Scalable Transfer Patterns

Hannah Bast* Matthias Hertel* Sabine Storandt*

Abstract

We consider the problem of Pareto-optimal route planning in public-transit networks of a whole country, a whole continent, or even the whole world. On such large networks, existing approaches suffer from either a very large space consumption, a very long preprocessing time or slow query processing. Transfer Patterns, a state-of-the-art technique for route planning in transit networks, achieves excellent query times, but the space consumption is large and the preprocessing time is huge. In this paper, we introduce a new scheme for the Transfer Pattern precomputation and query graph construction that reduces both the necessary preprocessing time and space consumption by an order of magnitude and more. Average query times are below 1 ms for local queries, independent of the size of the network, around 30 ms for non-local queries on the complete transit network of Germany, and an estimated 200 ms for a fictitious transit network covering the currently available data of the whole world.

1 Introduction

Route planning on real-world public-transit networks is a complex problem. When evaluating algorithms for this problem, a variety of criteria are important in practice: preprocessing time, space consumption of the precomputed auxiliary data, query processing time and space, the ability for multi-criteria optimization (travel times, number of transfers, prices etc.), the ability for realistic modeling (transfer buffers, foot paths), and the ability to deal with delays and other real-time information. There exist a variety of approaches, which hit different Pareto points among these criteria; see [2] for an overview. One such approach is Transfer Patterns [1, 3], which is excellent for all of the criteria above except two: the space consumption of the precomputed auxiliary data is large and the preprocessing time is huge.

Another important aspect mostly neglected so far is scalability. Most experimental evaluations of existing approaches are done on metropolitan areas (like London [9, 18] or Madrid [4]), and some on whole countries (like Germany [16, 18] or Sweden [6, 5]). The largest of these networks is Germany, with an average of 15 million connections per day. In comparison, the number of connections on an average day for the (currently available) public-transit data of the whole world is estimated to be 320 million [15]. A few papers also consider networks of whole continents. For example, [12] consider Europe, however only long-distance transport, which amounts for only a tiny fraction of all transport (1.7 million connections per day). In the original Transfer Patterns publication [1], a North America network was investigated (57 million connections). However, the reported preprocessing time is around 3,000 core hours (4 months on a single core) with suboptimal results for an (albeit very small) fraction of the queries. For the other route planning approaches, experimental results for networks of this size are not available.

We first provide an overview of existing route planning approaches and discuss their potential applicability to continental-size networks. We will see that for most approaches, the extrapolated query times or space consumption are too high to be of practical use. For Transfer Patterns, query times were shown to scale well (about 6 ms on New York, about 10 ms for North America [1]), so there is great potential for interactive query times even when considering the network of the whole world.

In this paper, we introduce a new scheme

*Department of Computer Science, University of Freiburg, 79110 Freiburg, Germany, bast@cs.uni-freiburg.de, hertel@cs.uni-freiburg.de, storandt@cs.uni-freiburg.de .

for Transfer Patterns precomputation and query graph construction that allows interactive query times for huge transit networks with manageable preprocessing times and space consumption.

1.1 Scalability of other routing approaches

We are interested in the setting where all Pareto-optimal routes regarding travel time and number of transfers are computed in a query – which is the most common objective function when it comes to route planning in public-transit networks. A route is Pareto-optimal if there exists no other route for the same departure time with smaller travel time and less number of transfers. We discuss the following approaches with regard to the ability of Pareto-optimal route computation and scalability: Dijkstra's algorithm on a time-expanded graph (TED), Dijkstra' algorithm on a time-dependent graph (TDD), round-based public-transit routing (RAPTOR) [9], the Connection Scan algorithm (CSA) [10] and its accelerated version (ACSA) [16], trip-based routing (TB) [18], and approaches based on labeling, namely public-transit labeling (PTL) [6] and time-table labeling (TTL) [17].

Time-expanded and time-dependent graphs are basic models for converting the timetable information of a public-transit network into a graph that enables routing via Dijkstra's algorithm. The TED approach was used in the original Transfer Pattern paper for preprocessing. The disadvantage of the time-expanded model is the relatively large graph size. For example, for the North America instance with about 338,000 stations the time-expanded graph contains over 100 million nodes and about 450 million edges. The query times for both TED and TDD exceed 100 milliseconds for Pareto-optimal routes (regarding travel time and number of transfers), already on metropolitan-sized networks. For whole countries or even continents, query times are on the order of several seconds and more.

RAPTOR takes advantage of the fact that optimal connections typically require only few transfers. RAPTOR computes Pareto-optimal connections in order of increasing number of transfers by operating in rounds. The number of rounds is bounded by the maximum number of transfers occurring in an optimal connection from start to destination. For cross-country queries this number can easily be five or more. Since RAPTOR considers all stations reachable within this maximum number of transfers, it is likely to scan large portions of the complete network for such queries. RAPTOR achieves good query times for metropolitan areas, but query times are over one second already on Germany.

CSA stores all elementary connections in a single array, sorted by departure time. For a given query, the connections are scanned starting from the given source station and departure time until the algorithm can be sure that all optimal connections to the given target station are found. The number of scanned connections is usually large already for small networks, yet this algorithm is fast due to its ideal data locality. On London, CSA outperforms RAPTOR. Nevertheless, CSA is expected to scale even worse than RAPTOR. When considering, e.g., the whole of Europe, CSA would scan connections in Italy, Greece and Norway even when the query is between stations in Lisbon. ACSA was developed to reduce the number of such unnecessary scans by partitioning the transit network (using METIS [13]), and restricting scan operations to connections in/between relevant partitions in a query. Unfortunately, ACSA was only evaluated for earliest arrival time queries (using the number of transfers only as tie breaker, not as additional optimization criterion). Minimizing only travel time is a significantly easier problem, and only one solution per departure time is produced (while for Pareto-optimal queries typically several routes are returned).

TB is based on a similar idea as RAPTOR and also enumerates optimal routes in increasing order of transfers. The main difference is that possible transfers between trips are pruned in a preprocessing phase in an optimality-preserving way. This allows to scan less trips in a query, which results in query times of about 40ms on Germany. Answering queries on Germany with Transfer Patterns takes 0.3ms (without hubs) on average which is two orders of magnitude faster. The gap between the query times is expected to become larger with growing network sizes, because

even with transfer pruning, TB has to consider a significant portion of the network in a cross-country or even cross-continental query.

