Project Number 288094

eCOMPASS

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

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

eCOMPASS – TR – 013

Parallel computation of best connections in public transportation networks

Daniel Delling, Bastian Katz, and Thomas Pajor

July 2012

Parallel Computation of Best Connections in Public Transportation Networks

DANIEL DELLING, Microsoft Research Silicon Valley
BASTIAN KATZ and THOMAS PAJOR, Karlsruhe Institute of Technology

Exploiting parallelism in route planning algorithms is a challenging algorithmic problem with obvious applications in mobile navigation and timetable information systems. In this work, we present a novel algorithm for the one-to-all *profile-search* problem in public transportation networks. It answers the question for all fastest connections between a given station S and any other station at any time of the day in a single query. This algorithm allows for a very natural parallelization, yielding excellent speed-ups on standard multicore servers. Our approach exploits the facts that, first, time-dependent travel-time functions in such networks can be represented as a special class of piecewise linear functions and, second, only few connections from S are useful to travel far away. Introducing the *connection-setting* property, we are able to extend Dijkstra's algorithm in a sound manner. Furthermore, we also accelerate station-to-station queries by preprocessing important connections within the public transportation network. As a result, we are able to compute all relevant connections between two random stations in a complete public transportation network of a big city (New York) on a standard multi-core server in real time.

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

General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Shortest paths, route planning, public transportation, profile queries

ACM Reference Format:
Delling, D., Katz, B., and Pajor, T. 2012. Parallel computation of best connections in public transportation networks. ACM J. Exp. Algor. 17, 4, Article 4.4 (October 2012), 26 pages.
DOI = 10.1145/2133803.2345678 http://doi.acm.org/10.1145/2133803.2345678

1. INTRODUCTION

Research on fast route planning algorithms has been undergoing a rapid development in recent years (see Delling et al. [2009c] for an overview). The fastest technique for static time-independent road networks yields query times of a few memory accesses [Abraham et al. 2011]. Recently, the focus has shifted to *time-dependent* transportation networks in which the travel time assigned to an edge is a *function* of the time of the day. Thus, the quickest route depends on the time of departure. It turns out that this problem is very different from computing distances in road networks [Bast 2009]. In general, two interesting questions arise for time-dependent route planning:

Partially supported by the DFG (project WA 654/16-1) and by the EU FP7/2007-2013 (DG INFSO.G4-ICT for Transport), under grant agreement no. 288094 (project eCOMPASS). Part of this work was done while the first author was at the Karlsruhe Institute of Technology. An extended abstract appeared at the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10).
Authors' addresses: D. Delling, Microsoft Research Silicon Valley, Mountain View 94043; email: dadellin@microsoft.com; B. Katz and T. Pajor, Karlsruhe Institute of Technology, P.O. Box 6980, 76128 Karlsruhe, Germany; email: {katz, pajor}@kit.edu.

DOI 10.1145/2133803.2345678 http://doi.acm.org/10.1145/2133803.2345678

compute the best connection for a given departure time and the computation of all best connections during a given time interval (e. g., a whole day). The former is called a *time-query*, while the latter is called a *profile-query*. Especially in public transportation, the use of time-queries is limited; specifying some fixed departure time will most probably lead to an awkward itinerary when a fast connection was just missed, thus, forcing the passenger to wait for a long time or letting him use a lot of slow trains. In this case, using the slightly earlier train would significantly improve the overall travel time. Hence, especially in public transportation networks, we are interested in the fast computation of profile-queries. Previous algorithms for computing profile-queries augment Dijkstra's algorithm by propagating travel-time functions instead of scalar values through the network [Dean 1999]. However, due to the fact that travel-time functions cannot be totally ordered, these algorithms lose the *label-setting* property, meaning that nodes are inserted multiple times into the priority queue. This implies a significant performance penalty, making the computation of profile-queries very slow. Furthermore, state-of-the-art algorithms typically do not involve parallel computation, and in fact, route planning is one of the rare large-scale combinatorial problems where parallelism seems to be of limited use to speed up single queries in the past.

Related Work. Modeling issues and an overview of basic route-planning algorithms in public transportation networks can be found in Pyrga et al. [2008], while Orda and Rom [1990] deal with time-dependent route planning in general. Basic speed-up techniques like goal-directed search have been applied to time-dependent railway networks in Disser et al. [2008], while SHARC [Bauer and Delling 2009] and Contraction Hiearchies [Geisberger et al. 2008] have been tested on such networks as well [Delling 2011; Geisberger 2010]. However, most of the algorithms fall short as soon as they are applied to bus networks [Bauer et al. 2011; Delling et al. 2009b]. The fastest (in terms of query times) known solution for computing connections in public transit networks is based on the concept of transfer patterns [Bast et al. 2010]. Although this concept accelerates query times greatly, it drops optimality and is based on preprocessing the input for thousands of CPU hours, which makes it hard to use in a dynamic scenario.

Most efforts in developing parallel search algorithms address theoretical machines such as the PRAM [Paige and Kruskal 1985; Driscoll et al. 1988] or the communication network model [Chandy and Misra 1982; Ramarao and Venkatesan 1992]. Even in these models, for a long time there was no algorithm known that is able to exploit parallelism beyond parallel edge relaxations and parallel priority queuing without doing substantially more work than a sequential Dijkstra implementation in general networks. Very recently, Delling et al. [2012] introduced a new algorithm for computing the distance from one node to any other that exploits parallelism on all levels. Implemented on a GPU, the algorithm is up to three orders of magnitude faster than Dijkstra's algorithm. However, this algorithm only works in road networks.

There also have been a few experimental studies of distributed single-source shortest path algorithms for example based on graph partitioning [Adamson and Tick 1991; Träff 1995] or on the Δ-stepping algorithm proposed in Meyer and Sanders [1998] (e.g., see Madduri et al. [2007]). For an overview on many related approaches, we refer the reader to Hribar et al. [2001]. All these approaches have in common that they do provide good speed-ups only for certain graph classes. Search algorithms for retrieving all quickest connections in a given time interval have been discussed in Dean [1999]. However, none of those algorithms have been parallelized and used for retrieving all quickest connections of a day in realistic dynamic public transportation networks.

Our Contribution. We present a novel parallel algorithm for the *one-to-all profile-search* problem asking for the set of all relevant connections between a given station

S and all other stations, that is, all connections that at any time constitute the fastest way to get from S to some other station. The key idea is that the number of possible connections is bounded by the number of outgoing connections from the source station S, and all time-dependent travel-time distances in such networks are piecewise linear functions that have a representation that is bounded by this number as well. Moreover, only few connections prove useful when traveling sufficiently far away. The algorithm we present in this work greatly exploits this fact by pruning such connections as early as possible. To this extent, we introduce the notion of *connection-setting*, which can be seen as an extension of the label-setting property of Dijkstra's algorithm, which usually is lost in profile-searches (e.g., road networks). The main idea regarding parallelism in transportation networks is that we may distribute different connections outgoing from S to the different processors. Furthermore, we show how connections can be pruned even across different processors.

While one-to-all queries are relevant for the preprocessing of many speed-up techniques [Delling and Wagner 2009; Delling et al. 2009a], we also accelerate the more common scenario of station-to-station queries explicitly. Therefore, we propose to utilize the very same algorithm for valuable preprocessing. The key idea is that we select a small number of important stations (called *transfer stations*) and precompute a full distance table between all these stations, which then can be used to prune the search during the query. We show the feasibility of our approach by running extensive experiments on real-world transportation networks. It turns out that our algorithm scales very well with the number of utilized cores.

Independent of our new algorithm, we also recap the widely used realistic time-dependent graph model and improve it by modeling conflicting trains inside stations more carefully. The key idea is then to compute a (minimum) coloring of a corresponding conflict graph such that each color represents a node in the model graph. Hence, using this *coloring model*, we are able to reduce the size of the graph significantly, which directly yields speed-ups for any graph search algorithm.

Moreover, for realistic queries *foot paths* are crucial to enable transfers between stations. However, often such data is not available from the transit agencies. Thus, we present a heuristic approach to generate artificial foot paths using the underlying road network. Our method is based on snapping stations to (nearest) intersections and introducing cliques between stations of the same intersection.

As an example, using our parallel algorithm on the coloring model, we are able to perform a one-to-all profile-search in less than 110ms and station-to-station queries in less than 17ms in all of our networks.

This article is the full version of the one published in the *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium* [Delling et al. 2010]. Compared to the conference version, we introduce a new approach for modeling public transit networks, show how to generate reasonable foot paths, improve the station-to-station scenario, and extend the experimental evaluation greatly.

Overview. This work is organized as follows. In Section 2, we briefly explain necessary definitions and preliminaries. Section 3 recaps the existing realistic time-dependent graph model and introduces our new coloring model. Here, we also show how we generate artificial foot paths. Section 4 then starts with our parallel one-to-all algorithm. Therefore, we first introduce the concept of connection-setting and show how some connections dominate others. In Section 5, we present how our algorithm can be utilized to accelerate station-to-station queries. A detailed review of our experiments can be found in Section 6. We conclude our work with a brief summary and possible future work in Section 7.

2. PRELIMINARIES

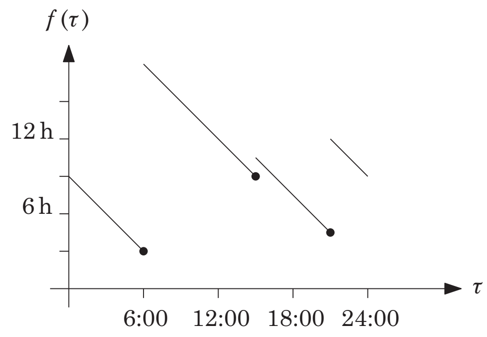
A *directed graph* is a tuple $G = (V, E)$ consisting of a finite set V of nodes and a set of ordered pairs of vertices, or *edges* $E \subseteq V \times V$. The node u is called the *tail* of an edge (u, v), and v the *head*. The reverse graph $\overleftarrow{G} = (V, \overleftarrow{E})$ is obtained from G by flipping all edges, i. e., $(u, v) \in \overleftarrow{E} \Leftrightarrow (v, u) \in E$.

Timetables. A *periodic timetable* is defined as a tuple $(\mathcal{C}, \mathcal{S}, \mathcal{Z}, \Pi, \mathcal{T})$, where $\mathcal{S}$ is a set of stations, $\mathcal{Z}$ a set of *trains* (or buses, trams, etc.), $\mathcal{C}$ a set of elementary connections, and $\Pi := \{0, \ldots, \pi - 1\}$ a finite set of discrete time points (think of it as the minutes or seconds of a day). We call π the periodicity of the timetable. Note that durations and arrival times can take values greater than π (think of a train arriving after midnight). Moreover, $\mathcal{T} : \mathcal{S} \to \mathbb{N}_0$ assigns each station a minimum transfer time required to change between trains. An elementary connection from $c \in \mathcal{C}$ is defined as a tuple $c := (Z, S_{\text{dep}}, S_{\text{arr}}, \tau_{\text{dep}}, \tau_{\text{arr}})$ and is interpreted as train $Z \in \mathcal{Z}$ going from station $S_{\text{dep}} \in \mathcal{S}$ to station $S_{\text{arr}} \in \mathcal{S}$, departing at S_{dep} at time $\tau_{\text{dep}} \in \Pi$ and arriving at $\tau_{\text{arr}} \in \mathbb{N}_0$. For simplicity, given an elementary connection c, $X(c)$ selects the X-entry of c, that is, $\tau_{\text{dep}}(c)$ refers to the departure time of c. Due to the periodic nature of the timetable, the length $\Delta(\tau_1, \tau_2)$ between two time points τ_1 and τ_2 is computed by $\tau_2 - \tau_1$ if $\tau_2 \geq \tau_1$ and $\pi + \tau_2 - \tau_1$ otherwise. Note that Δ is not symmetric.

