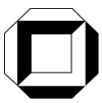
Universität Karlsruhe - Fakultät für Informatik - Bibliothek - Postfach 6980 - 76128 Karlsruhe

Parallel Computation of Best Connections in Public Transportation Networks

Autoren: Daniel Delling, Bastian Katz,
Thomas Pajor

Interner Bericht 2009-16



Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

ISSN 1432-7864

Fakultät für **Informatik**

Parallel Computation of Best Connections in Public Transportation Networks*

Daniel Delling, Bastian Katz, and Thomas Pajor

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

Abstract. We present a novel algorithm for the so-called 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. 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 that 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. Moreover, we are able to parallelize our algorithm in a very natural way, yielding excellent speed-ups on standard multicore servers. By preprocessing important connections within the public transportation network, we also accelerate station-to-station queries. 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 (Los Angeles) in less than 120 ms on average. This value is achieved on a standard multi-core server.

1 Introduction

The developement of fast route planning algorithms has been undergoing a rapid developement in recent years (cf. [8] for an overview). The fastest techniques for static road networks yield query times of a few microseconds [2]. Recently, the focus has shifted to *time-dependent* networks in which the travel time assigned to an edge is a *function* of the time of the day.

Most of recent research focused on time-dependent *road* networks. While one might expect that speed-up techniques perform similarly on public transportation networks as on road networks, it turned out that this does not hold. For example, time-dependent SHARC [6] yields speed-ups of up to 5 000 over DIJKSTRA's algorithm [9] in time-dependent road networks, while the speed-up for railways only reaches up to 20. The number is even lower for bus networks.

A query for a fast route usually does not always contain a fixed departure time. Hence, in time-dependent networks, a comprehensive answer for the travel-time between given stations is a function of the departure time. This type of query is called *profile search*. In this work, we present an easy-to-parallelize query algorithm that performs very well on public transportation networks, both of railways and buses. As DIJKSTRA's algorithm, it is designed to answer queries for travel times from one station to all others in a single run, but can be accelerated in the case of a station-to-station query.

Related Work. Modeling issues and an overview of basic route planning algorithms in public transportation networks can be found in [23]. Basic speed-up techniques like goal-directed search have been applied to time-dependent railway networks in [10], while SHARC has been tested on such networks as well [6]. However, most of the algorithms

* Partially supported by the DFG (project WA 654/16-1).

fall short as soon as they are applied to bus networks [3, 7]. Most efforts in developing parallel search algorithms address theoretical machines such as the PRAM [22, 11] or the communication network model [4, 24]. Even in these models, no algorithm is 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. There also have been a few experimental studies of distributed single-source shortest path algorithms for example based on graph partitioning [1, 26] or on the so-called Δ-stepping algorithm proposed in [19], e. g. [18]. For an overview on many related approaches, we refer the reader to [15]. 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 [5]. However, none of those algorithms have been parallelized and used for retrieving all quickest connections of a day in realistic public transportation networks.

Our Contribution. We present a novel and easy-to-parallelize algorithm for the so-called *one-to-all profile search* problem asking for the set of all relevant connections between a given station S and all other stations, i. e., 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 at most linear in this number of connections. 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 extend, we introduce the notion of *connection-setting*, that can be seen as an extension of the label-setting property of Dijkstra's algorithm, which usually is lost in profile searches, e. g., in road networks. Another main contribution of this work is that we are able to parallelize our approach such that we obtain faster query times on modern multi-core CPUs. The main idea for parallelization in transportation networks is that we may distribute different connections outgoing from S to the different cores. Obviously, a one-to-all profile search also answers a station-to-station query for a travel time profile. To accelerate these kind of queries, 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 uses 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 pretty well up to 4 cores. As an example, we are able to perform a one-to-all profile search in less than a second and station-to-station queries in less than 120 ms in all transportation networks.

This work is organized as follows: In Section 2 we briefly explain necessary definitions and preliminaries. Section 3 then introduces our one-to-all algorithm. Therefore, we first introduce the concept of *connection-setting* and show how some connections dominate others. Moreover, we present how our algorithm can be parallelized to run on multi-core systems. In Section 4 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 5. We conclude our work with a brief summary and possible future work in Section 6.

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), 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 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*, $\mathcal{C}$ a set of *elementary connections* and $\Pi := \{0, \ldots, \pi - 1\}$ a finite set of discrete time points (think of it as a day's minutes or seconds). 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, e. g. $\tau_{\text{dep}}(c)$ 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.

Models. For route planning, the timetable is modeled as a directed graph. Several approaches have been proposed [23, 7]. In our work we use the *realistic time-dependent model* as introduced in [23]. 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, if they run through the 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 (between the two given stations). See Figure 1 for an illustration.

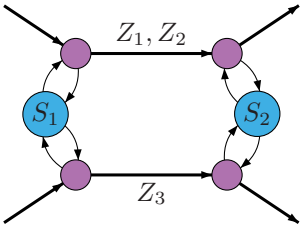


Fig. 1: Illustration of the realistic time-dependent model [23], showing two stations where two routes run through. Station nodes are blue, route nodes are purple.

