

# Accelerating Multi-Modal Route Planning by Access-Nodes<sup>\*</sup>

Daniel Delling, Thomas Pajor, and Dorothea Wagner

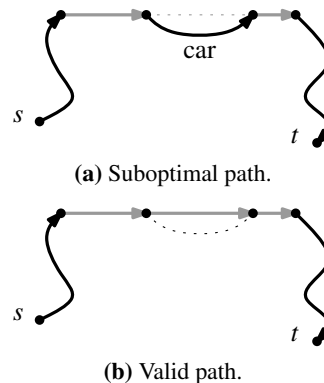
Department of Computer Science, Universität Karlsruhe (TH), P.O. Box 6980, 76128 Karlsruhe,  
Germany. {delling,pajor,wagner}@informatik.uni-karlsruhe.de

**Abstract.** Recent research on fast route planning algorithms focused *either* on road networks *or* on public transportation. However, on the long run, we are interested in planning routes in a *multi-modal* scenario: we start by car to reach the nearest train station, ride the train to the airport, fly to an airport near our destination and finally take a taxi. In other words, we need to *incorporate* public transportation into road networks. However, we do not want to switch the type of transportation too often. We end up in a *label constrained* variant of the shortest path problem. In this work, we present a first efficient solution to a restricted variant of this problem including experimental results for transportation networks with up to 125 Mio. edges.

## 1 Introduction

Computing the quickest path in graphs modeling transportation networks is one of the show-pieces of applied algorithms. In general, DIJKSTRA's algorithm [1] finds a shortest (or quickest) path between a given source  $s$  and target  $t$ . Unfortunately, the algorithm is far too slow to be used on huge datasets which appear frequently in route planning. Thus, several speed-up techniques have been developed that compute additional data during a *preprocessing* step in order to speed-up queries during the *online* phase.

However, all developed techniques so far suffer from one drawback: they only work either in road or railway networks. On the long run, we are interested in multi-modal queries where we change the type of transportation along our journey. Unfortunately, it is not sufficient to merge all networks and compute



**Fig. 1:** Motivation of a label-constrained shortest path: while the path at the top is the quickest connection, it requires us to use a car (black) between two trains (gray). The path to the bottom, however, is preferable since we do not need to leave the train.

<sup>\*</sup> Partially supported by the Future and Emerging Technologies Unit of EC (IST priority – 6th FP), under contract no. FP6-021235-2 (project ARRIVAL) and the DFG (project WA 654/16-1).

quickest paths in the resulting bigger network: the quickest path may force us to change the type of transportation too frequently. See Figure 1 for an example. A possible approach to this problem is the LABEL CONSTRAINED SHORTEST PATH PROBLEM. The idea is as follows. Each edge gets a label assigned depicting the type of transportation network it represents. Then, only a path between  $s$  and  $t$  is valid if certain constraints are fulfilled by the labels along the path. In this work, we present an approach how to accelerate multi-modal queries by skipping main parts of the network without losing correctness.

**Related Work.** Theoretical results on the hardness of the LABEL CONSTRAINED SHORTEST PATH PROBLEM can be found in [3, 4]. Experimental evaluations of basic algorithms are given in [5], while basic speed-up techniques like A\* [6] and bidirectional search [7] have systematically been examined in [8]. Route planning in uni-modal scenarios has been undergoing a rapid development in recent years with the fastest techniques yielding query times of several microseconds in road networks [9]. For a recent overview over uni-modal speed-up techniques, we direct the interested reader to [2]. However, to the best of our knowledge, there exists no route planning algorithm that can answer a multi-modal query in a huge combined transportation network within milliseconds. Since our approach is closely related to (uni-modal static) Transit-Node Routing [9], we briefly recap this technique. TNR selects a subset  $\mathcal{T} \subset V$  (normally 10 000 nodes) of so called transit nodes and stores distances between them in a table. Moreover, each node  $v \in V$  stores the distances to all *relevant* transit nodes, called *access-nodes*. Then, with good choice of  $\mathcal{T}$ , a long-range query can be reduced to three table lookups. In order to decide whether a query is a long-range query, a *locality* filter is introduced. In case  $s$  and  $t$  are too close to each other, an arbitrary speed-up technique is applied. The percentage of global queries can be increased by introducing several *layers* of transit nodes.

