Trip-Based Public Transit Routing Using Condensed Search Trees

Sascha Witt
sascha.witt@kit.edu

Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

We study the problem of planning Pareto-optimal journeys in public transit networks. Most existing algorithms and speed-up techniques work by computing subjourneys to intermediary stops until the destination is reached. In contrast, the trip-based model [26] focuses on trips and transfers between them, constructing journeys as a sequence of trips. In this paper, we develop a speed-up technique for this model inspired by principles behind existing state-of-the-art speed-up techniques, Transfer Patterns [1] and Hub Labelling [9]. The resulting algorithm allows us to compute Pareto-optimal (with respect to arrival time and number of transfers) 24-hour profiles on very large real-world networks in less than half a millisecond. Compared to the current state of the art for bicriteria queries on public transit networks, this is up to two orders of magnitude faster, while increasing preprocessing overhead by at most one order of magnitude.

1 Introduction

Finding optimal journeys in public transit networks is a complex problem. Efficient algorithms are required to allow real-time answering of queries by users in online systems such as Google Maps Transit[1] or those of local providers such as `bahn.de` or `fahrplan.sbb.ch`. In these systems, users enter a source location, a destination, and a rough point in time and expect a number of journeys that are optimal in some sense.

Precisely what constitutes an optimal journey is non-trivial to define, as it often depends on individual user preferences. Generally, passengers want to arrive as quickly as possible, so travel time should usually be minimized. However, some users may prefer a slightly longer

[1] `https://maps.google.com/transit`

journey with fewer transfers between different vehicles, as transfers reduce travel comfort and introduce additional uncertainty — connecting trains might be missed due to delays. How much extra travel time someone is willing to accept in exchange for fewer transfers differs from user to user and might depend on several factors, such as arrival time or even purpose of the journey.

Since no system can capture all these variables to compute the optimal journey for each query, we usually compute a set of possible journeys and let the user choose among them, possibly after applying some filtering [11,18]. A general approach to this is to define a number of criteria, such as arrival time and number of transfers, and compute a set of Pareto-optimal journeys, i.e. a set such that no journey is better than any other in all criteria.

1.1 Related Work

In the past, several algorithms based on different principles have been proposed. For an extensive survey, please refer to Bast et al. [2]. Pyrga et al. [23] reduce the problem of finding optimal journeys in public transit networks to finding shortest paths in graphs. They propose the time-extended and time-dependent model along with some speed-up techniques such as goal directed search, and optimize both travel time and number of transfers in the Pareto sense. Geisberger [20] applies the concept of contraction hierarchies, which have proved successful on road networks, to public transit networks. Only travel time is optimized. Berger et al. [5] introduce SUBITO and k-flags, two speed-up techniques that optimize both travel time and number of transfers in the Pareto sense.

RAPTOR [14] foregoes modelling the data as a graph and instead operates directly on the timetable data. In addition to travel time and number of transfers, they also consider price as a criteria. The Connection Scan Algorithm (CSA) [15] also eschews graphs and instead works on an ordered array of connections to find Pareto-optimal journeys with respect to travel time and number of transfers. Accelerated CSA [24] is a speed-up technique for CSA that works via partitioning of the network. Unlike the original CSA, it was only evaluated as a single-criterion algorithm, using the number of transfers as a tiebreaker between journeys with identical arrival time.

Public Transit Labelling (PTL) [12] uses, as the name implies, a hub labelling approach. It requires extensive preprocessing and produces a very large amount of auxiliary data, but leads to very low query times, even for multi-criteria queries. Timetable Labelling (TTL) [25] is another labelling-based approach, which has been extended in the context of databases by Efentakis [17]. However, TTL only performs single-criterion queries regarding arrival time.

Transfer Patterns (TP) [1,3,4] is a speed-up technique that precomputes the eponymous transfer patterns between all stops in the network. These transfer patterns are formed by the sequence of stops where passengers transfer between vehicles. At query time, these patterns are then used to quickly find all Pareto-optimal journeys.

1.2 Our Contribution

In this work, we present a speed-up technique based on Trip-Based Public Transit Routing (TB) [26]. Unlike other approaches, TB conceptually works on a graph where nodes represent trips, not stops. Edges represent possible transfers between trips, and are qualified using the indices of the stops where passengers exit or board a trip. These transfers are precomputed and can be looked up quickly during query processing. This has the advantage that minimum change times and footpaths do not have to be evaluated at query time, and allows fine-grained modelling without query-time overhead.

