mta.info/`.

PAJOR, T. 2009. Multi-Modal Route Planning. M.S. thesis, Universität Karlsruhe (TH).

PTV AG – PLANUNG TRANSPORT VERKEHR. 1979. `http://www.ptv.de`.

PYRGA, E., SCHULZ, F., WAGNER, D., AND ZAROLIAGIS, C. 2008. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics 12,* 2.4, 1–39.

RICE, M. AND TSOTRAS, V. 2011. 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)*. 69–80.

SANDERS, P. AND SCHULTES, D. 2005. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*. Lecture Notes in Computer Science Series, vol. 3669. Springer, 568–579.

SCHULZ, F., WAGNER, D., AND WEIHE, K. 2000. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics 5,* 12, 1–23.

WAGNER, D., WILLHALM, T., AND ZAROLIAGIS, C. 2005. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics 10,* 1.3, 1–30.