User-Constrained Multi-Modal Route Planning[*]

Julian Dibbelt[†] Thomas Pajor[‡] Dorothea Wagner[§]

Abstract

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.

1 Introduction

Research on route-planning algorithms in transportation networks has undergone a rapid development over the last years. See [19] 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 [26], it is too slow for practical applications in real-world transportation networks. They consist of several million nodes and edges, 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 [6, 10].

In the real-world the different modes of travel are linked extensively, and realistic transportation scenarios imply frequent modal changes. Even more so, with the emergence 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. 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 [6, 19]. Most techniques are composed of the following ingredients: bidirectional search, goal-directed search [27, 30, 33, 42], hierarchical techniques [7, 8, 24, 28, 40], and separator-based techniques [13, 14, 31]. Combinations have been studied [10, 41]. Regarding multi-modal route planning less work exists. An elegant approach to restricting modal transfers is the label constrained shortest paths problem (LCSPP) [34]: 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 [5, 34]. An experimental study of this approach, including basic goal-directed techniques, is conducted in [4]. In [36] 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 [17]. It skips the road network

[*]Partially supported by DFG grant WA 654/16-1.

[†]Karlsruhe Institute of Technology, dibbelt@kit.edu.
[‡]Karlsruhe Institute of Technology, pajor@kit.edu.
[§]Karlsruhe Inst. of Technology, dorothea.wagner@kit.edu.

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 [32]. It has fast preprocessing, but both preprocessing space and query times are high, and it also cannot handle arbitrary modal restrictions as query input.

Our Contribution. In this work we present 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. [24]. 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 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 *reverse* graph $\overleftarrow{G} = (V, \overleftarrow{E})$ of G is obtained from G by flipping all edges, i. e., $(u, v) \in E$ iff $(v, u) \in \overleftarrow{E}$. 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.

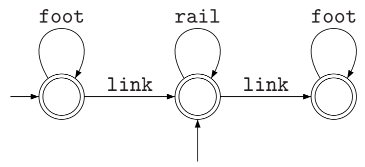
In time-dependent graphs there are two types of queries relevant to this work: A *time-query* has as input $s \in V$ and a departure time τ. It computes a shortest path tree to every node $u \in V$ when departing at s at time τ. In contrast, a *profile-query* computes a shortest path graph from s to 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* iff there exists a finite automaton $\mathcal{A}_L$ such that $\mathcal{A}_L$ accepts L.

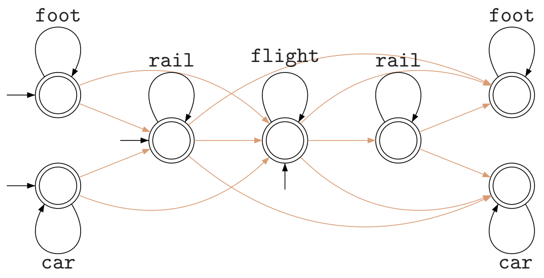
2.1 Models. Following [17], our multi-modal graphs are composed of different models for each mode of transportation. We briefly introduce each model and explain how they are combined.

In the *road network*, nodes model intersections and edges depict street segments. We either label edges by `car` for roads or `foot` for 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 realistic time-dependent model [39]. It consists of station nodes connected to route nodes. Trains are modeled between route nodes via time-dependent edges. Some station nodes are interconnected by time-independent foot paths. See [39] 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 [18]. 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 [15, 39]. 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 edges is computed from its geographical length and a walking speed of 4.5 kph.



(a) `foot-and-rail`.



(b) `hierarchical`.

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

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) [5]: 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, 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 [5].

2.3 Contraction Hierarchies (CH). Our algorithm is based on Contraction Hierarchies [24]. 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. Further acceleration techniques, such as stalling-on-demand [24], can be applied.

Partial Hierarchy. If the preprocessing is aborted prematurely, i. e., before all nodes are contracted, we obtain a partial hierarchy (PCH). Let $\mathrm{rank}(v) = \infty$ iff v is never contracted, then the same query algorithm as for Contraction Hierarchies is applicable and yields correct results. We call the subgraph of all uncontracted nodes the *core*, and the remaining (contracted) subgraph the *component*. Note that both the core and the 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 [2]. 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 *maximum node degree* encountered 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 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 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 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 $\mathrm{lbl}(e') = \mathrm{lbl}(e_1) \cdots \mathrm{lbl}(e_k)$. 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 $\mathrm{lbl}(e') = (\mathrm{lbl}_1, \ldots, \mathrm{lbl}_k)$. 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 $\mathrm{lbl}(v)$, 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 [23] and Timetable CH [22]. 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.

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 an easier witness search. We first turn toward preprocessing before we go into detail about the query algorithm.

