Connection Scan Accelerated[*]

Ben Strasser[†] Dorothea Wagner[‡]

Abstract

We study the problem of efficiently computing journeys in timetable networks. Our algorithm optimally answers profile queries, computing all journeys given a time interval. Our study demonstrates that queries can be answered optimally on large country-scale timetable networks within several milliseconds and fast delay integration is possible. Previous work either had to drop optimality or only considered comparatively small timetable networks. Our technique is a combination of the Connection Scan Algorithm and multi-level overlay graphs.

1 Introduction

We study the problem of answering queries in timetable networks. Efficient algorithms are needed as the foundation of complex web services such as the Google Transit or `bahn.de` - the German national railroad company's website. The user enters his desired departure stop, arrival stop and a vague moment in time and the system should compute an optimal journey telling the user when to take which train. In practice, trains do not adhere perfectly to the timetable and therefore it is necessary to be able to quickly adjust the scheduled timetable to the actual situation.

Formalizing the definition of an optimal query is non-trivial. The straight forward adaptation of the shortest path problem on graphs yields the earliest arrival problem [21]. Given a source time, a source stop and a target stop, the problem consists of finding a journey that departs after the source time and has a minimum arrival time. Unfortunately, earliest arrival journeys are flawed as the following example shows: The user asks for A@8:00$\rightarrow B$ where A is the source stop, B the target stop, and 8:00 the source time. Suppose that there is a train A@10:00$\rightarrow B$@11:00 and therefore the earliest arrival is 11:00. Intuitively the user is only interested in the direct journey $A \rightarrow B$, but this is not necessarily the only earliest arrival journey. Consider additional trains A@8:00$\rightarrow C$@9:00 and C@9:00$\rightarrow A$@10:00. Another earliest arrival journey is $A \rightarrow C \rightarrow A \rightarrow B$. However, intuitively it is clear that this journey is not the answer to the user's question; i.e. the earliest arrival journey problem does not model the user's expectations. One solution is to additionally minimize the number of transfers. This results in a bicriteria problem and is often optimized in the Pareto sense [11], i.e., find all journeys for which no journey exists that is better in both criteria. However, Pareto optimization is expensive. Therefore we chose an intermediate approach: We optimize the arrival time as a primary and the transfers as a secondary criterion, i.e., among all earliest arrival journeys we choose one with a minimum number of transfers. This variant was already proposed in [18].

Often the user is flexible with regard to his departure and/or arrival time. The basic earliest arrival problem does not reflect this flexibility. For this reason web services propose several journeys with varying departure times. We formalize this concept as profile queries. Given a source stop, a target stop, a source time and a target time compute all optimal journeys that depart after the source time and arrive before the arrival time. However, this formalization is still not perfect as the user often only wants to specify either the source or the target time. For this reason we also consider the earliest arrival *time* problem to estimate the target time based on the source time and vice versa.

Our approach uses the Connection Scan Algorithm (CSA) [9] as its core. CSA is a simple algorithm that does not use a priority queue and does not rely on computing auxiliary data in a preprocessing phase. Its strength lies in its very low running time constants allowing it to solve even moderately sized timetable networks such as the London urban region in about 150ms. However, sublinear running times are necessary on larger country-scale timetable networks such as the one used by `bahn.de`. We therefore extend CSA to use auxiliary data computed in a preprocessing phase. We adapt the basic ideas of CRP [5, 8] and its multi-level overlay predecessors [21, 16] to timetables. Previous studies [3] have shown that directly applying speed-up techniques designed for road graphs to timetables does not work. It is crucial to adapt the algorithms to the special timetable structure. CSA's ability to handle moderately sized timetable networks efficiently makes the approach practical even if overlays are not tiny.

[*]Partial support by DFG grant WA654/16-2 and EU grant 288094 (eCOMPASS) and Google Focused Research Award.

[†]Karlsruhe Institute of Technology, `strasser@kit.edu`

[‡]Karlsruhe Institute of Technology, `dorothea.wagner@kit.edu`

1.1 Related Work. In [18] timetable routing is reduced to variants of the shortest path problem in graphs. Time and transfers are optimized. Some speedup techniques are evaluated, among them goal direction using potentials. Only small instances are examined. RAPTOR [7] improves on this approach by not using graph data-structures. They also consider additional criteria such as price. In [22] multi-level graphs are evaluated on timetables. In [12] contraction hierarchies are adapted to timetables to optimize profile queries. However, they do not optimize transfers. In [4] the authors introduce two speedup techniques: SUBITO and k-flags. They optimize transfers in the Pareto sense. In [1], Transfer Patterns were introduced and were refined in [2]. This technique was developed in part by the Google Transit team and is the best competitor in our scenario. It is the only technique that has been shown to be practicable on large country-scale timetable networks with about 60 000 000 connections. This is slightly larger than our benchmark instance. Unfortunately, the preprocessing time is prohibitive. To make preprocessing feasible, correctness is dropped and even then multi-core CPU days are needed. Studied are Pareto optimization and optimizing a single composed criterion similar to ours. Profile times are not evaluated.

1.2 Our Contribution. Recent studies [7, 9] have shown that often algorithms not based on Dijkstra's [10] work better on timetable networks. We describe the first preprocessing based speedup technique that at its core is not based on Dijkstra's algorithm. We achieve running times of 8.7 ms for the earliest arrival time problem and of 78 ms for the profile problem on large-scale timetable networks with secondary transfer optimization. Depending on the formalization our preprocessing is either very fast (1 min) or fast (30 min). We know of no provably optimal speedup technique achieving similar running times for similar optimization criteria on large scale timetable networks. Our preprocessing is orders of magnitude faster than existing approaches.

2 Basics

In this section we formally define the notions of public transit network and journey and give a motivation for the design choices. We further formally define profiles and formulate the precise problems considered in this study.

2.1 Network. A (timetable) *network* consists of *stops, connections, trips* and a *transfer graph* (defined in the next subsection). A connection c represents a vehicle driving from a *departure stop* c_{depstop} at *departure time* c_{deptime} without intermediate halt to an *arrival stop* c_{arrstop} at *arrival time* c_{arrtime}. Travel times must be strictly positive. Connections served successively by the same vehicle are part of the same *trip* c_{tripid}.

Following [7], we only consider *aperiodic* timetables, where vehicles only drive once and are not repeat every day. To support queries involving overnight trains we *unroll* the network one day, i.e., we copy all connections and add 24h to the departure times and arrival times. However, queries are only allowed within the first 24h.

At first an aperiodic timetable seems less realistic than the periodic ones with perfectly repeating connections considered in other papers [18]. However, it is also unrealistic that schedules are never changed, especially when considering delays, and therefore both are models do not fully capture reality.