PTL uses hub labeling on a time-expanded graph. This leads to query times on the order of a few microseconds on London and also whole countries such as Sweden. The price to pay is a huge space consumption of the auxiliary data, which is about 26 GB for London alone. Even when neglecting amplification effects, the space consumption for considering, e.g., the whole of Europe would be gigantic. TTL is another indexing technique for transit networks which assigns labels to stations, leading again to query times in the microsecond range. Due to a custom-tailored compression technique, the space consumption of the auxiliary data is only about 1 GB for metropolitan areas. But TTL only computes optimal routes regarding travel time / earliest arrival time. Pareto-optimal solutions cannot be produced. Furthermore, all labeling techniques suffer from not being easily adaptable in case delays have to be considered.

1.2 Other related work While none of the existing approaches for public-transit routing have been evaluated on a whole-world network, web services like Rome2rio[1] offer the computation of routes between arbitrary locations on the globe. However, the goal there is not to come up with the full set of Pareto-optimal solutions, but rather a concise set of possible route options.

Improving the preprocessing time for Transfer Patterns was considered before in the frequency-labeling approach [5]. Frequency-labeling is based on the idea of handling vehicles that depart periodically not individually for every departure time but in a single operation whenever possible. This accelerates the Transfer Pattern precomputation by a factor of 60 on Germany (compared to the original TP paper [1], taking the difference of the used hardware into account) while also requiring less space during preprocessing. However, the large space consumption for the auxiliary data remains unchanged with this approach. And even with the improved preprocessing time, handling networks of continental size seems out of reach without massive parallelization.

Our new Transfer Pattern preprocessing scheme is inspired by the Customizable Route Planning framework (CRP) [7] which allows fast shortest-path computation and real-time updates on street networks. CRP starts by partitioning the street network into cells. Then, between all pairs of border nodes of a single cell (with border nodes being adjacent to an edge connecting to another cell), overlay edges are inserted with costs equal to the shortest driving time between the nodes. Queries are answered by a bi-directional Dijkstra computation in this overlay graph. As the overlay edges allow to skip over whole cells, query times are significantly better for CRP than for plain Dijkstra (especially if the partitioning is repeated recursively, resulting in a multi-layer graph). But CRP relies on several characteristics of street networks that public-transit networks are not compliant with. For example, every subpath of a shortest/quickest path is also an optimal path in the street network. This is not true for routes in public-transit networks when considering Pareto-optimal solutions, which increases the problem complexity significantly.

1.3 Contribution We present a new preprocessing scheme for Transfer Patterns that improves preprocessing time and space consumption for the precomputed auxiliary data structures by an order of magnitude and more. Local queries take below 1 ms, and non-local queries can be answered in 30 ms on Germany and an estimated 200 ms on the network of the whole world.

We first cluster the network and then show how these clusters and the natural hierarchy of local and long-distance transport can be exploited to limit the necessary effort for constructing transfer patterns in an optimality-preserving way. We describe several clustering methods and discuss their impact on the complete preprocessing scheme. We evaluate our new scheme on the large (but not huge) transit network of Germany and compare the outcome to the conventional Transfer Pattern scheme. We also extrapolate our results to a (so far fictitious) public-transit network of the whole

[1] http://www.rome2rio.com

world.

The deeper reason why our approach works is that very large (country-scale, continental-scale, or even world-scale) transit networks can be decomposed into fairly well-separated sub-networks of bounded size. Each such sub-network can be as large as a whole metropolitan area (e.g., New York and surrounding cities). This works because the state of the art has advanced to such a point that public-transit routing on whole metropolitan areas can be done with very fast query processing time and reasonable preprocessing effort. We can therefore consider whole metropolitan areas as "local" in our approach.

2 Preliminaries

We first introduce some basic notation and describe in detail the conventional Transfer Pattern construction and query processing scheme.

2.1 Transfer Patterns Transfer Patterns (TP) [1] is the approach behind public transportation route planning on Google Maps. The idea of TP is to precompute and compactly store all optimal routes in a time-independent abstraction. This abstraction of an optimal route, the so called transfer pattern, is the sequence of stations on the route at which a change of vehicle or transportation mode occurs (including the source and the target station).

To construct TP, a profile search is started for each station in the network, computing all Pareto-optimal paths to all other stations in a given time interval. Then all Pareto-optimal paths are backtracked and the transfer stations are identified. For each station, all these patterns are stored in reverse order in a directed acyclic graph (DAG). This reduces the space consumption, as common prefixes of patterns only have to be stored once. Still, if every station can be reached from every other station, the space consumption for all DAGs is (at least) quadratic in the number of stations in the network. For small networks, this is negligible compared to the space consumption for the timetables. But for the whole world, with millions of stations, this would be infeasible. Moreover, profile searches are expensive for large networks, even with algorithms designed for this purpose, like rRaptor [9], frequency-labeling [5] or profile TB [18]. With the latter, profile searches for Germany for a 24 hour range on a single core require about a day. For larger networks, we need more profile searches and each individual profile search is more expensive.

Query processing works as follows. For given start and target stations s and t, a query graph is constructed by extracting all transfer patterns from the DAG for s which lead to t (which is done in reverse starting from nodes in the DAG referring to t). The query graph is typically very small (on average less than a hundred nodes for queries on Germany). For a given departure time, Pareto-optimal routes can be found with the help of the query graph by running a time-dependent Dijkstra on it. Evaluating edge costs in the query graph amounts to a look-up in a direct connection data structure, which stores all routes that do not require transfers between each pair of stations, sorted by departure time. This enables query times of less than a millisecond on Germany.

2.2 Hubs Hubs were introduced to reduce the necessary preprocessing effort and especially the space consumption for TP. Hubs are a subset of "important" stations in the sense that they appear in many Pareto-optimal routes. Profile searches from hubs are done as described above. Profile searches from non-hubs can be stopped when all active labels encode a route over a hub. Every first hub on a Pareto-optimal path belongs to the access station set of the source station. At query time, the DAG for s is used to identify all optimal patterns to t and to the access stations of s. The DAGs of the access stations are then used to get the optimal patterns to the target t.

For example, for Switzerland, the space consumption for the DAGs was reduced from over 18 GB to about 830 MB when using hubs [1]. But preprocessing times were only reduced from 635 h to 586 h. To handle even larger networks, some heuristics were introduced in [1], as e.g. only considering routes with at most two transfers in the searches from non-hubs. It was shown that the number of suboptimal queries when using the heuristics is very small. Heuristic Transfer Pat-

tern construction for Switzerland only required 61 h. But for North America, even heuristic preprocessing requires over 3000 h.

