Intriguingly Simple and Fast Transit Routing*

Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany.
{dibbelt,pajor,strasser,dorothea.wagner}@kit.edu

Abstract. This paper studies the problem of computing optimal journeys in dynamic public transit networks. We introduce a novel algorithmic framework, called Connection Scan Algorithm (CSA), to compute journeys. It organizes data as a single array of connections, which it scans once per query. Despite its simplicity, our algorithm is very versatile. We use it to solve earliest arrival and multi-criteria profile queries. Moreover, we extend it to handle the minimum expected arrival time (MEAT) problem, which incorporates stochastic delays on the vehicles and asks for a set of (alternative) journeys that in its entirety minimizes the user's expected arrival time at the destination. Our experiments on the dense metropolitan network of London show that CSA computes MEAT queries, our most complex scenario, in 272 ms on average.

1 Introduction

Commercial public transit route planning systems are confronted with millions of queries per hour [12], making fast algorithms a necessity. Preprocessing-based techniques for computing point-to-point shortest paths have been very successful on road networks [8, 16], but their adaption to public transit networks [2, 10] is harder than expected [1, 3, 4]. The problem of computing "best" journeys comes in several variants [14]: The simplest, called *earliest arrival*, takes a departure time as input, and determines a journey that arrives at the destination as early as possible. If further criteria, such as the number of transfers, are important, one may consider *multi-criteria* optimization [7, 9]. Finally, a *profile query* [6, 7] computes a set of optimal journeys that depart during a period of time (such as a day). Traditionally, these problems have been solved by (variants of) Dijkstra's algorithm on an appropriate graph model. Well-known examples are the time-expanded and time-dependent models [6, 10, 14, 15]. Recently, Delling et al. [7] introduced RAPTOR. It solves the multi-criteria problem (arrival time and number of transfers) by using dynamic programming directly on the timetable, hence, no longer requires a graph or a priority queue.

In this work, we present the *Connection Scan Algorithm* (CSA). In its basic variant, it solves the earliest arrival problem, and is, like RAPTOR, not graph-based. However, it is not centered around *routes* (as RAPTOR), but elementary *connections*, which are the most basic building block of a timetable.

* Partial support by DFG grant WA654/16-1 and EU grant 288094 (eCOMPASS).

CSA organizes them as one single array, which it then scans once (linearly) to compute journeys to all stops of the network. The algorithm turns out to be intriguingly simple with excellent spatial data locality. We also extend CSA to handle multi-criteria profile queries: For a full time period, it computes Pareto sets of journeys optimizing arrival time and number of transfers. Finally, we introduce the *minimum expected arrival time problem* (MEAT). It incorporates uncertainty [5, 9, 11] by considering stochastic delays on the vehicles. Its goal is to compute a set of journeys that minimizes the user's *expected* arrival time (at the destination). The output can be viewed as a decision graph that provides all relevant alternative journeys at stops where transfers might fail (see Fig. 1). We extend CSA to handle these queries very efficiently. Moreover, we do not make use of heavy preprocessing, thus, enabling dynamic scenarios including train cancellations, route changes, real-time delays, etc. Our experiments on the dense metropolitan network of London validate our approach. With CSA, we compute earliest arrival queries in under 2 ms, and multi-criteria profile queries for a full period in 221 ms—faster than previous algorithms. Moreover, we solve the most complex of our problems, MEAT, with CSA in 272 ms, fast enough for interactive applications.

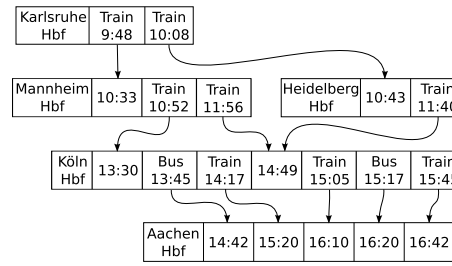

Fig. 1. Delay-robust itinerary from Karlsruhe to Aachen, Germany. A user should try to take the leftmost path. If transfers fail, alternatives are available.

This paper is organized as follows. Section 2 sets necessary notion, and Section 3 presents our new algorithm. Section 4 extends it to multi-criteria profile queries, while Section 5 considers MEAT. The experimental evaluation is available in Section 6, while Section 7 contains concluding remarks.