Piecewise Linear Functions. In general, there are two types of distances in a public transportation network: first, the distance between two stations S and T for a given departure time τ, denoted by $\text{dist}(S, T, \tau)$. The other type, which we are interested in is the distance between two stations S and T for *all* departure times $\tau \in \Pi$, denoted by $\text{dist}(S, T, \cdot)$. This type of query is called *profile search*.

In profile searches, distances or *travel-times* between any two nodes are functions $f : \Pi \to \mathbb{N}_0$, such that $f(\tau)$ denotes the travel-time when starting at time τ. This also includes the time-dependent edges in the graph G. For the remainder of this paper, 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 a good 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 Fig. 2 for an example. Hence, it is possible to represent f by a set of *connection-points* $\mathcal{P}(f) \subset \Pi \times \mathbb{N}_0$ such that $f(\tau)$ is $f(\tau) = \Delta(\tau, \tau_f) + w_f$ for the $(\tau_f, w_f) \in \mathcal{P}(f)$ which minimizes $\Delta(\tau, \tau_f)$. From the timetable, we can easily construct the travel-time functions f_e for the time-dependent edges between route nodes: For each elementary connection c assigned to some route edge e, we insert a connection point (τ, w) into $\mathcal{P}(f_e)$ where $\tau := \tau_{\text{dep}}(c)$, and $w := \Delta(\tau_{\text{dep}}(c), \tau_{\text{arr}}(c))$. Respecting periodicy in a senseful 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.

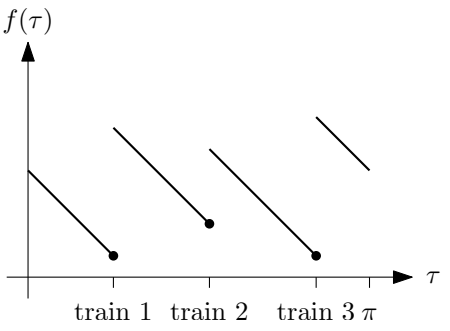


Fig. 2: A piecewise linear function f with 3 connection points, representing 3 relevant trains to start with.

Computing Distances. Computing $\text{dist}(S, \cdot, \tau)$ 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 non-decreasing 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 [5] is that the running time highly depends on the number of connection points assigned to the edges.

3 A Parallel Profile Search Algorithm

In this section we introduce a new profile search algorithm tailored to public transportation networks. The main property of our new approach is the concept of *connection-setting*: for each outgoing connection from S we guarantee that a node in the graph is settled at most once. Moreover, we exploit the fact that some connections are less important than others leading to the concept of *self-pruning*. Finally, we show how we are able to parallelize our algorithm to p processors (cores).

3.1 Self-Pruning Connection-Setting Algorithm

Let $G = (V, E)$ be a graph modeling a timetable $(\mathcal{C}, \mathcal{S}, \mathcal{Z}, \Pi, \mathcal{T})$, and $S \in \mathcal{S}$ a station. Furthermore, let $\text{conn}(S) \subseteq \mathcal{C}$ denote all *outgoing connections* from S defined by $\text{conn}(S) := \{c \in \mathcal{C} \mid S_{\text{dep}}(c) = S\}$. We make use of the following intuition: an itinerary from S always has to begin with one of the connections contained in the set $\text{conn}(S)$. This implies that regarding the distance function $\text{dist}(S, T, \cdot)$ where $T \in \mathcal{S}$ is an arbitrary station, 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 the distance to T for each outgoing connection. More precisely,

$$\mathcal{P}(\text{dist}(S, T, \cdot)) \subseteq \{(\tau, w) \mid \exists c \in \text{conn}(S) : \tau = \tau_{\text{dep}}(c), w = \text{dist}(S, T, \tau_{\text{dep}}(c))\} =: \widehat{\mathcal{P}}\,. \quad (1)$$

Our algorithm works as follows. First, in an *initialization* phase, the set $\text{conn}(S)$ is determined and a priority Q is initialized with queue items for each outgoing connection of $\text{conn}(S)$. Then, a best route is computed for all connections simultaneously in a similar way to DIJKSTRA's algorithm, which leads us to the concept of *connection-setting*. As a result, we obtain the set $\widehat{\mathcal{P}}$. Since $\widehat{\mathcal{P}}$ may contain unnecessary connections, $\widehat{\mathcal{P}}$ has to be reduced to $\mathcal{P}(\text{dist}(S, T, \cdot))$ which then induces the distance function $\text{dist}(S, T, \cdot)$. After showing how this can be done when the algorithm terminates, we introduce *self-pruning*, which detects and prunes unnecessary connections *during* the algorithm.

Initialization. At first, the set $\text{conn}(S)$ is determined and ordered non-decreasingly by the departure times of the elementary connections in $\text{conn}(S)$. Thus, we may say that a connection c_i has *index* i according to the ordering of $\text{conn}(S)$.