General Modeling Approaches. For the purpose of route planning, the timetable is usually modeled as a directed graph in such a way that a shortest path in the graph corresponds to a quickest journey in the timetable. Basically, there exist two categories of modeling approaches [Pyrga et al. 2008]: the time-expanded approach the time-dependent approach. In the time-expanded approach, the main idea is to create a node for each event of the timetable (departure of a train, arrival of a train, among others). To connect appropriate events in the graph, edges are inserted with length corresponding to the time difference of the respective events. Since a typical timetable contains millions of events, this approach leads to large graphs, and thus, to a large search space (and bad query times) for search algorithms [Müller–Hannemann and Schnee 2007].

To overcome this issue, in the time-dependent model, the key idea is to group connections between same pairs of stations. These groups of connections can be represented by a special form of piecewise linear functions (see next paragraph). This approach yields significantly smaller graph sizes (usually, the number of nodes is in the order of the number of stations); hence, a query algorithm has to explore a much smaller search space. Moreover, it allows for the computation of the distances between two stations for all departure times—as we are especially interested in. Hence, in this work, we focus on the time-dependent approach.

Piecewise Linear Functions. In general, there are two types of distances in a public transportation network. The first is the distance between two stations S and T for a given departure time τ, denoted by $\text{dist}(S, T, \tau)$. The second type, which we are interested in this work, the distance function between two stations S and T for all departure times $\tau \in \Pi$, denoted by $\text{dist}(S, T, \cdot)$. Such a function is defined as $f : \Pi \to \mathbb{N}_0$, where $f(\tau)$ denotes the travel-time when starting at time τ. For the remainder of this article, it is a crucial observation that in public transportation networks these functions can be represented as piecewise linear functions of a special form: The travel-time at time τ is composed of a waiting time for the next connection c starting at some $\tau_{\text{dep}}(c)$ plus the duration of the itinerary starting with c. Moreover, if the best choice at time τ is to wait for a connection c, the same holds for any $\tau \leq \tau' \leq \tau_{\text{dep}}$ in between. See Figure 1 for an example. Hence, it is possible to represent f by a set of connection-points



(a) Function.

No.	Dep.-time	Travel-time
1	06:00	3 h 00 min
2	15:00	9 h 00 min
3	21:00	4 h 30 min
⋮	⋮	⋮

(b) Respective connections.

Fig. 1. A piecewise linear function f with three connection points (left side), representing three relevant trains to start with (right side).

$\mathcal{P}(f) \subset \Pi \times \mathbb{N}_0$ such that $f(\tau) = \Delta(\tau, \tau_f) + w_f$ for the $(\tau_f, w_f) \in \mathcal{P}(f)$, which minimizes $\Delta(\tau, \tau_f)$. Respecting periodicity in a meaningful way, these travel-time functions have the FIFO-property if for any $\tau_1, \tau_2 \in \Pi$, it holds that $f(\tau_1) \leq \Delta(\tau_1, \tau_2) + f(\tau_2)$. In other words, waiting never gets you (strictly) earlier to your destination. Note that all our networks fulfill the FIFO-property.

Computing Distances. The sequential computation of $\text{dist}(S, \cdot, \tau)$ can be done by a time-dependent version of Dijkstra's algorithm, which we call *time-query*. It visits all nodes in the graph in nondecreasing order from the source S. Therefore, it maintains a priority queue Q, where the *key* of an element v is the tentative distance $\text{dist}(S, v)$. By using a priority queue, the algorithm makes sure that if an element v is removed from Q, $\text{dist}(S, v)$ cannot be improved anymore. This property is called *label-setting*.

Determining the complete distance function $\text{dist}(S, \cdot, \cdot)$, called a *profile-query*, from a given station S to any other station for all departure times $\tau \in \Pi$ can be computed by a profile-search algorithm being very similarly to Dijkstra. The main difference is that functions instead of scalars are propagated through the network. By this, the algorithm may lose its label-setting property, since nodes may be reinserted into the queue that have already been removed. Hence, we call such an algorithm a *label-correcting* approach. An interesting result from Dean [1999] is that the running time highly depends on the number of connection points assigned to the edges.

3. MODELING

As introduced before, we focus on the time-dependent modeling approach. More precisely, we use the realistic time-dependent model as defined in Pyrga et al. [2008]. It uses time-dependent edges and supports minimum transfer times at stations.

Given a timetable, the graph $G = (V, E)$ of the realistic time-dependent model is constructed as follows. First, the set $\mathcal{Z}$ of trains is partitioned into routes, where two trains $Z_1, Z_2 \in \mathcal{Z}$ are considered equivalent (i.e., belong to the same route) if they share the exact same sequence of stations. Regarding the nodes, for each station $S \in \mathcal{S}$, a station node is created. Moreover, for each route that runs through S, a route node is created. Route nodes are connected by edges to their respective station nodes with time-independent weights depicting the transfer time $\mathcal{T}(S)$. Furthermore, for each route and for each two subsequent stations S_1 and S_2 on that route, a time-dependent route-edge (u, v) is inserted between the route nodes u and v of the respective route at the stations S_1 and S_2. By these means, the time-dependent route-edges e get exactly those elementary connections $c \in \mathcal{C}$ assigned, where $Z(c)$ relates to a train of the respective route

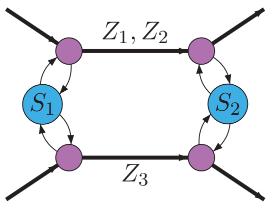


Fig. 2. Illustration of the realistic time-dependent model, showing two stations where two routes run through. Station nodes are blue, and route nodes are smaller and purple.

(between the two given stations). Note that the resulting time-dependent functions on the route-edges are exactly of the same type as introduced in the previous section (its connection points are exactly the respective elementary connections). In particular, they fulfill the FIFO-property. See Figure 2 for an illustration.

Some variations of this model to incorporate different levels of detail exist (e.g., to support quicker interchanges between trains stopping on the same platform, additional edges can be added). However, all variants rely on the notion of routes and add at least as many nodes per station to the graph as there are routes through the station. In fact, an analysis of the model reveals that the average number of route nodes per station is typically between 5 and 16, depending on the input (see Section 6), which is quite high. To reduce this number, in the next section we introduce a new model based on a formal notion of conflicting trains. Note that a smaller graph size immediately results in faster query times for any search algorithm.

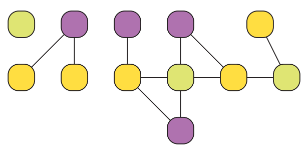
3.1. Coloring Model

One main reason of using the notion of routes in the realistic time-dependent model is the observation that in a journey, interchanges between two trains on the same route are never beneficial. Thus, when assigning trains of the same route to the same route node (i.e., assigning their respective elementary connections to edges incident to the route node), we ensure that we do not generate an itinerary with invalid transfers (i.e., violating the minimum transfer time at some station). However, this property can also be guaranteed by a more formal notion of conflicting trains in the following sense.

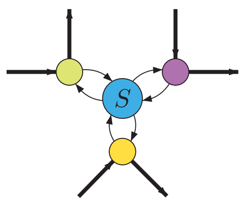
Consider two trains Z_1 and Z_2 that run through some station S. Let $\tau_{\text{arr}}(Z_1, S)$ be the arrival time of train Z_1 at S, and $\tau_{\text{dep}}(Z_2, S)$ the departure time of Z_2 at S. Then, these two trains conflict if and only if Z_2 departs after the arrival of Z_1, and the time in between is smaller than the minimum transfer time at S. More precisely, Z_1 and Z_2 conflict if and only if $\tau_{\text{dep}}(Z_2, S) \geq \tau_{\text{arr}}(Z_1, S)$ and $\tau_{\text{arr}}(Z_1, S) + \mathcal{T}(S) < \tau_{\text{dep}}(Z_2, S)$. In this case, putting Z_1 and Z_2 on the same route node could yield an illegal itinerary, which must be avoided.

Testing the conflict condition for all pairs of trains running through S naturally induces an undirected conflict graph $G_{\text{conf}}(S) = (V_{\text{conf}}(S), E_{\text{conf}}(S))$. The node set $V_{\text{conf}}(S) \subseteq \mathcal{Z}$ contains exactly those trains $Z \in \mathcal{Z}$ that run through S, that is, where there exists an elementary connection $c \in \mathcal{C}$ with $Z(c) = Z$ and $S_{\text{dep}}(c) = S$ or $S_{\text{arr}}(c) = S$. Two pairs of nodes $Z_i, Z_j \in V_{\text{conf}}(S)$ are connected by an edge $\{Z_i, Z_j\} \in E_{\text{conf}}(S)$ if and only if Z_i and Z_j are conflicting. Experiments on our instances (see Section 6) reveal that the number of conflicting trains indeed is small: We observe that of all possible train pairs per station, on average, less than 0.5 % are actually conflicting. Thus, we regard G_{conf} as sparse.

It is now easy to see that a valid node coloring on $G_{\text{conf}}(S)$, where no two adjacent nodes may share the same color, induces a set of route nodes of the station S in the model graph G. Let K be the number of distinct colors used for $G_{\text{conf}}(S)$, then for each



(a) Conflict graph.



(b) Model graph.

Fig. 3. An example of a conflict graph $G_{\text{conf}}(S)$ of some station S with a valid node coloring using three colors (a) and the corresponding induced subgraph of S having three route nodes in the model graph G (b). In (b), time-dependent route-edges are drawn bold, while time-independent transfer-edges are drawn thin.

color $k = 1 \ldots K$, we create a route node u in G, and put exactly those trains on u that have color k in $G_{\text{conf}}(S)$. An example of a conflict graph and its induced subgraph in the model are illustrated in Figure 3.

Computing Colorings. In general, our goal is to generate as few route nodes in G as possible. Thus, we aim for computing a coloring on $G_{\text{conf}}(S)$ with as few colors as possible. In fact, a lower bound on the number of route nodes for S in G is given by the chromatic number $\chi(G_{\text{conf}}(S))$. Since it is well-known that computing $\chi(G_{\text{conf}}(S))$ is NP-hard, we use the following greedy heuristic to color $G_{\text{conf}}(S)$ for every S. We start with an uncolored graph and process the nodes of $G_{\text{conf}}(S)$ in order of decreasing degree. When considering node u, we assign u the smallest color that is not used to color any of u's neighbors.

Note that this algorithm never uses more than $\text{maxdeg}(G_{\text{conf}}(S)) + 1$ colors, where $\text{maxdeg}(G_{\text{conf}}(S))$ is the maximum node degree of $G_{\text{conf}}(S)$. Since we consider $G_{\text{conf}}(S)$ to be sparse, the results of the greedy algorithm on $G_{\text{conf}}(S)$ are quite good in practice (see Section 6 for experimental details).

Merging Small Stations. To further reduce the number of nodes in the model graph G, we merge small stations S, which have only one route node (i.e., $G_{\text{conf}}(S)$ has been colored with one color). More precisely, we merge the station node with the (only) route node. Since there are no conflicting trains at S, we do not lose correctness by applying this procedure to all stations of this type in G.

3.2. Foot Paths

To enable transfers between nearby stations, the timetable is usually augmented with a set $\mathcal{F}$ of foot paths. Each foot path is defined as a tuple $(S_i, S_j) \in \mathcal{S} \times \mathcal{S}$ and an associated length $\ell(S_i, S_j)$, meaning that it is possible to walk from station S_i to station S_j in time $\ell(S_i, S_j)$. To incorporate foot paths into the model graph G, for each tuple $(S_i, S_j) \in \mathcal{F}$, we insert an edge (S_i, S_j) into G with (constant) weight $\ell(S_i, S_j)$, similarly to the transfer edges within stations of G.