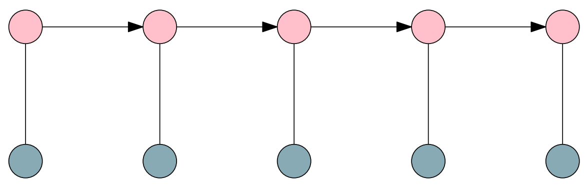
Preprocessing. The main reason behind the disadvantages discussed in Section 3.1 is the computation of shortcuts that span over boundaries of different modal networks. Instead, let Σ be the alphabet of labels of a multi-modal graph $\mathbf{G}$. We now process each subnetwork independently. We compute—in no particular order—a partial Contraction Hierarchy restricted to the subgraph $G_{\mathrm{lbl}} = (V_{\mathrm{lbl}}, E_{\mathrm{lbl}})$ (for every $\mathrm{lbl} \in \Sigma$). 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 $\mathrm{rank}(v) = \infty$, i. e., they are never contracted. Note that all other nodes have only incident edges labeled by lbl in **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 $\mathrm{lbl}(e)$, 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 [24]. 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.

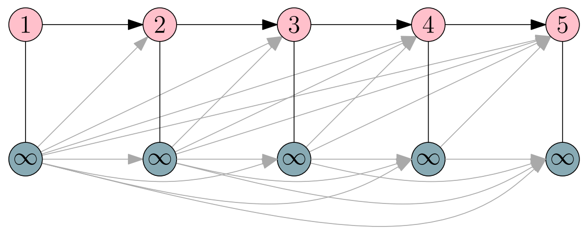
Our preprocessing results in a partial hierarchy for each modal network of **G**. Its transfer nodes are not contracted, thus, can be interpreted as staying at the top of the hierarchy. Recall that we call the subgraph induced by all nodes v with $\mathrm{rank}(v) = \infty$ the *core*. Because of the 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 **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, cp. [22]). It also hides information encoded in the timetable model (such as railway lines), further complicating query algorithms [11].

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 wrt. 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



(a) Input graph.



(b) Graph after contraction.

Figure 2: Contracting only route nodes in the realistic time-dependent rail model [39]. 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).

wrt. final states). Whenever we extract a tuple (v, q) from Q, we scan all edges $e = (v, w)$ in **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 **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 [19]. 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 (wrt. 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 [17], 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 [9, 16], 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 several improvements to our algorithm, some of which also apply to CH. Recall that whenever we contract a modal network, we never contract transfer nodes, even if they were of low importance in the context of the network. As a result, the number of added shortcuts may increase significantly. Thus, we stop the contraction process as soon as the *maximum node degree* 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.

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 20 %.

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 [12]. 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.

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 50 %. 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 2.1.

Finally, 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.

Combining all improvements yields a speedup of up to factor 4.9. (See Appendix A for detailed figures.)

4 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.

We assemble a total of six multi-modal networks where two are imported from [17]. Their size figures are reported in Table 1. For `ny-road-rail`, we combine New York's foot network with the public transit network operated by MTA [35]. We link bus and subway stops to road intersections that are no more than

Table 1: Comparing size figures of our input instances. The bottom two instances are taken from [17].

	Public Transportation		Road		
network	stations	connections	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 : 747
`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

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 1.5 km. 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 [17]. The data of the Western European and North American road networks (thus Germany and New York) was kindly provided to us by PTV AG [38] for scientific use. The timetable data of New York is publicly available through General Transit Feeds [25], while the data of the German and European railway networks was kindly provided by HaCon [29]. 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 1). 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).

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. Finally, the `everything` automaton allows arbitrary modal sequences in any order. See Figure 1 for transition graphs of `foot-and-rail` and `hierarchical`.

We evaluate both preprocessing and query performance. The contraction order is always computed according to the aggressive variant from [24]. 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 1 000 000 queries. We report the average number of: (1) nodes extracted 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 [24].

Evaluating Maximum Degree Limit. This experiment evaluates preprocessing performance with varying maximum degree α. We abort contraction when a node has degree greater than α. Table 2 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, the query time decreases (with smaller core size) up to $\alpha \approx 60$ 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 at $\alpha = 60$, hence, we use this value in subsequent experiments. Accordingly, we determine α

Table 2: Comparing preprocessing performance of UCCH on `de-road-rail` with varying maximum degree limit. For queries we use the `foot` automaton. We also report numbers for unconstrained unimodal CH.