Our algorithm maintains a priority queue Q of tuples where the first entry depicts a node $v \in V$ and the second entry a connection index $0 \leq i < |\text{conn}(S)|$. Each node $v \in V$ and for each connection i a label $\text{arr}(v, i)$ is assigned which depicts the *arrival time* at v when using connection i. In the beginning, each label $\text{arr}(v, i)$ is initialized with ∞. Then, for each connection $c_i \in \text{conn}(S)$ we insert (r, i) with key $\tau_{\text{dep}}(c_i)$ into Q, where r depicts the route node where connection c_i starts from. Note that in the beginning the 'arrival-time' $\text{arr}(r, i)$ is equal to the departure-time $\tau_{\text{dep}}(c_i)$.

Connection-Setting. Like DIJKSTRA's algorithm, we subsequently settle queue elements (v, i) assigning $\text{key}(v, i)$ as the final arrival time to $\text{arr}(v, i)$. Then, for each edge $e = (v, w) \in E$ we compute a tentative label $\text{arr}_{\text{tent}}(w, i)$ at w by $\text{arr}_{\text{tent}}(w, i) := \text{arr}(v, i) + f_e(\text{arr}(v, i))$ (for connection i). If w has not yet been discovered using connection i, we insert (w, i) into the priority queue with $\text{key}(w, i) := \text{arr}_{\text{tent}}(w, i)$, otherwise, the element (w, i) is already in the queue and we set $\text{key}(w, i)$ to $\min(\text{key}(w, i), \text{arr}_{\text{tent}}(w, i))$. Note that the following holds for every connection i: when a queue item (v, i) is settled, the label $\text{arr}(v, i)$ 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 $\text{arr}(v, i)$ for each node $v \in V$ and each connection $0 \leq i < |\text{conn}(S)|$ depicting the arrival time at v when starting with the i'th connection at S.

We like to mention two remarks. 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, since the arrival time is obtained by adding the departure time to the distance which is constant for all nodes.

Connection Reduction. For each node $v \in V$ the resulting label $\text{arr}(v, \cdot)$ induces the set of connection points $\widehat{\mathcal{P}}$ by $\widehat{\mathcal{P}} := \{(\tau_{\text{dep}}(c_i), \Delta(\tau_{\text{dep}}(c_i), \text{arr}(v, i))) \mid c_i \in \text{conn}(S)\}$. Unfortunately, the function f represented by $\widehat{\mathcal{P}}$ does not necessarily fulfill the FIFO-property: Taking an earlier train in the wrong direction from S and then the fastest route towards T in general is not better than waiting for the next train in the right direction. More formally: for two points $(\tau_i, w_i), (\tau_j, w_j) \in \widehat{\mathcal{P}}$ with $j > i$ it is possible that $\tau_j + w_j \leq \tau_i + w_i$. To remedy this issue, the set $\widehat{\mathcal{P}}$ is reduced to obtain $\mathcal{P}(\text{dist}(S, T, \cdot))$ by eliminating those points which 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 $\tau^{\text{arr}}_{\text{min}} := \tau_{i_{\text{min}}} + w_{i_{\text{min}}}$ along the way. Each time we scan a connection point $j < i_{\text{min}}$ with an arrival time $\tau^{\text{arr}}_j \geq \tau^{\text{arr}}_{\text{min}}$, the connection point is deleted. The remaining connection points are exactly those of $\mathcal{P}(\text{dist}(S, T, \cdot))$.

Self-Pruning. Performing the connection reduction after the algorithm has finished, 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: We introduce a *node-label* $\text{maxconn} : V \to \{0, \ldots, |\text{conn}(S)| - 1\}$ depicting the highest connection index with which the node v has been reached so far. Each time we settle a queue element (v, i) with $\text{arr}(v, i) := \text{key}(v, i)$, we check if $i > \text{maxconn}(v)$. If this is *not* the case, the node v has already been settled earlier—but with a later connection (remember that $j > i \Rightarrow \tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$), thus, implying $\text{arr}(v, j) \leq \text{arr}(v, i)$. Therefore, the current connection does not pay off, and we prune the connection i at v, i. e., we do not relax outgoing edges at v. Moreover, we set $\text{arr}(v, i) := \infty$, depicting that the i'th connection does not 'reach' v. In the case of $i > \text{maxconn}(v)$, we update $\text{maxconn}(v)$ to i, and continue with relaxing the outgoing edges of v regularly. Obviously, by applying self-pruning, the set of connection points $\mathcal{P}(\text{dist}(S, v, \cdot))$ at each node v induced by $\text{arr}(v, \cdot)$ fulfills the FIFO-property automatically (labels with $\text{arr}(v, i) = \infty$ have to be ignored).

Theorem 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 $\text{arr}(v, i)$ be the arrival time at v of the (optimal) i'th connection and assume that i has been pruned at v. Let j denote the connection which was responsible for pruning i. Then, it holds that $\text{arr}(v, j) \leq \text{arr}(v, i)$. Moreover, since j pruned i, it holds that $j > i$, which implies $\tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$. Therefore, it holds that $\Delta(\tau_{\text{dep}}(c_j), \text{arr}(v, j)) \leq \Delta(\tau_{\text{dep}}(c_i), \text{arr}(v, i))$. This is a contradiction to i being optimal: using the j'th connection results in an earlier arrival at v by departing later at S.