A third more realistic model assumes that the input only consists of a finite connection subset of an infinite aperiodic set of actual connections. However with this realistic model has some severe problems. For example one can only decide whether a journey exists that solely uses input connections. However, if no such journey is found, it does not mean that no journey exists using connections in the infinite set of all connections. It is not possible to decide whether a journey exists solely based on the input. We conclude that at some point perfect realism must be dropped.

In practice the differences between the periodic and aperiodic approaches are small, as the user is rarely interested in journeys longer than 24h. It makes no real difference whether a bogus journey is found or none is found at all. In both cases the user will look for different transportation options (such as planes).

2.2 Transfers. While real world datasets tend to agree on the modeling of vehicles, they differ significantly when it comes to modeling transfers. These differences are also reflected in the published papers but are rarely discussed.

For example, a widely used format is the General Transit Feed Specification [14]. It only allows for *footpaths* between two stops, i.e., the feed contains a list of weighted directed arcs that represents a transfer option from one stop to an adjacent one. No change times at stops are specified, allowing the user to instantly transfer between vehicles arriving and departing at the same stop. Many feeds model large train stations not as a single stop but each railway platform inside the station as individual stop and connect the platforms using footpaths.

Another specification is given by HAFAS [15]. Our benchmark instance is based on it. Here neighboring platforms are modeled as a single stop. To assure that the user is given enough time to transfer from one vehicle to another one, every stop is associated with a *minimum change time* $\tau_{\min}$, i.e., the difference between the departure time of the connecting vehicle and the arriving one must be at least $\tau_{\min}$. The dataset also contains a few footpaths to connect neighboring platform regions. A large train station might for example be modeled as three stops interconnected by footpaths: the main platforms, the subway platforms, and the forecourt platforms. HAFAS-based feeds tend to have fewer footpaths than GTFS-based ones.

There are GTFS feeds that contain footpath loops, such as for example the feed of Berlin [13]. These are footpaths with the same departure stop and arrival stop and a non-zero duration. The idea of these "footpaths" is to encode the minimum change time but the GTFS specification does not state whether this formulation is valid. However, GTFS does not provide a better way to encode this type of information.

At first glance the GTFS approach of modeling platforms as stops instead of whole stations as stops seems superior as it is more fine grained. However, in some countries such as France the operator does not planned in advanced at which platform a vehicle will arrive. Platforms are assigned to trains a few dozen minutes before the arrival of the vehicle. This can not be modeled without minimum change times.

When mixing footpaths and minimum change times a couple of questions arise. Let A and B be two stops with minimum change times of 5min and 10min and a connecting footpaths of 3min. Is the transfer time 3, 8, 13, or 18 minutes? We model the most general case allowing 3min transfers. A footpath can replace minimum change times. Note that this allows for transfers below the minimum change times. We observed instances of this actually happing in real data. However, one might argue that this is a data error.

Having motivated the need for both footpaths and minimum change times, we are ready to present our definition of the *transfer graph*, an abstraction that avoids the need to distinguish between both. It is a directed weighted graph. The stops are its nodes. We denote its directed arcs using $A \rightarrow B$ and refer to them as *transfers*. Their weight is denoted using $d\,(A \rightarrow B)$ and referred to as *duration*. Footpaths are modeled as interstop transfers and minimum change times as loops. Every stop is required to have a loop. The duration must be non-negative. Multi-arcs are not allowed. The user is by definition required to use at least one transfer when changing vehicles. To assure that one transfer is also always enough, we require additional restrictions on the transfer graph. Following [7] we require that the transfer graph is transitively closed. An equivalent characterization is to say that the transfer graph is a union of disjoint cliques. We further require that the triangle inequality is fulfilled, i.e, for every two transfers $A \rightarrow B$ and $B \rightarrow C$ the duration of the transfer $A \rightarrow C$ (which must exist) is never larger than the sum of the durations of $A \rightarrow B$ and $B \rightarrow C$. These requirements assure that all shortest paths are at most one hop long. If the input transfer graph does not fulfill these requirements, then it can easily be transformed to do so. In theory this can severely increase the transfer graph's size. However, transfers should not model the user walking from one station to a neighboring one. Transfer graphs are therefore highly disconnected, which results in only a small size increase. For unrestricted multi-modal routing a different approach is needed.

2.3 Journeys. We model journeys as sequence of connections and transfers. We first define partial journeys that can end and/or depart in the middle of a trip. Based on these we define complete journeys that are required to depart and arrive at a stop and must use an initial and final transfer.

A *partial journey* is a sequence of connections and transfers in temporal order, i.e., a connection must arrive before the next one can depart. Two successive connections c and c' either share the same trip and $c_{\text{arrstop}} = c'_{\text{depstop}}$ or are separated by a transfer t with $c_{\text{arrstop}} = t_{\text{depstop}}$ and $t_{\text{arrstop}} = c'_{\text{depstop}}$. Without loose of generality we can forbid successive transfers, as they can always be replace by a direct transfer that is not shorter. If a sequence begins (ends) with a transfer we say that it departs (arrives) at the source (target) stop of the transfer and has a departure (arrival) time. Otherwise it begins (ends) with a connection. We say that a partial journey departs (arrives) at that connection.

A partial journey is called a *complete journey* (or just journey) if it begins and ends with a transfer. To avoid a border case in the profile definition, we do not consider journeys consisting of a single transfer to be complete. This is no real restriction as testing for single transfer journeys is simple. We say that a (complete) journey is an $A@\tau \rightarrow B$-journey, if it departs at A no earlier than τ and arrives at B. Partial journeys are only used in intermediate steps of our algorithms. Unless stated otherwise journeys are complete. A complete journey can be represented more compactly by only storing the connections adjacent to transfers. The intermediate connections can easily be inferred.

2.4 Optimization Criteria. We consider different optimization criteria. The basic ones consist of minimizing arrival time or the number of transfers. However, these two can also be combined yielding further more complex criteria. For example we also consider determining a journey with a minimum number of transfers among those with a minimum arrival time. To avoid having to formulate algorithm variations for each criterion, we define an abstract journey cost. Unless stated otherwise, our non-accelerated profile algorithms can minimize any cost that fulfills a short set of axioms. All the afore mentioned criteria fulfill these requirements.

A *cost* is a function that maps partial and complete journeys onto values that can be totally ordered. A $A@\tau\to B$-journey is *optimal*, if no other $A@\tau\to B$-journey with lower costs exists. We require that the cost function fulfills the *suffix exchangeability* condition: Let $s_1 s_2 \ldots s_n$ denote an optimal $A@\tau\to B$-journey. Consider any suffix $s_i \ldots s_n$ that is a complete journey departing at stop C at time τ'. Replacing the suffix with any optimal $C@\tau'\to B$-journey must yield another optimal $A@\tau\to B$-journey.

The combined criteria can be realized using a cost function that maps journeys onto lexicographically ordered pairs, consisting of the arrival time and the number of transfers. What component is compared first decides what criteria is more important. As optimization we encode these pairs as single integer where the important criterion is encoded within the higher bits and the unimportant criterion in the lower bits. This allows to perform a lexicographical comparison using a single integer comparison.