**Our Contribution.** In this work, we present an efficient approach to a special case of multi-modal route planning. We assume that we want to use the road network only at the beginning and the end of a journey and that the public transportation network is much smaller than the road network. Then, by adapting some ideas from Transit-Node Routing, we may “skip” the road network with a table lookup and restrict the search to the much smaller public transportation network (PTN). We present how to compute so-called *access-nodes* to the PTN for each node efficiently. With this information at hand, we are able to present Access-Node Routing, accelerating multi-modal queries (in our scenario) by more than 4 orders of magnitude. However, the main achievement is that we are able to separate the public transportation network from the road network in a multi-modal context. This allows us to run different query algorithms on the public transportation and the road networks. Although, in this work we use an augmentation of DIJKSTRA’s algorithm on the public transportation network, it would also be possible to use a speed-up technique or even multi-criteria search.

This work is organized as follows. Section 2 settles basic definitions we need for this work. In Section 3 we briefly recap existing and new approaches for modeling transportation networks. Moreover, we show how to obtain a multi-modal network and

present that a *label-constrained* variant of the shortest path problem is a possible approach for better routes in multi-modal networks. It turns out that we need an automaton that restricts the number of network changes. Starting from analyzing our networks and automata, we develop our main contribution (Section 4) of this work: Access-Node Routing. By running several experiments on transportation networks with up to 125 Mio. edges (Section 5), we show that Access-Node Routing is up to 31 000 times faster than a label-constrained variant of DIJKSTRA. This value is achieved without using a speed-up technique within the public transportation network. Section 6 concludes our work with a brief summary and interesting open questions.

## 2 Preliminaries

A *graph* is a tuple  $G = (V, E)$  consisting of a finite set  $V$  of *nodes* and a set  $E \subseteq V \times V$  of *edges* which are ordered pairs  $(u, v)$  if the graph is *directed*. The node  $u$  is called the *tail* of the edge,  $v$  the *head*. The reverse graph  $\overleftarrow{G} = (V, \overleftarrow{E})$  is the graph obtained from  $G$  by substituting each  $(u, v) \in E$  by  $(v, u)$ . The main difference between uni- and multi-modal route planning is that the nodes and edges of a graph are *labeled* by a finite set  $\Sigma$  of symbols which is often called an *alphabet*. The node-label is denoted by  $\text{lab} : V \rightarrow \Sigma$ , the edge-label by  $\text{lab} : E \rightarrow \Sigma$ . Throughout the whole work we restrict ourselves to directed *labeled* graphs which are weighted by a length function  $\text{len}$  depicting the travel time from  $u$  to  $v$ . Depending on the edge-label, edge weights may be time-independent or time-dependent. For time-independent weights, we use a positive length function  $\text{len} : E \rightarrow \mathbb{R}^+$ , while for time-dependent edges, we use a time-dependent length function  $\text{len} : E \rightarrow \mathbb{F}$  with the function space  $\mathbb{F}$  consisting of positive *periodic* functions  $f : \Pi \rightarrow \mathbb{R}^+$ ,  $\Pi = [0, p]$ ,  $p \in \mathbb{N}$  such that  $f(0) = f(p)$  and  $f(x) + x \leq f(y) + y$  for any  $x, y \in \Pi, x \leq y$ . Note that these functions respect the FIFO property [10], and hence our networks fulfill the FIFO property as well. In the following, we call  $\Pi$  the *period* of the input. The upper bound of  $f$  is denoted by  $\bar{f} = \max_{x \in \Pi} f(x)$ , the lower by  $\underline{f} = \min_{x \in \Pi} f(x)$ . Note that we can obtain a time-independent labeled lower/upper bound graph  $\underline{G}/\overline{G}$  from  $G$  by replacing each edge function with their lower/upper bounds.

A sequence  $w := \sigma_1, \sigma_2, \dots, \sigma_k$  of symbols from  $\Sigma$  is called a *word*. The *length* of a word is the number of symbols it is composed of. A not necessarily finite set  $L$  of words over  $\Sigma$  is called a *language* over  $\Sigma$ . A *non-deterministic finite automaton*  $\mathcal{A} := (Q, \Sigma, \delta, S, F)$  consists of a finite set  $Q$  of states, an alphabet  $\Sigma$ , the transition function  $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ , a set  $S$  of initial states and a set  $F$  of final states. A language  $L$  is called *regular* if it can be accepted by a non-deterministic finite automaton [11, 12]. Throughout the whole work we restrict ourselves to regular languages.

## 3 Models and Basic Algorithms

We now briefly present how to model transportation networks as graphs and how to construct a multi-modal graph from these ingredients. Finally, we present why the LABEL CONSTRAINED SHORTEST PATH PROBLEM is useful for our application of reasonable route planning in multi-modal scenarios.