3.2 Parallelization

We parallelize our algorithm in the following natural way. Given p processors (cores), we partition the set $\text{conn}(S)$ into p subsets $\text{conn}(S)_0, \ldots, \text{conn}(S)_{p-1}$ during the initialization

phase. Then, we create p threads where each thread independently runs the self-pruning connection-setting algorithm on its restricted subset $\text{conn}(S)_i$. After termination of each thread, a master thread merges the labels $\text{arr}_i(v, \cdot)$ of each thread i to a common label $\text{arr}(v, \cdot)$ while preserving the ordering of the connections. Note that the common label $\text{arr}(v, \cdot)$ is not necessarily FIFO, since we do not self-prune between threads. 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 $\text{arr}(v, \cdot)$ with our connection reduction method described above.

Choice of the Partition. The speed-up that can be achieved by the parallelization of our algorithm depends on the partitioning of $\text{conn}(S)$. 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 $\text{conn}(S)$ heuristically. We propose the following simple methods.

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 $\text{conn}(S)_i$ turn out to be very unbalanced, at least in our scenario. The reason for this is that the connections in $\text{conn}(S)$ 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 $\text{conn}(S)$ 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 [17] can be applied. However, preliminary tests showed that the improvement on the query performance is rather insignificant over the simple methods, thus, in our experiments (cf. Section 5) we use the equal number of connections method as a reasonable compromise.

Impact on Self-Pruning. 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 $\text{conn}(S)$. In this case, our algorithm basically corresponds to computing $|\text{conn}(S)|$ time-queries in parallel—without any pruning. However, in real-world inputs (especially in local bus networks) the average number of outgoing connections is significantly larger than the number of processor cores on today's computers. Hence, the impact on self-pruning imposed by the parallelization is almost negligible. See also Section 5 for an experimental evaluation.

4 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 [8]. In this section, we present how some of the ideas, i. e., the so called *stopping criterion*, map to our new algorithm. Moreover, we

show how the precomputation of certain connections improves the performance of our algorithm in a station-to-station scenario.

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, i. e., station-to-station, we can stop the query as soon as the target station T has its final label $\text{arr}(T, i)$ 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 $q = (v, i) \in \mathrm{Q}$ with $i \leq T_m$ (at any node v). We may stop the query as soon as the queue is empty.

Theorem 2 *The stopping criterion is correct.*

Proof. We need to show that no entry $q = (v, i) \in \mathrm{Q}$ 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 $\tau_{\text{dep}}(c'_i) \geq \tau_{\text{dep}}(c_i)$ holds as well. Moreover, since q is settled after q', we know that $\text{arr}(v', i') \leq \text{arr}(v, i)$ holds. In other words, it does not pay off to board train i at station S. □

Pruning with a Distance Table. Next, we show how to acceletate our station-to-station algorithm by pruning via a distance table. We therefore consider the *station graph* $G_S = (\mathcal{S}, E_S)$ where an edge (S_1, S_2) indicates at least one train running from S_1 to S_2. For a node u of the timetable graph, $\text{st}(u)$ denotes the station a node belongs to. We are given a subset $\mathcal{S}_{\text{trans}} \subseteq \mathcal{S}$ of stations (called *transfer stations*) 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 stations $S, T \in \mathcal{S}_{\text{trans}}$, the arrival time at T when departing from S at $\tau \in \Pi$ (without any transfer times at S and T). Before explaining the pruning rule in detail, we need the notion of *local* and *via stations*.

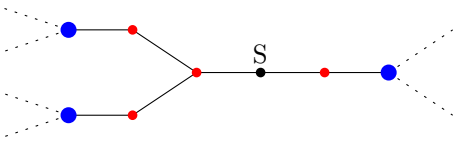


Fig. 3: Local and via stations of a station S. Local stations are indicated in red, while via stations are marked thicker in blue.

The set of local stations $\text{local}(S) \subseteq \mathcal{S}$ of an arbitrary station S includes all stations L such that there is a simple path from L to S that contains only non-transfer stations in the station graph G_S. The set of transfer stations that are adjacent to at least one local station of S are called the *via stations* of S, denoted by $\text{via}(S) \subseteq \mathcal{S}_{\text{trans}}$. They basically separate $S \cup \text{local}(S)$ from any other station in G_S. Figure 3 gives a small example. In the special case of S being a transfer station, we set $\text{local}(S) = \emptyset$ and $\text{via}(S) = \{S\}$.