It is straight forward to additionally define *prefix exchangeability*: Let $s_1 s_2 \ldots s_n$ denote an optimal $A@\tau\to B$-journey. Consider any prefix $s_1 \ldots s_i$ that is a complete journey arriving at C. Replacing the prefix with any optimal $A@\tau\to C$-journey must yield another optimal $A@\tau\to B$-journey. However, only the earliest arrival criterion fulfills this property. Minimizing the number of transfers does not and therefore the combined criteria do not either. For example suppose that the prefix $s_1 \ldots s_i$ contains 2 transfers and that a $A@\tau\to C$-journey with 1 transfer exists that arrives later. It is impossible to guarantee that exchanging the prefix does not invalidate the transfer connecting the prefix to the rest of the journey sequence. Transfer minimization not fulfilling prefix exchangeability severely limits the concept's usefulness. We therefore only exploit postfix exchangeability in our profile algorithms. This is the reason why we constructs journeys backward starting at the target stop.

2.5 Profile. *Profiles* are functions that map departure times onto arrival times (or generalized costs). Given two stops A and B we define a $A\to B$-profile to be the function that maps τ onto the costs of an optimal complete $A@\tau\to B$-journey (or $+\infty$ if no journey exists). As by definition all journeys must contain at least one connection and there are only finitely many connections, there may only be a finite number of τ values, where the function changes its value. The profile functions are therefore always step functions. Further as a $A@\tau\to B$-journey is also a $A@\tau'\to B$-journey for all $\tau' < \tau$, the profile functions are non-decreasing. Profiles can be identified by a sequence of $(\tau_i, \mathcal{C}_i)$-pairs, where the function changes its value. In each pair τ_i is the departure time and $\mathcal{C}_i$ the cost. The sequence is simultaneously ordered by τ_i and $\mathcal{C}_i$. These pairs correspond to departing vehicles.

To compute actual journeys (instead of just profiles) we tag every pair with two connection IDs. These represent the connections, where the user enters the first vehicle and exits the first vehicle of the journey. This allows to extract the first part of an optimal journey. The remaining parts can be extracted by iteratively repeating the process at intermediate stops. We refer to these connection IDs as *journey pointers*.

2.6 Problems. The input to the *earliest arrival time problem* consists of a network, a source stop p_s, a source time τ_s, and a target stop p_t. The goal is to compute the minimum arrival time over all $p_s@\tau_s\to p_t$-journeys. Analogously the *latest departure time problem* consists of maximizing the departure time among all journeys arriving not later than a given target time τ_t.

In the *profile problem*, one is given a network, a source stop p_s, and a target stop p_t. The objective is to compute the sequence of $(\tau_i, \mathcal{C}_i)$-pairs corresponding to the $p_s\to p_t$-profile. Note that source and target times are not restricted. The problem consists of computing full profiles.

The *range profile problem* takes as additional input a source time τ_s and a target time τ_t. All pairs in the $p_s\to p_t$-profile sequence should be computed that depart no earlier than τ_s and arrive no later than τ_t.

Note that the (range) profile problems are parametrized on the cost function used. First optimizing arrival time and then optimizing transfers as a secondary criterion is the cost function we want to solve. As an intermediate step in the preprocessing we will also need to solve profile problems that solely optimize transfers.

Further note that the earliest arrival time and latest departure time problems are basically equivalent. Solving one on a network N consists of solving the other on a transformed network N' constructed as following: The stops are the same. The transfer arcs have their direction inverted. All connections have exchanged departure and arrival stops. The departure and arrival times are negated. The range profile problem can be reduced to the profile problem by removing connections with departure time or arrival times outside of the range.

In our envisioned websystem the user is first prompted for a source stop and a target stop. He must additionally enter a source time or a target time (or both). If he enters both then the query consists of solving the range profile problem. If only one is given, the system must estimate a sensible value for the other one. If he indicates only a source time τ_s, the system solves an intermediate earliest arrival time problem, to determine the minimum travel time τ. The system then suggests $\tau_t = \tau_s + a \cdot \tau$, where a is some tuning parameter. We set $a = 2$. Analogously, if only a target time τ_t is given, the system solves the latest arrival time problem and suggests $\tau_s = \tau_t - a \cdot \tau$.

2.7 SIMD. Many modern processor architectures support single instruction multiple data (SIMD). These are special hardware registers that store fixed length vectors of scalars. The processors contain special instructions, that can apply operations componentwise to the vectors. These special instructions are as fast as their single scalar counterparts. Using bitmasks simple conditional operations can be done componentwise. However, most algorithms have to be adjusted to exploit such SIMD parallelism, because they contain branches that diverge too much to be done using bitmasks. A current x86 processor supports integer vectors of width 4 containing 32-bit integers. Next generation processors are planned to support vectors of 8 integers. Using accelerator cards widths of 16 integers are possible.

3 Connection Scan without Auxiliary Data

In this section we recapitulate the basic Connection Scan approach to journey planning introduced in [9]. We extend the profile algorithm to minimize generalized cost functions and parallelize it using SIMD.

All algorithms are based on dynamic programming. We maintain a solution that only considers the scanned connections. (The solution, however, always contains all stops and trips. The trips initial contain no connections though.) Initially no connection was scanned, which allows for a trivial initial solution. We then iteratively introduce connections ordered by their departure time. At each step we modify the solution to incorporate the new connection c. For this it is sufficient to only consider journeys containing c, as the other journeys can not be modified by introducing a new connection. As the connections are scanned ordered by departure time, the new connection c can only appear at the end of a journey (or beginning if scanned by decreasing departure time). Our objective is to compute journeys

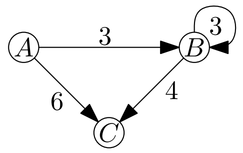


Figure 1: The capital letters denote stops. The arrows represent transfers and are annotated with their duration. A first vehicle arriving at $A@1$ sets $\tau(B) = 4$ and $\tau(C) = 7$. A second vehicle arriving at $B@2$ can improve $\tau(C)$ but not $\tau(B)$.

between a source stop p_s and a target stop p_t. However, during the algorithm we do not maintain such a one-to-one solution but a one-to-all solution, i.e., for a fixed source stop p_s and all possible target stops p_t. (When scanning the connection decreasing by departure time, we maintain an all-to-one solution.)