2 Preliminaries

Our public transit networks are defined in terms of their aperiodic *timetable*, consisting of a set of *stops*, a set of *connections*, and a set of *footpaths*. A *stop* p corresponds to a location in the network where a passenger can enter or exit a vehicle (such as a bus stop or train station). Stops may have associated minimum change times, denoted $\tau_{\text{ch}}(p)$, which represent the minimum time required to change vehicles at p. A *connection* c models a vehicle departing at a stop $p_{\text{dep}}(c)$ at time $\tau_{\text{dep}}(c)$ and arriving at stop $p_{\text{arr}}(c)$ at time $\tau_{\text{arr}}(c)$ without intermediate halt. Connections that are subsequently operated by the same vehicle are grouped into *trips*. We identify them by $t(c)$. We denote by c_{next} the next connection (after c) of the same trip, if available. Trips can be further grouped into *routes*. A route is a set of trips serving the exact same sequence of stops. For correctness, we require trips of the same route to not overtake each other. *Footpaths* enable walking transfers between nearby stops. Each footpath consists of two stops

with an associated walking duration. Note that our footpaths are transitively closed. A *journey* is a sequence of connections and footpaths. If two subsequent connections are not part of the same trip, their arrival-departure time-difference must be at least the minimum change time of the stop. Because our footpaths are transitively closed, a journey never contains two subsequent footpaths.

In this paper we consider several well-known problems. In the *earliest arrival problem* we are given a source stop p_s, a target stop p_t, and a departure time τ. It asks for a journey that departs from p_s no earlier than τ and arrives at p_t as early as possible. The *profile problem* asks for the set of all earliest arrival journeys (from p_s to p_t) for every departure at p_s. Besides arrival time, we also consider the number of transfers as criterion: In multi-criteria scenarios one is interested in computing a *Pareto set* of nondominated journeys. Here, a journey J_1 *dominates* a journey J_2 if it is better with respect to every criterion. Nondominated journeys are also called to be *Pareto-optimal*. Finally, the *multi-criteria profile problem* requests a set of Pareto-optimal journeys (from p_s to p_t) for all departures (at p_s).

Usually, these problems have been solved by (variants of) Dijkstra's algorithm on an appropriate graph (representing the timetable). Most relevant to our work is the realistic *time-expanded model* [15]. It expands time in the sense that it creates a vertex for each *event* in the timetable (such as a vehicle departing or arriving at a stop). Then, for every connection it inserts an arc between its respective departure/arrival events, and also arcs that link subsequent connections. Arcs are always weighted by the time difference of their linked events. Special vertices may be added to respect minimum change times at stops. See [14, 15] for details.

3 Basic Connection Scan Algorithm

We now introduce the Connection Scan Algorithm (CSA), our approach to public transit route planning. We describe it for the earliest arrival problem and extend it to more complex scenarios in Sections 4 and 5. Our algorithm builds on the following property of public transit networks: We call a connection c *reachable* iff either the user is already traveling on a preceding connection of the same trip $t(c)$, or, he is standing at the connection's departure stop $p_{\text{dep}}(c)$ on time, i. e., before $\tau_{\text{dep}}(c)$. In fact, the time-expanded approach encodes this property into a graph G, and then uses Dijkstra's algorithm to obtain optimal sequences of reachable connections [15]. Unfortunately, Dijkstra's performance is affected by many priority queue operations and suboptimal memory access patterns. However, since our timetables are aperiodic, we observe that G is acyclic. Thus, its arcs may be sorted topologically, e. g., by departure time. Dijkstra's algorithm on G, actually, scans (a subsequence of) them in this order.

Instead of building a graph, our algorithm assembles the timetable's connections into a single array C, sorted by departure time. Given source stop p_s and departure time τ as input, it maintains for each stop p a label $\tau(p)$ representing the earliest arrival time at p. Labels $\tau(\cdot)$ are initialized to all-infinity, except $\tau(p_s)$, which is set to τ. The algorithm scans all connections $c \in C$ (in order), testing

if c can be *reached*. If this is the case and if $\tau_{\text{arr}}(c)$ improves $\tau(p_{\text{arr}}(c))$, CSA *relaxes* c by updating $\tau(p_{\text{arr}}(c))$. After scanning the full array, the labels $\tau(\cdot)$ provably hold earliest arrival times for all stops.

Reachability, Minimum Change Times and Footpaths. To account for minimum change times in our data, we check a connection c for reachability by testing if $\tau(p_{\text{dep}}(c)) + \tau_{\text{ch}}(p_{\text{dep}}(c)) \leq \tau_{\text{dep}}(c)$ holds. Additionally, we track whether a preceding connection of the same trip $t(c)$ has been used. We, therefore, maintain for each connection a flag, initially set to 0. Whenever the algorithm identifies a connection c as reachable, it sets the flag of c's subsequent connection c_{next} to 1. Note that for networks with $\tau_{\text{ch}}(\cdot) = 0$, trip tracking can be disabled and testing reachability simplifies to $\tau(p_{\text{dep}}(c)) \leq \tau_{\text{dep}}(c)$. To handle footpaths, each time the algorithm relaxes a connection c, it scans all outgoing footpaths of $p_{\text{arr}}(c)$.