In the following, we call an S-T station query *local*, if $S \in \text{local}(T)$, otherwise the

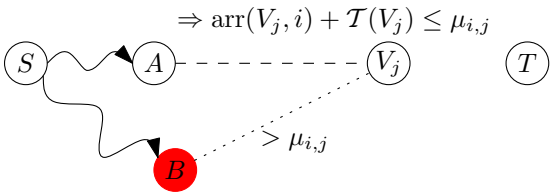


Fig. 4: Example for pruning via a distance table, given an S-T query. A and B are transfer stations, V_j the via station of T. When settling a node at station A, we obtain that the arrival time at V_j plus the transfer time at V_j is smaller or equal to $\mu_{i,j}$. Hence, we may prune the query at B if the lower bound obtained from the distance table yields an arrival time at V_j greater than $\mu_{i,j}$.

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 station V_j of T. Whenever we settle a queue entry $q = (v, i)$ with $\text{st}(v) \in \mathcal{S}_{\text{trans}}$, we set $\mu_{i,j} := \min\{\mu_{i,j}, \mathcal{D}(\text{st}(v), V_j, \text{arr}(v, i) + \mathcal{T}(\text{st}(v))) + \mathcal{T}(V_j)\}$ 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 change the train at V_j. So, we may prune the search regarding q if

$$\forall V_j \in \text{via}(T) : \mathcal{D}(\text{st}(v), V_j, \text{arr}(v, i)) > \mu_{i,j} \tag{2}$$

holds. In other words, we prune the search at v for a connection i if the path through $\text{st}(v)$ is provably not important for the best path to any via station of $V_j \in \text{via}(T)$. Figure 4 gives a small example.

Theorem 3 *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 it is no improvement over the path corresponding to $\mu_{i,j}$.

Special Cases. Obviously, we may immediately stop the search if $S, T \in \mathcal{S}_{\text{trans}}$ since the distance table already includes all best connections from S to T. However, we may also apply an additional pruning rule if $T \in \mathcal{S}_{\text{trans}}$, which we call *target pruning*. For each connection i, we maintain a tentative lower bound γ_i on the arrival time at T, initialized with ∞. Whenever we settle an element $q = (v, i) \in \text{Q}$ with $\text{st}(v) \in \mathcal{S}_{\text{trans}}$, we update γ_i to $\min\{\gamma_i, \mathcal{D}(\text{st}(v), T, \text{arr}(v, i))\}$. As soon as all elements $q = (v, i) \in \text{Q}$ for a given connection i have a transfer station as ancestor, γ_i is a feasible lower bound on the arrival time at T. When we then remove a queue element $q = (v, i) \in \text{Q}$ with $\text{st}(v) \in \mathcal{S}_{\text{trans}}$, we may stop the search for i if $\mathcal{D}(\text{st}(q), T, \text{arr}(v, i) + \mathcal{T}(\text{st}(v))) = \gamma_i$ holds. We set $\text{arr}(T, i) = \mathcal{D}(\text{st}(q), T, \text{arr}(v, i) + \mathcal{T}(\text{st}(q)))$ and prune the search for any $q = (v, i) \in \text{Q}$.

Theorem 4 *Target pruning is correct.*

The proof of Theorem 4 follows from the observation that γ_i is a valid lower bound to the target station and that when we prune the search, we already have found the optimal arrival time at T (for i). The full proof can be found in Appendix A.

Determining $\text{via}(T)$. We determine the via stations of T on-the-fly: During the initialization phase of the algorithm, we run a DFS on the reverse station 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 local from global queries when computing $\text{via}(T)$: as soon as our DFS visits S, the query is local, otherwise it is global.

Selection of Transfer Stations. The success of pruning via a distance table highly depends on which stations are selected for $\mathcal{S}_{\text{trans}}$. In [25], 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 here propose to use the concept of contraction [12]

which proved useful in road networks. A contraction routine iteratively removes unimportant nodes from the graph and adds shortcuts to the graph in order to preserve the distances between non-removed nodes. We mark any station as important which has not been removed after the contraction of c stations.

Another possibility to select important stations is via their degree in the station graph. More precisely, we mark any station as transfer station having a degree $> k$ in the station graph.

5 Experiments

We conducted our experiments on up to eight cores of a dual Intel Xeon 5430 running SUSE Linux 11.1. The machine is clocked at 2.6 GHz, has 32 GiB of RAM and 2×1 MiB of L2 cache. The program was compiled with GCC 4.3, using optimization level 3. Our implementation is written in C++ using solely the STL and Boost at some points. As priority queue we use a binary heap.

Inputs. We use five different public transportation networks as input: the local networks of Oahu Transit Services [21], Hawaii (3 918 stops and 1 408 559 elementary connections), Los Angeles County Metro [16] (15 792 stops and 5 023 877 elementary connections), and the network of Washington Metropolitan Area Transit Authority [27] (10 764 stops and 3 387 987 elementary connections). Moreover, we use railway networks of Germany and Europe. The former has 6 822 stations and 554 996 elementary connections, while the latter has 30 517 stations and 1 775 533 elementary connections. Note, that the local networks are much denser than the railway networks, i. e., the connections per station ratio is significantly higher there.

The timetable data of the local city networks is publicly available through Google Transit Data Feeds [13], while the timetable data of the German and European railway networks was kindly given to us by HaCon [14].