### 3.1 Models

Modeling a *road network* or *foot networks* as a graph  $G_{\text{road}}$  is straightforward. Junctions are modeled as nodes and an edge  $e = (u, v)$  between two junctions  $u, v \in V$  is inserted if and only if a road segment from  $u$  to  $v$  exists in the road network. The edge weights  $\text{len}(e)$  in the road network represent the average travel time on the specific road segment. For foot edges, we assign a weight based on the assumption that we walk the segment with 4 km/h on average. Note that our foot networks compared to our road networks have the same node set whereas the edge sets may differ due to motorways (not open to pedestrians), one-ways and pedestrian zones (not open to traffic). For *railway networks*  $G_{\text{rail}}$ , we use the *realistic* time-dependent approach as presented in [13] where edges are time-dependent and depict several trains running on the same route from one station to another. We model *flight networks*  $G_{\text{flight}}$  as follows. Our realistic raw data incorporates two major flight alliances: StarAlliance [14] and Oneworld [15]. Thus, for each airport we introduce a supernode and two departure and arrival nodes—one for each flight alliance. Edges between supernodes and departure nodes model check-in, arrival-departure edges the changing of planes (with different weights for transfers between flight alliances) and arrival-supernode edges the check-out. Time-dependent edges between two airports  $A$  and  $B$  depict direct flights between  $A$  and  $B$ . Depending on the fact which flight alliance operates the respective flight, head and tail are chosen as the corresponding departure and arrival nodes. See [16] for details.

Let  $\Sigma = \{\text{road}, \text{foot}, \text{rail}, \text{flight}\}$  be an alphabet. We construct a *multi-modal* network from our four types of transportation networks  $G_\sigma = (V_\sigma, E_\sigma)$ ,  $\sigma \in \Sigma$  by first labeling each node  $u$  and each edge  $e$  with a label  $\text{lab}(u), \text{lab}(e) \in \Sigma$ . Then, we unify the node and edge-sets to a graph  $G_{\text{multi}} = (\bigcup_{\sigma \in \Sigma} V_\sigma, \bigcup_{\sigma \in \Sigma} E_\sigma) =: (V_{\text{multi}}, E_{\text{multi}})$ . In order to connect the networks among each other, we introduce so called *link-edges*  $(u, v)$  with  $\text{lab}((u, v)) = \text{link}$  and  $\text{lab}(u) \neq \text{lab}(v)$ . These edges depict possible switches from one transportation type to another. So, we have  $\Sigma = \{\text{road}, \text{foot}, \text{rail}, \text{flight}, \text{link}\}$ . We connect each station node (railways) and supernode (flights) with its closest node from the road network. Moreover, we connect each airport with its closest train stations by a link edge. However, head and tail must not be more than 5 km away from each other.

### 3.2 The Label-Constrained Shortest Path Problem

The LABEL CONSTRAINED SHORTEST PATH PROBLEM is an augmentation of the classic SHORTEST PATH PROBLEM. The idea is that only such paths between  $s$  and  $t$  are valid that form a word of a language  $L$ . More precisely, given an alphabet  $\Sigma$ , a language  $L \subset \Sigma^*$ , a weighted, directed graph  $G = (V, E)$  with  $\Sigma$ -labeled edges and source and target nodes  $s, t \in V$ , we ask for a shortest path  $P = (u_1, \dots, u_k)$  from  $s = u_1$  to  $t = u_k$ , where the sequence of labels along the edges of the path forms a word of  $L$ . In other words,  $\text{lab}((u_1, u_2)) \dots \text{lab}((u_{k-1}, u_k)) \in L$  must hold.

It turns out that the complexity of this problem depends on the restrictions to  $L$  [4]. In our case, where  $L$  is regular, the problem remains polynomially solvable. Then, we can use a straightforward adaption of DIJKSTRA's algorithm for computing a label-constrained shortest  $s$ - $t$  path. Besides the inputs for a normal DIJKSTRA, we also

require an automaton  $\mathcal{A} := (Q, \Sigma, \delta, S, F)$  that accepts our language  $L$ . We initialize pairs  $(s, q)$ ,  $q \in S$  with  $\text{dist}_s((s, q)) = 0$  and insert them with key 0 into a priority queue. Any other node-state tuple  $(u, q')$  is initialized with  $\text{dist}_s((u, q')) = \infty$ . Then, we remove a node-state tuple  $(u, q')$  with minimum key from the queue, relax all outgoing edges  $e = (u, v)$ , determine all states  $q'' \in \delta(q', \text{lab}(e))$  and check whether  $\text{dist}_s((u, q')) + \text{len}(e) < \text{dist}_s((v, q''))$  holds. If so, we update the distance label and enqueue  $(v, q'')$ . We may stop the search as soon as we settle a tuple  $(t, q')$  with a final state  $q' \in F$ .