Improvements. Clearly, connections departing before time τ can never be reached and need not be scanned. We do a binary search on C to identify the first relevant connection and start scanning from there (*start criterion*). If we are only interested in *one-to-one queries*, the algorithm may stop as soon as it scans a connection whose departure time exceeds the target stop's earliest arrival time. Also, as soon as one connection of a trip is reachable, so are all subsequent connections of the same trip (and preceding connections of the trip have already been scanned). We may, therefore, keep a flag (indicating reachability) per trip (instead of per connection). The algorithm then operates on these *trip flags* instead. Note that we store all data sequentially in memory, making the scan extremely cache-efficient. Only accesses to stop labels and trip flags are potentially costly, but the number of stops and trips is small in comparison. To further improve spatial locality, we subtract from each connection $c \in C$ the minimum change time of $p_{\text{dep}}(c)$ from $\tau_{\text{dep}}(c)$, but keep the original ordering of C. Hence, CSA requires random access only on small parts of its data, which mostly fits in low-level cache.

4 Extensions

CSA can be extended to profile queries. Given the timetable and a source stop p_s, a profile query computes for every stop p the set of all earliest arrival journeys to p for every departure from p_s, discarding dominated journeys. Such queries are useful for preprocessing techniques, but also for users with flexible departure (or arrival) time. We refer to the solution as a Pareto set of $(\tau_{\text{dep}}(p_s), \tau_{\text{arr}}(p_t))$ pairs.

In the following, we describe the *reverse* p–p_t-profile query, which is needed in Section 5. The forward search works analogously. Our algorithm, pCSA (p for profile), scans once over the array of connections sorted by *decreasing* departure time. For every stop it keeps a partial (tentative) profile. It maintains the property that the partial profiles are correct wrt. the subset of already scanned connections. Every stop is initialized with an empty profile, except p_t, which is set to a constant identity-profile. When scanning a connection c, pCSA *evaluates* the partial profile at the arrival stop $p_{\text{arr}}(c)$: It asks for the earliest arrival time τ^* at p_t over

all journeys departing at $p_{\text{arr}}(c)$ at $\tau_{\text{arr}}(c)$ or later. It then *updates* the profile at $p_{\text{dep}}(c)$ by potentially adding the pair $(\tau_{\text{dep}}(c), \tau^*)$ to it, discarding newly dominated pairs, if necessary.

Maintaining Profiles. We describe two variants of maintaining profiles. The first, pCSA-P (P for Pareto), stores them as arrays of Pareto-optimal (τ_{dep}, τ_{arr}) pairs ordered by decreasing arrival (departure) time. Since new candidate entries are generated in order of decreasing departure time, profile updates are a constant-time operation: A candidate entry is either dominated by the last entry or is appended to the array. Profile evaluation is implemented as a linear scan over the array. This is quick in practice, since, compared to the timetable's period, connections usually have a short duration. The identity profile of p_t is handled as a special case. By slightly modifying the data structure, we obtain pCSA-C (C for constant), for which evaluation is also possible in constant time: When updating a profile, pCSA may append a candidate entry, even if it is dominated. To ensure correctness, we set the candidate's arrival time τ^* to that of the dominating entry. We then observe that, independent of the input's source or target stop, profile entries are always generated in the same order. Moreover, each connection is associated with only two such entries, one at its departure stop, relevant for updating, and, one at its arrival stop, relevant for evaluation. For each connection, we precompute *profile indices* pointing to these two entries, keeping them with the connection. Furthermore, its associated departure time and stop may be dropped. Note that the space consumption for keeping all (even suboptimal) profile entries is bounded by the number of connections. Following [6], we also collect—in a quick preprocessing step—at each stop all arrival times (in decreasing order). Then, instead of storing arrival times in the profile entries, we keep *arrival time indices*. For our scenarios, these can be encoded using 16 (or fewer) bits. We call this technique *time indexing*, and the corresponding algorithm pCSA-CT.