5.1 One-to-All Queries

Our first set of experiments focuses on the question how well our self-pruning connection-setting algorihm performs if executed on a varying number of cores. Therefore, we run 1 000 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 1 reports these figures for a varying number (between 1 and 8) of cores. For comparison, we also report the performance of a label-correcting approach (cf. Section 2). For better comparability, the number of connections figure 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 all networks except Europe, the number of settled nodes only increases by $\approx$10–20%. This is less than we expected since we cannot self-prune between different threads. So, on 4 cores we have a speed-up of a factor 3 compared to an execution on one core. On 8 cores, the speed-up is of a factor of 5. The reason for this is that memory management also plays a crucial role for the scalability of a parallel algorithm. Still, on eight cores, we

Table 1: One-to-all profile queries with our parallel self-pruning connection-setting algorithm (CS) on 1,2,4, and 8 cores, compared to a label-correcting approach (LC). Column *spd-up* indicates the time speed-up of a multi-core run over a single-core execution.

		Oahu			**Los Angeles**			**Washington D.C.**			**Germany**			**Europe**		
		Settled	Time	Spd	Settled	Time	Spd	Settled	Time	Spd	Settled	Time	Spd	Settled	Time	Spd
	p	Conns	[ms]	Up	Conns	[ms]	Up	Conns	[ms]	Up	Conns	[ms]	Up	Conns	[ms]	Up
CS	1	931829	369.4	1.0	4311920	2660.5	1.0	2039578	1122.7	1.0	1409515	835.4	1.0	2825883	1913.3	1.0
	2	944064	192.7	1.9	4369988	1411.1	1.9	2060926	576.0	1.9	1465211	439.7	1.9	3112617	1036.9	1.8
	4	968933	125.4	2.9	4493353	927.6	2.9	2106210	352.3	3.2	1577633	256.9	3.3	3658800	677.8	2.8
	8	1019357	79.9	4.6	4735290	602.2	4.4	2188936	242.3	4.6	1791296	177.6	4.7	4539940	527.1	3.6
LC	1	13448000	1056.0	—	52838500	4372.3	—	16467900	1219.1	—	11290200	1127.2	—	20315700	2875.2	—

are able to compute all quickest connections of a day in less than 1 second. Note that this value is achieved without any preprocessing, hence, we can directly use this approach in a fully dynamic scenario as discussed in [20]. On Europe however, the scalability falls short when executed on more than 2 cores: on 8 cores, the number of settled nodes increases by 60% compared to a single-threaded execution yielding a speed-up of only 3.6. The reason for this is that the average number of outgoing connections per station on Europe is rather small. Hence, each thread has only a small number of connections assigned yielding less self-pruning. Moreover, the running times of the threads are more biased than for networks with more connections.

Comparing our connection-setting (CS) with the label-correcting (LC) approach (cf. Section 2), we observe that CS clearly outperforms LC, even when CS is executed on only one core. The main reason for this is that the number of connections investigated during execution is much smaller for CS than for LC. However, the number of priority queue operations for LC is up to 4 times lower than for CS. Hence, the advantage of CS in number of settled connections does not yield the same speed-up in query times.

5.2 Station-to-Station Queries with Pruning by Distance Tables

Next, we evaluate our algorithm in a station-to-station scenario. We use 8 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. The distance tables are computed by running our parallel one-to-all algorithm on 8 cores from every transfer station. As strategies for selecting transfer stations, we use contraction with varying number of removed stations and selection via degree in the station graph. Table 2 gives an overview over the obtained results. We observe that compared to Tab. 1, the stopping criterion accelerates queries by approximately 20%. Moreover, we observe that the size of the distance table has a high impact on the query performance. While augmenting only 1% of the stations to transfer stations hardly accelerates queries, 5% transfer stations yields additional speed-ups between 1.8 and 4.4, 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 5% of the stations as transfer stations seems to be a good compromise. For this scenario, we are able to compute all quickest connections on all inputs in less than 117 ms.

Table 2: Performance of our parallel self-pruning query algorithm with stopping criterion enabled. Moreover, we prune by a distance table as described in Section 4. The number of transfer stations is given in percentage of input stations.

	Oahu					Los Angeles					Washington D.C.				
	Prepro		Query			Prepro		Query			Prepro		Query		
	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up
0.0%	—	—	853744	66.2	1.0	—	—	3718790	473.6	1.0	—	—	1869023	208.3	1.0
1.0%	0:08	1.2	748666	64.0	1.0	3:19	18.9	2397412	318.8	1.5	1:01	8.8	1671548	233.1	0.9
2.5%	0:17	5.9	524648	47.5	1.4	7:26	99.4	958027	140.8	3.4	2:34	61.2	999040	157.3	1.3
5.0%	0:30	20.3	329952	31.7	2.1	14:00	346.5	625114	106.5	4.4	4:46	212.0	689025	112.7	1.8
10.0%	0:54	65.7	262030	28.6	2.3	26:10	1252.6	555303	106.9	4.4	8:45	705.0	553261	93.6	2.2
20.0%	1:43	225.6	231840	25.9	2.6	50:23	4490.3	514539	105.7	4.5	—	—	—	—	—
30.0%	2:35	477.5	202118	24.6	2.7	—	—	—	—	—	—	—	—	—	—
deg > 2	2:26	376.2	188963	21.9	3.0	57:24	5331.6	428983	90.1	5.3	27:12	5450.0	394984	68.1	3.1