Note that this procedure can be adapted to a time-dependent scenario easily: if we want to compute the shortest path for a given departure time  $\tau$ , we simply have to evaluate edge weights for their correct departure time when relaxing them. If we want to compute the shortest path for all departure times, we have to use a label-correcting algorithm that propagates functions instead of constants through the network (see [17, 18] for details).

**A Special Case.** When planning a typical multi-modal voyage, we observe that in most cases we start in the road or foot network. Then, we enter the public transportation network without using the road network again (except for transfers) until the end of the journey. There, we either use a taxi or rental car or go by foot to reach our final destination. This observation is characterized by the following definition on languages.

**Definition 1 (Enclose Property).** *Let  $L$  be a regular language over the alphabet  $\Sigma$  of edge-labels. If  $L$  is of the form  $L = \sigma_{r_1}^* l \sigma_t^* l \sigma_{r_2}^*$ , where  $\sigma_{r_1}, \sigma_{r_2} \in \{\text{road}, \text{foot}\}$  and  $\sigma_t \in \{\text{rail}, \text{flight}\}$  and  $l = \text{link}$ , we say that  $L$  fulfills the enclose property.*

In other words, the public transportation is enclosed by the road network part. An example for an automaton for such languages only allows railway connections enclosed by foot-edges. In the following, we denote this automaton by `foot-and-rail`. By substituting `foot` by `road` and `rail` by `flight`, we obtain a second automaton which we call `road-and-flight`.

**Trees and Profile-Graphs.** In the following, we build label-constrained shortest path trees and profile graphs. As discussed in Section 3.1, our railway- and flight-networks are time-dependent. So, running a label-constrained DIJKSTRA with a given departure time  $\tau$  from a node  $u$  until the priority queue is empty yields a tree rooted at  $u$ . Similarly, running a label-constrained label-correcting algorithm (cf. [17, 18]) for all possible departure times from  $u$  yields a so-called profile-graph. For each node  $v$ , we obtain a function depicting the travel time from  $v$  to  $u$  (the profile) with changing parent node during the period.

## 4 Access-Node Routing

Analyzing the networks obtained from our multi-modal approach, we observe that most part of the graph is made up of road segments and, hence, a multi-modal DIJKSTRA spends most its time settling road nodes. Access-Node Routing (ANR) accelerates

queries with an automaton fulfilling the enclose property as described in Section 3.2. The main idea is to precompute distances to all *relevant* access points to the public transportation network. With this information at hand, we may “skip” the road network and restrict the search to a much smaller network. In the following, we define access-nodes, how they can be computed efficiently and the resulting query algorithm. It turns out that space consumption of this approach is rather high, hence, we present how to reduce space consumption by using the concept of contraction [19]. Note that in the following, we explain how to skip the road network, however, this approach can also be used to skip a foot network. Experiments for both variants can be found in Section 5.

#### 4.1 Access-Nodes

Not every node in the public transportation network is suited as “access-node”. For example, in the flight network the departure and arrival nodes are used to model internal procedures at airports and should not be accessed directly from the road network. More precisely, a node  $v$  is called *access-node candidate* if  $\text{lab}(v) \neq \text{road}$  and at least one incident edge is a link-edge. The set of all access-node candidates is denoted by  $A$ . In our case the set  $A$  includes exactly all station nodes regarding the railway network and all supernodes of the flight network (cf. Section 3). Nodes  $v \in A$  can be interpreted as entry (or exit) points to/from the public transportation network. Computing distances from every road network node to every access-node would require too much space. Moreover, not every entry point to the public transportation network is mandatory for correct shortest paths. More precisely, a node  $v \in A$  with  $\text{lab}(v) \neq \text{road}$  is an access-node for all nodes  $u$  with  $\text{lab}(u) = \text{road}$  if there exists another node  $w \in A$  with  $\text{lab}(w) \neq \text{road}$  for which the shortest  $u$ - $w$  path (for at least one departure time) uses  $v$  to enter the public transportation network (i.e., all ancestors of  $v$  are road-labeled). The set of (forward) access-nodes for a node  $u$  is denoted by  $\vec{A}(u)$ . Note that we can define *backward* access-nodes  $\overleftarrow{A}(u)$  analogously.

#### 4.2 Computing Access-Nodes

In the following, we explain how to compute forward access-nodes for each  $v \in V_{\text{multi}}$  with  $\text{lab}(v) = \text{road}$  efficiently. Computing backward access-nodes is done analogously. In general, we present two approaches for computing access-nodes, a forward and an inverse approach, which we both explain in detail.