Minimum Change Times and Footpaths. We incorporate minimum change times by evaluating the profile at a stop p for time τ at $\tau + \tau_{\text{ch}}(p)$. The trip bit is replaced by a trip arrival time, which represents the earliest arrival time at p_t when continuing with the trip. When scanning a connection c, we take the minimum of the trip arrival time and the evaluated profile at $p_{\text{arr}}(c)$. We update the trip arrival time and the profile at $p_{\text{dep}}(c)$, accordingly. *Footpaths* are handled as follows. Whenever a connection c is relaxed, we scan all incoming footpaths at $p_{\text{dep}}(c)$. However, this no longer guarantees that profile entries are generated by decreasing departure time, making profile updates a non-constant operation for pCSA-P. Also, we can no longer precompute profile indices for pCSA-C. Therefore, we expand footpaths into *pseudoconnections* in our data, as follows. If p_a and p_b are connected by a footpath, we look at all reachable (via the footpath) pairs of incoming connections c_{in} at p_a and outgoing connections c_{out} at p_b. We create a new pseudoconnection (from p_a to p_b, departure time $\tau_{\text{arr}}(c_{\text{in}})$, and arrival time $\tau_{\text{dep}}(c_{\text{out}})$) iff there is no other pseudoconnection with a later or equal departure time and an earlier or equal arrival time. Pseudoconnections can be identified by a simultaneous sweep over the incoming/outgoing connections

of p_a and p_b. During query, we handle footpaths toward p_t as a special case of the evaluation procedure. Footpaths at p_s are handled by merging the profiles of stops that are reachable by foot from p_s.

One-to-One Queries. So far we described *all-to-one profile queries*, i. e., from all stops to the target stop p_t. If only the *one-to-one profile* between stops p_s and p_t is of interest, a well-known pruning rule [6, 14] can be applied to pCSA-P: Before inserting a new profile entry at any stop, we check whether it is dominated by the last entry in the profile at p_s. If so, the current connection cannot possibly be extended to a Pareto-optimal solution at the source, and, hence, can be pruned. However, we still have to continue scanning the full connection array.

Multi-Criteria. CSA can be extended to compute *multi-criteria* profiles, optimizing triples $(\tau_{\text{dep}}(p_s), \tau_{\text{arr}}(p_t), \#\, t)$ of departure time, arrival time and number of taken trips. We call this variant mcpCSA-CT. We organize these triples hierarchically by mapping arrival time $\tau_{\text{arr}}(p_t)$ onto *bags* of $(\tau_{\text{dep}}(p_s), \#\, t)$ pairs. Thus, we follow the general approach of pCSA-CT, but now maintain profiles as $(\tau_{\text{arr}}(p_t)$, bag) pairs. Evaluating a profile, thus, returns a bag. Where pCSA-CT computes the minimum of two departure times, mcpCSA-CT *merges* two bags, i. e., it computes their union and removes dominated entries. When it scans a connection c, $\#\, t$ is increased by one for each entry of the evaluated bag, unless c is a pseudoconncetion. It then merges the result with the bag of trip $t(c)$, and updates the profile at $p_{\text{dep}}(c)$, accordingly. Exploiting that, in practice, $\#\, t$ only takes small integral values, we store bags as fixed-length vectors using $\#\, t$ as index and departure times as values. Merging bags then corresponds to a component-wise minimum, and increasing $\#\, t$ to shifting the vector's values. A variant, mcpCSA-CT-SSE, uses SIMD-instructions for these operations.

5 Minimum Expected Arrival Time

In this section we aim to provide delay-*robust* journeys that offer sensible backup alternatives at every stop for the case that transfers fail. A tempting approach might be to optimize *reliability*, introduced in [9], possibly together with other criteria. While this produces journeys that have low failure probabilities on their transfers, they are not necessarily robust in our sense: The set of reliable journeys may already diverge at the source stop, and in general, no fall-back alternatives are guaranteed at intermediate stops. On the other hand, on high-frequency urban routes (such as subways) an unreliable transfer might not be a problem, if the next feasible trip is just a few minutes away. To ensure that the user is never left without guidance, we compute a *subset* of connections (rather than journeys) such that at any point along the way, the user is provided with a good (in terms of arrival time) option for continuing his journey toward the destination. We propose to minimize the *expected arrival time* to achieve these goals.

We assume the following simple delay model: A connection c arrives at a random time $\tau^R_{\text{arr}}(c)$ but departs on time at $\tau_{\text{dep}}(c)$. All random arrival times are

independent. No connection arrives earlier than its scheduled arrival time $\tau_{\text{arr}}(c)$. To make computations meaningful, we assume an upper bound on all $\tau^R_{\text{arr}}(c)$. We further assume that walking is exact. Note that more complex stochastic models have been considered in [5, 11], containing dependent random variables to model delays. In this case, however, such models also propagate data errors (besides delays), therefore, requiring precise delay data [5], which is hard to obtain in practice. Also, even basic operations in [11] have super-quadratic running time (in the number of connections), making the approach impractical, already for medium-sized timetables.