3 Making Transfer Patterns Scalable

In the following, we present a multi-phase TP preprocessing scheme which takes several characteristics of transit networks into account to reduce the time and space necessary to compute and store all optimal Transfer Patterns.

3.1 Clustering transit networks The first and basic step in our preprocessing pipeline consists of dividing the stations of the transit network into suitable partitions/clusters. Ideally, we would like every cluster to be convex.

Definition 1. (Convex Transit Cluster) *A subset $C \subseteq S$ of stations in a transit network is called a convex transit cluster if for all pairs $s, t \in C$ all optimal routes from s to t only traverse stations in C.*

The complete network is naturally convex. But our goal is to break the network down into smaller clusters. In general, we expect from a good clustering that (1) clusters have moderate size, (2) there are only few connections between different clusters (3) each cluster contains at least one station where long-distance vehicles depart.

Many large transit networks are implicitly clustered because of different transport agencies running the vehicles in certain areas. But this data is not always available and the induced clustering might not meet our requirements (in terms of size, number of connections between clusters and convexity). Therefore, we will propose methods that work independently of this kind of information. Our clustering is computed based on only the local-transport connections and foot paths and not on the long-distance connections (like ICE, long-distance buses, etc.). If the given transit data does not provide a distinction between local and long-distance transport, a simple classifier based on the number of stations on the trip, distance between consecutive stations on the trip and speed of the vehicle can be used to come up with a heuristic distinction.

Based on all trips specified in the transit data, we construct a graph by creating a vertex for every station and connecting consecutive stations in a trip with edges. So for a trip that traverses k stations, we insert $k-1$ edges. Edge weights reflect the frequency of the connection over the period of a year. If several trips would lead to the insertion of the same edge, edge weights are accumulated. Moreover we add edges between stations when they are connected via a foot path ($\leq$ 400 m). They receive an artificially high edge weight because foot paths can be used at any time and stations in close proximity of each other should preferably end up in the same cluster. In our experiments, we use the weight 200,000 which corresponds to $\approx 365 \cdot 24 \cdot 20$, i.e. the foot path is used every three minutes. This leads to foot paths exhibiting a higher weight than 99.89% of the transit edges.

In the following, we outline four methods for computing clusters in public-transit networks, namely k-Means, merging, METIS and PUNCH.

3.1.1 k-Means clustering [14] This approach is oblivious of the graph structure but only considers the geo-coordinates of the stations. The k is a parameter to be set by the user. For a given k, an initial set of k seed stations is selected (e.g., randomly) and all other stations are assigned to the cluster of their closest seed. k-Means then minimizes the overall distance from the data points to the midpoints of the clusters to which they are assigned. It operates in rounds, where each round starts by selecting the means of the clusters as new seeds, followed by a reassignment of the stations to these seeds. The algorithms stops when the clustering does not change anymore from one round to the next. Note that k-Means does not necessarily compute connected partitions, since it ignores the structure of the network and uses only spatial data to form the clusters.

3.1.2 Merge-based clustering [11] This hierarchical approach was originally designed to find a partitioning of a road network. In the beginning, every node of a graph forms its own partition of size 1. Neighboring partitions are merged until their size reaches a given upper bound U. The order

of the pairs of partitions that are merged is determined by a utility function: in each step, the algorithm selects the pair of neighboring partitions of combined size $\leq U$ with the maximum value of the utility function. Since the aim of the algorithm is to find a partitioning with moderate partition sizes and small edge weights between the partitions, the utility function punishes big partitions and rewards pairs of partitions with high edge weights between them. We use the following utility function in our experiments:

$$f(u,v) = 1/s(u) \cdot 1/s(v) \cdot \left(w(u,v)/\sqrt{s(u)} + w(v,u)/\sqrt{s(v)}\right)$$

with $s(u), s(v)$ denoting the sizes of the clusters u, v and $w(u,v)$ is the sum of the weights of all edges with one endpoint in u and one endpoint in v. The algorithm stops when no pair of neighboring partitions with combined size $\leq U$ is left. Instead of an upper bound size U, one can also specify the number k of partitions. Then the algorithm merges partitions until only k partitions are left, without considering the size of the partitions. Just like for k-Means, the right choice of k is crucial for the result.

3.1.3 METIS [13] and PUNCH [8] METIS is a general-purpose graph-clustering algorithm that runs in three phases. In the first phase (called *coarsening*) the size of the graph is reduced. In the second phase, a partitioning of the coarse graph is computed. In the third phase, the partitioning is projected back to the original graph and refined. We use METIS with the *-contig* parameter, which enforces contiguous partitions. METIS is used in the preprocessing of ACSA [16].

PUNCH (partitioning using natural cut heuristics) was originally developed for road networks and runs in two phases. The first phase (called *filtering*) shrinks the input graph without changing its natural properties (as exhibiting small cuts along rivers and mountains). The second phase (called *assembly*) computes an initial partitioning of the shrunk graph and then revisits parts of the shrunk graph in order to improve the partitioning.

3.1.4 Types of stations in a cluster Let $\mathcal{C}$ be the set of clusters from a clustering. For each cluster $C \in \mathcal{C}$ we define the set of border stations $b(C)$ as those stations from C that are served by a trip to or from another cluster. That is, $s \in b(C)$ iff $s \in C$ and there is a trip containing s and $s' \in C'$ with $C' \neq C$ (see Figure 1 for an example). We also define $long(C)$ to be the set of those stations from C where long-distance transport departs or arrives.

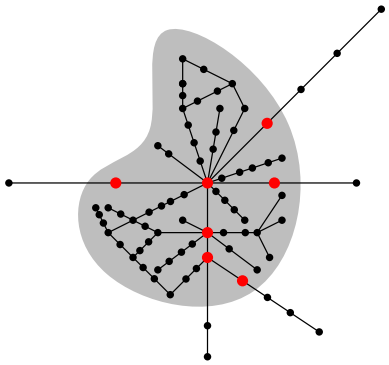

Figure 1: Illustration of a cluster (gray area). Dots indicate stations, trips are represented by straight lines. The large red dots indicate the border stations of the cluster. These stations are served by trips that enter/leave the cluster.

In the following, we will use C_x to refer to the cluster of a station x.