For the *forward* approach, we construct a profile-graph (with an automaton fulfilling the enclose property)  $T_v$  from each  $v$  with  $\text{lab}(v) = \text{road}$ . Whenever we settle a candidate node  $a \in A$ , we add  $a$  to  $\vec{A}(v)$  if a parent—with respect to  $T_v$ —of  $a$  is a road node. We may stop the search as soon as all nodes  $u$  in the priority queue are *covered*. A node  $u$  is called *covered* if at least one of its ancestors—with respect to  $T_v$ —is a non-road node. Note that this approach requires two profile-graph constructions (forward and backward) per road node. Hence, this approach is only useful if the number of access-node candidates is high and thus, the construction terminates early.

For the *backward* approach, we compute for each access-node candidate  $a \in A$  all nodes  $u \in A^{-1}(a)$ . Therefore, we construct a backward profile-graph *not* relaxing edges whose tail is a road node. By this, we obtain travel time functions from each  $a' \in A \setminus$

$\{a\}$  to  $a \in A$ . Next, we construct a backward shortest path forest: we initialize  $a$  with distance 0 and any  $a'$  with an upper bound on the travel time functions to  $a$ . This time, we do not relax public transportation edges. We may stop the search as soon as all nodes  $u$  in the priority queue have a node  $a' \in A \setminus \{a\}$  as ancestor. Finally, we add all nodes with ancestor  $a$  to  $A^{-1}(a)$ . After performing this task for all candidates, we obtain  $\vec{A}(v)$  for all road nodes by inverting the relation  $A^{-1}(\cdot)$ . Note that by bounding the functions, we may compute a *superset* of the actual access-nodes.

### 4.3 Query

With the preprocessed data at hand, we can skip the road network part for multi-modal queries with automata fulfilling the `enclose` property. Assume  $\text{lab}(s) = \text{lab}(t) = \text{road}$ . In a first step, we add node-state tuples  $(a_s, q_s)$  (with corresponding distances) for all  $a_s \in \vec{A}(s)$  and all  $q_s \in Q$  obtained by a `link`-labeled transition originating from an arbitrary initial state of the automaton to the priority queue. Node-state tuples  $(a_f, q_f)$  for all  $a_f \in \overleftarrow{A}(t)$  and all  $q_f \in Q$  such that there is a `link`-labeled transition from  $q_f$  to a final state in the automaton are accumulated in a target node set  $T$ . In a second step, we run a multi-modal query in the public transportation network to  $T$  not relaxing edges whose head is a road node. We may stop the search if all  $(a_f, q_f) \in T$  have a final label assigned or the priority queue runs empty. We end up having distances  $\text{dist}_s((a_f, q_f))$  for all  $(a_f, q_f) \in T$ . Then, the length of the shortest path from  $s$  to  $t$  is  $\min_{(a_f, q_f) \in T} \{\text{dist}_s((a_f, q_f)) + \text{dist}(a_f, t)\}$ . Note that we can run time-, profile-, or even multi-criteria-queries within the public transportation network. The main gain we obtain is skipping the road network.

Note that the paths found by our access-node query algorithm must contain a public transportation node. While for long-range queries this may be a meaningful restriction (nobody wants to drive 30 hours by car), low-range queries may contain *only* road nodes. Fortunately, computations of quickest paths in road networks can be done in microseconds. So, we run a *check-query* with the CHASE-algorithm [20] that outputs the length of the quickest path if only the road network is used.

### 4.4 Core Access-Node Routing

One of the main drawbacks of pure Access-Node Routing is its high space consumption. For each node  $v$  of the road network we store two sets of access-nodes together with corresponding distances. Although the number of access-nodes per road node is relatively small, it is not necessary to store these sets for all nodes but only for important ones, i.e., the *core* of the graph. We use a contraction routine that removes unimportant nodes and adds shortcuts in order to preserve distances between core nodes.

*Preprocessing* for Core-Based Access-Node Routing is done in two steps. First, we contract the graph by a node-reduction followed by an edge-reduction which we obtain by a straightforward adaption of the concepts presented in [20]. However, in order to preserve correctness, we may only remove road nodes having no incident link-edges. As a result, the embedded time-dependent public transportation network is fully contained in the core. See [19] for more details on uni-modal contraction. In a second step, we compute forward and backward access-nodes for all core nodes  $u$  with  $\text{lab}(u) = \text{road}$ .

The *query* is a three-phase algorithm. During phase 1, we run a bidirectional DIJKSTRA, not relaxing edges outgoing from core-nodes. Whenever the forward search settles a core node, we add it to a set  $S$ . A set  $T$  is maintained for backward search analogously. Phase 1 ends if both priority queues are empty. Then, a (two-phase) access-node query is started with  $S$  as source nodes and  $T$  as target nodes.