Inspired by the principles behind Transfer Patterns [1] and Hub Labelling [9], our speed-up technique achieves sub-millisecond query times for profile queries on country-sized networks, while keeping preprocessing overhead low.

2 Preliminaries

A public transit network is defined by an aperiodic *timetable*, which contains a set of stops, a set of footpaths, and a set of trips. A *stop* is a physical location where passengers can enter or exit a vehicle, such as a bus or train. Depending on the granularity of the model, a stop may represent an entire train station, a single platform, or some subset of all platforms within a train station. Transferring from one vehicle to another at the same stop s may require a certain amount of time, which we call minimum change time $\Delta\tau_{\text{ch}}(s)$. If the time between the arrival of the previous vehicle and the departure of the subsequent one is less than $\Delta\tau_{\text{ch}}(s)$, no transfer between them is possible at this station. *Footpaths* connect two stops and indicate the time required to walk from one to the other. We use the most general model of directed, non-transitive footpaths. We denote the time required to walk from stop s_1 to s_2 as $\Delta\tau_{\text{fp}}(s_1, s_2)$, with $\Delta\tau_{\text{fp}}(s_1, s_2) := \infty$ if no footpath from s_1 to s_2 exists.

Trips represent vehicles. Each trip t travels along a sequence of stops $\vec{s}(t) = \langle t@1, \ldots, t@n \rangle$. A trip may visit a stop multiple times. For each $t@i$, the timetable contains the arrival time time $\tau_{\text{arr}}(t@i)$ and the departure time $\tau_{\text{dep}}(t@i)$ of trip t at that stop index. Trips that travel along the same sequence of stops are grouped into *lines*. We require that trips never overtake another trip of the same line; more specifically, we require that the trips of a line can be totally ordered with respect to

$$t_1 \preceq t_2 \iff \forall i \in [1, |\vec{s}(t_1)|] : \tau_{\text{arr}}(t_1@i) \leq \tau_{\text{arr}}(t_2@i). \tag{1}$$

Trips that violate this requirement are assigned to different lines. We denote the line of a trip t as $L(t)$, and define $\vec{s}(L(t)) := \vec{s}(t)$.

Transfers indicate connections between trips. We denote transfers as $t_1@e \to t_2@b$, meaning that passengers can exit trip t_1 at stop index e in order to board trip t_2 at stop index b. Transfers may occur at a single stop, in which case

$$\tau_{\text{arr}}(t_1@e) + \Delta\tau_{\text{ch}}(t_1@e) \leq \tau_{\text{dep}}(t_2@b) \tag{2}$$

must hold, or they may involve a footpath, in which case the requirement is

$$\tau_{\text{arr}}(t_1@e) + \Delta\tau_{\text{fp}}(t_1@e, t_2@b) \leq \tau_{\text{dep}}(t_2@b). \tag{3}$$

A *journey* describes how and when to get from a source stop s_{src} to a destination stop s_{dest}. It can be defined by a sequence $\langle t_1@b_1, t_2@b_2, \ldots, t_n@b_n\rangle$ with the following requirements:

$$s_{\mathrm{src}} = t_1@b_1 \vee \Delta\tau_{\mathrm{fp}}(s_{\mathrm{src}}, t_1@b_1) < \infty \tag{4}$$
$$\forall i \in [1,n) : \exists e > b_i : t_i@e \to t_{i+1}@b_{i+1} \tag{5}$$
$$\exists i : t_n@i = s_{\mathrm{dest}} \vee \Delta\tau_{\mathrm{fp}}(t_n@i, s_{\mathrm{dest}}) < \infty. \tag{6}$$

These requirements ensure that the first trip can be reached (4), that transfers are possible between subsequent trips (5), and that the final trip arrives at the destination (6).

We consider two well-known bicriteria problems, optimizing arrival time and number of transfers required. It has been shown [22] that for these criteria, computing the full set of Pareto-optimal journeys is feasible. A journey is Pareto-optimal if no other journey dominating it exists. A journey dominates another if it is better or equal in all criteria — if they are equal in all criteria, we break the tie arbitrarily and keep only one of them in the set.