Incorporating foot paths into the model turns out crucial for finding realistic itineraries with reasonable transfers. Even worse, the graph obtained from some real-world timetables may even get disconnected into components when foot paths are omitted. Unfortunately, foot path data was not included with the timetable data available to us. Thus, we use the following heuristic to generate an artificial set $\mathcal{F}$ of foot paths.

Let R be the road network covering (at least) the geographical area of the public transportation network for which we are about to generate foot paths. We assign each station $S \in \mathcal{S}$ to a bucket b using R. We find the intersection $b \in R$ that is geographically closest to S, and assign S to b if the geographical distance is no greater than a parameter (typically set to 100m). We then look at all buckets b created, and between all pairs of

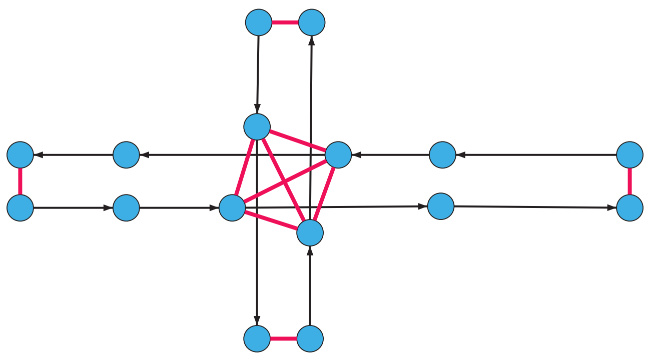


Fig. 4. Example of heuristically generated foot paths in a typical U. S. bus network (two bus lines along two streets). Foot edges are depicted in bold magenta.

different stations $S_i, S_j \in b$, we add a foot path (S_i, S_j) to $\mathcal{F}$ with a length computed by the sum of the distances from S_i to b to S_j divided by an assumed average walking speed (typically 4kph).

Note that since each station is assigned to exactly one bucket, our heuristic obtains many small components of stations that are interconnected by foot paths near intersections. In particular, we avoid connecting large regions of the network through sequences of foot paths. See Figure 4 for an example.

4. A PARALLEL SELF-PRUNING PROFILE SEARCH ALGORITHM

In this section, we describe our new parallel profile-search algorithm tailored to public transportation networks. A crucial observation in such networks is the fact that each itinerary from a source station S to any other station has to begin with an elementary connection originating at S. Let this set of outgoing connections be denoted by $\text{conn}(S) := \{c \in \mathcal{C} \mid S_{\text{dep}}(c) = S\}$. A naïve and obvious way to compute the full distance function $\text{dist}(S, \cdot, \cdot)$ would be to compute a time-query $\text{dist}(S, \cdot, \tau)$ for each elementary connection $c \in \text{conn}(S)$ with respect to its departure time $\tau = \tau_{\text{dep}}(c)$. However, such a connection does not necessarily contribute to $\text{dist}(S, T, \cdot)$. A connection c_i with departure time $\tau_{\text{dep}}(c_i)$ may as well be dominated by a connection c_j with later departure time $\tau_{\text{dep}}(c_j) > \tau_{\text{dep}}(c_i)$ in the following sense: If the earliest arrival time at T starting with c_j is not greater than the earliest arrival time starting with c_i, we can—and must, for the sake of correctness—prune the result of the search regarding connection c_i, since starting with c_i never yields the shortest travel time. Note that this observation implies that for any $T \in \mathcal{S}$, the set of connection points $\mathcal{P}(\text{dist}(S, T, \cdot))$ of the distance function $\text{dist}(S, T, \cdot)$ is a subset of the set of connection points induced by $\text{conn}(S)$ and their distances to T. More precisely, the following holds:

$$\begin{aligned}\mathcal{P}(\text{dist}(S, T, \cdot)) \subseteq \{(\tau, w) \mid \exists c \in \text{conn}(S) :\; & \\ \tau = \tau_{\text{dep}}(c), & \\ w = \text{dist}(S, T, \tau_{\text{dep}}(c))\}. &\end{aligned} \tag{1}$$

The problem to run $|\text{conn}(S)|$ time-queries and then pruning dominated connections from $\text{dist}(S, T, \cdot)$ afterward is an embarrassingly parallel problem. Going much further, we show how to extend the above observation to obtain a pruning rule that we call *self-pruning*. It can be applied to eliminate “unnecessary” connections as soon as possible. Thereby, we use self-pruning within the restricted domain of each single thread, but we also take advantage of communication between the different threads yielding a rule we call *inter-thread-pruning*. Therefore, we require a fixed assignment of the outgoing connections to the processors where each processor handles a set of connections simultaneously.

The outline of our parallel algorithm is as follows. First, we partition the set $\mathrm{conn}(S)$ to a given set of processors. Second, every processor runs a single thread, applying our main sequential profile search algorithm restricted to its subset of outgoing connections. In a third step, the partial results by the different threads are combined, thereby eliminating dominated connections that could not be pruned earlier, a step we will refer to as *connection reduction*.

4.1. The Main (Sequential) Algorithm

From the point of view of a single processor that has some subset of $\mathrm{conn}(S)$ as input, it basically makes no difference to the profile-search algorithm that some of the connections are ignored. We simply obtain $\mathrm{dist}_k(S, \cdot, \cdot)$ restricted to the connections assigned to the particular processor k. Hence, we describe the main algorithm as if it was a purely sequential profile-search algorithm and turn toward the parallel issues like merging the results from each processor, the choice of the partitioning of $\mathrm{conn}(S)$, and the inter-thread-pruning rule, afterward.

The naïve approach of running a separate time-query for each $c \in \mathrm{conn}(S)$ by Dijkstra's algorithm would require an empty priority queue for every connection c. By contrast, our algorithm maintains a single priority queue and handles all of its connections simultaneously. Moreover, we use tentative arrival times as keys (instead of distances). By these means, we enable both the connection-setting property as well as our self-pruning rule.

Initialization. At first, the set $\mathrm{conn}(S)$ is determined and ordered nondecreasingly by the departure times of the elementary connections in $\mathrm{conn}(S)$. Thus, we may say that a connection c_i has index i according to the ordering of $\mathrm{conn}(S)$. The elements of the priority queue are pairs (v, i), where the first entry depicts a node $v \in V$ and the second entry a connection index $0 \leq i < |\mathrm{conn}(S)|$. For each node $v \in V$ and for each connection i, a label $\mathrm{arr}(v, i)$ is assigned that depicts the (tentative) arrival time at v when using connection i. In the beginning, each label $\mathrm{arr}(v, i)$ is initialized with ∞. Then, for each connection $c_i \in \mathrm{conn}(S)$, we insert (r, i) with key $\tau_{\mathrm{dep}}(c_i)$ into Q, where r depicts the route node where connection c_i starts from. Note that in the beginning, the "arrival time" $\mathrm{arr}(r, i)$ equals the departure time $\tau_{\mathrm{dep}}(c_i)$.

Connection-Setting. Like Dijkstra's algorithm, we subsequently settle queue elements (v, i) assigning $\mathrm{key}(v, i)$ as the final arrival time to $\mathrm{arr}(v, i)$. Then, for each edge $e = (v, w) \in E$, we compute a tentative label $\mathrm{arr}_{\mathrm{tent}}(w, i)$ at w by $\mathrm{arr}_{\mathrm{tent}}(w, i) := \mathrm{arr}(v, i) + f_e(\mathrm{arr}(v, i))$ (for connection i). If w has not yet been discovered using connection i, we insert (w, i) into the priority queue with $\mathrm{key}(w, i) := \mathrm{arr}_{\mathrm{tent}}(w, i)$, otherwise the element (w, i) is already in the queue, and we set $\mathrm{key}(w, i)$ to $\min(\mathrm{key}(w, i), \mathrm{arr}_{\mathrm{tent}}(w, i))$. Note that the following holds for every connection i: When a queue item (v, i) is settled, the label $\mathrm{arr}(v, i)$ is final; thus, the label-setting property holds with respect to each connection i, which we call *connection-setting*. The algorithm ends as soon as the priority queue runs empty. We end up with labels $\mathrm{arr}(v, i)$ for each node $v \in V$ and each connection $0 \leq i < |\mathrm{conn}(S)|$, depicting the arrival time at v when starting with the ith connection at S.

We would like to stress two things. First, although the computation is done for all connections simultaneously, they can be regarded as independent, since the labels and the queue items refer to a specific connection throughout the algorithm. Second, the original variant of Dijkstra's algorithm uses distances instead of arrival times as keys. However, this has no impact on the correctness of the algorithm. For each connection the distance can be obtained by subtracting the respective departure time from the arrival time, which is constant for all nodes.

Connection Reduction and Self-Pruning. For each node $v \in V$, the final labels $\mathrm{arr}(v, \cdot)$ induce a set of connection points $\widehat{\mathcal{P}}$ by $\widehat{\mathcal{P}} := \{(\tau_{\mathrm{dep}}(c_i), \Delta(\tau_{\mathrm{dep}}(c_i), \mathrm{arr}(v, i))) \mid c_i \in \mathrm{conn}(S)\}$. Unfortunately, the function f represented by $\widehat{\mathcal{P}}$ does not account for domination of connections and hence does not necessarily fulfill the FIFO-property. Formally, for two points $(\tau_i, w_i), (\tau_j, w_j) \in \widehat{\mathcal{P}}$ with $j > i$, it is possible that $\tau_j + w_j \leq \tau_i + w_i$. The aforementioned connection reduction, which remedies this issue at the end of the algorithm, reduces $\widehat{\mathcal{P}}$ to obtain $\mathcal{P}(\mathrm{dist}(S, T, \cdot))$ by eliminating those points that are dominated by another point with a later departure time and an earlier arrival time. More precisely, we scan backward through $\mathcal{P}$ keeping track of the minimum arrival time $\tau^{\mathrm{arr}}_{\mathrm{min}} := \tau_{i_{\mathrm{min}}} + w_{i_{\mathrm{min}}}$ along the way. Each, time we scan a connection point $j < i_{\mathrm{min}}$ with an arrival time $\tau^{\mathrm{arr}}_j \geq \tau^{\mathrm{arr}}_{\mathrm{min}}$, the connection point is deleted. The remaining connection points are exactly those of $\mathcal{P}(\mathrm{dist}(S, T, \cdot))$.

Performing this connection reduction after termination of the algorithm results in the computation of many unnecessary connections and, therefore, many unnecessary queue operations. Recall that the keys in our queue are arrival times. Thus, we propose a more sophisticated approach to eliminate dominated connections during the algorithm: We introduce a *node-label* $\mathrm{maxconn} : V \to \{0, \dots, |\mathrm{conn}(S)| - 1\}$ depicting the highest connection index with which node v has been reached so far. Each time we settle a queue element (v, i) with $\mathrm{arr}(v, i) := \mathrm{key}(v, i)$, we check if $i > \mathrm{maxconn}(v)$. If this is not the case, the node v has already been settled earlier—but with a later connection (remember that $j > i \Rightarrow \tau_{\mathrm{dep}}(c_j) \geq \tau_{\mathrm{dep}}(c_i)$), thus implying $\mathrm{arr}(v, j) \leq \mathrm{arr}(v, i)$. Therefore, the current connection does not pay off, and we prune the connection i at v, that is, we do not relax outgoing edges at v. Moreover, we set $\mathrm{arr}(v, i) := \infty$, depicting that the ith connection does not "reach" v. In the case of $i > \mathrm{maxconn}(v)$, we update $\mathrm{maxconn}(v)$ to i and continue with relaxing the outgoing edges of v regularly. Obviously, by applying self-pruning, the set of connection points $\mathcal{P}(\mathrm{dist}(S, v, \cdot))$ at each node v induced by $\mathrm{arr}(v, \cdot)$ fulfills the FIFO-property automatically (labels with $\mathrm{arr}(v, i) = \infty$ have to be ignored).