#### 4.5 Comparison to Transit-Node Routing

As already mentioned, Access-Node Routing (ANR) adapts ideas from Transit-Node Routing (TNR). Recall that TNR has three ingredients: a table for storing distances between transit nodes, stored distances for each node to its relevant transit (access) nodes, and a locality filter for deciding whether  $s$  and  $t$  are far away enough from each other (cf. Section 1). In fact, ANR can be interpreted as a multi-modal variant of TNR. Our access-node candidates become transit nodes and we store distances to the relevant access-nodes. Unfortunately, due to poor upper bounds on the travel time functions of the time-dependent edges of the public transportation network, we cannot make up an efficient locality filter as for TNR: we have to run a local path query in almost all cases. In fact, the only time we do not need to run a check query is when  $s$  and  $t$  are not in the same component of the road network.

Note that TNR uses a (sophisticated) forward approach for determining the relevant access-nodes. Unfortunately, our public transportation networks tend to be sparse and they are time-dependent as well. As a result, the inverse approach turns out to be better for our scenario. Summarizing, ANR can be interpreted as a variant of TNR with higher flexibility but worse query performance. However, TNR has neither been adapted to time-dependent nor label-constrained route planning so far. Both aspects are crucial preconditions for route planning in our scenario.

## 5 Experiments

We conducted our experiments on one core of an AMD Opteron 2218 running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 32 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2, using optimization level 3. Our implementation is written in C++ using solely the STL at some points. As priority queue we use a binary heap.

*Inputs.* We use two different networks and two automata. The first network depicts the foot network of Germany ( $|V| \approx 4.5$  Mio,  $|E| \approx 11.2$  Mio) merged with all long distance trains from the timetable of the winter period 2000/2001 (498 stations and 18069 connections). This includes InterRegio (IR), InterCity (IC) and InterCityExpress (ICE) trains. We call the resulting graph `de-road-rail(long)` and apply the `foot-and-rail` automaton to this input. Our second input consists of the road network of Western Europe ( $|V| \approx 30$  Mio,  $|E| \approx 73$  Mio) and North America (including Canada,  $|V| \approx 20$  Mio,  $|E| \approx 51$  Mio) and the flight network of both flight alliances (359 airports with 32 621 flights). We use this graph together with the `road-and-flight` automaton. In the following, this input is referenced to by `na-eur-road-flight`. Note that second input has about 50 Mio. nodes and 125 Mio. edges.

*Methodology.* In the following, we report preprocessing times and the overhead of the preprocessed data in terms of *additional* bytes per node. We evaluate query performance by *random queries*, i.e., the nodes  $s$  and  $t$  are picked uniformly at random. Since public transportation networks are time-dependent, we additionally need a departure time  $\tau$ , which we pick uniformly at random as well. We provide the average number of settled nodes, i.e., the number of nodes extracted from the priority queue and the average query time. Unless otherwise stated, all figures in this paper are based on 1 000 000 random  $s$ - $t$  queries and refer to the scenario that only the lengths of the shortest paths have to be determined, without outputting a complete description of the paths. Efficient methods for unpacking table lookups have been published in [9]. Local paths are computed by CHASE [20] which is a combination of Contraction Hierarchies [19] and Arc-Flags [21].

**Preprocessing.** In Section 4, we presented two approaches to compute access-nodes: a forward variant and an inverse approach. It turns out that the former is too slow for our multi-modal inputs, hence, we only use the inverse approach. Table 1 reports key figures for computing access-nodes. Note that `na-eur-road-flight`, we only use the core-based approach (cf. Section 4.4). Also note that we report the preprocessing effort for CHASE. Since we need to keep the road network as a *uni-modal* graph in memory, the space consumption here *includes* the graph.

Regarding `de-road-rail(long)`, the average number of forward and backward access-nodes per road node is 32.4 resp. 20.9. Thus, 32.4 railway stations are important (on average) to enter the railway network. While this seems to be a very high number (even more if we consider that these railway stations have to be reached by foot), there are two good reasons for this. First, the railway network is sparsely embedded into the road network, thus, for a single road network node a lot of stations are important at least once a day. Second, long distance trains do not operate very frequently on some parts of the network. As a consequence, the upper bounds are of poor quality yielding many unnecessary access-nodes. The same effect can be observed in the `na-eur-road-flight` network, as flights are even more infrequent. This makes it attractive to cover far distances by car, thus, including many (also far away) airports into the set of relevant access-nodes.