3.1.5 Border station reduction We formulated as goals for our clustering that clusters should have moderate size and there should only be few connections between different clusters. These two requirements are important as they will determine the preprocessing time and the query time later on. To be more specific, our preprocessing will require to compute all Pareto-optimal routes within each cluster. Obviously, the smaller the clusters the less time is needed per cluster. But we will also need to compute Pareto-optimal routes between clusters. It will turn out that this requires an expensive profile search for each border station of each cluster. Also, for an s-t-query the number of border stations of C_s and C_t will determine the query graph size. Hence small border station sets are desirable. But this leads to a conflict with our goal of having small clusters. For example, if we aim for clusters with around 1,000 stations each, large metropolitan

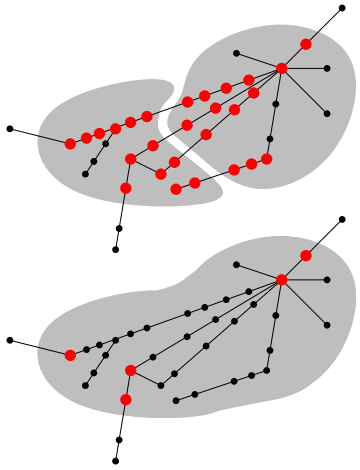

Figure 2: Border station reduction by merging two clusters – only 5 out of 27 remain.

areas, as Berlin or London, with more than 10,000 stations are split in several parts. As the public-transit in such areas is densely connected, splitting them leads to a huge number of border stations. But there are typically only a few of such densely connected areas per country. Hence allowing larger clusters only for such areas will not increase the total time for precomputing intra-cluster solutions too severely, but will decrease the maximum number of border stations per cluster.

We realize this by a post-processing phase in which we merge clusters if the number of border stations reduces dramatically, see Figure 2 for an example.

3.2 Local transfer patterns The next step after the clustering is to compute local transfer patterns, for which we need the optimal connections between all stations inside a cluster. Since we assume that each cluster is of moderate size (at most of the size of one metropolitan area), we can use the basic Transfer Patterns approach without hubs here (using, e.g., rRaptor for the profile searches). So for each cluster C, we start a profile query for each station $s \in C$ until all Pareto-optimal routes to all other stations in C are known. We do not consider connections with transfer stations that are not in C at this stage.

If the cluster is convex, this already guarantees optimal local queries, that is, between two stations from the same cluster. Determining which clusters are convex will be a by-product of the last step of our framework.

3.3 Long-distance transfer patterns between clusters In the next step, we concentrate on long-distance transport. Typically, the number of stations serving long-distance transport (called long-distance stations in the following) is very small compared to the total number of stations. For example, for the whole network of Germany less than 3% of all stations are long-distance in this sense. We now want to compute the transfer patterns between all pairs of long-distance stations in different clusters. To this end, we run a profile search from each $s \in \mathit{long}(C)$ until all optimal connections to stations in $\bigcup_{C' \in \mathcal{C} \setminus C} \mathit{long}(C')$ are known.

During the search, whenever a new cluster C' is entered, we use the local transfer pattern from the previous step to compute labels for all stations in $b(C')$ and $\mathit{long}(C')$. This saves us the consideration of the complete local transport in every cluster, thus speeding up the profile searches significantly.

If considering not only countries but whole continents or the public transport world-wide (including flights), the computation of these long-distance patterns might still be too expensive. Therefore, we use hubs in this phase to limit the search radius, for example large airports and train stations (e.g., Frankfurt Airport).

Having completed this step, we can compute *approximately* optimal routes between two stations s and t (in different clusters) efficiently as follows. We first look up the local transfer patterns from s to all stations in $\mathit{long}(C_s)$ and from all stations in $\mathit{long}(C_t)$ to t. Then we look up the long-distance transfer patterns from the long-distance stations in $\mathit{long}(C_s)$ to the long-distance stations is $\mathit{long}(C_t)$. Note that in a suitable clustering neither $\mathit{long}(C_s)$ nor $\mathit{long}(C_t)$ should be empty, i.e., every cluster contains at least one long-distance station. The resulting query graph can be evaluated as usual.

3.4 Local connections between clusters It remains to take care of optimal connections that demand local transport between different clusters. A naive approach would be to run a complete

profile search from every station of each cluster considering local and long-distance transport. But this would take effort comparable to that of the conventional Transfer Patterns approach. We use two modifications in order to reduce the necessary effort, based on the output of the previous steps above:

1. We only consider border stations of clusters instead of all stations as starting points for profile searches.
2. We accelerate each search by considering the (approximate) solutions that can be computed via the the precomputed local and long-distance Transfer Patterns.

We now describe in detail how these two modifications help to decrease the preprocessing time.

In the conventional Transfer Pattern computation, the profile searches from each station do a lot of redundant work. For example, consider a station s which only has elementary connections to another station s'. Then every optimal connection from s could be deduced by using the query graph for s' and augmenting it with the edge (s, s'). A separate profile search starting at s would be unnecessary. Hubs were introduced in the original approach as one method to reduce redundancy: the parts of the optimal routes beyond hub stations are only computed once. We will use the border stations of each cluster like hubs. Obviously, every connection between different clusters C_s and C_t involves visiting at least a border station in C_s and a border station in C_t. Since we already computed the optimal connections between all pairs of stations inside a cluster in Step 2, it remains to compute the optimal connections between the border stations.

A profile search from a border station $b \in C_b$ works as follows. We use a time-dependent Dijkstra computation and consider only local transport connections to other border stations. Every time we improve a label at a border station b' of a cluster $C_{b'}$, we try to improve the labels at all other border stations of C'_b by using the local transfer patterns from Step 2. Each improved label is added to the priority queue (PQ) of our Dijkstra computation. Moreover, we consider the long-distance connections leaving the cluster as follows. We first evaluate the local transfer patterns from b' to all stations in $long(C_{b'})$. Then we use the transfer patterns for each station from $long(C'_b)$ (precomputed in Step 3) to improve the labels of all other (far away) stations $q \in long(C_q)$. Since we are only interested in border-station labels, we use the local transfer pattern in C_q to check the optimal routes from q to all border nodes of C_q. Only if a border-station label is improved, we add it to the PQ.