THEOREM 4.1. *Applying self-pruning is correct.*

PROOF. Let $v \in V$ be an arbitrary node. We show that no optimal connection to v has been pruned by contradiction. Let $\mathrm{arr}(v, i)$ be the arrival time at v of the (optimal) ith connection and assume that i has been pruned at v. Let j denote the connection that was responsible for pruning i. Then, it holds that $\mathrm{arr}(v, j) \leq \mathrm{arr}(v, i)$. Moreover, since j pruned i, it holds that $j > i$, which implies $\tau_{\mathrm{dep}}(c_j) \geq \tau_{\mathrm{dep}}(c_i)$. Therefore, it holds that $\Delta(\tau_{\mathrm{dep}}(c_j), \mathrm{arr}(v, j)) \leq \Delta(\tau_{\mathrm{dep}}(c_i), \mathrm{arr}(v, i))$. This is a contradiction to i being optimal: Using the jth connection results in an earlier arrival at v by departing later at S. □

Putting things together, the complete (sequential) algorithm can be found in Algorithm 1 in pseudocode notation.

4.2. Parallelization

Unlike the trivial parallelization that would assign a connection $c \in \mathrm{conn}(S)$ for an arbitrary idle processor, which then runs Dijkstra's algorithm on c, our algorithm needs a fixed assignment of the connections to the processors beforehand. Let p denote the number of processors available. In the first step, we partition $\mathrm{conn}(S)$ into p subsets where each thread k runs our main algorithm on its restricted subset $\mathrm{conn}_k(S)$.

After termination of each thread, we obtain partial distance functions $\mathrm{dist}_k(S, \cdot, \cdot)$ restricted to the connections that were assigned to thread k. Thus, the master thread merges the labels $\mathrm{arr}_k(v, \cdot)$ of each thread k to a common label $\mathrm{arr}(v, \cdot)$ while preserving

ALGORITHM 1: Self-Pruning Connection-Setting (SPCS)

```
   Input: Graph $G = (V, E)$, source station $S$, outgoing connections $\mathrm{conn}(S)$
   Side Effects: Distance labels $\mathrm{arr}(\cdot, \cdot)$ for each node and connection
   // Initialization
1  Q ← new(PQueue);
2  $\mathrm{maxconn}(\cdot) \leftarrow -\infty$;
3  $\mathrm{arr}(\cdot, \cdot) \leftarrow \infty$;
4  $\mathrm{discovered}(\cdot, \cdot) \leftarrow$ false ;
5  sort($\mathrm{conn}(S)$);
6  forall the $c_i \in \mathrm{conn}(S)$ do
7      $r \leftarrow$ route node belonging to $c_i$;
8      Q.insert($(r, i)$, $\tau_{dep}(c_i)$);
9      $\mathrm{discovered}(r, i) \leftarrow$ true ;

   // Main Loop
10 while not Q.empty() do
       // Settle next node/connection
11     $(v, i) \leftarrow$ Q.minElement();
12     $\mathrm{arr}(v, i) \leftarrow$ Q.minKey();
13     Q.deleteMin();
       // Self-Pruning Rule
14     if $\mathrm{maxconn}(v) > i$ then
15         $\mathrm{arr}(v, i) \leftarrow \infty$;
16         continue ;
17     else
18         $\mathrm{maxconn}(v) \leftarrow i$;
       // Relax outgoing edges
19     forall the outgoing edges $e = (v, w) \in E$ do
20         $\mathrm{arr}_{\mathrm{tent}}(w, i) \leftarrow \mathrm{arr}(v, i) + f_e(\mathrm{arr}(v, i))$;
21         if not $\mathrm{discovered}(w, i)$ then
22             Q.insert($(w, i)$, $\mathrm{arr}_{\mathrm{tent}}(w, i)$);
23             $\mathrm{discovered}(w, i) \leftarrow$ true ;
24         else if $\mathrm{arr}_{\mathrm{tent}}(w, i) <$ Q.key($(w, i)$) then
25             Q.decreaseKey($(w, i)$, $\mathrm{arr}_{\mathrm{tent}}(w, i)$);
```

the ordering of the connections. This can be done by a linear sweep over the labels. Note that the common label $\mathrm{arr}(v, \cdot)$ is not necessarily FIFO, since we do not self-prune between threads so far. For that reason, the connection points $\mathcal{P}(S, T, \cdot))$ of the final distance function are obtained by reducing the connection points induced by the common label $\mathrm{arr}(v, \cdot)$ with our connection reduction method described earlier. The pseudocode of the main parallel algorithm is presented in Algorithm 2.

Choice of the Partition. The speed-up achieved by the parallelization of our algorithm depends on the partitioning of $\mathrm{conn}(S)$. As the overall computation time is dominated by the thread with the longest computation time (for computing the final distance function, all threads have to be in a finished state), nearly optimal parallelism would be achieved if all threads share the same amount of queue operations, thus, approximately sharing the same computation time. However, this figure is not known beforehand, which requires us to partition $\mathrm{conn}(S)$ heuristically. We propose the following simple methods.

ALGORITHM 2: Parallel SPCS (PSPCS)

```
   Input: Graph $G = (V, E)$, source station $S$, outgoing connections $\mathrm{conn}(S)$,
          $p$ processors
   Side Effects: Distance labels $\mathrm{arr}(\cdot,\cdot)$ for each node and connection
   // Initialization
1  $\{\mathrm{conn}_1(S), \ldots, \mathrm{conn}_p(S)\} \leftarrow$ partition($\mathrm{conn}(S)$);
   // Parallel Computation
2  for $k \leftarrow 1 \ldots p$ do in parallel
3      $\mathrm{arr}_k(\cdot,\cdot) \leftarrow \infty$;
       // Invoke the sequential self-pruning connection-setting algorithm
4      SPCS($\mathrm{conn}_k(S)$);
   // Connection-Reduction
5  $\mathrm{arr}(\cdot,\cdot) \leftarrow$ merge($\mathrm{arr}_1(\cdot,\cdot), \ldots, \mathrm{arr}_p(\cdot,\cdot)$);
6  forall the $v \in V$ do
7      last $\leftarrow \infty$;
8      for $i \leftarrow |\mathrm{conn}(S)| \ldots 1$ do
9          if $\mathrm{arr}(v, i) <$ last then
10             last $\leftarrow \mathrm{arr}(v, i)$;
11         else
12             $\mathrm{arr}(v, i) \leftarrow \infty$;
```

The equal time-slots method partitions the complete time-interval Π into p intervals of equal size. While this can be computed easily, the sizes of $\mathrm{conn}(S)_i$ turn out to be very unbalanced, at least in our scenario. The reason for this is that the connections in $\mathrm{conn}(S)$ are not distributed uniformly over the day due to rush hours and operational breaks at night. The equal number of connections method tries to improve on that by partitioning the set $\mathrm{conn}(S)$ into p sets of equal size (i.e., containing equally many subsequent elementary connections). This is also very easy to compute and improves over the equal time slots method regarding the balance. Besides these simple heuristics, in principle, more sophisticated clustering methods like k-means [MacQueen 1967] can be applied. However, our experimental evaluation (see Section 6.2) shows that the improvement in query performance is negligible compared to the simple methods; thus, we use the equal number of connections method as a reasonable compromise. We stress that for the correctness of our algorithm, it is not necessary to partition $\mathrm{conn}(S)$ into cells of subsequent connections. However, it is intuitive to see that the self-pruning rule is most effective on neighboring (regarding the departure time) connections.

Impact on Self-Pruning and Pruning between Threads. When computing the partial profile functions independently in parallel, the speed-up gained by self-pruning may decrease, since a connection j cannot prune a connections i if i is assigned to a different thread than j. Thus, with an increasing number of threads, the effect achieved by self-pruning vanishes to the extreme point where the number of threads equals the number of connections in $\mathrm{conn}(S)$. In this case, our algorithm basically corresponds to computing $|\mathrm{conn}(S)|$ time-queries in parallel—without any pruning. To remedy this issue, the self-pruning rule can be augmented in order to make use of dominating connections across different threads. In the case that the partitioning of $\mathrm{conn}(S)$ is chosen such that each cell $\mathrm{conn}(S)_k$ only contains subsequent connections, we can define a total ordering on the cells by $\mathrm{conn}(S)_k \prec \mathrm{conn}(S)_l$ if for all connections $c \in \mathrm{conn}(S)_k$ and all connections $c' \in \mathrm{conn}(S)_l$ it holds that $\tau_{\mathrm{dep}}(c) \leq \tau_{\mathrm{dep}}(c')$. Without loss of generality, let $k < l \Leftrightarrow \mathrm{conn}(S)_k \prec \mathrm{conn}(S)_l$. We introduce an additional label $\mathrm{minarr}_k : V \to \Pi$

for each thread k that depicts for every node v the earliest arrival time at v using connections assigned to the kth thread. In the beginning, we initialize $\text{minarr}_k(v) = \infty$ and update $\text{minarr}_k(v) := \min(\text{minarr}_k(v), \text{arr}(v, i))$ each time thread k settles v for some connection i. Then, in addition to our self-pruning rule, we propose the following inter-thread-pruning rule: Each time we settle a queue element (v, i) with $\text{arr}(v, i) = \text{key}(v, i)$ in thread k, we check if there exists a thread l with $l > k$ for which $\text{minarr}_l(v) \leq \text{arr}(v, i)$. If this is the case, we know by the total ordering of the partition cells that there exists a connection j assigned to thread l with $\tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$ but $\text{arr}(v, j) \leq \text{arr}(v, i)$. In other words, connection i assigned to thread k is dominated by a connection j assigned to thread l. Thus, we may prune i at v the same way we do for self-pruning, that is, we do not relax outgoing edges of v for connection i. Correctness of this rule can be proven analogue to the the self-pruning rule described earlier.

In a shared memory set-up like in multicore servers, the values of $\text{minarr}_k(\cdot)$ can be communicated through the main memory, thus not imposing a significant overhead to the algorithm. Moreover, for practical use, it is sufficient to only check a constant number c of threads $\{k+1, \dots, k+c\}$, since dominating connections are less likely to be "far in the future," (i.e., assigned to threads $l \gg k$). Furthermore, we like to mention that our inter-thread-pruning rule does not guarantee pruning of dominated connections, since the priority queue is not shared across threads. However, in most cases, connections j with small arrival times prune connections i with high arrival time with respect to their particular thread. Hence, j is likely to be settled before i, thus, enabling pruning of i. An illustration of the inter-thread-pruning rule is presented in Algorithm 3.

ALGORITHM 3: Inter-Thread-Pruning Rule

```
Input: Thread number $k$, number of processors $p$, ...
1  ...;
2  $\text{minarr}_k(\cdot) \leftarrow \infty$;
3  ...;
4  while not Q.empty() do
5      ...
       // Inter-thread-pruning rule
6      if $\exists l$ with $k < l \leq p$ for which $\text{minarr}_l(v) \leq \text{arr}(v, i)$ then
7          $\text{arr}(v, i) \leftarrow \infty$;
8          continue ;
9      $\text{minarr}_k(v) \leftarrow \min(\text{minarr}_k(v), \text{arr}(v, i))$;
10     ...
```

5. STATION-TO-STATION QUERIES

Dijkstra's algorithm can be accelerated by precomputing auxiliary data as soon as we are only interested in point-to-point queries [Delling et al. 2009c]. In this section, we present how some of the ideas, for example, the stopping criterion, map to our new algorithm. Moreover, we show how the precomputation of certain connections improves the performance of our algorithm. The enhancements introduced in this section refer to the sequential algorithm (see Section 4.1). Thus, all results translate to our parallel algorithm naturally. Also note that they require a target station as input; in particular, they do not accelerate one-to-all queries.