The input to the *earliest arrival query* consists of a source stop s_{src}, a destination stop s_{dest}, and a departure time τ. The result is a set of tuples $(\tau_{\mathrm{dest}}, n)$, one for each Pareto-optimal journey leaving s_{src} no earlier than τ and arriving at s_{dest} at time τ_{dest} after n transfers. For the *profile query*, we are given a source stop s_{src}, a destination stop s_{dest}, an earliest departure time τ_{edt}, and a latest departure time τ_{ldt}. Here, we consider the departure time of journeys as an additional criterion, with later departures dominating earlier ones. Thus, we compute all Pareto-optimal journeys departing at s_{src} at some time τ_{src} with $\tau_{\mathrm{edt}} \leq \tau_{\mathrm{src}} \leq \tau_{\mathrm{ldt}}$ and arriving, after n transfers, at s_{dest} at time τ_{dest}. The answer to the query is then the set of tuples $(\tau_{\mathrm{src}}, \tau_{\mathrm{dest}}, n)$ corresponding to these journeys.

3 Trip-Based Public Transit Routing

This section provides a quick explanation of the Trip-Based Public Routing (TB) algorithm [26]. For more details, please refer to the original publication.

3.1 Preprocessing

As mentioned in section 1.2, TB uses trips and transfers between them as its basic building blocks. During a short preprocessing phase, all possible transfers between trips are computed. However, it can be shown that many of these transfers can never be part of an optimal journey, for example transfers that lead to trips that run in the opposite direction, or transfers to several trips of the same line. Therefore, the second step of preprocessing is discarding these superfluous transfers, which may constitute up to 90% of total transfers. Since each trip can be processed separately, preprocessing is trivially parallelized and can be performed within minutes even for very large networks.

3.2 Queries

Queries are similar to a breadth-first search on the graph formed by trips and the transfers between them. For an earliest arrival query, we first identify the trips reachable from the source stop, and insert them into a queue. Then, each trip is processed by scanning its outgoing transfers. Newly reached trips are in turn added to the queue. Trips are marked as reached by, conceptually, assigning labels consisting of trip, stop index, and number of transfers needed to reach that trip to lines.[2] Branches of the search are pruned if they are dominated by existing labels. The graph is explored until all Pareto-optimal journeys to the destination stop are found.

For a profile query, we essentially repeat the above multiple times. Observe that the departure time is an additional criteria for journeys in profile queries, with later journeys dominating earlier ones. If all other criteria are equal, the journey with the later departure has less travel time and is therefore preferable. Thus, we start with the latest possible departure at the source stop, and perform an earliest arrival query. We can add the resulting journeys to the result set. Then, without resetting labels, we perform an earliest arrival query for the second-latest departure, and so on. By preserving the labels between runs, we allow later journeys to dominate earlier ones, avoiding redundant work.

4 Storing One-to-All Search Trees

In this section, we show how some of the principles behind Transfer Patterns [1] can be applied to the Trip-Based model. The core idea of the Transfers Patterns algorithm is to precompute, for all pairs of source and destination stop, the *transfer patterns* for all optimal journeys. The transfer pattern of a journey is the sequence of stops where a change of vehicle occurs. In practice, optimal journeys between two given stops share a limited number of transfer patterns. If all optimal transfer patterns between source and destination are known, queries can be answered quickly by looking up direct connections between the stops forming the transfer patterns.

We use the same property as the foundation for our speed-up technique. Since we operate on trips — or more generally, lines, which are ordered sets of trips — we do not precompute sequences of stops where transfers occur. Instead, we precompute the sequence of *lines* that correspond to an optimal journey, together with the stop indices where each of these lines is boarded. As we show in the next section, these line sequences form a natural generalization of one-to-all profile search trees.

4.1 Prefix Trees

We compute one-to-all profiles from each stop to find all potential line sequences. These one-to-all profiles are at their core identical to the one-to-one profiles described in the original publication [26] and summarized in section 3.2.

First, all departures at the source stop are ordered by departure time and then processed backwards. For each distinct departure time, we then perform a breadth-first search as described in section 3.2. This results in a breadth-first tree, with the source stop as the root node, the visited trips as internal nodes, and the reached stops as leaves. In contrast to one-to-one profiles, we also assign labels to all stops, consisting of arrival time and number

[2] In the implementation, we unroll the "trip" and "number of transfers" dimensions for faster lookup and to allow the use of SIMD instructions.

of transfers. We update these using the breadth-first tree, pruning branches that do not lead to an improved stop label. The remaining tree is generalized by replacing all trips with their respective line and the index of the stop where the trip was boarded. We then restart the search using the next (earlier) departure, preserving all labels.