After the profile run is completed, all optimal connections are backtracked in order to construct the DAG. This includes the integration of local and long-distance transfer patterns (from Steps 2 and 3) which were used in the search. In order to later construct query graphs between border stations more efficiently, we do not construct a DAG for each individual border station of C but rather one DAG per cluster, containing all patterns to a border station of the cluster. Note that a simple merging of the DAGs does not necessarily lead to a DAG again. We take care of potential cycles by keeping multiple DAGs per cluster if necessary.

The described approach limits the search effort that is necessary per border station. Intuitively, a route to a far-away destination almost always involves long-distance transport. This is because using only local transport the target is either unreachable (think about going from the U.S. East Coast to the West Coast) or the travel time and/or the number of transfers are unbearably high. Using our (approximate) solutions via local and long-distance transfer patterns, we are able to prune non-optimal local connections early. Moreover, we only consider non-border stations in a cluster if they are part of a transfer pattern between two border or long-distance stations. This saves many edge relaxation operations and – since only border stations are pushed into the PQ – also reduces the space consumption of the profile search.

Over the course of this last preprocessing step, we can easily decide whether the clusters produced in the first step are convex. If no new optimal patterns are identified between border stations of a cluster, the local transfer Patterns already captured all optimal connections between all stations inside the cluster. In that case, the cluster is con-

vex. Otherwise it is not.

3.5 Query processing For a given query, we proceed as follows. We first check whether the source station s and the target station t are in the same cluster. If this is the case and the cluster is marked convex, we only need to evaluate the query graph built from the local transfer patterns between s and t, which can easily be extracted.

If s and t are in different clusters or in the same non-convex cluster, we additionally extract the local query graph from s to all border stations of C_s. We do the same for the border stations of C_t and t. Then we extract the transfer patterns between the border stations of the two clusters which provides us with the final query graph. If hubs are used, they are incorporated as described in Section 2.2.

Running a time-dependent Dijkstra on the query graph (using the direct connection data structure to evaluate edge costs) completes the query answering.

3.6 Correctness proof We prove that our approach returns the full set of Pareto-optimal solutions. For that purpose, we first show an infix-optimality property of Pareto-optimal transfer patterns.

LEMMA 3.1. *If a Pareto-optimal route r from s to t is encoded with the transfer pattern $s, q_1, \cdots, q_r, t$, then either every subpattern $q_i, \cdots, q_j$ also encodes a Pareto-optimal route or it can be replaced by a pattern that encodes a Pareto-optimal solution from q_i to q_j without changing the costs of the route from s to t.*

Proof. Let $q_i, \cdots, q_j$ be an arbitrary subpattern. Let *dep* be the departure time at q_i in r and *arr* the arrival time at q_j. If $q_i, \cdots, q_j$ is not Pareto-optimal, there has to exist another route from q_i to q_j which (1) departs no earlier than *dep*, (2) has no more than $j - i - 1$ transfers and (3) arrives no later at q_j than *arr*. Because of (1) and (3), we can create a new valid route r' from s to t that uses the alternative route from q_i to q_j. Because of (1)-(3), r' can only lead to the same or a better arrival time at t and to less or the same number of transfers as r. Therefore, if r' would be better than r in one aspect, r would not have been Pareto-optimal in the first place. Hence we conclude that r and r' lead to the same costs. ∎

THEOREM 3.1. *The query answering procedure returns all Pareto-optimal solutions from s to t.*

Proof. If s and t are in the same convex cluster ($C_s = C_t$), the correctness argument from the conventional Transfer Patterns algorithm applies.

Otherwise let $s, q_1, q_2, \cdots, q_r, t$ be a pattern of a Pareto-optimal solution. Let q_i be the first station in the pattern that is not contained in C_s. Then the pattern from s to q_{i-1} has to be contained in the DAG for s because it is an optimal local transfer pattern (possibly $q_{i-1} = s$). Analogously, let q_j be the last station in the pattern that is not contained in C_t. Then the pattern q_{j+1} to t has to be contained in the DAG for q_{j+1} from the local transfer patterns computation (possibly $q_{j+1} = t$).

Now q_{i-1} and q_i are in different clusters, and so are q_j and q_{j+1}. Hence q_{i-1} is a border station of C_s and q_{j+1} is a border station of C_t. Therefore all optimal patterns between q_{i-1} and q_{j+1} are encoded in the DAG for the border stations of C_s. According to Lemma 3.1, if $s, \ldots, q_{i-1}, \ldots, q_{j+1}, \ldots, t$ is a pattern encoding a Pareto-optimal solution, either the subpattern from q_{i-1} to q_{j+1} has to be optimal as well for some departure time or there exists an alternative pattern that leads to the same Pareto-optimal solution from s to t. Therefore, the extracted query graph (encoding the precomputed Pareto-optimal solutions from s to the border nodes of C_s, from all border nodes of C_s to all border nodes of C_t and from the border nodes of C_t to t) allows to recreate the Pareto-optimal solution in question. As this holds for every pattern leading to a Pareto-optimal solution, the time-dependent Dijkstra run on the query graph will return the full set of Pareto-optimal solutions. ∎

We want to emphasize that our definition of border stations (all stations on inter-cluster trips) is

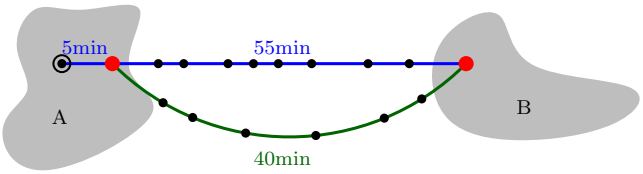

Figure 3: Illustrating the necessity of all stations on inter-cluster trips being border stations: If only the red marked nodes would be border stations, the only Pareto-optimal route from the border of A to the border of B would be via the green line, taking 40 minutes and no transfers. Combining local patterns with border patterns, the route from the circled node to B would cost 45 minutes and one transfer. But the Pareto-optimal solution of using the blue line directly, taking 1 hour and zero transfers, would be missed. If the circled node is a border station as well this problem can not occur.

crucial for correctness. Every station with a direct connection to another cluster is a border station according to our definition. Figure 3 shows that defining the border stations as those with an *elementary* connection to another cluster would not suffice, as Pareto-optimal solutions would be missed.

4 Experimental Results

We implemented the new Transfer Pattern construction and query processing scheme in C++, along with the conventional Transfer Pattern approach with and without hubs. Run times are measured on a single core of an Intel i5-3360M CPU with 2.80GHz and 16GB RAM.