3.1 Earliest Arrival Time Algorithm. We solve the earliest arrival time problem by scanning the connections increasingly by departure time. We fix a source stop p_s and maintain a one-to-all solution, that consists of a tentative arrival time $\tau(p)$ for every possible target stop p. We further store at every trip q a reachability bit $r(q)$ that indicates whether a partial journey exists that ends in a connection of this trip. Initially no $r(q)$-bit is set. All $\tau(p)$ are $+\infty$ except if a transfer $p_s \to p$ exists. In this case we set $\tau(p) = \tau_s + d(p_s \to p)$. When scanning a connection c we first determine, whether $r(c_{\text{tripid}})$ must be updated and then adjust the tentative arrival times. If $r(c_{\text{tripid}})$ is set, a partial journey ending in a connection of trip c_{tripid} must exist. This partial journey can be extended by appending c, resulting in a partial journey ending in c. If it is not set then c can only be part of a journey, if the user can enter at c_{depstop}. This can be checked using $\tau(p_t) \leq c_{\text{deptime}}$ and if it is the case, $r(c_{\text{tripid}})$ must be set. If the updated $r(c_{\text{tripid}})$ is set, a partial journey ending in c must exist. By appending a transfer departing at c_{arrstop} the journey can be completed. We do this by iterating over all outgoing transfers $c_{\text{arrstop}} \to p$ and decreasing the tentative arrival time $\tau(p)$ if a better journey is found.

As basic optimizations the scan can be started at the first connection c with $c_{\text{deptime}} \geq \tau_s$ and aborted once $\tau(p_t) \leq c_{\text{deptime}}$ holds. A further more complex optimization is called *limited propagation*. It consists of only iterating over the outgoing transfers, if $c_{\text{arrtime}} \leq \tau(c_{\text{arrstop}})$. Intuitively the idea behind limited propagation is, that if we can not ameliorate the solution at c_{arrstop}, we can not do it anywhere, as the better journey ending in c_{arrstop} can be append with another transfer resulting in a better journey everywhere. However, note that the afore mentioned test is not equivalent with $\tau\left(c_{\text{arrstop}}\right)$ being decreased as the situation in Figure 1 demonstrates.

3.2 Profile Algorithm. The profile algorithm uses a very similar approach. The main difference is that the connections are scanned decreasing by departure time to exploit suffix exchangeability. We fix a target stop p_t and maintain an all-to-one instead of a one-to-all solution. Further storing full profiles is more complex, which reflects in more complex data structures. We first introduce the data structures used, then describe how they relate to the profiles and what their counterparts in the basic earliest arrival case are. We then describe how a new connection c can be introduced and what modifications in the data structure are necessary.

Our datastructure consists of: the *tentative cost functions* F_p, the *tentative trip costs* $\mathcal{C}_q$, and the *transfer-to-target durations* w_p. The tentative cost functions F_p replace the tentative arrival times of the base algorithm. They are profile functions that map the departure time at the stop p onto the minimum cost over all journeys with target p_t. The tentative trip costs $\mathcal{C}_q$ replace the trip reachability bit of the base algorithm. These are values that store the minimum cost over all partial journeys that depart in a connection of this trip and arrive at p_t. Finally the transfer-to-target durations w_p have no real counterpart in the base algorithm. If a transfer $p \to p_t$ exists then w_p is set to $d\left(p \to p_t\right)$ and $+\infty$ otherwise.

We initially set the w_p to $+\infty$ at the beginning of the program and before and after each query we iterate over all incoming transfers of p_t. By definition each complete journey must contain a connection and therefore initially no journey can exist. This allows us to initially let each tentative cost function map every value onto $+\infty$ and also set all tentative trip costs to $+\infty$.

Let c be the currently processed connection. We first compute the minimum cost over all journeys departing in c to update $\mathcal{C}_{c_{\text{tripid}}}$. To do this consider all options the user has upon the arrival of c: Sometimes he can directly transfer to the target, or he can remain seated, or use a transfer and enter another vehicle. The costs of the first two cases can be computed using w_p and $\mathcal{C}_q$. The last one requires evaluating $F_{c_{\text{arrstop}}}$. The correctness of the later can be seen as following: Suppose an optimal journey did exit upon the arrival of c. The suffix exchangeability allows us to exchange the journey suffix after c, with the journey that caused the value of $F_{c_{\text{arrstop}}}$. After updating $\mathcal{C}_{c_{\text{tripid}}}$ we prefix the corresponding partial journey with all incoming transfers $p \to c_{\text{depstop}}$ and adjust F_p if needed. Limited propagation can be applied: It is sufficient to only iterate over the transfers, if inserting a $(c_{\text{deptime}}, \mathcal{C}_{c_{\text{tripid}}})$ would not modify the function.

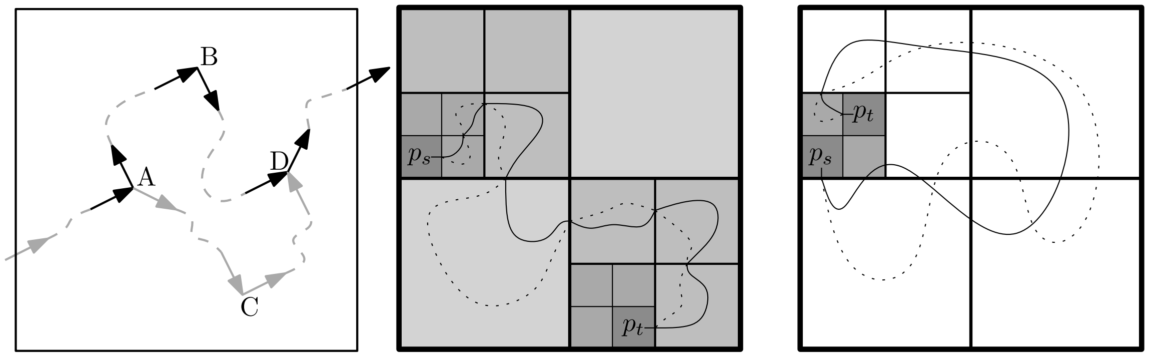
3.3 Tentative Cost Function Operations. Recall that profile functions can be identified by an ordered sequence of $(\tau_i, \mathcal{C}_i)$-pairs. We store these in dynamic arrays that can be enlarged at the front. A function is evaluated at τ by sequentially iterating over all pairs starting at lower values. In theory this *evaluation loop* can have many iterations. However, experiments in Section 6 show that the contrary is the case: The loop nearly always aborts directly. The newly introduced connection c has a minimum departure time. It therefore tends to only change profiles for small values of τ_i, i.e., precisely those stored at the front of the array. New pairs are also inserted using a sequential scan. We call this loop the *insertion loop*. The same effect can be observed. In essence all operations require quasi constant running time, giving the whole profile algorithm a running time quasi proportional to the number of scanned connections. In [9] we have shown, that if no interstop transfers (i.e., footpaths) exist, the algorithm can be modified such that amortized constant worst case running time bounds for the cost function operations can be proven.

3.4 Multi-Query Connection Scan. The profile algorithm can be parallelized using SIMD, resulting in lower amortized times. Denote by n the width of the SIMD vectors. Our algorithm can compute profiles for n different target stops in parallel. This is useful in preprocessing. A different scenario involves a server with many simultaneous requests. It can answer several queries in parallel using a single thread, resulting in a higher throughput.