For a given target stop p_t, we define for every subset S of connections of the timetable and for every connection c the *expected arrival time* $\hat{\tau}(S, c)$ at p_t, recursively. Let $c_1 \dots c_n \subseteq S$ be the connections that the user can transfer to at c's arrival stop $p_{\text{arr}}(c)$, ordered by departure time $\tau_{\text{dep}}(c_i)$ (adjusted for footpaths and minimum change times). We define

$$\hat{\tau}(S, c) = \min \left\{ \hat{\tau}(S, c_{\text{next}}), \sum_{i=1}^{n+1} P\left[\tau_{\text{dep}}(c_{i-1}) \leq \tau^R_{\text{arr}}(c) < \tau_{\text{dep}}(c_i)\right] \cdot \hat{\tau}(S, c_i) \right\}$$

where $\tau_{\text{dep}}(c_0) = \tau_{\text{arr}}(c)$, $\tau_{\text{dep}}(c_{n+1}) = \infty$, $\hat{\tau}(S, c_{n+1}) = \infty$, and $\hat{\tau}(S, c_{\text{next}}) = \infty$ if c is the last connection of trip $t(c)$. The base of the recursion is defined by the connections c arriving at p_t, for which we define $\hat{\tau}(S, c) = E[\tau^R_{\text{arr}}(c)]$. If the possibility of the user not reaching the target is non-zero, the expected arrival time is trivially ∞. Since a connection is assumed to never arrive early, $\hat{\tau}(S, c)$ only depends on connections departing later than c, which guarantees termination. (This is where we require aperiodicity; in periodic networks infinite recursions may occur.) In short, we compute the average over the expected arrival times of each outgoing connection from the stop $p_{\text{arr}}(c)$, weighted by the probability of the user catching it. We define the *minimum expected arrival time* $\hat{\tau}^*(c)$ of a connection c as the minimum $\hat{\tau}(S, c)$ over all subsets S. A subset S^* minimizes $\hat{\tau}^*(c)$, if for every stop p the set of pair $(\tau_{\text{dep}}(c), \hat{\tau}(S^*, c))$ induced by those $c \in S^*$ that depart at p, does not include dominated connections. (A pair is dominated, if, wrt. another pair, it departs earlier with higher expected arrival time.) Note that removing a dominated pair's connection improves $\hat{\tau}(\cdot)$. Also, all subsets with this property have the same $\hat{\tau}(\cdot)$ and therefore S^* is globally optimal. At least one subset S^* exists that is optimal for every c, because removing dominated connections is independent of c.

To solve the *minimum expected arrival time problem* (MEAT), we compute a set S^*, and output the reachable connections for the desired source stop and departure time. Our algorithm is based directly on pCSA-P, with a different meaning for its stop labels: Instead of mapping a departure time τ_{dep} to the corresponding earliest arrival time τ_{arr} at p_t, the algorithm now maps τ_{dep} to the corresponding minimum expected arrival time $\hat{\tau}^*$ at p_t. It does so by maintaining an array of nondominated $(\tau_{\text{dep}},\ \hat{\tau}^*)$ pairs. For a connection c, the label at stop $p_{\text{arr}}(c)$ is evaluated by a linear scan over that array: Following from the recursive definition above, the minimum expected arrival time $\hat{\tau}^*(c)$ is computed by a weighted summation of each of the expected arrival times $\hat{\tau}^*$ collected during

Table 1. Size figures for our timetables including figures of the time-dependent (TD), colored time-dependent (TD-col), and time-expanded (TE) graph models [6, 14, 15].

Figures	London		Germany		Europe	
Stops	20 843		6 822		30 517	
Trips	125 537		94 858		463 887	
Connections	4 850 431		976 678		4 654 812	
Routes	2 135		9 055		42 547	
Footpaths	45 652		0		0	
Expanded Footpaths	8 436 763		0		0	
TD Vertices (Arcs)	97 k	(272 k)	114 k	(314 k)	527 k	(1 448 k)
TD-col Vertices (Arcs)	21 k	(71 k)	20 k	(86 k)	79 k	(339 k)
TE Vertices (Arcs)	9 338 k	(34 990 k)	1 809 k	(3 652 k)	8 778 k	(17 557 k)

this scan multiplied with the success probability of the corresponding transfer at $p_{\text{arr}}(c)$. An optimization, called *earliest arrival pruning*, first runs an earliest arrival query from the source stop and then only processes connections marked reachable during that query. Note that, since during evaluation we scan over several outgoing connections, pCSA-C is not applicable.