4.1 Data sets The main reason for missing experimental results on public-transit network of continental size is the unavailability of data. While more and more GTFS feeds become openly available, most of them contain only the data for one metropolitan area. Even when aggregating all available data for a country, coverage is often poor. In particular, data in rural areas and long-distance transport connecting the metropolitan areas is often missing. Exceptions are, for example, the UK, Sweden and the Netherlands, for which feeds for the whole country were published. Moreover, the complete public-transit of Germany is described in the HAFAS data provided by Deutsche Bahn. But creating a meaningful feed for the whole of Europe or other continents is still out of reach. Therefore, we will first focus on Germany to show the ability of our new scheme to improve over the conventional Transfer Pattern approach. Then, we extrapolate our results to a (fictitious) transit network of the whole world. Table 1 shows the characteristics of our data sets; the numbers for the whole world are based on [15]. In Table 1 and all the result tables that follow, we mark estimated / extrapolated figures with a *.

	Germany	World
stations	0.25 million	5 million *
connections/day	15 million	320 million *

Table 1: Characteristics of the public-transit network of Germany (HAFAS data 2015/16) and estimated values for the whole world for a single day.

4.2 Transfer Pattern baseline We first present results for the conventional Transfer Pattern construction. For Germany, without hubs and using rRaptor for the profile searches, preprocessing takes about 372 hours [5] and produces auxiliary data of a total size of about 140 GB. This corresponds to profile search times of about 5 seconds per station. Using hubs (and accepting a small fraction of suboptimal results), preprocessing times can be reduced to 350 hours and auxiliary data size to 10 GB. In view of the profile search times reported for Germany when using TB [18], preprocessing times could be reduced to 24 hours.

Considering the whole world, with an estimated number of stations on the order of 5 million, we expect a single run of rRaptor to take at least 100 seconds per station on average. Without hubs, this would result in a preprocessing time of about 140,000 core hours (15 years on a single core) and a space consumption of around 50 TB (if all stations are reachable pair-wisely). With hubs, assuming that searches from non-hubs visit about 5% of all stations, the space consumption would decrease to about 3 TB, but the preprocessing time would still

be around 20,000 core hours (over 2 years on a single core).

4.3 Clustering We first evaluate how our clustering approaches behave on the public transit network of Germany.

4.3.1 Baseline The HAFAS data groups the stations of Germany into clusters of stations from 187 distinct sub-agencies. Cluster sizes according to this grouping range from 1 to 33,385 with an average of 1,337 stations and a standard deviation of 3,456. Most clusters exhibit slightly less than a thousand stations. The total cut size (i.e., the sum of all weights of edges with stations in different clusters) is around 96 million.

4.3.2 Comparison of clustering approaches We computed our own clusterings for the transit network of Germany, using k-Means, merge-based clustering, METIS and PUNCH as described in Section 3.1.

Germany contains about 250,000 stations. Thus, matching the average cluster size of around 850 in the HAFAS data leads to about 300 clusters. Table 2 shows a comparison of all approaches when fixing the number of clusters to 300. k-Means produces good results with respect to the total cut size and the number of border nodes. This is surprising, given the limited amount of information this approach takes as input. Merge-based clustering performs best in terms of cut size and number of border nodes. METIS produces even slightly less cut edges than merging, but the cut size is much worse. Using PUNCH also did not result in satisfactory cut sizes. PUNCH was developed to work on unweighted graphs. We tried to incorporate edge weights and used a utility function similar to merging to overcome this. But PUNCH was still outperformed by the other approaches. As PUNCH consists of many sub-algorithms which all could be further tuned, there might be a way to produce better results with PUNCH, though.

We conclude that in our implementation, merge-based clustering leads to the clustering most suitable for our application, since the number of border nodes and the number of inter-cluster trips indicated by the cut size determine the running time of the profile searches in the border transfer pattern computation phase.

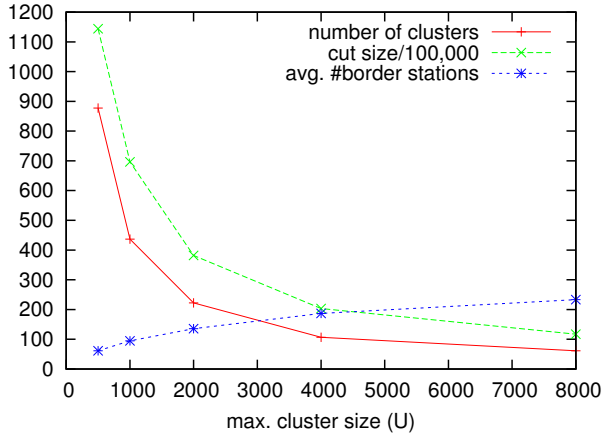

Figure 4: Experimental results for computing clusters on Germany with merging.

4.3.3 Varying the cluster size We also tested different cluster size bounds in the merging approach to evaluate the impact on the cut size and the number of border nodes.

In general, very small clusters/many clusters lead to efficient computability of local transfer patterns. But then the cut size and the total number of border stations is huge, hence the border transfer pattern computation gets expensive. On the other hand, very large clusters/few clusters lead to expensive profile searches in the local transfer pattern construction process. Moreover, large clusters lead to many border stations per cluster which renders non-local queries (with source and target in different clusters) inefficient. Both extremes, namely every station forming its own cluster and all stations being in the same cluster, yield a scheme that is equivalent to the conventional Transfer Pattern scheme with excellent query times but huge preprocessing effort in terms of space and time. We aim for a clustering which provides us with a good trade-off between preprocessing time, space consumption and query processing.

In Figure 4, merge-based clusterings are analyzed for different values of U (determining the maximal allowed number of stations in a cluster).

	k-Means	Merging	METIS	PUNCH
avg. cluster size	833	833	833	886
std. dev. cluster size	336.4	632.8	360.5	562.8
cut size	$20.1 \cdot 10^7$	$3.1 \cdot 10^7$	$6.5 \cdot 10^7$	$71.0 \cdot 10^7$
cut edges	16,124 (2.9 %)	10,887 (2.0 %)	10,828 (2.0 %)	15,162 (6.1 %)
border nodes	20,212 (8.1 %)	14,669 (5.9 %)	21,089 (8.4 %)	21,234 (8.5 %)

Table 2: Comparison of different clustering approaches on the Germany data set. The number of clusters was set to 300, therefore the average cluster contains 833 stations. For PUNCH, the number of clusters is not an input parameter. Reported numbers refer to a decomposition with 282 clusters.