Finally, the trees are merged, resulting in one *prefix tree* [10] for each source stop, containing the optimal line sequences to all destination stops. In essence, this prefix tree represents a condensed, time-independent result of a one-to-all profile search. Note that prefix trees are functionally equivalent to the transfer pattern graphs used by Transfer Patterns [1], except that internal nodes represent lines instead of stops.

4.2 Queries

We can use these prefix trees to quickly answer queries. First, we construct the *query graph*. To do so, we find the nodes corresponding to the destination stop in the prefix tree of the source stop. We follow the paths from these nodes to the root, adding edges from parent to child nodes to the query graph. Multiple occurrences of the same $L@b$ in the prefix tree are mapped onto the same node in the query graph. Again, note the similarity to the query graph used by Transfer Patterns [1].

To answer the query, we run a simple multi-criteria label-correction shortest path algorithm [16] on the query graph. Labels consist of a trip, the number of transfers, and, for profile queries, the departure time at the source stop. Finding the initial trips at the source stop is straightforward. Given a label (t, n, τ_{dep}), we relax an edge between $L_1@i$ and $L_2@j$ by finding a transfer $t@k \rightarrow s@j$ such that $k > i$ and $L(s) = L_2$. We then add a label $(s, n+1, \tau_{dep})$ to the node representing $L_2@j$. Once the algorithm terminates, we can extract the arrival times at the destination stop from the labels. Intuitively, the prefix tree tells us which paths through the networks optimal journeys can take. The query then follows these paths to find the actual journeys for the given departure time(s).

5 Splitting Trees

Unfortunately, for large networks, prefix trees grow unfeasibly large, and memory usage becomes an issue. Each tree spans the entire network, and in addition, many subtrees are duplicates of each other, with slightly different prefixes. Furthermore, subtrees are often duplicated across different trees, since stops can only be reached through a limited set of lines.

We can reduce this redundancy by removing branches from the prefix trees and instead storing them in *postfix trees*. These postfix trees are essentially reverse prefix trees: They are rooted at a *destination* stop and describe optimal line sequences for reaching that stop. Storing these sequences once for each destination stop instead of once or even multiple times for each source stop greatly improves space efficiency. Optimal line sequences can be recovered by concatenating branches of the source's prefix tree with matching branches of the destination's postfix tree.

5.1 Postfix Trees

We construct the postfix trees from the prefix trees as follows. For each path from the root (that is, the source stop) to a leaf (a destination stop), we select an internal node N_{cut} where we "cut" this path. Section 5.3 explains how this node is chosen. We add the subpath from N_{cut} (inclusive) to the leaf — in reverse order — to the postfix tree for the destination node. Then, we remove the leaf node and, recursively, all internal nodes that no longer have any children from the prefix tree, until we reach N_{cut}. Thus, if the prefix tree originally contained the path $\langle S, N_1, \ldots, N_l, N_{\text{cut}}, N_{l+1}, \ldots, N_n, T\rangle$, we end up with $\langle S, N_1 \ldots, N_l, N_{\text{cut}}\rangle$ in the prefix tree and $\langle T, N_n, \ldots, N_{l+1}, N_{\text{cut}}\rangle$ in the postfix tree.

However, recall that each internal node represents a line together with the stop index where the line is boarded, $L@b$. This breaks the symmetry between prefix and postfix trees. As a result, we end up with many postfixes that are identical except for the board index at N_{cut} — depending on the source stop, there are many ways to reach a line, but only a limited number of (optimal) ways from that line to the destination stop. We can merge these nodes by setting the index to the *exit* of the next transfer, which is identical for all of them. Note that we only do this for the N_{cut} in postfix trees, not for any other nodes in either prefix or postfix trees. Thus, if $N_{\text{cut}} \triangleq L@b$ and the original path was $\langle S, N_1, \ldots, L@b, \ldots, N_n, T\rangle$, we now have $\langle S, N_1, \ldots, L@b\rangle$ in the prefix tree for S and $\langle T, N_n, \ldots, L@e\rangle$ in the postfix tree for T, with $b < e$. This results in a greatly reduced number of leaves in postfix trees, while still allowing us to recover the original line sequence.