	Germany					Europe				
	Prepro		Query			Prepro		Query		
	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up	Time [m:s]	Space [MiB]	Settled Conns	Time [ms]	Spd Up
0.0%	—	—	1400841	140.5	1.0	—	—	3339884	387.0	1.0
1.0%	0:19	0.6	1352037	149.9	0.9	4:04	5.9	2492665	309.6	1.3
2.5%	0:52	5.5	777683	88.9	1.6	11:40	55.6	1202247	152.6	2.5
5.0%	1:44	23.3	470028	52.4	2.7	23:27	216.2	860923	116.8	3.3
10.0%	3:17	87.4	338192	40.5	3.5	—	—	—	—	—
20.0%	6:12	315.5	286733	35.7	3.9	—	—	—	—	—
deg > 2	10:11	804.0	227275	29.2	4.8	—	—	—	—	—

6 Conclusion

In this work, we have presented a novel algorithm for computing all relevant connections from a given station to any other station in a transportation network for a full timetable period. 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 travelling 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. Our algorithm is easy to use in a multi-core setup. Moreover, utilizing the algorithm to precompute connections between important stations, we can greatly accelerate station-to-station queries.

Regarding future work, it will be interesting to incorporate multi-criteria connections, e. g., minimizing the number of transfers. 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, we are interested in checking whether some methods from [6] also work for our connection-setting approach.

References

1. P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest path problem. *International Journal of Parallel Programming*, 20:271–298, 1991.
2. R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In *WEA'08*, LNCS 5038, 303–318. 2008.
3. R. Bauer, D. Delling, and D. Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 2009. Accepted for publication, to appear.

4. K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Comm. ACM*, 25(11):833–837, 1982.
5. B. C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, MIT, 1999.
6. D. Delling. Time-Dependent SHARC-Routing. *Algorithmica*, July 2009.
7. D. Delling, T. Pajor, and D. Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Proceedings of ATMOS'08*, Dagstuhl Seminar Proceedings. September 2008.
8. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, LNCS 5515, pp. 117–139. Springer, 2009.
9. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Math.*, 1:269–271, 1959.
10. Y. Disser, M. Müller–Hannemann, and M. Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In *WEA'08*, LNCS 5038, pp. 347–361. 2008.
11. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Comm. ACM*, 31(11):1343–1354, 1988.
12. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA'08*, LNCS 5038, pp. 319–333. 2008.
13. Google Transit Data Feed. http://code.google.com/p/googletransitdatafeed/, 1967.
14. HaCon - Ingenieurgesellschaft mbH. http://www.hacon.de, 2008.
15. M. R. Hribar, V. E. Taylor, and D. E. Boyce. Implementing parallel shortest path for parallel transportation applications. *Parallel Computing*, 27:1537–1568, 2001.
16. Los Angeles County Metropolitan Transportation Authority. http://www.metro.net, 1993.
17. J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.
18. K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. In *ALENEX'07*, pp. 23–35. SIAM, 2007.
19. U. Meyer and P. Sanders. Δ-Stepping : A Parallel Single Source Shortest Path Algorithm. In *ESA'98*, LNCS 1461, pp. 393–404, 1998.
20. M. Müller–Hannemann, M. Schnee, and L. Frede. Efficient On-Trip Timetable Information in the Presence of Delays. In *Proceedings of ATMOS'08*, Dagstuhl Seminar Proceedings. 2008.
21. O'ahu Transit Services, Inc. http://www.thebus.org, 1971.
22. R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. pp. 553–556, 1985.
23. E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.
24. K. V. S. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *J. Algorithms*, 13:235–257, 1992.
25. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *WAE'99*, LNCS 1668, pp. 110–123. Springer, 1999.
26. J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.
27. Washington Metropolitan Area Transit Authority. http://www.wmata.com, 1967.

A Proofs

Proof of Theorem 3

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 connection i. Note that if S-T is a local query, no pruning is applied. Furthermore, let $\text{arr}_{\text{opt}}(T, i)$ denote the (optimal) arrival time at T when using P. We show a series of lemmas before proving the main theorem.