6 Experiments

We ran experiments pinned to one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. We compiled our C++ code using g++ 4.7.1 with flags `-O3 -mavx`.

We consider three realistic inputs whose sizes are reported in Table 1. They are also used in [6, 10, 7], but we additionally filter them for (obvious) errors, such as duplicated trips and connections with non-positive travel time. Our main instance, London, is available at [13]. It includes tube (subway), bus, tram, Dockland Light Rail (DLR) and is our only instance that also includes footpaths. However, it has no minimum change times. The German and European networks were kindly provided by HaCon [12]. Both have minimum change times. The German network contains long-distance, regional, and commuter trains operated by Deutsche Bahn during the winter schedule of 2001/02. The European network contains long-distance trains, and is based on the winter schedule of 1996/97. To account for overnight trains and long journeys, our (aperiodic) timetables cover one (London), two (Germany), and three (Europe) consecutive days.

We ran for every experiment 10 000 queries with source and target stops chosen uniformly at random. Departure times are chosen at random between 0:00 and 24:00 (of the first day). We report the running time and the number of label comparisons, counting an SSE operation as a single comparison. Note that we disregard comparisons in the priority queue implementation.

Earliest Arrival. In Table 2, we report performance figures for several algorithms on the London instance. Besides CSA, we ran realistic time-expanded

Table 2. Figures for the earliest arrival problem on our London instance. Indicators are: • enabled, ○ disabled, – not applicable. "Sta." refers to the start criterion. "Trp." indicates the method of trip tracking: connection flag (○), trip flag (•), none (×). "One." indicates one-to-one queries by either using the stop criterion or pruning.

Alg.	Sta.	Trp.	One.	# Scanned Arcs/Con.	# Reachable Arcs/Con.	# Relaxed Arcs/Con.	# Scanned Footpaths	# L.Cmp. p. Stop	Time [ms]
TE	–	–	○	20 370 117	—	5 739 046	—	977.3	876.2
TD	–	–	○	262 080	—	115 588	—	11.9	18.9
TD-col	–	–	○	68 183	—	21 294	—	3.2	7.3
CSA	○	○	○	4 850 431	2 576 355	11 090	11 500	356.9	16.8
CSA	•	○	○	2 908 731	2 576 355	11 090	11 500	279.7	12.4
CSA	•	•	○	2 908 731	2 576 355	11 090	11 500	279.7	9.7
TE	–	–	•	1 391 761	—	385 641	—	66.8	64.4
TD	–	–	•	158 840	—	68 038	—	7.2	10.9
TD-col	–	–	•	43 238	—	11 602	—	2.1	4.1
CSA	•	•	•	420 263	126 983	5 574	7 005	26.6	2.0
CSA	•	×	•	420 263	126 983	5 574	7 005	26.6	1.8

Dijkstra (TE) with two vertices per connection [15] and footpaths [14], realistic time-dependent Dijkstra (TD), and time-dependent Dijkstra using the optimized coloring model [6] (TD-col). For CSA, we distinguish between scanned, reachable and relaxed connections. Algorithms in Table 2 are grouped into blocks.

The first considers one-to-all queries, and we see that basic CSA scans *all* connections (4.8 M), only half of which are reachable. On the other hand, TE scans about half of the graph's arcs (20 M). Still, this is a factor of four more entities due to the modeling overhead of the time-expanded graph. Regarding query time, CSA greatly benefits from its simple data structures and lack of priority queue: It is a factor of 52 faster than TE. Enabling the start criterion reduces the number of scanned connections by 40 %, which also helps query time. Using trip bits increases spatial locality and further reduces query time to 9.7 ms. We observe that just a small fraction of scanned arcs/connections actually improve stop labels. Only then CSA must consider footpaths. The second block considers one-to-one queries. Here, the number of connections scanned by CSA is significantly smaller; journeys in London rarely have long travel times. Since our London instance does not have minimum change times, we may remove trip tracking from the algorithm entirely. This yields the best query time of 1.8 ms on average. Although CSA compares significantly more labels, it outperforms Dijkstra in almost all cases (also see Table 4 for other inputs). Only for one-to-all queries on London TD-col is slightly faster than CSA.