We observe that doubling U halves the number of clusters and the cut size but also increases the average number of border nodes per cluster. A value of U between $1,000$ and $4,000$ leads to the best trade-off. We will use the merge-based clustering with $U = 1,500$ as basis for the following experiments. There, most clusters contain about 1,100 stations and have about 70-110 border nodes. We will use those numbers as basis for our later extrapolations.

The maximum number of border stations per cluster is $1,242$, though. Hence there are clusters with nearly all stations being border stations. Using our post-processing scheme, we could reduce this number to 986 while creating a few larger clusters with up to $6,109$ stations. Clustering methods that take metropolitan areas more explicitly into consideration might help to reduce the maximum occurring number of border stations further. Figure 5 depicts our final clustering.

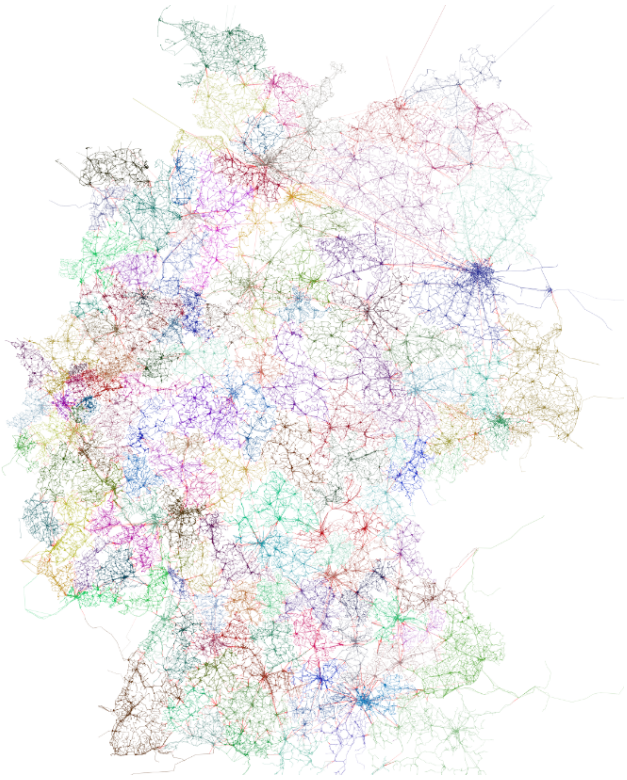

Figure 5: Outcome of our transit network clustering routine.

4.4 Local transfer pattern computation

The next step to evaluate is the local transfer pattern (LTP) computation in each cluster. Using rRaptor, we observed an average profile search time per station in the cluster of about 150 ms (including backtracking of optimal routes and DAG construction). Processing a single cluster then takes about 2.5 minutes on average. For Germany, this results in a total LTP computation time of 12.5 hours. For TB, profile times on London (with 20,800 stations and about 5 million connections) were reported to be 70 ms on average [18]. The average size of our clusters is about 15 times smaller than London. But it is unclear if this translates to the same factor of time reduction. Assuming 10 ms per profile query in a cluster, LTP computation would take 10 seconds per cluster and less than an hour in total. Experiments on the Netherlands with an average cluster size of 1,000 stations led to similar results. For Sweden and the UK, the timings were even better, since clusters contained fewer interior trips on average. Considering the whole world, we expect to get about 5,000 clusters with the same characteristics as for the clusters in Germany. Then the LTP computation would take about 200 hours

	avg. per station	avg. per cluster	total Germany	total World
rRaptor	150 ms	2.5 min	12.5 h	200 h
TB	10 ms *	10 secs *	1 h *	15 h *

Table 3: Timings for local transfer pattern computation with different profile search algorithms.

	# long-distance stations	time per station	time total
Germany	7,530	1.3 secs	2.7 h
World	150,000 *	30 secs *	1,200 h *

Table 4: Number of source stations and timings for long-distance transfer pattern computation.

with rRaptor, and about 15 hours with TB. Table 3 summarizes these results.

Regarding space consumption, the DAGs for a single cluster require 4 MB on average. For Germany, the total space consumption is 760 MB. For the whole world, this extrapolates to around 20 GB.

4.5 Long-distance transfer pattern computation In Germany, 7,530 stations feature long-distant transport (served e.g. by ICE/IC or Eurostar trains). Almost all of these stations are also border stations of their respective clusters. Since we only push border stations and long-distance stations in the PQ of the Dijkstra during long-distance pattern construction, the PQ contains 16,000 elements at most.

For Germany, a profile search using TDD (time-dependent Dijkstra) requires 15.2 seconds on average. Our accelerated search – using precomputed LTPs from the previous step, and only pushing long-distance and border stations in the PQ – takes only 1.3 seconds on average. Note that we keep the query graphs between border nodes explicitly in memory and do not recompute them from scratch every time a cluster is visited. The total time for the precomputation of the long-distance patterns on Germany is 2.7 hours.

Estimating the number of long-distance stations for the whole world is difficult. Assuming that, like in Germany, about 3% of the stations world-wide feature long-distance transport (presumably a huge overestimation), we get 150,000 such stations. On Germany, evaluating a single cluster takes 3-6 ms per profile search on average. Experiments on the Netherlands give similar results. For the whole world, we expect a single profile search using precomputed LTPs to take around 30 seconds. This results in a total time consumption of about 1250 hours for this phase. Table 4 provides an overview of the timings for long-distance pattern computation.

For Germany, the space consumption to store long-distance patterns is 180 MB. For the whole world, storing all such patterns in the naive way requires around 60 GB. Using hubs, the space consumption is reduced to around 4 GB.

4.6 Border transfer pattern computation With the merge-based clustering on Germany, about 16,000 stations are border stations. From each of these, we ran profile searches using both local and long-distance patterns. The average profile search time was about 0.6 seconds. For long-distance stations, no new searches are necessary, because we do full searches for them in the previous phase. The total time consumption for this phase is therefore 1.25 hours. For the whole world, assuming the same percentage of border nodes, about 300,000 profile searches are necessary. With an extrapolated running time of 15 seconds per search, and assuming that half of the border stations are long-distance stations, the precomputation time for this phase would be about 600 hours.

Long-distance patterns can be deleted after this phase. The space consumption for border patterns is about 400 MB for Germany, with some compression coming from the combined DAGs for the border stations of a cluster. For the whole world, border DAGs require about 180 GB without hubs, and about 10 GB with hubs. Table 5 summarizes the results for the complete Transfer Pattern construction pipeline.