	Preprocessing					Query			
	max degree	core-nodes	avg core-degree	shortcut-edges	time [min]	settled nodes	relaxed edges	touched edges	time [ms]
UCCH	29	25 629	11.0	42.5 %	7	12 886	23 741	142 526	4.79
	39	17 064	14.2	43.0 %	8	8 611	17 592	123 275	3.23
	45	15 495	15.5	43.2 %	9	7 833	16 508	121 852	3.04
	60	11 337	22.5	44.0 %	14	5 795	14 002	130 801	2.81
	79	9 728	30.3	44.7 %	18	5 032	13 536	152 577	3.06
	97	9 138	35.7	45.1 %	20	4 770	13 638	169 629	3.25
	167	8 101	56.3	46.6 %	42	4 357	14 837	243 132	4.37
PCH	29	11 337	12.7	41.7 %	7	5 910	11 983	75 194	2.00
PCH	31	6 764	14.9	41.8 %	8	3 640	7 980	54 199	1.33
CH	48	—	—	41.8 %	9	677	1 290	16 549	0.32

for all our instances. Note that α has little effect on the total number of shortcuts, but preprocessing time increases by factor 6 and query times vary up to 70 %.

Comparison to Unimodal CH. In Table 2 we also compare UCCH to CH when run on the unimodal road network. Computing a full hierarchy results in a maximum degree of 48, and query times are about a factor 9 faster. 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 11 337—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 6 764 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 2.1 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.

Preprocessing. Table 3 shows preprocessing figures for UCCH on all our instances. Besides the maximum degree, we evaluate the core in terms of total and fractional number of core nodes, the average degree 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 542 MiB of data. Note that 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 : 53) than we do, for example, on `wo-road-rail-flight` (1 : 1 248).

Comparing the preprocessing effort of UCCH to scaled figures of ANR, we observe that UCCH is 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.

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 4. 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 10, while on `wo-road-flight` we achieve query times of less than a millisecond—a speedup of over 45 000. Note that most of the time is spent inside the core (particularly, in the public transit network), which we do not accelerate. Appendix A 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.

Table 3: Preprocessing figures for UCCH and Access-Node Routing on the road subnetwork. Figures for the latter are taken from [17]. We scale the preprocessing time with respect to running time figures compared to Dijkstra.

	UCCH							Access-Node	
	max	core nodes		avg core-	shortcuts	shortcuts	time	space	time
network	degree	total	ratio	degree	percent	[MiB]	[min]	[MiB]	[min]
`ny-road-rail`	36	10 864	1 : 53	7.7	47.1 %	8	< 1	—	—
`de-road-rail`	60	11 337	1 : 446	22.5	44.0 %	62	14	—	—
`europe-road-rail`	95	40 153	1 : 752	24.4	38.9 %	323	42	—	—
`wo-road-rail-flight`	115	40 265	1 : 1 248	27.5	39.1 %	557	162	—	—
`de-road-rail(long)`	63	1 148	1 : 4 403	31.2	42.2 %	60	14	504	26
`wo-road-flight`	98	694	1 : 72 247	36.5	38.0 %	542	89	14 050	184

Table 4: Query performance of UCCH compared to plain multi-modal Dijkstra and Access-Node Routing. Figures for the latter are taken from [17]. We scale the running time with respect to Dijkstra.

		Dijkstra		Access-Node			UCCH		
		settled	time	settled	time	speed-	settled	time	speed-
network	automaton	nodes	[ms]	nodes	[ms]	up	nodes	[ms]	up
`ny-road-rail`	`foot-and-rail`	455 328	250	—	—	—	50 109	25.12	10
`de-road-rail`	`foot-and-rail`	2 668 177	2 053	—	—	—	75 333	38.11	54
`europe-road-rail`	`car-and-rail`	30 265 558	24 559	—	—	—	338 014	179.32	137
`wo-road-rail-flight`	`hierarchical`	36 454 341	36 207	—	—	—	455 279	236.90	153
`de-road-rail(long)`	`foot-and-rail`	2 735 426	2 075	13 524	3.45	602	12 531	3.35	619
`wo-road-flight`	`car-and-flight`	36 582 904	33 862	4 200	1.07	31 551	1 654	0.73	46 386

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 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 [17] 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

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In Pardalos and Rebennack [37], pages 230–241.

[2] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In M. Charikar, editor, *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 782–793. SIAM, 2010.

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

[4] C. Barrett, K. Bisset, M. Holzer, G. Konjevod, M. V. Marathe, and D. Wagner. Engineering Label-Constrained Shortest-Path Algorithms. In Demetrescu et al. [20], pages 309–319.

[5] C. Barrett, R. Jacob, and M. V. Marathe. Formal-Language-Constrained Path Problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[6] H. Bast. Car or Public Transport – Two Worlds. In S. Albers, H. Alt, and S. Näher, editors, *Efficient Algorithms*, volume 5760 of *Electronic Notes in Theoretical Computer Science*, pages 355–367. Springer, 2009.