We replace $\mathcal{C}_q$ and $w_{c_{\text{arrstop}}}$ by SIMD-vectors and all operations on them by SIMD-instructions in a straightforward way. The varying lengths of the dynamic arrays underlying the tentative cost functions result in diverging code paths prohibiting a direct parallelization. We observe that these arrays may contain duplicated entries without producing incorrect results. We store a single array that is used by all n queries. It contains $(\tau_i, \mathcal{C}_i)$-pairs, where τ_i is a non-vectorized departure time and $\mathcal{C}_i$ is a SIMD-vector of costs. A pair is inserted if is not dominated in every query. Dominated cost vector components are replaced by the dominating cost. The arrays are longer but SIMD-parallelization is possible.

4 Accelerating Connection Scan

We utilize the core idea from CRP [5, 8]. It consists of subdividing a road graph into cells and computing for each cell a replacement graph that preserves shortest



(a) The solid black connections are transit connections.

(b) The transit connection sets participating in a normal query.

(c) The transit connection sets participating in a local query.

Figure 2: Boxes represent cells. In the right two figures every cell is subdivided into 4 subcells. Curved lines are journey parts consisting of several connections. Arrows denote single connections. Letters denote stops. The shade of a cell indicates the density of a cell. A darker shade signals a denser set.

paths. Translated into the timetable setting, we recursively partition the stop set into k cells over ℓ levels. The top cell is the full stop set. We define a connection to be *interior (exterior)* to a cell z, if it (does not) depart in z. An *entry* (*exit*) connection of a z is departs (arrives) outside of z but arrives (departs) inside.

4.1 Partitioning. The stops are partitioned by running a graph partitioner [17] on the following graph: Stops are mapped onto the same node if there is a transfer between them. This prohibits transfers from crossing cell borders. An edge exists if a corresponding connection exists and is weighted by the number of such connections.

4.2 Transit Connection Set $T(z)$. For every cell z we store a set of transit connections $T(z)$. The transit connections are interior connections and every optimal journey that goes through the cell can be replaced by a journey, that only changes vehicles at connections in $T(z)$ (or are exterior to z). Consider the example in Figure 2a. Suppose that the journeys $A \to B \to D$ and $A \to C \to D$ are both optimal. It is sufficient to add one of them (in the example the one through B) into the transit connection set, because any subjourney though C can be replaced by the one through B without modifying the cost. Note that several transit connection sets are valid, as we could have included the journeys through C. Our algorithm computes some small valid sets but not necessarily a smallest one. The set only contains the connections adjacent to a transfer in the journey. The intermediate connections are not contained. We do this because the base algorithms introduced in Section 3 successfully find the journeys traversing the cell, even if only those connections are scanned. Also note that connections that depart outside of z are not part of the transit set of z, but of the set of its neighboring cell.

Transit connections are computed using the algorithms in Section 5. In this section we assume that a blackbox is available that does this. The input of the blackbox is the set of entry and exit connections of z and a set of connections to scan.

4.3 Long Distance Connection Set $D(z)$. For every cell z we additionally define a set of *long distance connections* $D(z)$ as following: If z is at the bottom level then $D(z)$ is the set of all interior connections, and otherwise it is the union of the transit connection sets of all direct children cells of z, i.e., $D(z) = \bigcup T(z_{\text{child}})$. At query time we merge all $D(z)$ of cells z containing the source stop or the target stop (or both) resulting in one large connection subset Q. The set Q is illustrated in Figure 2b. Let the dotted journey j be optimal. We can not guarantee that our algorithms will find j. However, one can iteratively replace the subjourneys in each cell and obtain the solid line journey j'. We can guarantee that j' is optimal and our algorithms find j'. We run the algorithms from Section 3 but only scanning the connections in Q.

4.4 Improving Local Queries. *Local queries*, where the source stop and the target stop are nearby, are common and therefore we optimize them. We achieve this by not merging all sets up to the top level but only up to the lowest common ancestor cell z_{lca}. Consider the situation in Figure 2c. The top shaded cell is z_{lca}. Most journeys will be fully contained within z_{lca} and will therefore be found. However a few journeys exist that exit z_{lca} and reenter it (possibly multiple times such as the solid journey). We therefore store an additional set of loop connections $L(z_{\mathrm{lca}})$ at every cell, that contains all connections necessary to find journeys that exit and reenter z_{lca}. The set Q is the union of the long distance connection set of all sets containing either p_s or p_t (but not both), and $D(z_{\mathrm{lca}})$ and $L(z_{\mathrm{lca}})$.

The loop connection set of a cell z can be computed by passing z's entry and exit connections to the blackbox and letting it scan every connection (including those outside of z). The connections not needed for the extremely rare journeys that enter and exit z multiple times and are inside of $D(z)$ can be removed from $L(z)$.

4.5 Efficiently Computing $T(z)$, $D(z)$ and $L(z)$. Computing the connection sets by considering the whole network is too slow. We therefore exploit the hierarchy already during their construction and restrict the connections scanned by the blackbox. In a first phase we compute $T(z)$ and $D(z)$ in an alternating bottom up fashion. In the second phase $L(z)$ is computed top down. The main idea of the first phase is that transit connections are long distance connections, i.e., $T(z) \subseteq D(z)$. It is therefore sufficient if the blackbox scans $D(z)$. Long distance connections are computed using using $D(z) = \bigcup T(z_{\mathrm{child}})$, with the exception of the lowest level where $D(z)$ is the set of interior connections. The second phase exploits that loop connections are either loop or long distance connection of the parent cell, i.e., $L(z) \subseteq D(z_{\mathrm{parent}}) \cup L(z_{\mathrm{parent}})$. The exception is the top cell where the loop set is empty. To improve locality we remove stops without connections or transfers before running the blackbox. Operations on the same level can be parallelized but we found that parallelizing the blackbox is more effective because of unbalanced workloads per cell. Recall that we scan the connections ordered by departure time. We therefore assign ids to the connections in this order. All computed sets are stored as ordered connection ID arrays. This allows efficient merging without loading any connection details. If a time range is given, we first discard the connections outside of the range and then merge the remaining connections.

4.6 Tailored for the Earliest Arrival Problem. We can solve the earliest arrival problem by first computing Q and then running the basic CSA algorithm. However, we can accelerate the base algorithm by not explicitly merging the connection sets and processing the connections out of order.