Since we no longer store destination stops in prefix trees (or source stops in postfix trees), but still want to preserve directional information, a bit vector is stored with each N_{cut}. We partition the stops and set the ith bit if N_{cut} connects to the postfix tree of a stop in partition i, and vice versa for the N_{cut} in postfix trees. In practice, we use 64-bit integers and simply partition the stops by ID, taking advantage of the pre-existing locality in the data sets.

5.2 Queries

The algorithm for query graph construction follows from the construction of the postfix trees. First, we take the prefix tree for the source stop and select all N_{cut} where the bit vector has the bit corresponding to the destination stop set. Similarly, we select the N'_{cut} from the postfix tree for the destination stop where the bit corresponding to the source stop is set. Then, we find all pairs $(N_{\text{cut}}, N'_{\text{cut}})$ such that $N_{\text{cut}} \triangleq L@b$ and $N'_{\text{cut}} \triangleq L@e$ with $b < e$. Each such pair defines a path $\langle S, N_1, \ldots, N_l, N_{\text{cut}}, N_{l+1}, \ldots, N_n, T\rangle$, and we need to ensure that the query graph contains all edges in that path. By ordering the nodes by their corresponding line, we can find these pairs using an algorithm similar to a coordinated sweep. Due to the generalizations performed during postfix tree construction, we will find some prefix-postfix combinations that do not correspond to an optimal line sequence. Thus, the resulting query graph will usually be larger than in section 4.2, but this only affects performance of the query, not correctness. The query algorithm itself is the same as before.

Essentially, we find optimal paths during preprocessing, split them at some intermediary node for more efficient storage, and then reassemble them at query time. Note the similarity to the concept of hub labelling [9]. In hub labelling, optimal journeys are split at some intermediary hub, then stored in compressed form at the source and destination. We do the same, except we only store the more general line sequences instead of the journeys, which we can then reconstruct at query time. Indeed, as we show in section 6, our approach shares some properties with existing labelling approaches.

The flags we use to filter possible connections are reminiscent of arc flags [21]. Without them, many long prefixes would connect to long postfixes for stops that are close together on the network, without a corresponding optimal journey. Exploring these unnecessary nodes during the query would be costly and is avoided by this pre-filtering.

5.3 Cut Selection

It is clear that the choice of N_{cut} has a large effect on the resulting trees. In general, we want smaller trees, which are more space efficient. We examined two fundamentally different strategies.

The first is to simply cut paths in half. Unsurprisingly, this results in rather large trees, since paths are cut at more or less arbitrary lines. This results in many different prefixes and postfixes for each stop, which translates to large trees.

The second strategy exploits the underlying network's structure by selecting the most "important" lines. To find these lines, we construct the *line graph* [6] of the network. In the (undirected) line graph, nodes correspond to lines, and two nodes share an edge if and only if a transfer between these lines is possible. We then use this line graph to compute the betweenness centrality [19] of each line using Brandes' algorithm [7].[3] This gives us an ordering of the lines, and when choosing N_{cut}, we select the node which corresponds to the most central line on the path. This ensures that the choice is consistent across different paths, which allows better merging of prefixes and postfixes. As we show in section 6, this strategy gives good results on country-sized networks, which typically exhibit good structure. Unfortunately, it is less successful on the less structured metropolitan networks. On these, using the simpler strategy of cutting paths into two equal halves leads to better results. Exploration of further criteria for selecting cut nodes is a subject of future research.

6 Experiments

We performed experiments using a quad 8-core Intel Xeon E5-4640 clocked at 2.4 GHz with 512 GB of DDR3-1600 RAM, using 64 threads for parallel preprocessing. Except where otherwise noted, computations are sequential. Code was written in C++ and compiled using g++ 5.2.0 with optimizations enabled. We consider five real-world data sets, three covering countries of varying size and two metropolitan networks: Germany, provided to us by Deutsche Bahn, Switzerland, available at `gtfs.geops.ch`, and Sweden, available at `trafiklab.se`, contain both long-distance and local transit, and cover two consecutive days to allow for overnight journeys. London, available at `data.london.gov.uk`, and Madrid, available at `emtmadrid.es`, cover a single day only. For Madrid, we computed footpaths using

[3]We chose this algorithm for simplicity; since the exact centrality is not required, one could also use an approximate algorithm [8] instead.

Table 1: Instances used for experiments.