**Table 1:** Preprocessing Figures for (Core-Based) Access-Node Routing. We report the number of access-node candidates, the average number of forward and backward access-nodes per road node and the preprocessing time for computing access-nodes as well as the additionally required space per node. We also report preprocessing effort for CHASE. Note that for CHASE we report the space consumption *including* the graph.

network	Access-Node Routing							CHASE	
	core-based	AN-cand.	forward access-nodes	backward access-nodes	time [min]	space [B/n]	time [min]	space [B/n]	
<code>de-road-rail(long)</code>		473	32.4 (6.8%)	20.9 (4.4%)	143	435	17	56	
<code>de-road-rail(long)</code>	✓	473	31.0 (6.5%)	19.7 (4.1%)	26	56	17	56	
<code>na-eur-road-flight</code>	✓	359	118.7 (33.0%)	119.1 (33.1%)	161	224	233	57	

Preprocessing times are in the range of several hours (between 26 minutes for the core of the German network and almost three hours for the core of the continental network). As expected, switching to Core-Based Access-Node Routing drastically reduces both preprocessing time and the required space for the access-nodes. The additional effort for preprocessing CHASE is comparable to ANR. We need 56 bytes per node (including a uni-modal graph) and preprocessing times are within a reasonable range.

**Query Performance.** Table 2 reports query performance of a multi-modal DIJKSTRA, ANR, and of our CHASE check-query for all our inputs. Note that the figures for plain DIJKSTRA are based on 1 000 random queries. We observe a drastic drop in both the number of settled nodes and the query time when using Access-Node Routing. In `de-road-rail(long)` we observe that the query time increases from 3.9 to 5.8 milliseconds when switching to the core-based variant. This is due to the computational overhead of the initialization phase. The performance of Access-Node Routing on `na-eur-road-flight` is better than on `de-road-rail(long)`. This is due to the fact that the flight network is significantly smaller than the railway network embedded in `de-road-rail(long)`. Moreover, the number of access-nodes per road node is only 14.2 whereas in Germany it is twice that much. The highest speed-up of 31 551 is achieved when applying our biggest input. We are able to perform intercontinental queries with an average time of 2.3 ms compared to over 72 sec when the standard algorithm is used. We also observe that the running time for CHASE is negligible compared to the execution time of ANR: query times are between 51 and 111  $\mu$ s for settling only very few nodes.

**Three Phases.** The query algorithm for Core-Based Access-Node Routing is made up of three distinct phases (cf. Section 4.4). Table 3 reports the distribution of the running time among the particular phases of the query algorithm. We observe that the public transportation query makes up the major part of the running time (between 73.9% and 96.2% depending on the network). This is expected since we do not use a speed-up technique within the public transportation network. The time for looking up the access-nodes is negligible as it is less than 2% on `de-road-rail(long)` and 7.9% on `na-eur-road-flight` of the running time. This is due to the fact that the number of average access-nodes per road node is higher than on the other two networks (cf. Table 1).

**Table 2:** Query performance of (Core-Based) Access-Node Routing without local queries (i.e., all shortest paths use the transportation network) compared to plain multi-modal DIJKSTRA. Note that the figures for the latter are based on only 1 000 random queries.

network	DIJKSTRA		Access-Node Routing			Local (CHASE)		
	settled nodes	time [ms]	core-based	#settled nodes	time [ms]	speed-up	#settled nodes	time [ms]
<code>de-road-rail(long)</code>	2483030	3492		13295	3.9	895	168	0.111
<code>de-road-rail(long)</code>	2483030	3492	✓	13524	5.8	602	168	0.111
<code>na-eur-road-flight</code>	46244703	72566	✓	4200	2.3	31551	75	0.051

**Table 3:** In-depth analysis of *Core-Based* Access-Node Routing. This table reports the distribution of query time among the particular phases of the query algorithm: the bidirectional initialization, the table-lookups of access-nodes, and DIJKSTRA on the public transportation network. Furthermore, we report the amount of local queries (paths that do not use the transportation network) when generating 1000 (*de-road-rail(long)*, *ny-de-road-flight*) and 100 (*na-eur-road-flight*) random queries.

Network	QUERY				total [ms]	local queries
	initialization phase	access-node lookup	public transport			
<i>de-road-rail(long)</i>	0.15 (2.4%)	0.08 (1.4%)	5.87 (96.2%)		5.8	2.3%
<i>na-eur-road-flight</i>	0.42 (18.2%)	0.18 (7.9%)	1.70 (73.9%)		2.3	24%

Furthermore, we report the relative number of local paths, i.e., how many quick-est paths do not use the public transportation network. We observe a great variation depending on the network. While in *de-road-rail(long)* only 2.3% of the queries do not use the railway, while for *na-eur-road-flight* the amount of local queries is 24%. Still, as observable in Table 2, running a CHASE query comes almost for free compared to the running time of the multi-modal query. In general, the high number of local queries is also due to the fact that we pick  $\tau$  randomly. By computing a departure time with minimal travel time via profile-searches [18], the amount of local queries most probably will decrease. Note that in our scenario, it is sufficient to use profile-searches within the public-transportation network in order to answer such requests.