4.7 Query processing We differentiate between queries with the source and the target station in the same cluster (local queries) and with the source and the target station in different clus-

	Germany		World	
	time	space	time	space
scalable TP, local patterns	12.5 h	760 MB	200 h *	20 GB *
scalable TP, long-distance patterns	2.7 h	(180 MB)	1,250 h *	(4 GB) *
scalable TP, border patterns	1.3 h	400 MB	600 h *	10 GB *
scalable TP, total	**16.5 h**	**1160 MB**	**2,050 h** *	**30 GB** *
conventional TP, without hubs	372 h	140 GB	140,000 h *	50 TB *
conventional TP, with hubs	350 h	2 GB	20,000 h *	3 TB *

Table 5: Overview of time and space consumption of the different steps of our new scalable Transfer Patterns precomputation. The space consumption for the long-distance patterns is enclosed in parantheses, since they are only required during preprocessing but not for query processing. The last two lines show numbers for the conventional Transfer Patterns approach; note that with hubs, a small fraction of the results may be sub-optimal.

	local		non-local	
	convex	non-convex	Germany	World
edges	30	380	4,500	15,000 *
time	0.1 ms	5 ms	32 ms	200 ms *

Table 6: Number of edges in the query graph and runtime for different kind of queries. Values are averaged over 1,000 runs with randomly selected source and target station.

ters (non-local queries). For a local query, it is also relevant whether the cluster is convex or not. Table 6 shows all results.

For Germany, local queries in convex clusters take 0.1 ms on average, with a query graph containing less than 30 edges on average. For non-convex clusters (92 out of 300), query times are 5.0 ms on average, and query graphs contain around 380 edges on average. Actually, in a non-convex cluster, on average over 90% of all pairs of source/target stations do not demand the addition of connections between border nodes. One could check this property for all station pairs in the preprocessing and then flag non-convex pairs. The query times for such "convex station pairs" would then be as fast as for station pairs in convex clusters.

For non-local queries, the combined query graphs contains about 3,500 edges on average. Constructing those graphs from scratch for every query takes about 20 ms on average. If the query graphs between clusters are cached, they can be merged with the local pattern from the source to the border stations and from the border stations to the target in 2 ms on average. The time-dependent Dijkstra computation on the final query graph takes 30 ms on average. This results in a total average query time of 50 ms or 32 ms, respectively.

For the whole world, query times are expected to be similar to the reported results for Germany. In particular, note that the time to answer local queries does not depend on how large the network is in total. For non-local queries, the patterns between border stations might contain more transfer nodes. Also, using hubs might introduce an additional chunk of nodes and edges. But even when we pessimistically expect that for each pair of border nodes 3 additional nodes are created in the query graph (on Germany, it is 0.3 nodes per border station pair), the query graph would contain around 15,000 edges on average. The average total query time is then around 200 milliseconds.

5 Conclusions and Future Work

We introduced a public-transit route planning scheme that is based on clustering the network but still produces the full set of Pareto-optimal solutions regarding earliest arrival time and number of transfers. On the transit network of Germany, our new approach reduces the preprocessing time by a factor of more than 20 and the space consumption by factor of more than 100, with exact query results in all cases. Using other profile search meth-

ods than rRaptor, the preprocessing times could be further improved. Our extrapolated values for the whole world predict even more dramatic reductions: down to a very reasonable 2,000 hour precomputation and 30 GB space consumption. We are aware that the structure of transit networks in other parts of the world differs significantly from the structure of the German network. However, we consider Germany a worst-case basis for extrapolation due to the generally high network density, which makes the partitioning into well-separated sub-networks harder. Of course, as soon as larger-scale public-transit data is openly available, experiments should be conducted to check the validity of our extrapolations.

In future work, a clustering approach which is designed to produce (possibly overlapping) convex clusters could accelerate the preprocessing and lead to even better query times for local (intra-cluster) queries. An important open problem is to deal with those (few) clusters with a relatively large number of border nodes. These do not significantly affect average query times, but are critical for the maximum query time. One promising approach could be a more metropolitan-aware clustering that allows some clusters to be significantly larger than others if this helps to reduce the maximal number of border nodes. We also consider it worth investigating which search techniques besides time-dependent Dijkstra searches allow for the incorporation of precomputed transfer patterns into the profile search. This could further reduce the profile-search times in the precomputation of the long-distance and border patterns.

References

[1] 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 2010*, pages 290–301. Springer, 2010.

[2] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *arXiv:1504.05140*, 2015.

[3] H. Bast, J. Sternisko, and S. Storandt. Delay-robustness of transfer patterns in public transportation route planning. In *ATMOS*, pages 42–54, 2013.

[4] H. Bast and S. Storandt. Flow-based guidebook routing. In *ALENEX*, pages 155–165, 2014.

[5] H. Bast and S. Storandt. Frequency-based search for public transit. In *SIGSPATIAL*, pages 13–22, 2014.

[6] D. Delling, J. Dibbelt, T. Pajor, and R. Werneck. Public transit labeling. In *SEA*, pages 273–285, 2015.

[7] D. Delling, A. Goldberg, T. Pajor, and R. Werneck. Customizable route planning. In *SEA*, pages 376–387. Springer, 2011.

[8] D. Delling, A. Goldberg, I. Razenshteyn, and R. Werneck. Graph partitioning with natural cuts. In *IPDPS*, pages 1135–1146, 2011.

[9] D. Delling, T. Pajor, and R. Werneck. Round-based public transit routing. *Transportation Science*, 2014.

[10] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Intriguingly simple and fast transit routing. In *SEA*, pages 43–54. 2013.

[11] I. Flinsenberg, M. Van Der Horst, J. Lukkien, and J. Verriet. Creating graph partitions for fast optimum route planning. *WSEAS*, 3(3):569–574, 2004.

[12] R. Geisberger. Contraction of timetable networks with realistic transfers. In *Experimental Algorithms*, pages 71–82. Springer, 2010.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. on Scientific Computing*, 20(1):359–392, 1998.

[14] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297, 1967.

[15] Private Communication, 2015. With a large commercial transit-routing provider.

[16] B. Strasser and D. Wagner. Connection scan accelerated. In *ALENEX*, pages 125–137, 2014.

[17] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *SIGMOD*, pages 967–982, 2015.

[18] S. Witt. Trip-based public transit routing. In *ESA*, pages 1025–1036, 2015.