Instance	Stops	Conn.	Trips	Lines	Footp.	Transfers
Germany	247.9 k	27 061 k	1 432 k	192.8 k	98.8 k	84 950 k
Sweden	50.7 k	6 054 k	261 k	17.6 k	0.8 k	16 455 k
Switzerland	27.8 k	4 650 k	611 k	14.4 k	34.3 k	12 626 k
London	20.8 k	4 991 k	129 k	2.2 k	27.6 k	15 883 k
Madrid	4.6 k	5 280 k	190 k	1.4 k	1.4 k	9 256 k

Table 2: Preprocessing figures. Listed are the average time required to compute the full prefix tree for a stop, the total time required to compute the split trees for all stops (sequential and parallel), the average number of nodes in those trees (per stop, i.e. the sum of prefix and postfix), and the total space consumption. Sequential preprocessing for the Germany instance was performed on a different machine.

Instance	p. prefix tree [ms]	seq. [h:m]	par. [h:m]	speed up	avg. # of nodes	mem. [GB]
Germany	2 143.6	(231:16)	13:48	(16.8 x)	7 305	23.2
Sweden	166.7	4:33	0:18	15.2 x	2 433	1.6
Switzerland	209.3	3:18	0:12	16.5 x	4 315	1.6
London	1 368.1	15:19	0:42	21.9 x	20 390	6.0
Madrid	497.3	1:22	0:04	17.0 x	32 293	2.0

a known heuristic [13], for all other instances, they are part of the input. These data sets are summarized in Table 1.

Preprocessing figures can be found in Table 2. Due to scheduling conflicts, sequential preprocessing of the Germany instance was performed on a different machine[4]. We report the total time required to perform the computation of prefix and postfix trees, as described in section 5. This includes the time required to compute the betweenness centrality, which is negligible in most cases. For Germany, Switzerland and Sweden we use the betweenness centrality to select cut nodes; for London and Madrid we use the simpler method of cutting paths in half. The reverse generally leads to larger trees and therefore higher memory consumption. For most instances, the difference is about 1–2 GB; for Germany, the difference is almost 50 GB. It is interesting to note that the metropolitan networks require more space than the two small country-sized networks. This indicates that the topology of the network is more important than the raw size in terms of stops or connections. A similar effect can be seen in the labelling approaches, Public Transit Labelling (PTL) [12] and Timetable Labelling (TTL) [25].

We evaluate query times in Table 3. We measured the average times for 10 000 queries with source and destination stop chosen uniformly at random. For earliest arrival queries, the departure time was chosen uniformly at random on the first day; for profile queries, the departure time range is the entire first day. We evaluated queries for three different variants:

[4]Dual 8-core Intel Xeon E5-2650s v2, 2.6 GHz, 128 GB DDR3-1600 RAM, 20 MB L3 cache

Table 3: Query figures. Listed are the query graph size (nodes + edges), the time required to construct the query graph, and the time required to perform an earliest arrival and a 24h profile query. The first block refers to the basic trip-based algorithm, where no query graph is used. The second block uses a prefix tree for each source stop, as in section 4. The third block uses the split trees for source and destination stop, as in section 5.

Instance	Var.	Query graph size [N+E]	Query graph time [μs]	EA [μs]	profile [μs]
Germany	TB	—	—	30 856	192 952
Sweden	TB	—	—	2 760	16 532
Switzerland	TB	—	—	1 780	18 104
London	TB	—	—	1 374	96 114
Madrid	TB	—	—	711	54 118
Germany	PT	$41 + 58$	994.4	63.3	155.0
Sweden	PT	$23 + 32$	24.6	40.4	88.6
Switzerland	PT	$38 + 59$	34.0	45.8	155.9
London	PT	$91 + 196$	138.2	101.1	2 786.6
Madrid	PT	$150 + 407$	306.9	81.7	6 913.8
Germany	ST	$124 + 232$	81.1	75.0	430.5
Sweden	ST	$66 + 122$	32.5	27.2	207.1
Switzerland	ST	$118 + 233$	76.1	32.7	327.6
London	ST	$331 + 1242$	1 583.3	141.4	14 545.4
Madrid	ST	$456 + 2073$	11 822.9	165.8	28 919.0

The basic trip-based algorithm (TB), using prefix trees as described in section 4 (PT), and using both prefix and postfix trees as described in section 5 (ST). The ST variant leads to larger query graphs than the PT variant. This is to be expected, as some information gets lost in the transformation, and some prefixes may connect to more postfixes than required. This does not affect correctness, because all optimal line sequences are still contained in the query graph. It does, however, lead to increased query times for ST in comparison to PT. Nevertheless, the time required to construct the query graph on the Germany instance is lower for ST, since the split trees contain fewer nodes in total than the original prefix tree. Profile query times are much higher on the metropolitan networks than on the generally larger country-sized networks. In part, this is because they are less structured than the larger networks, which leads to larger query graphs. However, on the metropolitan networks, the set of optimal journeys is also much larger than on the others, which slows down the query algorithm.