Lemma 1. *For all tuples $(v, V_j) \in V \times \text{via}(T)$ with $\text{st}(v) \in \mathcal{S}_{trans}$ it holds that*

$$\begin{aligned}\text{arr}_{\text{opt}}(T, i) \leq & \underbrace{\mathcal{D}(\text{st}(v), V_j, \text{arr}(v, i) + \mathcal{T}(\text{st}(v)))}_{=: \mu_{i,v,j}} \\ & + \mathcal{T}(V_j) + \text{dist}(V_j, T, \mu_{i,v,j}).\end{aligned} \tag{3}$$

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

Corollary 1. *Let $\mu_{i,j} := \min_{v \in V, \text{st}(v) \in \mathcal{S}_{trans}}(\mu_{i,v,j})$, then it holds that $\text{arr}_{\text{opt}}(T, i) \leq \mu_{i,j} + \text{dist}(V_j, T, \mu_{i,j})$.*

Lemma 2. *For all tuples $(v, V_j) \in V \times \text{via}(T)$ with $\text{st}(v) \in \mathcal{S}_{trans}$ it holds that*

$$\text{arr}_{V_j}(T, i) \geq \underbrace{\mathcal{D}(\text{st}(v), V_j, \text{arr}(v, i))}_{=: \gamma_{i,v,j}} + \text{dist}(V_j, T, \gamma_{i,v,j}) \tag{4}$$

where $\text{arr}_{V_j}(T, i)$ depicts the arrival time of the combined shortest S-v-V_j-T path.

Proof. Assume that the right hand side of the equation evaluates to $\text{arr}'_{V_j}(T, i)$ with $\text{arr}'_{V_j}(T, i) < \text{arr}_{V_j}(T, i)$, but this is a contradiction to the correctness of the distance table $\mathcal{D}$ yielding the earliest arrival time at V_j, since $\text{dist}(V_j, T, \cdot)$ fulfills the FIFO-property and $\gamma_{i,v,j}$ is the earliest possible arrival time at V_j (without transfer at $\text{st}(v)$).

Lemma 3. *Let $v \in V$ be a node with $\text{st}(v) \in \mathcal{S}_{trans}$, and let $\gamma_{i,v,j} > \mu_{i,j}$. Then*

$$\gamma_{i,v,j} + \text{dist}(V_j, T, \gamma_{v,i,j}) \geq \mu_{i,j} + \text{dist}(V_j, T, \mu i, j) \tag{5}$$

holds.

Proof. This follows immediately from the FIFO-property of $\text{dist}(V_j, T, \cdot)$.

Proof of Theorem 3. Given a global S-T-query with via stations $\text{via}(T)$. Let $v \in V$ be a node with $\text{st}(v) \in \mathcal{S}_{\text{trans}}$, where the pruning rule is potentially applied. Then from Lemma (2), (3) and Corollary (1) we get for a via node $V_j \in \text{via}(T)$ that

$$\gamma_{v,i,j} > \mu_{i,j} \quad \Longrightarrow \quad \text{arr}_{V_j}(T, i) \geq \underbrace{\mu_{i,j} + \text{dist}(V_j, T, \mu_{i,j})}_{=: \psi} \geq \text{arr}_{\text{opt}}(T, i) \tag{6}$$

Since our algorithm keeps track of $\mu_{i,j}$ which is the minimum over all $\mu_{i,x,j}$ with $\text{st}(x) \in \mathcal{S}_{\text{trans}}$, the path which 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 (5) holds for *all* $V_j \in \text{via}(T)$, it follows that $v \notin P$, thus, v not being important for the shortest S-T-path. □

Proof of Theorem 4

Similar to the proof of Theorem 3, we show correctness of Theorem 4 for each connection i separately. Again, let i be a fixed connection and $P = [S, \ldots, T]$ the shortest path of a global S-T-query of connection i and let $\text{arr}_{\text{opt}}(T, i)$ denote the (optimal) arrival time at T when using P.

We know that for all nodes v with $\text{st}(v) \in \mathcal{S}_{\text{trans}}$, the inequation $\text{arr}_{\text{opt}}(T, i) \leq \mathcal{D}(\text{st}(v), T, \text{arr}(v, i) + \mathcal{T}(\text{st}(v)) =: \mu_{i,v}$ holds. Moreover, for all nodes $u \in P$ with $\text{st}(u) \in \mathcal{S}_{\text{trans}}$, we know that $\text{arr}_{\text{opt}}(T, i) \geq \mathcal{D}(\text{st}(v), T, \text{arr}(v, i)) =: \gamma_{i,v}$ holds as well. From this follows that

$$\min_{\substack{\text{st}(v) \in \Gamma \subseteq \mathcal{S}_{\text{trans}} \\ \exists\, \text{st}(u) \in \Gamma : \text{st}(u) \in P}} \gamma_{i,v} := \gamma_i \leq \text{arr}_{\text{opt}}(T, i) \leq \min_{\text{st}(v) \in \mathcal{S}_{\text{trans}}} \mu_{i,v} \tag{7}$$

holds. In other words, as soon as a transfer station on the shortest path has contributed to γ_i, γ_i is a feasible lower bound on the arrival time at T. So, we have found the optimal arrival time at T as soon as $\gamma_i = \mu_i$ holds. By enabling target pruning only when all elements in the queue have a node u with $\text{st}(u) \in \mathcal{S}_{\text{trans}}$ as ancestor, we ensure that a transfer station on the shortest path contributes to γ_i. Hence, Theorem 4 is correct. □