To achieve this we must first modify the base algorithm to support the scanning of connections in

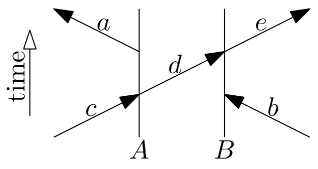


Figure 3: The vertical lines are stops and the arrows connections. The upper case letters denote stops and the lower case letters connections. The connections c, d and e are part of the same trip.

arbitrary order and not only by increasing departure time. Consider the situation depicted in Figure 3. Suppose that initially only b is reachable, i.e., only a partial journey ending in b exists. The connections are scanned in following order: b, e, c, a. After scanning b the stop B is reached. After processing e, the reachability bit of the trip e_{tripid} is set. As $e_{\mathrm{tripid}} = c_{\mathrm{tripid}}$ the next connection c is considered reachable as $r(c_{\mathrm{tripid}})$ is set, even though no partial journey exists that ends in c. As c is considered reachable the stop A is reached and thus the next connection a is also erroneously considered reachable. This problem can be fixed by replacing $r(q)$ by an integer representing the position of the first reachable connection inside trip q. When scanning a connection c the algorithm does not test whether the bit is set, but whether $r(c_{\mathrm{tripid}})$ is smaller than c's position.

Consider a query with source cell z_s, target cell z_t and source time τ_s. If a journey exists then an optimal journey exists that first uses connections from $D(z_s)$, then from $D(z_s^{\mathrm{parent}})$, ..., $D(z_{\mathrm{lca}})$, $L(z_{\mathrm{lca}})$, $D(z_{\mathrm{lca}})$, ..., $D\left(z_t^{\mathrm{parent}}\right)$, and finally from $D(z_t)$. Instead of merging those sets we "guess" that an earliest arrival journey arrives before $\tau_s + c$ for some constant c. Next we try to verify if we guessed correctly and if not increase c and repeat. We verify our guess by iterating over the sets in the order given above. For each set S we scan the connections in S departing before $\tau_s + c$ increasing departure time. If a journey arriving before $\tau_s + c$ exists, it is found this way. As optimization, we do not process connections multiple times and ignore those departing before τ_s by running a binary search on each set.

5 Computing Transit Connections

In this section we implement the blackbox from Section 3. Given a set of entry and exit connections and a set of connections to scan, the blackbox computes the transit connections. We describe two constructions $T_a(z)$ and $T_t(z)$ with different properties. The *arrival time transit set* $T_a(z)$ can be computed quickly but only guarantees that an earliest arrival journey is found. It has no guarantees with respect to transfers. Further

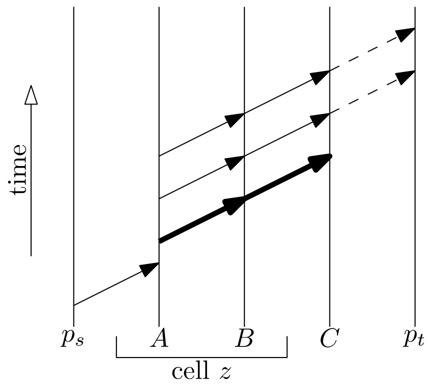


Figure 4: The vertical lines are stops. The arrows are connections. The vertical position is the time. Connections arriving and departure at the same moment are part of the same trip. Stops A and B are part of the same cell z. Stop A is an interior border stop of z, whereas C is an exterior border stop. Only the thick connections are in $T_a(z)$, whereas all interior connections are in $T_t(z)$.

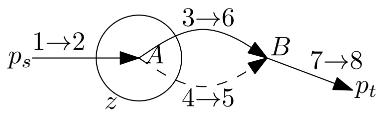


Figure 5: The capital letters represent stops. The arrows are connections. The solid arrows are part of the same trip. The dashed one is in its own trip. The circle represents a cell z containing only the stop A. The stop A is an interior border stop and B is an exterior one. The connections are annotated with departure and arrival times. Only the dashed connection is in $T_a(z)$.

non-zero duration transfer loops, i.e. minimum change times, are not supported. On the other hand, the *transfer transit set* $T_t(z)$ needs more preprocessing time but does not have these limitations. The transit sets are constructed such that the subjourney between an entry and exit connection pair can be replaced with an equivalent subjourney found in the transit set. Both constructions are parallelized using threads and SIMD.

5.1 Arrival Time Transit Set $T_a(z)$**.** We first identify all *interior (exterior) border stops* as the arrival stops of entry (exit) connections. Note that there are significantly less border stops than border connections. In the next step we compute profiles from every interior border to every exterior border stop. We optimize arrival time for correctness. We optimize transfers secondarily to heuristically improve the results. Figure 4 depicts an example $T_a(z)$.

This construction is only correct if all transfer loops have zero-duration. Consider the situation in Figure 5. As the dashed connection dominates the solid one, only the dashed one is included in $T_a(z)$. If B has a zero transfer loop an earliest arrival journey between p_s and p_t is found, as the user can transfer at B from the dashed vehicle back into the solid one. However, if the transfer duration of the loop at B is greater than 2, no journey is found at all.

5.2 Transfer Transit Sets $T_t(z)$**.** To compute $T_t(z)$ we compute for each pair of entry and exit connections a journey with a minimum number of transfers. Note that for a specific pair all journeys are required to end in the same connection. All valid journeys therefore have the same arrival time and thus it is sufficient to optimize transfers. We do this by computing profiles between every pair of entry and exit connections. As profiles can only be computed between stops, we split each entry and exit connection into two and add a dummy stop in the middle. We compute the profiles between pairs of dummy stops.

5.3 Parallelization. Both transit sets are computed by enumerating all profiles between a set of source stops S and a set of target stops T. For each $p_t \in T$ we compute an all-to-one profile solution and extract for each $p_s \in S$ the corresponding journeys by following the journey pointers. We parallelize this operation by subdividing T into work packages of n elements, where n is the width of the SIMD vector. A thread then processes a work package by computing all profiles simultaneously using SIMD. However, the journey extraction can not be done using SIMD.

5.4 Intermediate Transit Sets. The transfer transit set $T_t(z)$ is constructed such that it conserves all Pareto optimal journeys. However we do not exploit this property afterwards. We only compute a journey with a minimum amount of transfers among all earliest arrival journeys. Is it possible to design a transit set larger than the arrival time transit set $T_a(z)$ but smaller than $T_t(z)$ that only conserves these journeys? We can not give a definitive answer about its existence but it is very unlikely that it could be computed faster than $T_t(z)$ and would therefore be useless. Consider the situation in Figure 4. Every transit set must contain the thick connections because otherwise the earliest arrival journey from p_s to C is not conserved. The question therefore is: Which of the thin interior connections are superfluous? For an efficient transit set computation, it is necessary that we only use information depending on the interior of the cell. We therefore can not tell which

transfer	SIMD	run. time [ms]	extra time for journey pointers	inserted pair count $/10^6$	pairs earliest when inserted	average insert loop iterations	average evaluation loop iterations
○	○	2 640	509	4.3	97.5%	1.006	1.013
●	○	3 361	812	5.7	97.6%	1.009	1.021
●	●	1 468	405	11.6	97.1%	1.023	1.055

Table 1: Details of the all-to-one profile algorithm on the full network. The time range is not restricted.

of the dashed connections exist. We have to compute one transit connection set, that is correct for each combination of dashed connections existing. If both of them exist, all thin connections must be transit connections. We conclude that it is not possible to conserve fewer than all Pareto optimal journeys, without looking at the outside of the cell.