Profile and Multi-Criteria Queries. In Table 3 we report experiments for (multi-criteria) profile queries on London. Other instances are available in Table 4. We compare CSA to SPCS-col [6] (an extension of TD-col to profile queries) and rRAPTOR [7] (an extension of RAPTOR to multi-criteria profile queries).

Table 3. Figures for the (multi-criteria) profile problem on London. "# Tr." is the max. number of trips considered. "Arr." indicates minimizing arrival time, "Tran." transfers. "Prof." indicates profile queries. "# Jn." is the number of Pareto-optimal journeys.

Algorithm	# Tr.	Arr.	Tran.	Prof.	One.	# Jn.	# L.Cmp. p. Stop	Time [ms]
SPCS-col	–	●	○	●	○	98.2	477.7	1 262
SPCS-col	–	●	○	●	●	98.2	372.5	843
pCSA-P	–	●	○	●	○	98.2	567.6	177
pCSA-P	–	●	○	●	●	98.2	436.9	161
pCSA-C	–	●	○	●	–	98.2	1 912.5	134
pCSA-CT	–	●	○	●	–	98.2	1 912.5	104
rRAPTOR	8	●	●	●	○	203.4	1 812.5	1 179
rRAPTOR	8	●	●	●	●	203.4	1 579.6	878
rRAPTOR	16	●	●	●	●	206.4	1 634.0	922
mcpCSA-CT	8	●	●	●	–	203.4	15 299.8	255
mcpCSA-CT-SSE	8	●	●	●	–	203.4	1 912.5	221
mcpCSA-CT-SSE	16	●	●	●	–	206.4	3 824.9	466

Note that in [7] rRAPTOR is evaluated on two-hours range queries, whereas we compute full profile queries. A first observation is that, regarding query time, one-to-all SPCS is outperformed by all other algorithms, even those which additionally minimize the number of transfers. Similarly to our previous experiment, CSA generally does more work than the competing algorithms, but is, again, faster due to its cache-friendlier memory access patterns. We also observe that one-to-all pCSA-C is slightly faster than pCSA-P, even with target pruning enabled, although it scans 2.7 times as many connections because of expanded footpaths. Note, however, that the figure for pCSA-C does not include the post-processing that removes dominated journeys. Time indexing further accelerates pCSA-C, indicating that the algorithm is, indeed, memory-bound. Regarding multi-criteria profile queries, doubling the number of considered trips also doubles both CSA's label comparisons and its running time. For rRAPTOR the difference is less (only 12 %)—most work is spent in the first eight rounds. Indeed, journeys with more than eight trips are very rare. This justifies mcpCSA-CT-SSE with eight trips, which is our fastest algorithm (221 ms on average). Note that using an AVX2 processor (announced for June 2013), one will be able to process 256 bit-vectors in a single instruction. We, therefore, expect mcpCSA-CT-SSE to perform better for greater numbers of trips in the future.

Minimum Expected Arrival Time. In Table 5 we present figures for the MEAT problem on all instances. Besides running time, we also report output complexity in terms of number of stops and arcs of the decision graph (see Fig. 1 for an example). Real world delay data was not available to us. Hence, we follow Disser et al. [9] and assume that the probability of a train being delayed by t minutes (or less) is $0.99 - 0.4 \cdot \exp(-t/8)$. After 30 min (10 min on London) we

Table 4. Evaluating other instances. Start criterion and trip flags are always used.

Algorithm	# Tr.	Arr.	Tran.	Prof.	One.	Germany # Jn.	Germany # L.Cmp. p. Stop	Germany Time [ms]	Europe # Jn.	Europe # L.Cmp. p. Stop	Europe Time [ms]
TE	–	●	○	○	○	1.0	317.0	117.1	0.9	288.6	624.1
TD-col	–	●	○	○	○	1.0	11.9	3.5	0.9	10.0	21.6
CSA	–	●	○	○	○	1.0	228.7	3.4	0.9	209.5	19.5
TE	–	●	○	○	●	1.0	29.8	11.7	0.9	56.3	129.9
TD-col	–	●	○	○	●	1.0	6.8	2.0	0.9	5.3	11.5
CSA	–	●	○	○	●	1.0	40.8	0.8	0.9	74.2	8.3
pCSA-CT	–	●	○	●	–	20.2	429.5	4.9	11.4	457.6	46.2
rRAPTOR	8	●	●	●	○	29.4	752.1	161.3	17.2	377.5	421.8
rRAPTOR	8	●	●	●	●	29.4	640.1	123.0	17.2	340.8	344.9
mcpCSA-CT-SSE	8	●	●	●	–	29.4	429.5	17.9	17.2	457.6	98.2

Table 5. Evaluating pCSA-P for the MEAT problem on all instances.