[7] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. 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.

[8] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. 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.

[9] G. V. Batz, R. Geisberger, S. Neubauer, and P. Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Festa [21], pages 166–177.

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

[11] A. Berger, D. Delling, A. Gebhardt, and M. Müller–Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In ATMOS'09 [3].

[12] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. 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.

[13] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In Pardalos and Rebennack [37], pages 376–387.

[14] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Routing. In Demetrescu et al. [20], pages 73–92.

[15] D. Delling, B. Katz, and T. Pajor. Parallel Computation of Best Connections in Public Transportation Networks. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE Computer Society, 2010.

[16] D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.

[17] D. Delling, T. Pajor, and D. Wagner. Accelerating Multi-Modal Route Planning by Access-Nodes. In A. Fiat and P. 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.

[18] D. Delling, T. Pajor, D. Wagner, and C. Zaroliagis. Efficient Route Planning in Flight Networks. In ATMOS'09 [3].

[19] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[20] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

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

[22] R. Geisberger. Contraction of Timetable Networks with Realistic Transfers. In Festa [21], pages 71–82.

[23] R. Geisberger, M. Kobitzsch, and P. 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.

[24] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In C. C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[25] General Transit Feed. `http://code.google.com/p/googletransitdatafeed/`, 2010.

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

[27] A. V. Goldberg and C. 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.

[28] R. 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.

[29] HaCon - Ingenieurgesellschaft mbH. `http://www.hacon.de`, 2008.

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

[31] M. Holzer, F. Schulz, and D. Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[32] D. Kirchler, L. Liberti, T. Pajor, and R. W. 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)*, 2011.

[33] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mo-*

bilität - von der Forschung zur praktischen Anwendung, volume 22, pages 219–230. IfGI prints, 2004.

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

[35] Metropolitan Transportation Authority. `http://www.mta.info/default.html`, 1965.

[36] T. Pajor. Multi-Modal Route Planning. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009. Online available at `http://i11www.ira.uka.de/extra/publications/p-mmrp-09.pdf`.

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

[38] PTV AG - Planung Transport Verkehr. `http://www.ptv.de`, 2008.

[39] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2007.

[40] P. Sanders and D. 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.

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

[42] D. Wagner, T. Willhalm, and C. Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10(1.3):1–30, 2005.

A Further Experiments

In this appendix we present two further experiments. We analyze our improvements to UCCH and present more detailed figures of UCCH's query performance.

A.1 Improvements. In Table 5 we report figures on the improvements to UCCH described in Section 3.3. 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. Note that from the number of settled nodes we can deduce which of the improvements impact cache efficiency and which impact the search space.

A.2 In-Depth Analysis of Query Performance. Table 6 reports in-depth figures for the UCCH query. We see that a very large fraction of the query is spent on the public transportation part of the multi-modal network: up to 91 % of the settled nodes and up to 85 % 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) 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 the `de-road-rail` instance with the figures of Table 2 (where we use the same instance but with the smaller `foot` automaton), we observe that the `foot-and-rail` automaton yields a factor 2 slowdown. The reason is that the `foot-and-rail` automaton actually has two "foot-states" (cf. Figure 1), and thus, has to do twice the work on the road subnetwork.

Table 5: 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), and only forward search on the core (fo).

network	automaton	improvement	settled nodes	time [ms]	speed-up
europe-road-rail	car	none	49 069	67.76	—
		rn	49 069	34.30	2.0
		rn,re	49 069	28.70	2.4
		rn,re,bi	31 995	18.91	3.6
		rn,re,fo	24 586	13.76	4.9
wo-road-rail-flight	car	none	36 942	52.91	—
		rn	36 942	27.20	1.9
		rn,re	36 942	22.83	2.3
		rn,re,bi	30 868	19.22	2.8
		rn,re,fo	18 553	10.98	4.8

Table 6: 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.	—	—	225	—	—
		road-core	415 989	227.45	10 262	5.31	43
		rail	39 339	22.55	39 339	19.81	1.1
de-road-rail	foot-and-rail	road-comp.	—	—	173	—	—
		road-core	2 599 364	1991.3	6 678	5.60	336
		rail	68 813	61.73	68 813	32.51	1.9
europe-road-rail	car-and-rail	road-comp.	—	—	206	—	—
		road-core	29 973 823	24 250	48 965	34.73	698
		rail	291 735	309.26	291 735	144.59	2.1
wo-road-rail-flight	hierarchical	road-comp.	—	—	214	—	—
		road-core	36 059 431	35 794	64 509	43.45	767
		rail	393 927	410.90	393 927	192.60	2
		flight	982	1.90	928	0.86	2.1