6 Experiments

We ran experiments on a dual 8-core Intel Xeon E5-2670 processor clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. We use g++ 4.7.1 with `-O3`. In the tables an enabled feature is marked by a "●" and a disabled one by a "○". SIMD is done using SSE with registers containing 4 integers of 32 bits. We ran 10^4 queries with random source and target stops and a random source time within the first 24 hours.

6.1 Instance. Our test instance is based on the data of `bahn.de` during winter 2011/2012. The data contains European long distance trains, German local trains, many buses inside of Germany and even more exotic vehicles such as rack railways in the Alps. To obtain an instance comparable in size with [1] we extracted all trips regardless of their day of operation and consider two successive identical days. We removed data noise such as exactly duplicated trips, vehicles driving at more than 300 km/h or footpaths at more than 50 km/h. More than half of the connections are buses. Minimum change times are modeled as transfer loops and footpaths as interstop transfer arcs. We transitively closed the transfers and obtain the sizes reported in Table 2. For comparison with [7] we indicate the number of FIFO-routes, i.e., into how many parts the set of trips has to be partitioned (at most) such that all trips in one part have the same stop sequence and do not overtake eachother. For comparison with [1] the time-expanded (TE) graph size is indicated.

#stops	252 374
#connections	46 218 148
#trips	2 395 656
#footpaths	103 535
#FIFO-routes	229 666
#TE-nodes	82 017 803
#TE-arcs	202 073 458

Table 2: Instance size

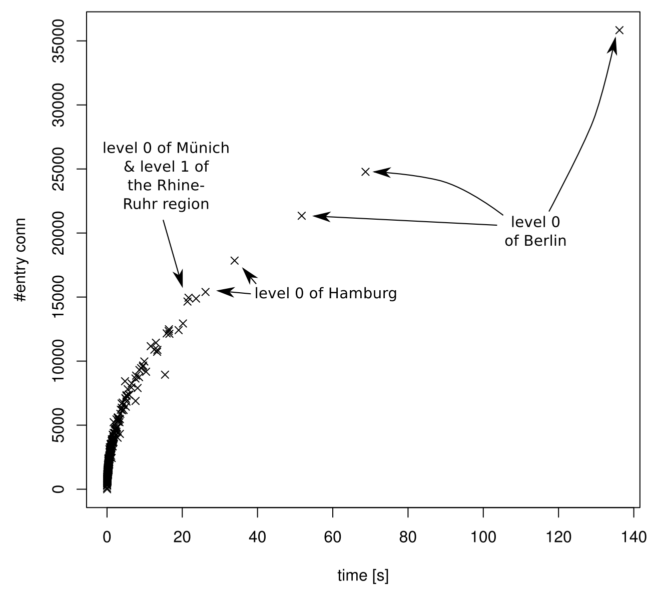
6.2 Reference Times. We ran a reasonably tuned version of Dijkstra's algorithm with a binary heap. The advanced optimizations from [18] were not included. On average it settled 10 554 570 nodes and needed 2 960ms for a one-to-one query optimizing only arrival time. A non-profile one-to-one CSA as described in Section 3 needed on average 298.6ms.

6.3 Profile Queries. In Table 1 we evaluate the unaccelerated CSA described in Section 3. We report running times with and without journey pointers. SIMD running times are divided by the number of simultaneous queries. We report the number of inserted pairs and how many were inserted at the front of the array (i.e. have an earliest departure time when inserted). We further report how many iterations the insertion and the evaluation loops needed. Computing pointers increases running times by 25%. Optimizing transfers inserts additional pairs which costs in time. SIMD halves the running time. The two loops nearly always terminate right away. SIMD increases the number of iterations but they remain negligible. Unsurprisingly a near constant insertion loop implies most pairs being inserted at the array's front.

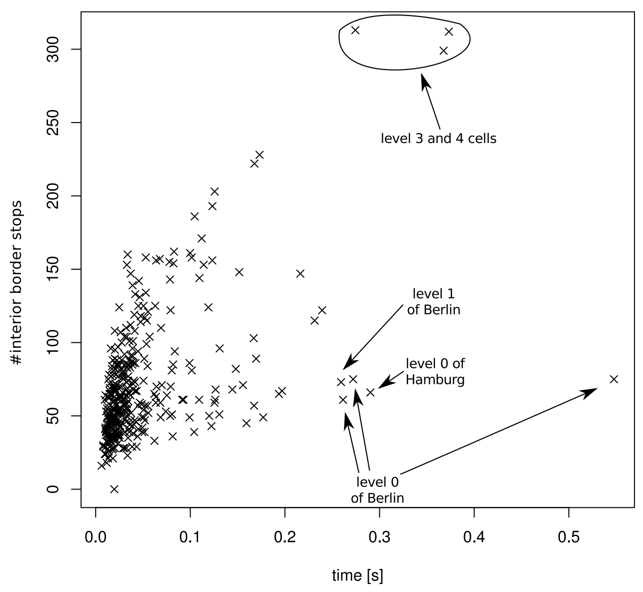
6.4 Computing Overlays. In Table 4 we report the times needed to compute the transit sets introduced in Section 5. To compute the $T_a(z)$ we set all transfer loops to zero in this experiment. "transfer" indicates whether $T_t(z)$ or $T_a(z)$ is computed. All 16 cores are used. We recursively subdivide the stops 5 times into 3

SIMD	transfer	running time [s]
○	○	45.4
●	○	49.2
○	●	2007.8
●	●	1794.7

Table 4: Parallelized running time needed to compute overlays.



(a) Computing $T_t(z)$.



(b) Computing $T_a(z)$.

Figure 6: Running times needed to compute the transit connections against their border size. The data points are annotated with the geographical regions that they belong to. Thread parallelization was used. SIMD was not active.

	only time $T_a(z)$		with transfers $T_t(z)$	
level	$\lvert D(z)\rvert$	$\lvert L(z)\rvert$	$\lvert D(z)\rvert$	$\lvert L(z)\rvert$
0	193 029	11 232	193 029	11 109
1	70 781	15 204	115 660	14 042
2	93 933	20 046	138 036	18 836
3	125 642	24 046	181 689	24 065
4	154 790	25 155	222 143	24 596
5	168 080	0	277 454	0

Table 3: Number of long distance and loop connections averaged over all cells on the same level. Level 5 is the top.

partitions. $T_a(z)$ can be computed in under a minute enabling real time updates. The alternative $T_t(z)$ can be computed in half an hour, which should be fast enough for most applications. SIMD-speedup is smaller than in the base query case because following journey pointers is sequential. Further interleaving pointers from different queries destroys cache locality, which explains the slight slowdown for $T_a(z)$. We use $T_t(z)$ in all query experiments.