5.1. Stopping Criterion

For point-to-point queries, Dijkstra's algorithm can stop the query as soon as the target node has been taken from the priority queue. In our case, that is, station-to-station, we

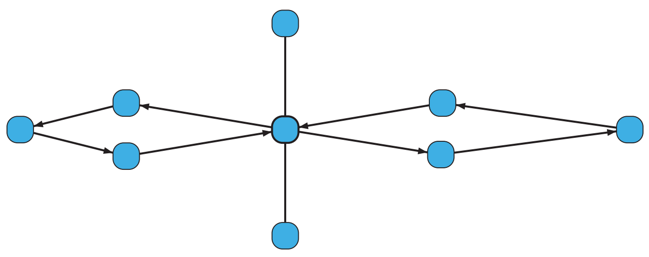


Fig. 5. The superstation graph G^* corresponding to the network as depicted by Figure 4.

can stop the query as soon as the target station T has its final label $\mathrm{arr}(T, i)$ for all i assigned. This can be achieved as follows. We maintain an index T_m, initialized with $-\infty$. Whenever we settle a connection i at our target station T, we set $T_m := \max\{i, T_m\}$. Then, we may prune all entries $q = (v, i) \in Q$ with $i \leq T_m$ (at any node v). We may stop the query as soon as the queue is empty.

THEOREM 5.1. *The stopping criterion is correct.*

PROOF. We need to show that no entry $q = (v, i) \in Q$ with $i \leq T_m$ can improve on the arrival time at T for the connection i. Let $q' = (v', i')$ be the responsible entry that has set T_m. Since $i \leq T_m$ holds, we know that regarding the departure times of the connections $\tau_{\mathrm{dep}}(c'_i) \geq \tau_{\mathrm{dep}}(c_i)$ holds as well. Moreover, since q is settled after q', we know that $\mathrm{arr}(v', i') \leq \mathrm{arr}(v, i)$ holds. In other words, it does not pay off to use train i at station S. □

5.2. Pruning with a Distance Table

Next, we show how to accelerate our station-to-station algorithm by pruning with the help of a distance table. Since a distance table computed directly on the model graph G would be too large to be practical, we use the smaller superstation network to compute the distance table. Intuitively, superstations are obtained by merging stations that are connected by a foot path.

Constructing Superstations. Consider the foot graph $G_F = (\mathcal{S}, \mathcal{F})$ whose nodes are exactly the stations in the timetable, and edges correspond to foot paths. As mentioned in Section 3, G_F is composed of small connected components of stations near the same intersections of the road network. Thus, we use G_F to obtain the superstation graph $G^* = (\mathcal{S}^*, E^*)$ in the following way. For each connected component in G_F, we create a superstation S^* in $\mathcal{S}^*$. An edge (S^*_i, S^*_j) is contained in E^* if and only if there exists a train running from any of the stations inside S^*_i to any of the stations inside S^*_j. We use $\mathrm{sst}(S)$ to denote the superstation of a station $S \in \mathcal{S}$. See Figure 5 for the superstation graph obtained from the network as depicted in Figure 4.

Furthermore, for our pruning rule, we need the notion of the diameter of a superstation S^*, which we define as the longest shortest path inside a component of G_F, but additionally taking the transfer times at its respective source and target stations into account. Formally, let $\mathrm{dist}_F(S_i, S_j)$ denote the shortest path distance between two stations S_i and S_j in G_F, then

$$\mathrm{diam}(S^*) := \max_{S_i, S_j \in S^*} \{\mathcal{T}(S_i) + \mathrm{dist}_F(S_i, S_j) + \mathcal{T}(S_j)\}. \tag{2}$$

Think of the diameter as an upper bound on the time you can spend walking inside a superstation.

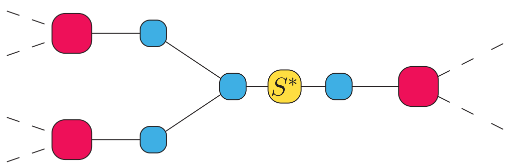


Fig. 6. Local and via superstations of a superstation S^*. Local superstations are indicated in blue, while via superstations are marked thicker in red.

Transfer Superstations. We are now given a subset $\mathcal{S}^*_{\text{trans}} \subseteq \mathcal{S}^*$ of superstations, called *transfer superstations* (think of them as important hubs in the network), and a distance table $D : \mathcal{S}^*_{\text{trans}} \times \mathcal{S}^*_{\text{trans}} \times \Pi \to \mathbb{N}_0$. The distance table returns, for each pair of superstations $S^*, T^* \in \mathcal{S}^*_{\text{trans}}$ the quickest way of getting from S^* to T^* at time $\tau \in \Pi$, that is, the earliest possible arrival time at T^* for any of the combinations of a station inside S^* and a station inside T^*. Note that we do not consider the diameters of S^* and T^* here. In other words, the distance table returns a lower bound on the distance between S^* and T^* at time τ.

Before explaining the pruning rule in detail, we need additional notion of local and via superstations. The set of local superstations $\text{local}(S^*) \subseteq \mathcal{S}^*$ of an arbitrary superstation S^* includes all superstations L^* such that there is a simple path from L^* to S^* that contains only nontransfer superstations in the superstation graph G^*. The set of transfer superstations that are adjacent to at least one local superstation of S^* are called the via super stations of S^*, denoted by $\text{via}(S^*) \subseteq \mathcal{S}^*_{\text{trans}}$. They basically separate $S^* \cup \text{local}(S^*)$ from any other superstation in G^*. Figure 6 gives a small example. In the special case of S^* being a transfer superstation, we set $\text{local}(S^*) = \emptyset$ and $\text{via}(S^*) = \{S^*\}$.

Applying the Distance Table. In the following, we call an S-T (with respective superstations S^* and T^*) query *local* if $S^* \in \text{local}(T^*)$, otherwise the query is called *global*. Note that a best connection of a global query must contain a via station of T^*. We accelerate global S-T queries by maintaining an upper bound $\mu_{i,j}$—initialized with ∞—for each connection i and each via superstation V^*_j of T^*. Whenever we settle a queue entry $q = (v, i)$ with $\text{sst}(v) \in \mathcal{S}^*_{\text{trans}}$, we set $\mu_{i,j} := \min\{\mu_{i,j}, \mathcal{D}(\text{sst}(v), V^*_j, \text{arr}(v, i) + \text{diam}(\text{sst}(v))) + \text{diam}(V^*_j)\}$ for all $V^*_j \in \text{via}(T^*)$. In other words, $\mu_{i,j}$ depicts an upper bound on the earliest train we can catch at V^*_j, even if we had to transfer (and potentially walk) at V^*_j. So, we may prune the search regarding q if

$$\forall V^*_j \in \text{via}(T^*) : \mathcal{D}(\text{sst}(v), V^*_j, \text{arr}(v, i)) > \mu_{i,j} \tag{3}$$

holds. In other words, we prune the search at v for a connection i if the path through $\text{sst}(v)$ is provably not important for the best path to any via station of $V^*_j \in \text{via}(T^*)$. Figure 7 gives a small example.

THEOREM 5.2. *Pruning based on a distance table is correct.*

The proof can be found in Appendix A. It follows the intuition that arriving at a time $\leq \mu_{i,j}$ at V^*_j ensures catching the optimal train toward T. Moreover, when we prune at v, the path through v yields a later arrival time at V^*_j than $\mu_{i,j}$. Thus, the path at v can be pruned, since there is no improvement over the path corresponding to $\mu_{i,j}$.

Determining $\text{via}(T^*)$. We determine the via superstations of T^* on the fly. During the initialization phase of the algorithm, we run a depth first search on the reverse superstation graph from T^*, pruning the search at stations $V^* \in \mathcal{S}^*_{\text{trans}}$. Any station $V^* \in \mathcal{S}^*_{\text{trans}}$ touched during the DFS is added to $\text{via}(T^*)$. Note that we may distinguish

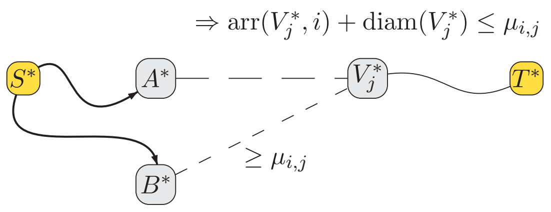

$\Rightarrow \mathrm{arr}(V_j^*, i) + \mathrm{diam}(V_j^*) \leq \mu_{i,j}$
S^*
A^*
V_j^*
T^*
B^*
$\geq \mu_{i,j}$

Fig. 7. Example for pruning via a distance table, given an S-T query. A^* and B^* are transfer superstations, and V_j^* is the via superstation of T^*. When settling a node at superstation A^*, we obtain that the arrival time at V_j^*, plus the diameter at V_j^* is smaller or equal to $\mu_{i,j}$. Hence, we may prune the query at B^* if the lower bound obtained from the distance table yields an arrival time at V_j^* greater than $\mu_{i,j}$.

local from global queries when computing via(T^*). As soon as our DFS visits S^*, the query is local, otherwise it is global.

5.3. Selecting Transfer Superstations

The efficiency of pruning via a distance table highly depends on which superstations are selected for $\mathcal{S}^*_{\text{trans}}$. In Schulz et al. [1999], the authors propose to identify important stations by a given "importance" value provided by the input. However, such values are not available for all inputs. Hence, we compute importance values heuristically. Consider the aforementioned superstation graph G^*. We augment G^* with constant edge weights $\ell(S_i^*, S_j^*)$. Therefore, consider all connection points of trains running from any station inside S_i^* to any station inside S_j^*, denoted by $\mathcal{P}(S_i^*, S_j^*)$. Then, we define $\ell(S_i^*, S_j^*)$ to be the expected travel time to get from S_i^* to S_j^*, solely using connections from $\mathcal{P}(S_i^*, S_j^*)$. Note that the expected travel time also includes waiting times between connections. We now use G^* together with ℓ to select important stations by one of the following methods.

Contraction Hierarchies. A fast approach for selecting important superstations is using contraction hierarchies [Geisberger et al. 2008]. A contraction routine iteratively removes unimportant nodes from G^* and adds shortcuts in order to preserve the distances between nonremoved nodes. We stop as soon as the number of noncontracted nodes is c and mark these superstations as important.

Shortest Path Covers. Abraham et al. [2011] observed that contraction hierarchies do a poor job picking the most important nodes of a road network. Hence, they use shortest path covers for selecting them. Unfortunately, computing such covers is hard, but the authors propose a polynomial time $O(\log n)$ approximation algorithm, which we adapt to our problem in the following way. Beginning with $\mathcal{S}^*_{\text{trans}}$ as an empty set, we iteratively determine the next most important superstation as the one that covers most (yet uncovered by $\mathcal{S}^*_{\text{trans}}$) shortest paths in G^*. We stop as soon as we selected c transfer superstations. Note that this algorithm requires c times the computation of all-pairs shortest path in G^*. However, G^* is sufficiently small for this approach to still be practical.

6. EXPERIMENTS

We conducted our experiments on up to 48 cores (4 CPUs, 8 NUMA-nodes, 6 cores per NUMA-node) of an AMD Opteron 6172 machine running SUSE Linux. The machine is clocked at 2.1GHz, has 256GiB of RAM, 512KiB of L2 cache per core, and 6MiB of L3 cache per NUMA-node. The program was compiled with GCC 4.5, using optimization level 3. Our implementation is written in C++, solely using the STL and Boost at some points. As parallelization framework, we use OpenMP, and a 4-heap as priority queue.

To avoid congestion of the memory bus, we keep a copy of the graph in the designated memory area of each NUMA-node.

Inputs. We use three different public transportation networks as inputs: the Los Angeles County Metro (15,146 stops and 979,283 elementary connections) and the complete network of MTA New York including buses, ferries, and subways (16,897 stops and 2,062,846 elementary connections). Moreover, we use the railway network of Europe. It has 30,517 stations and 1,691,691 elementary connections. The networks of Los Angeles, and New York were created based on the timetable of March 1, 2011. The European railway network is based on the timetable of the winter period 1996/1997. Note that the local networks are much denser than the railway network, that is, the connections per station ratio is significantly higher there. Moreover, our data of the European railway network contains real minimum transfer times for all stations. For the bus networks of New York and Los Angeles, this data was not available to us. Hence, we set a minimum transfer time of 90 seconds for all bus stops. Foot paths are computed on all networks (see Section 3.2).