## 6 Conclusion

In this work we presented a first efficient approach to a special variant of multi-modal routing in large transportation networks. Using the reasonable assumption that we want to use a car only at the beginning and the end of the journey, we can split the search by adapting some ideas from Transit-Node Routing. The key idea is to skip the road network during the query. Experiments on real-world multi-modal networks with up to 125 Mio. edges confirm the feasibility of our approach: random queries are up to 31 000 times faster than with a multi-modal variant of DIJKSTRA. We want to stress out that we could achieve further speed-ups by using a better routing algorithm than DIJKSTRA within the public transportation network.

Regarding future work, it would be interesting to develop a multi-modal speed-up technique that does not restrict the choice of the automaton combined with reasonable speed-ups. Preliminary results from [16] confirm that the adaption of some speed-up techniques, e.g., Contraction Hierarchies, Arc-Flags or Landmarks, is much more challenging than one might expect. So, such a technique is non-trivial. Furthermore, we want to develop better foot networks and accelerate public transportation queries.

**Acknowledgments.** We would like to thank Martin Holzer and Christos Zaroliagis for interesting discussions on multi-modal route planning. Moreover, we thank PTV AG and HaCon for providing us with real-world data for scientific use.

## References

1. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* **1** (1959) 269–271
2. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering Route Planning Algorithms. In Lerner, J., Wagner, D., Zweig, K.A., eds.: *Algorithmics of Large and Complex Networks*. Volume 5515 of *Lecture Notes in Computer Science*. Springer (2009) 117–139
3. Mendelzon, A.O., Wood, P.T.: Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing* **24**(6) (1995) 1235–1258
4. Barrett, C., Jacob, R., Marathe, M.V.: Formal-Language-Constrained Path Problems. *SIAM Journal on Computing* **30**(3) (2000) 809–837
5. Barrett, C., Bisset, K., Jacob, R., Konjevod, G., Marathe, M.V.: Classical and Contemporary Shortest Path Problems in Road Networks: Implementation and Experimental Analysis of the TRANSIMS Router. In Möhring, R.H., Raman, R., eds.: *ESA 2002*. LNCS, vol. 2461, pp. 126–138. Springer, Heidelberg (2002).
6. Hart, P.E., Nilsson, N., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* **4** (1968) 100–107
7. Dantzig, G.B.: *Linear Programming and Extensions*. Princeton University Press (1962)
8. Barrett, C., Bisset, K., Holzer, M., Konjevod, G., Marathe, M.V., Wagner, D.: Engineering Label-Constrained Shortest-Path Algorithms. In: *Shortest Paths: Ninth DIMACS Implementation Challenge*. DIMACS Book. American Mathematical Society (2009) To appear.
9. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast Routing in Road Networks with Transit Nodes. *Science* **316**(5824) (2007) 566
10. Orda, A., Rom, R.: Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM* **37**(3) (1990) 607–625
11. Kleene, S.C.: Representation of Events in Nerve Nets and Finite Automata. In Shannon, C.E., McCarthy, J., eds.: *Automata Studies*. *Annals of Mathematics Studies*. Princeton University Press (1956) 3–42
12. Rabin, M.O., Scott, D.: Finite Automata and their Decision Problems. *IBM Journal of Research and Development* **3** (1959) 114–125
13. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient Models for Timetable Information in Public Transportation Systems. *ACM J. of Exp. Algorithmics* **12** (2007) Article 2.4
14. Star Alliance: <http://www.staralliance.com> (1997)
15. Oneworld Management Ltd: <http://www.oneworld.com> (1999)
16. Pajor, T.: Multi-Modal Route Planning. Master’s thesis, Universität Karlsruhe (TH), Fakultät für Informatik (2009).
17. Dean, B.C.: Continuous-Time Dynamic Shortest Path Algorithms. Master’s thesis, Massachusetts Institute of Technology (1999)
18. Delling, D.: Time-Dependent SHARC-Routing. *Algorithmica* (2009). To appear.
19. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch, C.C., ed.: *WEA 2008*. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008).
20. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra’s Algorithm. In McGeoch, C.C., ed.: *WEA 2008*. LNCS, vol. 5038, pp. 303–318. Springer, Heidelberg (2008).
21. Hilger, M., Köhler, E., Möhring, R.H., Schilling, H.: Fast Point-to-Point Shortest Path Computations with Arc-Flags. In: *Shortest Paths: Ninth DIMACS Implementation Challenge*. DIMACS Book. American Mathematical Society (2009) To appear.