Network	Max. Delay [min]	Decision # Stops	Graph # Arcs	All-To-One Time [ms]	One-To-One Time [ms]	One-To-One Dis. Time [ms]
Germany	30	8	19	68.1	31.0	24.6
Europe	30	20	46	205.0	169.0	112.0
London	10	2 724	30 243	668.0	491.0	272.0

set this value to 1. Moreover, we also evaluate performance when discretizing the probability function at 60 equidistant points [9]. We run pCSA-P on 10 000 random queries and evaluate both the all-to-one and one-to-one (with earliest arrival pruning enabled) setting. Regarding output complexity, on the German and European networks the resulting decision graphs are sufficiently small to be presented to the user. They consist of 8 stops and 19 arcs on average (Germany), roughly doubling on Europe. However, for London these figures are impractically large, increasing to 2 724 (stops) and 30 243 (arcs). Note that in a dense metropolitan network (such as London), trips operate much more frequently, therefore, many more alternate (and fall-back) journeys exist. These must all be captured by the output. Regarding query time, pCSA-P computes solutions in under 205 ms on Germany and Europe for all scenarios. On London, all-to-one queries take 668 ms, whereas one-to-one queries can be computed in 272 ms time. Note that all values are still practical for interactive scenarios.

7 Final Remarks

In this work, we introduced the Connection Scan framework of algorithms (CSA) for several public transit route planning problems. One of its strengths is the conceptual simplicity, allowing easy implementations. Yet, it is sufficiently flexible

to handle complex scenarios, such as multi-criteria profile queries. Moreover, we introduced the MEAT problem which considers stochastic delays and asks for a robust set of journeys minimizing (in its entirety) the user's expected arrival time. We extended CSA to MEAT queries in a sound manner. Our experiments on the metropolitan network of London revealed that CSA is faster than existing approaches, and computes solutions to the MEAT problem surprisingly fast in 272 ms time. All scenarios considered are fast enough for interactive applications. For future work, we are interested in investigating network decomposition techniques to make CSA more scalable, as well as more realistic delay models. Also, since CSA does not use a priority queue, parallel extensions seem promising. Regarding multimodal scenarios, we like to combine CSA with existing techniques developed for road networks.

References

1. H. Bast. Car or Public Transport – Two Worlds. In S. Albers, H. Alt, and S. Näher, editors, *Efficient Algorithms*, LNCS 5760, pp. 355–367. Springer, 2009.
2. H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger. Fast Routing in Very Large Public Transportation Networks using Transfer Patterns. In *ESA*, LNCS 6346, pp. 290–301. Springer, 2010.
3. R. Bauer, D. Delling, and D. Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 57(1):38–52, 2011.
4. A. Berger, D. Delling, A. Gebhardt, and M. Müller–Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *ATMOS*, OpenAccess Series in Informatics (OASIcs), 2009.
5. A. Berger, A. Gebhardt, M. Müller–Hannemann, and M. Ostrowski. Stochastic Delay Prediction in Large Train Networks. In *ATMOS*, pp. 100–111. 2011.
6. D. Delling, B. Katz, and T. Pajor. Parallel Computation of Best Connections in Public Transportation Networks. *ACM JEA*, 2012. To appear.
7. D. Delling, T. Pajor, and R. F. Werneck. Round-Based Public Transit Routing. In *ALENEX*, pp. 130–140. SIAM, 2012.
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. Y. Disser, M. Müller–Hannemann, and M. Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In *WEA*, LNCS 5038, 347–361. Springer, 2008.
10. R. Geisberger. Contraction of Timetable Networks with Realistic Transfers. In *SEA*, LNCS 6049. Springer, May 2010.
11. M. Goerigk, M. Knoth, M. Müller–Hannemann, M. Schmidt, and A. Schöbel. The Price of Robustness in Timetable Information. In *ATMOS*, pp. 76–87. 2011.
12. HaCon website, 2013. http://www.hacon.de/hafas/.
13. London Data Store. http://data.london.gov.uk.
14. M. Müller–Hannemann, F. Schulz, D. Wagner, and C. Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization*, LNCS 4359, pp. 67–90. Springer, 2007.
15. E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM JEA*, 12(2.4):1–39, 2008.
16. C. Sommer. Shortest-Path Queries in Static Networks, 2012. Submitted. Preprint available at http://www.sommer.jp/spq-survey.htm.