The timetable data of the local city networks is publicly available through General Transit Data Feeds, while the timetable data of the European railway networks was kindly given to us by HaCon Ingenieurgesellschaft. See Figure 8 for a visualization of the Los Angeles superstation graph.

6.1. Modeling

Our first set of experiments focuses on evaluating the models, as presented in Section 3. In particular, we compare the realistic time-dependent model with our new coloring-based model. Table I shows figures on all of our inputs for both models. We observe that using the coloring-based model reduces the graph size for all inputs. The average number of route nodes per station can be reduced by a factor of between 6.1 (New York) and 12.3 (Los Angeles).

In addition, we observe for many stations that there exists no conflict between any connections. In fact, we can merge the only route node with its station node for 79.5 % of the stations in the Los Angeles network. On the other hand, on the European railway network about two thirds of the stations contain more than one route node, which stems from the fact that in this network the transfer times are much higher, thus, increasing the likelihood of two trains conflicting.

Since the coloring-based model yields smaller graphs, which improves performance of all our algorithms compared to the classic route-based model, we use the coloring-based model for all subsequent experiments.

6.2. One-to-All Queries

Our second set of experiments focuses on the question how well our parallel self-pruning connection-setting algorithm (PSPCS) performs if executed on a varying number of cores. Therefore, we run 1000 one-to-all queries with the source station picked uniformly at random. We report the average number of connections taken from the priority queue (sum over all cores) and the average execution time of a query. Table II reports these figures for a varying number (between 1 and 48) of cores and different load balancing strategies. To evaluate the load balancing, we report the standard deviation with respect to the execution times of the individual threads. In other words, a low deviation shows a good balance, whereas a high deviation indicates that some threads are often idle. For comparison, we also report the performance of a label-correcting (LC) approach (see Section 2), as well as of our connection-setting algorithm (CS) without self-pruning enabled (think of it as a simultaneous execution of a bunch of Dijkstras for every connection out of the source station). Regarding LC, for better comparability,

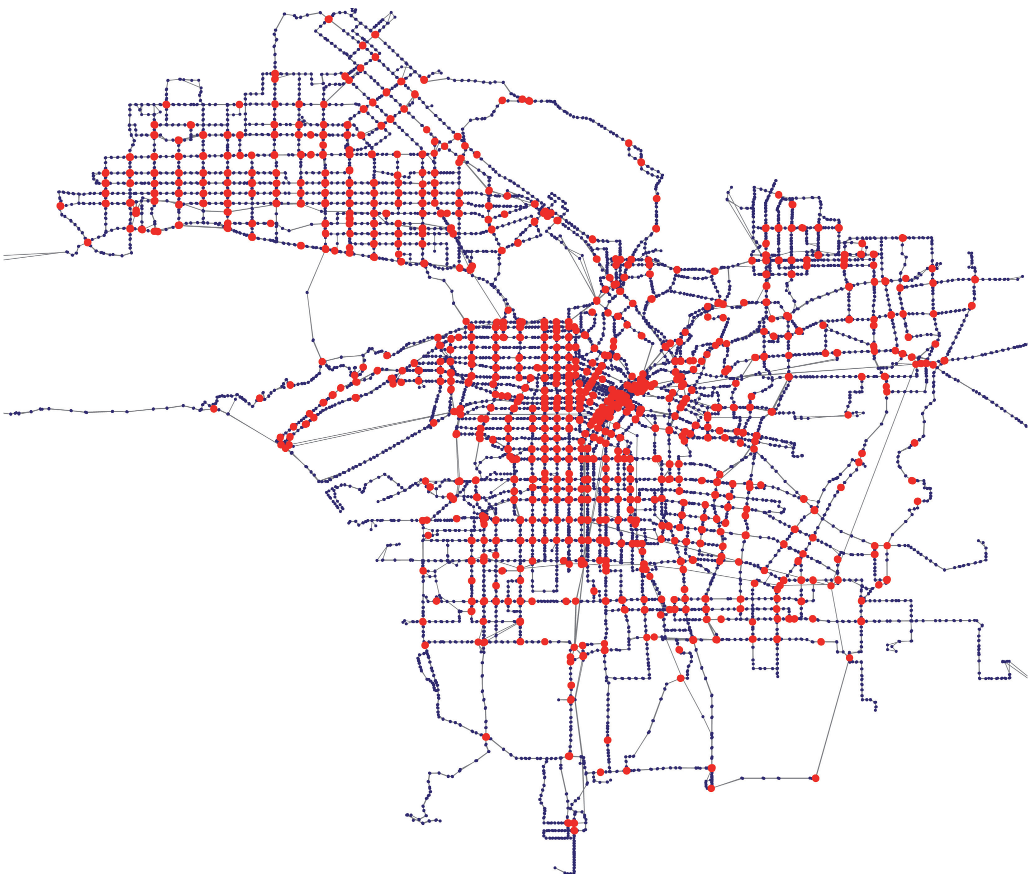


Fig. 8. Excerpt of the superstation graph of one of the Los Angeles County Metro network. Transfer superstations are highlighted in thick red (see Section 5.2). In this figure, we used the contraction hierarchy method to select 10 % of the stations as a transfer station.

Table I. Comparison of the Route-Based Model to Our Coloring-Based Model
We report the number of stations in the timetable, the number of nodes and edges in the graph, as well as the number of route nodes per station and the percentage of stations that are merged (i.e., consist of only one route node).

	Los Angeles		New York		Europe	
	Routes	Colored	Routes	Colored	Routes	Colored
# Stations	15,146	15,146	16,897	16,897	30,517	30,517
# Nodes	89,111	21,680	79,881	27,203	515,062	83,732
# Edges	235,394	54,896	198,232	67,105	1,412,082	392,675
Rt.-Nodes p. St.	4.9	0.4	3.7	0.6	15.9	1.7
% Merged St.	—	79.5	—	71.7	—	33.2

the number of connections here indicates the sum of the sizes of the connection-labels taken from the priority queue.

We observe that our algorithm scales pretty well with increasing number of cores. On both the Los Angeles and New York networks, the number of settled connections is only increasing mildly with the number of cores. Therefore, on 12 cores, we have a speed-up factor of around 4 to 8 compared to an execution on 1 core. On 48 cores, the

Table II. One-to-All Profile-Queries with Our Parallel Self-Pruning Connection-Setting Algorithm (PSPCS) on 1, 3, 6, 12, 24, and 48 Cores with Different Load Balancing Strategies, Compared to a Label-Correcting (LC) Approach

The column *spd.-up* indicates the time speed-up of a multicore run over a single-core execution of PSPCS. The column *Dev* reports the standard deviation with respect to the execution times of the individual threads indicating how well the threads are balanced (lower values are better).

	Los Angeles				New York				Europe			
	Settl.	Time	Spd.	Dev.	Settl.	Time	Spd.	Dev.	Settl.	Time	Spd.	Dev.
p	Conns	[ms]	Up	[%]	Conns	[ms]	Up	[%]	Conns	[ms]	Up	[%]
1	844,852	374.0	1.0	—	1,606,515	931.5	1.0	—	550,912	394.9	1.0	—
					EQUICONN							
3	855,676	131.5	2.8	9.1	1,625,545	391.5	2.4	13.9	666,889	162.4	2.4	15.3
6	871,978	72.1	5.2	12.9	1,654,798	165.9	5.6	12.6	843,695	139.5	2.8	18.8
12	904,149	66.1	5.7	20.9	1,711,439	118.1	7.9	16.8	1,172,269	100.9	3.9	15.0
24	967,339	46.4	8.1	22.6	1,822,735	106.9	8.7	20.5	1,709,985	125.8	3.1	21.4
48	1,079,224	21.4	17.5	13.9	2,038,022	57.0	16.3	18.5	2,393,664	109.7	3.6	20.9
					EQUITIME							
3	853,629	153.5	2.4	18.9	1,623,518	384.6	2.4	24.5	651,022	163.7	2.4	17.5
6	865,679	85.6	4.4	25.6	1,645,273	201.0	4.6	26.4	799,641	172.6	2.3	23.4
12	891,822	90.7	4.1	24.9	1,692,424	132.9	7.0	23.7	1,065,354	116.5	3.4	18.2
24	943,625	55.2	6.8	23.4	1,783,835	117.5	7.9	22.2	1,474,137	136.1	2.9	21.4
48	1,022,931	38.2	9.8	21.1	1,953,405	69.7	13.4	19.9	1,970,312	117.3	3.4	21.2
					KMEANS							
3	852,122	142.2	2.6	17.8	1,619,993	361.8	2.6	22.7	648,190	166.0	2.4	19.1
6	864,301	87.2	4.3	24.5	1,643,853	190.9	4.9	25.1	810,833	113.9	3.5	18.8
12	893,412	89.5	4.2	24.7	1,693,146	171.5	5.4	21.3	1,128,571	118.0	3.3	18.0
24	949,905	44.6	8.4	21.5	1,795,074	92.2	10.1	19.8	1,644,280	122.6	3.2	21.3
48	1,057,201	31.0	12.0	20.8	2,002,726	58.5	15.9	19.0	2,276,361	107.2	3.7	21.8
CS	1,352,894	586.7	—	—	3,327,697	1,965.4	—	—	4,377,790	3,843.3	—	—
LC	2,529,009	445.9	—	—	4,656,646	748.4	—	—	1,278,093	635.3	—	—

speed-up factor is between 3.6 (Europe) and 17.5 (Los Angeles). The relatively mild speed-ups on Europe compared to the other networks are explained by the fact that the average number of connections at a station is much smaller than in the dense bus networks. Still, on all cores, we are able to compute all quickest connections of a day in less than 0.2 seconds. Note that this value is achieved without any preprocessing; hence, we can directly use this approach in a fully dynamic scenario (as discussed in Müller–Hannemann et al. [2008]).

Regarding load balancing, we observe that using an equal number of connections (equiconn) yields. On average, the lowest query times (and deviation). In few occasions, equal time slots (equitime) or *k*-means yields better results, but over all inputs and number of cores, equiconn seems to be the best choice. Hence, we use equiconn as default strategy for all further multicore experiments. Another, not very surprising, observation is that the deviation increases with increasing number of cores. The more cores we use, the harder a perfect balancing can be achieved.

Comparing our new connection-setting to the LC approach, we observe that PSPCS outperforms LC on Los Angeles and Europe even when PSPCS is executed on a single core only. The main reason for this is that the number of connections investigated during execution is much smaller for PSPCS than for LC. On the network of New York, LC is slightly faster than PSPCS on a single core, but already on 3 cores, PSPCS outperforms LC by a factor of 2. Note that the number of priority queue operations for LC is up to four times lower than for PSPCS. Hence, the advantage of PSPCS in number of settled connections does not yield the same speed-up in query times.

Table III. Comparing Our Self-Pruning Connection-Setting Algorithm with and without Inter-Thread-Pruning Enabled on a Varying Number of Cores p
The column spd.-up refers to the speed-up in time over a single-core execution of the same algorithm.