We compare variant ST, using prefix and postfix trees, to other state of the art algorithms in Table 4. Algorithms based on labelling approaches are generally the fastest. In particular, for single criterion queries, they dominate other preprocessing-based approaches with regard to query times, preprocessing time and memory consumption. PTL [12] supports multi-criteria queries, at the cost of massive increases in both preprocessing time and memory consumption,

Table 4: Comparison with the state of the art. Results taken from [2–4, 12, 24, 25]. Algorithms computing Pareto-optimal journeys with respect to the number of transfers in addition to arrival time are marked in column "tr." Profile queries are marked in column "pr."

algorithm	instance	stops [10^3]	conn. [10^6]	tr.	pr.	mem. [GB]	pre. [h]	query [µs]
CSA [24]	Germany	252.4	46.2	○	○	—	—	298.6 k
ACSA [24]	Germany	252.4	46.2	○	○	n/a	0.2	8.7 k
TP [4]	Germany	248.4	13.9	•	○	140.0	372.0	300.0
Sc-TP [3]	Germany	250.0	15.0	•	○	1.2	16.5	32.0 k
TB	Germany	247.9	27.1	•	○	23.2	231.3	156.1
TTL [25]	Sweden	51.4	n/a	○	○	≈ 0.5	0.2	≈ 10.0
PTL [12]	Sweden	51.1	12.7	•	○	12.3	36.2	27.6
TB	Sweden	50.7	6.1	•	○	1.6	3.8	59.7
PTL [12]	Switzerland	27.1	23.7	•	○	12.7	61.6	21.7
TB	Switzerland	27.8	4.7	•	○	1.6	2.7	108.8
CSA [15]	London	20.8	4.9	○	○	—	—	1.8 k
PTL [12]	London	20.8	5.1	•	○	26.2	49.3	30.0
TB	London	20.8	5.0	•	○	6.0	11.6	1.7 k
TTL [25]	Madrid	4.6	n/a	○	○	≈ 0.4	0.1	≈ 30.0
PTL [12]	Madrid	4.7	4.5	•	○	9.9	10.9	64.3
TP [2]	Madrid	4.6	4.8	•	○	n/a	185.0	3.1 k
TB	Madrid	4.6	5.3	•	○	2.0	1.1	12.0 k
ACSA [24]	Germany	252.4	46.2	○	•	n/a	0.2	171.0 k
TP [4]	Germany	248.4	13.9	•	•	140.0	372.0	5.0 k
TB	Germany	247.9	27.1	•	•	23.2	231.3	511.6
PTL [12]	Sweden	51.1	12.7	○	•	0.7	0.5	12.1
TB	Sweden	50.7	6.1	•	•	1.6	3.8	239.6
PTL [12]	Switzerland	27.1	23.7	○	•	0.7	0.7	24.5
TB	Switzerland	27.8	4.7	•	•	1.6	2.7	403.7
PTL [12]	London	20.8	5.1	○	•	1.3	0.9	74.3
CSA [15]	London	20.8	4.9	•	•	—	—	466.0 k
TB	London	20.8	5.0	•	•	6.0	11.6	16.1 k
PTL [12]	Madrid	4.7	4.5	○	•	0.4	0.4	111.9
TB	Madrid	4.6	5.3	•	•	2.0	1.1	40.7 k

while TTL [25] only performs single-criterion queries. TP [1,3,4] can answer bicriteria profile queries in a few milliseconds, even on large networks. The original TP had the drawbacks of very long preprocessing times and a large memory consumption. More recently, Scalable Transfer Pattern [3] has made impressive improvements on this front, at the cost of increased query times.