Table 3 shows the sizes of the connection sets corresponding to Table 4. A first observation is that the arrival time transit sets $T_a(z)$ are smaller, than the transfer transit sets $T_t(z)$. This is a direct result of the effect illustrated in Figure 4. As expected the number of loop connections is significantly smaller than the number of long distance connections. With the exception of the bottom level the average size of each long distance connection set increases with each level. The size of the bottom level suggests that a further subdivision might be useful to decrease query running times. We decided against it after doing some quick preliminary experiments. The higher levels dominate the amount of scanned connections and therefore the maximum impact on the query running times is small. However, an additional level exponentially increases the number cells, which resulted in significantly longer preprocessing times.

In Table 4 we observed that computing $T_t(z)$ is significantly more expensive than computing $T_a(z)$. We therefore analyzed the running times per cell in greater detail in Figure 6. Figure 6a shows that the running times needed to compute the $T_t(z)$ is quadratic in the number of entry connections of z. This was to be expected, given that there are about as many exit connections as entry connections and we consider all pairs of exit and entry connections. A more detailed investigation reveals that computing $T_t(z)$ for the metropolitan areas on the lowest levels are by far the most expensive cells. Figure 6b depicts similar information for $T_a(z)$. A first observation is that the square of the interior border stops only is a lower bounds on the running time. The interior of the cells has also a significant influence on performance. For both transit

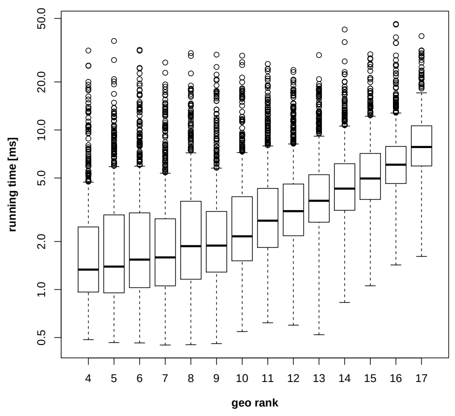


Figure 7: Geo-rank showing the running times of the tailored EA query

range	accel.	merge [ms]	scan [ms]	processed conn $/10^6$
○	○	—	4 173	46.2
○	●	12	159	1.2
●	●	6	72	0.6

Table 5: Running times for the accelerated algorithm split up into the merge and scan phases and the number of processed connections.

set types the same Berlin cell needs the most time.

6.5 Earliest Arrival Time and Local Queries. We examine in Figure 7 the speedup of local queries. We use a geo-rank instead of the common Dijkstra-rank used for road networks. With the later it is unclear what source time to use. Further, especially in rural areas, travel times between neighboring villages can be huge. A Dijkstra-rank [19] does not recognize these queries as being local. A geo-rank picks a random source stop and orders all other stops by geographical distance. It runs queries towards the 2^r-th stop. The *geo-rank* of the query is r. Local queries are on average by a factor of 6 faster than long distance connections. The figure also shows that the running times have a lot of variance. The largest outliers are queries where no journey exists. With random source and target stops our query averages at 8.66ms, which is about the running time of query with maximum geo-rank. This is the case because picking stops uniformly at random nearly never picks source and target stops that are close.

6.6 Accelerated Profile Queries. We evaluate in Table 5 the speedup of our technique. We assume that a earliest arrival time query ($\approx$ 8.66ms) was run to compute a minimum travel time τ while the user selects the time interval. As $\tau = +\infty$ is easy to test, we assume that a journey exists. Journey pointers were computed. "accel" indicates whether transit sets were used. "range" indicates whether a source and target time restricted the scan. The source time τ_s is chosen at random and the target time τ_t is set to $\tau_s + 2\tau$. Using transit connections achieves a speedup of 24. This is less than the ratio of processed connections because the relevant connection data is no longer adjacent in memory. Restricting the time interval yields another factor 2 totaling in a speedup of 49 over a CSA profile baseline. Interestingly the number of processed connections is still large, which explains why the scan phase dominates.

6.7 Comparison with Transfer Pattern [1]. Fair comparisons are hard because the problem formulations and instances differ. Further their algorithm is heuristic. However, no detailed error study was published making the gain of optimality hard to quantify. The TE-node ratio between our instance and their largest North America instance is 0.72. The instances are roughly comparable in size. Their processor is probably a bit slower than ours but they publish no details. The fastest (non-adjusted) reported running time for a one-to-one earliest arrival variant is 7ms compared to our 9ms. Because of the various differences in setup the numbers can not directly be compared. We conclude that the order of magnitude is the same. However, we have provable correctness. Further there is a huge difference in preprocessing time. We need 8 hours single core. They report $2632\text{h} + 571\text{h} \geq 4$ months single core.

6.8 Comparison with RAPTOR [7]. In [9] we compared RAPTOR with the non-accelerated CSA and concluded that it is slower than our profile CSA baseline. Further the London instance used in [9, 7] has significantly more trips per route. A higher ratio benefits RAPTOR. We expect the running time gap to be larger. They compute full Pareto sets optimally.

7 Conclusion.

We showed that profile queries can be solved optimally on country-sized timetable networks within 78ms. Our algorithm optimizes realistic transfers and has reasonable preprocessing times. We equate Transfer Pattern, the fastest competitor, in terms of query times without dropping optimality. But we are significantly better in terms of preprocessing time.

Directions for future research include: (1) Evaluating the impact of other graph partitioner than METIS [17], such as PUNCH [6] or KaHip [20]; (2) Better algorithms to compute transfer transit sets in dense metropolitan areas; (3) Solving more complex queries such as MEAT [9] or complex Pareto queries.

References

[1] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks using Transfer Patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[2] Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-Robustness of Transfer Patterns in Public Transportation Route Planning. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*, OpenAccess Series in Informatics (OASIcs), pages 42–54, September 2013.

[3] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, OpenAccess Series in Informatics (OASIcs), 2009.

[4] Annabell Berger, Martin Grimmer, and Matthias Müller–Hannemann. Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 35–46. Springer, May 2010.

[5] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[6] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011.

[7] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140. SIAM, 2012.

[8] Daniel Delling and Renato F. Werneck. Faster Customization of Road Networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013.

[9] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013.

[10] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.

[12] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer, May 2010.

[13] VBB Verkehrsverbund Berlin-Brandenburg GmbH. `http://daten.berlin.de/datensaetze/vbb-gtfs-daten-jun-dez-2013`, 2013.

[14] GTFS. `https://developers.google.com/transit/gtfs/reference`, 2013.

[15] HAFAS. `http://www.hacon.de/hafas`, 2013.

[16] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[17] Dominique LaSalle and George Karypis. Multi-Threaded Graph Partitioning. In *27th International Parallel and Distributed Processing Symposium (IPDPS'13)*. IEEE Computer Society, 2013.

[18] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

[19] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[20] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

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

[22] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.