	Without ITP			With ITP		
p	Settl. Conns	Time [ms]	Spd. Up	Settl. Conns	Time [ms]	Spd. Up
	Los Angeles					
1	844,852	374.0	1.0	838,331	381.5	1.0
3	855,676	131.5	2.8	836,759	215.9	1.8
6	871,978	72.1	5.2	835,494	72.7	5.2
12	904,149	66.1	5.7	836,186	41.7	9.1
24	967,339	46.4	8.1	856,631	47.6	8.0
48	1,079,224	21.4	17.5	919,060	32.9	11.6
	New York					
1	1,606,515	931.5	1.0	1,595,121	958.3	1.0
3	1,625,545	391.5	2.4	1,594,007	413.8	2.3
6	1,654,798	165.9	5.6	1,594,153	173.9	5.5
12	1,711,439	118.1	7.9	1,600,842	158.0	6.1
24	1,822,735	106.9	8.7	1,625,629	104.7	9.2
48	2,038,022	57.0	16.3	1,711,238	59.5	16.1
	Europe					
1	550,912	394.9	1.0	511,203	373.7	1.0
3	666,889	162.4	2.4	528,588	224.5	1.7
6	843,695	139.5	2.8	610,796	100.2	3.7
12	1,172,269	100.9	3.9	824,653	119.5	3.1
24	1,709,985	125.8	3.1	1,230,380	109.8	3.4
48	2,393,664	109.7	3.6	1,753,982	106.7	3.5

When comparing the single-core execution of PSPCS to a connection-setting algorithm without self-pruning (CS), we observe that enabling self-pruning makes a significant difference in both settled connections and running time. Most notably, on Europe, the number of connections drops from 4.3 million to 0.5 million, together with a drop from 3.8 to 0.4 seconds in running time. The difference is less pronounced on our bus networks, which is due to the fact that these networks inherit a weaker hierarchy, that is, there are fewer express trains (buses, respectively) that prune local (slow) trains.

Inter-Thread-Pruning. In our previous experiment (see Table II), we did not enable inter-thread-pruning (see Section 4). Hence, in Table III, we compare our self-pruning connection-setting algorithm with and without inter-thread-pruning on a varying number of cores. Thereby, we limit the number of threads we check for a dominating connection to 1.

We observe that enabling inter-thread-pruning helps to reduce the number of settled connections in all scenarios. Interestingly, even for a sequential execution, we are able to reduce the number of settled connections. Here, the “thread” we check for a dominating connection is the thread itself. By these means, we are able to prune over the boundary of the time period, for example, for a connection after midnight to prune a connection in the late evening (remember that the timetable is periodic).

While the number of settled connections decreases with inter-thread-pruning, the additional computational overhead in the algorithm does not always justify the smaller number of settled connections. Hence, the gain in query time is mostly only small. In the network of New York, enabling inter-thread-pruning even leads to slightly worse query times. We conclude that the benefit of inter-thread-pruning is small. Thus, for the sake of simplicity and reduced communication overhead of the algorithm, we disable inter-thread-pruning in subsequent experiments.

6.3. Station-to-Station Queries

In this section, we evaluate our algorithm in a station-to-station scenario. We use all 48 cores as default and evaluate the impact of different distance table sizes. Since these tables need to be precomputed, we also report the preprocessing time and the size of the tables in megabytes. Furthermore, we report the average number of via superstations per superstation if it were the target of a query. The distance tables are computed by running our parallel one-to-all algorithm on 48 cores from every transfer superstation. As strategies for selecting transfer stations, we evaluate both the greedy cover (GC) and the contraction hierarchies (CH) approach (see Section 5.3). Table IV gives an overview over the obtained results.

We observe that compared to Table II, the stopping criterion alone (which requires no preprocessing) already accelerates queries by up to 89 % (Europe).

When we additionally use a distance table, we can accelerate our queries further. We observe that the size of the distance table has a high impact on the query performance, especially for smaller tables. Augmenting only 2.5 % of the superstations to transfer superstations hardly accelerates queries. In fact, especially on the very dense network of Los Angeles, the performance even degrades for small tables, as the average number of required via superstations per target superstation is too high. Note that we need to separate the target superstation by via stations from the network (see Section 5.2) hence, when more superstations are augmented as transfer stations, fewer of them are required to separate the target superstation.

On the other hand, augmenting 10% of the superstations to transfer superstations yields additional speed-ups between 2.0 and 3.6, depending on the input. Larger distance tables hardly pay off: The size of the table increases significantly, and the gain in query performance is little. Hence, selecting 10% to 15% of the stations as transfer stations seems to be a good compromise.

Regarding the preprocessing effort, we observe that with increasing number of transfer stations, the size of the tables and the preprocessing time increases as well. Moreover, while the part of the preprocessing time spent on selecting transfer superstations is negligible when using the CH method (CH), it is significant for the gc method. This is because for each selected superstation, we need to run an all-pairs shortest-path computation on the (sparse) superstation graph, each of which takes time $O(|\mathcal{S}^*|^2 \log |\mathcal{S}^*|)$. Recall that $\mathcal{S}^*$ is the set of superstations.

However, when using 10% transfer super stations selected by the CH method, we can compute the distance tables between 6 and 10 minutes, while the tables consume less than 1.5GiB space for all of our inputs. For this scenario, we are able to compute all quickest connections on all inputs in less than 16.1ms time.

6.4. A Different Machine

In this final experiment, we run our parallel algorithm on different hardware. Here, we use a dual Intel Xeon 5430 machine that has 8 cores on two NUMA-nodes clocked at 2.6GHz, 32GiB of RAM and 2 × 1MiB of L2 cache. To evaluate our algorithm on this machine, we use the one-to-all scenario, similarly to Section 6.2; however, for the sake of simplicity, only for the equal connections distribution strategy. Table V shows the obtained results.

We observe that the figures of the sequential algorithms coincide with those in Table II, except that they are scaled: The Xeon machine is slightly faster, since it has a higher clock frequency (2.6GHz compared to 2.1GHz of the Opteron machine).

Table IV. Performance of Our Parallel Self-Pruning Connection-Setting Algorithm (PSPCS) with Stopping Criterion Enabled

As load-balancing strategy we use the equal connections method. Moreover, we prune by a distance table as described in Section 5.2. The number of transfer superstations is given in percentage of input super stations.

	Los Angeles						New York					
	Preprocessing			Queries			Preprocessing			Queries		
Size [%]	Time [m:s]	Space [MiB]	Via St.	Settl. Conns	Time [ms]	Spd. Up	Time [m:s]	Space [MiB]	Via St.	Settl. Conns	Time [ms]	Spd. Up
0	—	—	—	614,254	19.8	1.0	—	—	—	1,188,870	35.4	1.0
	GC											
2.5	2:48	52.5	39.3	392,872	25.7	0.8	5:07	115.0	20.0	547,307	32.3	1.1
5.0	5:15	171.7	8.4	214,620	12.8	1.5	9:57	394.8	4.7	280,011	18.9	1.9
10.0	10:50	577.5	3.7	141,348	10.0	2.0	20:29	1,352.7	2.6	198,315	15.7	2.3
15.0	16:29	1,189.5	2.8	126,509	9.9	2.0	31:45	2,807.8	2.1	181,965	14.8	2.4
20.0	22:24	1,980.7	2.5	121,244	9.8	2.0	43:57	4,791.7	1.9	174,438	14.5	2.4
	CH											
2.5	1:09	53.3	295.4	565,832	60.7	0.3	1:43	110.9	165.2	784,445	82.8	0.4
5.0	2:41	196.3	28.0	250,452	26.0	0.8	4:31	412.0	8.1	299,851	12.8	2.8
10.0	6:12	659.0	3.8	128,265	9.8	2.0	10:50	1,500.4	2.6	183,729	10.7	3.3
15.0	9:35	1,323.1	2.7	109,229	8.0	2.5	17:23	3,126.8	1.9	167,777	9.5	3.7
20.0	13:12	2,166.4	2.3	110,502	8.8	2.2	24:53	5,213.8	1.7	162,283	11.9	3.0

	Europe					
	Preprocessing			Queries		
Size [%]	Time [m:s]	Space [MiB]	Via St.	Settl. Conns	Time [ms]	Spd. Up
0	—	—	—	1,266,720	58.0	1.0
	GC					
2.5	44:59	71.9	5.9	347,156	21.5	2.7
5.0	84:13	261.3	3.0	261,894	17.8	3.3
10.0	161:41	930.6	2.2	256,514	18.4	3.2
15.0	216:31	1,956.0	2.1	263,867	19.4	3.0
20.0	280:02	3,354.7	2.0	260,812	18.0	3.2
	CH					
2.5	2:32	72.7	42.7	507,466	41.8	1.4
5.0	5:21	269.5	4.9	280,494	19.1	3.0
10.0	12:01	985.8	2.2	220,550	16.1	3.6
15.0	18:37	2,068.6	1.9	208,599	14.1	4.1
20.0	27:01	3,492.4	1.7	218,388	15.4	3.8

Table V. One-to-All Profile-Queries as in Table II, but on 1, 2, 4, and 8 Cores of an Intel Xeon 5430 Machine

Regarding PSPCS, we only report results for the Equiconn distribution strategy.

	Los Angeles				New York				Europe			
p	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]
1	844,852	303.5	1.0	—	1,606,515	725.2	1.0	—	550,912	293.6	1.0	—
	EQUICONN											
2	849,553	196.1	1.5	9.5	1,615,576	440.0	1.6	7.4	608,951	166.2	1.8	12.6
4	861,235	92.1	3.3	9.6	1,636,196	224.4	3.2	10.4	726,382	113.4	2.6	15.8
8	882,905	53.3	5.7	13.7	1,674,558	131.0	5.5	11.7	955,412	83.7	3.5	14.2
CS	1,352,894	451.1	—	—	3,327,697	1,363.7	—	—	4,377,790	2,881.5	—	—
LC	2,529,009	356.2	—	—	4,656,646	589.0	—	—	1,278,093	519.3	—	—

Regarding the parallel performance, we observe speed-ups in the range of 3.5 to 5.7 on 8 cores. Again, the number of settled connections on the networks of Los Angeles and New York is almost independent of the number of cores, even without inter-thread-pruning (which is, again, disabled in this experiment). Concluding, we are able to compute all best connections to all stations in under 131ms on average, in all of our networks on this machine.

7. CONCLUSION

In this work, we presented a novel parallel algorithm for computing all best connections of a day from a given station to all other stations in a public transportation network in a single query. To this extent, we exploited the special structure of travel-time functions in such networks and the fact that only few connections are useful when traveling sufficiently far away. Introducing the concept of connection-setting, we showed how to transfer the label-setting property of Dijkstra's algorithm to profile-searches in transportation networks. By the fact that the outgoing connections of the source station can be distributed to different processors, our algorithm is easy to use in a multicore set-up yielding excellent speed-ups on today's computers. Moreover, utilizing the very same algorithm to precompute connections between important stations, we can greatly accelerate station-to-station queries.

Regarding future work, it will be interesting to incorporate multicriteria connections, for example, minimizing the number of transfers or incorporating fare zones, which is relevant especially in local networks. The main challenge here is to keep up the connection-setting property and to find efficient criteria for self-pruning in such a scenario. Moreover, our algorithm can be seen as a replacement for Dijkstra's algorithm which is the basis for most of today's speed-up techniques (e. g., Delling [2011]). Hence, we are interested in applying those techniques to our new connection-setting approach.

APPENDIX

A. ADDITIONAL PROOFS

Proof of Theorem 5.2

We are proving the overall correctness by showing the correctness for each connection i separately. Thus, let i be a fixed connection index and $P = [S, \dots, T]$ the shortest path of a global S-T-query of connetion i. Note that if S-T is a local query, no pruning is applied (and there is nothing to prove).

Now, let $\mathrm{arr}_{\mathrm{opt}}(T, i)$ denote the (optimal) arrival time at T of the path P (i.e., using connection i). Moreover, let T^* be the superstation of the target station T. To show the main theorem, we prove a series of lemmas first.