On the metropolitan networks, our algorithm performs notably worse than could be expected, although query times are still in the low milliseconds. As previously mentioned, this is mostly due to the much higher number of journeys compared to the country-sized networks. For bicriteria queries on the country-sized networks, our algorithm has preprocessing costs one order of magnitude less than PTL, while query times are similar. Note, however, that PTL has not been evaluated for bicriteria profile queries, making direct comparison difficult. In comparison to Scalable TP, our query times are two orders of magnitude lower, at the cost of one order of magnitude for preprocessing costs. As such, our algorithm enables the currently fastest bicriteria profile queries on large realistic instances, with reasonable preprocessing overhead. On very large instances, such as Germany, preprocessing time and memory consumption may be prohibitive for some use cases. This is a subject of future research.

7 Conclusion

We introduced a speed-up technique for the basic trip-based public transit routing algorithm [26]. This technique applies principles sharing some similarities to those behind Transfer Patterns [1,4] and Hub Labelling [9] to the trip-based model and expands on them. The resulting algorithm enables query times on the microsecond scale on large realistic public transit networks with moderate preprocessing cost, occupying a Pareto-optimal spot among current state of the art algorithms.

Future work includes the study of different methods for cut node selection, with the goal of further reducing memory consumption and query graph size, developing tailored query algorithms to speed up queries on metropolitan networks, and making preprocessing more scalable by avoiding the computation of full one-to-all queries for all stops. We are also interested in adapting this speed-up technique to different scenarios, such as other and/or more criteria, and stop-based routing.

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 *European Symposium on Algorithms (ESA)*, volume 6346, pages 290–301, 2010.

[2] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. *ArXiv e-prints*, April 2015, 1504.05140.

[3] Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable Transfer Patterns. In *Algorithm Engineering and Experiments (ALENEX)*, pages 15–29, 2016.

[4] Hannah Bast and Sabine Storandt. Frequency-Based Search for Public Transit. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22. ACM Press, November 2014.

[5] Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann. Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In *Symposium on Experimental Algorithms (SEA)*, pages 35–46. Springer Berlin Heidelberg, 2010.

[6] J.C. Bermond, M.C. Heydemann, and D. Sotteau. Line graphs of hypergraphs I. *Discrete Mathematics*, 18(3):235–241, 1977.

[7] Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[8] Ulrik Brandes and Christian Pich. Centrality Estimation in Large Networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[9] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[10] Rene De La Briandais. File Searching Using Variable Length Keys. In *Western Joint Computer Conference 1959*, IRE-AIEE-ACM '59 (Western), pages 295–298, New York, NY, USA, 1959. ACM.

[11] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F Werneck. Computing Multimodal Journeys in Practice. In *Experimental Algorithms*, pages 260–271. Springer, 2013.

[12] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public Transit Labeling. In *Experimental Algorithms*, volume 9125 of *Lecture Notes in Computer Science (LNCS)*, pages 273–285. Springer, 2015.

[13] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics (JEA)*, 17, 2012.

[14] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. *Transportation Science*, 49(3):591–604, 2015.

[15] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science (LNCS)*, pages 43–54. Springer, Heidelberg, 2013.

[16] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria Shortest Paths in Time-Dependent Train Networks. In *Workshop on Experimental Algorithms (WEA)*, pages 347–361. Springer Berlin Heidelberg, 2008.

[17] Alexandros Efentakis. Scalable Public Transportation Queries on the Database. In *International Conference on Extending Databse Technology (EDBT)*, pages 527–538. OpenProceedings.org, 2016.

[18] Marco Farina and Paolo Amato. A Fuzzy Definition of "Optimality" for Many-Criteria Optimization Problems. *Systems, Man and Cybernetics, Part A: Systems and Humans*, 34(3):315–326, 2004.

[19] Linton C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, 1977.

[20] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science (LNCS)*, pages 71–82. Springer, Heidelberg, 2010.

[21] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *J. Exp. Algorithmics*, 11, February 2007.

[22] Matthias Müller-Hannemann and Karsten Weihe. On the cardinality of the Pareto set in bicriteria shortest path problems. *Annals of Operations Research*, 147(1):269–286, 2006.

[23] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *Journal of Experimental Algorithmics*, 12:1, 2008.

[24] Ben Strasser and Dorothea Wagner. Connection Scan Accelerated. In *Algorithm Engineering and Experiments (ALENEX)*, 2014.

[25] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 967–982, New York, NY, USA, 2015. ACM.

[26] Sascha Witt. Trip-Based Public Transit Routing. In *European Symposium on Algorithms (ESA)*, pages 1025–1036. Springer Berlin Heidelberg, 2015.