LEMMA A.1. *For all tuples* $(v, V_j^*) \in V \times \mathrm{via}(T^*)$ *with* $\mathrm{sst}(v) \in \mathcal{S}^*_{trans}$, *it holds that*

$$\begin{aligned}\mathrm{arr}_{\mathrm{opt}}(T, i) \leq \, & \underbrace{\mathcal{D}\big(\,\mathrm{sst}(v), V_j^*, \mathrm{arr}(v, i) + \mathrm{diam}(\mathrm{sst}(v))\big) + \mathrm{diam}(V_j^*)}_{=:\, \mu_{i,v,j}} \\ & + \mathrm{dist}(V_j^*, T, \mu_{i,v,j}).\end{aligned} \tag{4}$$

PROOF. Assume that the equation is false, and the right-hand side yields an arrival time at T, which is earlier than $\mathrm{arr}_{\mathrm{opt}}(T, i)$. Then, the path induced by the right-hand side of the equation yields a shorter path to T, which is a contradiction to $\mathrm{arr}_{\mathrm{opt}}(T, i)$ being optimal. □

This proves that using the distance table via V_j^* at any node v yields an upper bound on the arrival time at T (for connection i). Since this is true at all nodes $v \in V$ (for which $\mathrm{sst}(v) \in \mathcal{S}^*_{\mathrm{trans}}$), the following corrolary follows immediately.

COROLLARY A.2. *Let* $\mu_{i,j} := \min_{v\in V, \mathrm{sst}(v)\in\mathcal{S}^*_{trans}}(\mu_{i,v,j})$*. Then, it holds that* $\mathrm{arr}_{\mathrm{opt}}(T,i) \leq \mu_{i,j} + \mathrm{dist}(V^*_j, T, \mu_{i,j})$*.*

Note that in the algorithm, $\mu_{i,j}$ is maintained exactly the way it is defined in Lemma A.1, and the minimum operation is applied iteratively each time we settle a node v for which $\mathrm{sst}(v) \in \mathcal{S}^*_{\mathrm{trans}}$ holds. Hence, the inequality of Corollary A.2 holds in the algorithm, as well.

Next, consider the combined shortest S-v-V^*_j-T path of connection i with arrival time $\mathrm{arr}_{V^*_j}(T,i)$ at T. We can lower bound $\mathrm{arr}_{V^*_j}(T,i)$ by the distance table as in the following lemma.

LEMMA A.3. *For all tuples* $(v, V^*_j) \in V \times \mathrm{via}(T^*)$ *with* $\mathrm{sst}(v) \in \mathcal{S}^*_{trans}$*, it holds that*

$$\mathrm{arr}_{V^*_j}(T,i) \geq \underbrace{\mathcal{D}(\mathrm{sst}(v), V^*_j, \mathrm{arr}(v,i))}_{=:\,\gamma_{i,v,j}} + \mathrm{dist}(V^*_j, T, \gamma_{i,v,j}), \tag{5}$$

where $\mathrm{arr}_{V^*_j}(T,i)$ *depicts the arrival time of the combined shortest* S-v-V^*_j-T *path.*

PROOF. Let us assume that the right-hand side of the equation evaluates to $\mathrm{arr}'_{V^*_j}(T,i)$ with $\mathrm{arr}'_{V^*_j}(T,i) > \mathrm{arr}_{V^*_j}(T,i)$. Then, because both $\mathcal{D}(\mathrm{sst}(v), V^*_j, \cdot)$ and $\mathrm{dist}(V^*_j, T, \cdot)$ are fulfilling the FIFO-property, the departure time τ of $\mathcal{D}(\mathrm{sst}(v), V^*_j, \tau)$ of the path corresponding to $\mathrm{arr}_{V^*_j}(T,i)$ on the left-hand side of the inequation has to be strictly smaller than $\mathrm{arr}(v,i)$ at v. But, this cannot be true, since the path induced by $\mathrm{arr}_{V^*_j}(T,i)$ is assumed to be the shortest path. □

Intuitively, Lemma A.3 proves that any valid (shortest) S-T-path that goes via v and V^*_j has to be at least as long as the "path" that completely ignores walking at both $\mathrm{sst}(v)$ and V^*_j (and basically acts as if you can catch any train at $\mathrm{sst}(v)$ and V^*_j instantaneously).

Next, we establish that when we apply our pruning rule during the algorithm, we do not prune a path that is important (i.e., we only prune paths which are provably not shortest to T).

LEMMA A.4. *Let* $v \in V$ *be a node with* $\mathrm{sst}(v) \in \mathcal{S}^*_{trans}$*, and let* $\gamma_{i,v,j} > \mu_{i,j}$*. Then,*

$$\gamma_{i,v,j} + \mathrm{dist}(V^*_j, T, \gamma_{v,i,j}) \geq \mu_{i,j} + \mathrm{dist}(V^*_j, T, \mu i, j) \tag{6}$$

holds.

PROOF. This follows immediately from the FIFO-property of $\mathrm{dist}(V^*_j, T, \cdot)$. □

We can now conclude our prove of Theorem 5.2. Hence, let $v \in V$ be a node with $\mathrm{sst}(v) \in \mathcal{S}^*_{\mathrm{trans}}$, where the pruning rule is potentially applied. Then, from Lemma (A.3), (A.4) and Corollary (A.2), we get for a via super station $V^*_j \in \mathrm{via}(T^*)$ that

$$\gamma_{v,i,j} > \mu_{i,j} \Rightarrow \mathrm{arr}_{V^*_j}(T,i) \geq \underbrace{\mu_{i,j} + \mathrm{dist}(V^*_j, T, \mu_{i,j})}_{=:\,\psi} \geq \mathrm{arr}_{\mathrm{opt}}(T,i). \tag{7}$$

Since our algorithm keeps track of $\mu_{i,j}$ as the minimum over all $\mu_{i,x,j}$ with $\mathrm{sst}(x) \in \mathcal{S}^*_{\mathrm{trans}}$, the path that corresponds to $\mu_{i,j}$ is not pruned. Hence, at the point where v is pruned, a path with arrival time ψ toward V^*_j is guaranteed to be found. Since v is only pruned if Equation (6) holds for *all* $V^*_j \in \mathrm{via}(T^*)$, it follows that $v \notin P$; thus, v is not important for the shortest S-T-path. □

REFERENCES

ABRAHAM, I., DELLING, D., GOLDBERG, A. V., AND WERNECK, R. F. 2011. A hub-based labeling algorithm for shortest paths on road networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Lecture Notes in Computer Science, vol. 6630, Springer, Berlin. 230–241.

ADAMSON, P. AND TICK, E. 1991. Greedy partitioned algorithms for the shortest path problem. *Int. J. Parallel Program. 20*, 271–298.

AHUJA, R. K., MÖHRING, R. H., AND ZAROLIAGIS, C., Eds. 2009. *Robust and Online Large-Scale Optimization*. Lecture Notes in Computer Science Series, vol. 5868. Springer, Berlin.

BAST, H. 2009. Car or public transport—two worlds. In *Efficient Algorithms*, Lecture Notes in Computer Science, vol. 5760. Springer, Berlin. 355–367.

BAST, H., CARLSSON, E., EIGENWILLIG, A., GEISBERGER, R., HARRELSON, C., RAYCHEV, V., AND VIGER, F. 2010. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. Lecture Notes in Computer Science, vol. 6346. Springer, Berlin. 290–301.

BAUER, R. AND DELLING, D. 2009. SHARC: fast and robust unidirectional routing. *ACM J. Exp. Algor. 14,* 2.4, 1–29.

BAUER, R., DELLING, D., AND WAGNER, D. 2011. Experimental study on speed-up techniques for timetable information systems. *Networks 57,* 1, 38–52.

CHANDY, K. M. AND MISRA, J. 1982. Distributed computation on graphs: shortest path algorithms. *Comm. ACM 25,* 11, 833–837.

DEAN, B. C. 1999. Continuous-time dynamic shortest path algorithms. M.S. thesis, Massachusetts Institute of Technology.

DELLING, D. 2011. Time-dependent SHARC-routing. *Algorithmica 60,* 1, 60–94.

DELLING, D., GOLDBERG, A. V., NOWATZYK, A., AND WERNECK, R. F. 2012. PHAST: Hardware-accelerated shortest path trees. *J. Parallel Distrib. Comput.*

DELLING, D., KATZ, B., AND PAJOR, T. 2010. Parallel computation of best connections in public transportation networks. In *Proceeding of the 24th International Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE, Los Alamitos, CA. 1–12.

DELLING, D., PAJOR, T., AND WAGNER, D. 2009a. Accelerating multi-modal route planning by access-nodes. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*. Lecture Notes in Computer Science Series, vol. 5757, Springer, Berlin. 587–598.

DELLING, D., PAJOR, T., AND WAGNER, D. 2009b. Engineering time-expanded graphs for faster timetable information. Lecture Notes in Computer Science, vol. 5868, Springer, Berlin. 182–206.

DELLING, D., SANDERS, P., SCHULTES, D., AND WAGNER, D. 2009c. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*. Lecture Notes in Computer Science, vol. 5515, Springer, Berlin. 117–139.

DELLING, D. AND WAGNER, D. 2009. Time-dependent route planning. Lecture Notes in Computer Science, vol. 5868, Springer, Berlin. 207–230.

DISSER, Y., MÜLLER–HANNEMANN, M., AND SCHNEE, M. 2008. Multi-criteria shortest paths in time-dependent train networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Lecture Notes in Computer Science, vol. 5038, Springer, Berlin. 347–361.

DRISCOLL, J. R., GABOW, H. N., SHRAIRMAN, R., AND TARJAN, R. E. 1988. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Comm. ACM 31,* 11, 1343–1354.

GEISBERGER, R. 2010. Contraction of timetable networks with realistic transfers. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Lecture Notes in Computer Science, vol. 6049, Springer, Berlin. 71–82.

GEISBERGER, R., SANDERS, P., SCHULTES, D., AND DELLING, D. 2008. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, Lecture Notes in Computer Science, vol. 5038, Springer, Berlin. 319–333.

HRIBAR, M. R., TAYLOR, V. E., AND BOYCE, D. E. 2001. Implementing parallel shortest path for parallel transportation applications. *Parallel Comput. 27*, 1537–1568.

MACQUEEN, J. 1967. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. 281–297.

MADDURI, K., BADER, D. A., BERRY, J. W., AND CROBAK, J. R. 2007. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 23–35.

McGeoch, C. C., Ed. 2008. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Lecture Notes in Computer Science, vol. 5038, Springer, Berlin.

Meyer, U. and Sanders, P. 1998. Δ-stepping : a parallel single source shortest path algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA'98)*. Lecture Notes in Computer Science, vol. 1461, Springer, Berlin. 393–404.

Müller–Hannemann, M. and Schnee, M. 2007. Finding all attractive train connections by multi-criteria Pareto search. In *Algorithmic Methods for Railway Optimization*. Lecture Notes in Computer Science, vol. 4359. Springer, Berlin. 246–263.

Müller–Hannemann, M., Schnee, M., and Frede, L. 2008. Efficient on-trip timetable information in the presence of delays. In *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*. OpenAccess Series in Informatics (OASIcs).

Orda, A. and Rom, R. 1990. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *J. ACM 37,* 3, 607–625.

Paige, R. C. and Kruskal, C. P. 1985. Parallel algorithms for shortest path problems. In *Proceedings of the International Conference on Parallel Processing*. IEEE, 14–20.

Pyrga, E., Schulz, F., Wagner, D., and Zaroliagis, C. 2008. Efficient models for timetable information in public transportation systems. *ACM J. Exp. Algor. 12,* 2.4, 1–39.

Ramarao, K. V. S. and Venkatesan, S. 1992. On finding and updating shortest paths distributively. *J. Algo. 13*, 235–257.

Schulz, F., Wagner, D., and Weihe, K. 1999. Dijkstra's algorithm on-line: an empirical case study from public railroad transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*. Lecture Notes in Computer Science, vol. 1668, Springer, Berlin. 110–123.

Träff, J. L. 1995. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Comput. 21*, 1505–1532.

Received August 2011; accepted May 2012