Faster Transit Routing by Hyper Partitioning*

Daniel Delling[1], Julian Dibbelt[2], Thomas Pajor[3], and Tobias Zündorf[4]

1 **Apple Inc., USA**
ddelling@apple.com
2 **Apple Inc., USA**
jdibbelt@apple.com
3 **Apple Inc., USA**
tpajor@apple.com
4 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
tobias.zuendorf@kit.edu

Abstract

We present a preprocessing-based acceleration technique for computing bi-criteria Pareto-optimal journeys in public transit networks, based on the well-known RAPTOR algorithm [16]. Our key idea is to first partition a hypergraph into cells, in which vertices correspond to routes (e.g., bus lines) and hyperedges to stops, and to then mark routes sufficient for optimal travel across cells. The query can then be restricted to marked routes and those in the source and target cells. This results in a practical approach, suitable for networks that are too large to be efficiently handled by the basic RAPTOR algorithm.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Routing, speed-up techniques, public transport, partitioning

Digital Object Identifier 10.4230/OASIcs.ATMOS.2017.8

1 Introduction

Computing best journeys in transportation networks is a success story of applied algorithms research. Initiated by the publication of a continental road network in 2005 [17], a vast amount of results have been published [3], yielding exact algorithms that compute shortest path distances in a constant number of random memory accesses [1]. Some of these techniques turned out to be very practical and have been picked up by industry for widespread applications. As a result, millions of people today use algorithms developed by researchers over the last decade.

While most research focused on computing routes in road networks, computing best journeys in public transit networks is equally important [10, 20, 27, 29]. However, adapting techniques that work well on road networks to public transit proved to be harder than expected [6, 7, 8, 9, 13, 22, 34]. The most striking differences are the inherent time-dependency of the transit schedule as well as the need of multicriteria optimization in practice [26].

Five techniques specialized for public transit networks have been considered in recent years: Connection Scan (CSA)[18], RAPTOR [16], Transfer Patterns [2], Trip-Based Public Transit Routing [35], and specific labeling techniques [12, 33]. Unlike previous techniques, both CSA and RAPTOR do not rely on a graph representation but instead work directly

* This work was done while the last author was interning at Apple Inc.

17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017).
Editors: Gianlorenzo D'Angelo and Twan Dollevoet; Article No. 8; pp. 8:1–8:14

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

on the underlying timetable. This allows for very lightweight preprocessing (to index or re-organize the timetable) and queries fast enough for metropolitan networks. Notably, RAPTOR has been extended to a large set of optimization criteria (e.g., arrival time, number of transfers, reliability, walking duration, and ticket price) as well as to fully multimodal networks with unrestricted walking, biking and taxi [11, 16]. Trip-Based Public Transit Routing employs light preprocessing to achieve improvements in query performance of about an order of magnitude over CSA and RAPTOR. Even faster queries are possible, however, at considerable preprocessing cost on larger instances [36]. Transfer Patterns employs most heavy preprocessing to achieve very fast queries, but preprocessing takes too long to incorporate delays without sacrificing exactness [5]. Labeling techniques achieve even higher query performance but also require high preprocessing effort, especially for multi-criteria optimization, making the approach less applicable for real production systems.

Partitioning-based Approaches. For accelerating route planning in road networks, methods based on partitioning [14, 15, 19, 23, 24, 30, 31] turned out to be the most practical ones. The main observation is that in transportation networks, topology changes are much less frequent than metric changes due to, e. g., delays or traffic [14]. Partition-based approaches exploit this by running a rather expensive preprocessing step using the topology only, whereas a much faster second step (sometimes referred to as *customization*) produces auxiliary routing data based on real-time information [14, 19].

Transfer Patterns and Connection Scan have been enhanced to exploit this partition-based paradigm: A recent study reduces the preprocessing time of transfer patterns greatly [4], but it remains high in comparison. Accelerated Connection Scan [32] shows good preprocessing and query performance, however, it has not been evaluated for full multicriteria optimization (such as finding a Pareto set of arrival time and number of trips).

In this work, we study how RAPTOR can benefit from a partition of the public transit network. We argue that ideas from previous works [4, 32] do not translate well, i. e., RAPTOR needs to be adapted in a non-trivial way, leading to several interesting insights. More precisely, we introduce a novel approach to partition a public transport network and evaluate different algorithms for finding such partitions. We also present several algorithms to exploit such a partition and introduce optimizations for faster queries. We demonstrate feasibility of our approach on publicly available country-sized networks.

This work is organized as follows. Section 2 settles some preliminaries. Section 3 discusses two different approaches to partitioning public transit networks. Section 4 presents our main algorithm in full detail. Section 5 presents several experiments on large public transit networks showing the feasibility of approach. We conclude our work in Section 6.

2 Preliminaries

We consider *aperiodic* timetables, consisting of sets of stops S, events E, trips T, and footpaths F. Stops are locations where one can board or alight from transit vehicles. Stop events are scheduled departures or arrivals of vehicles. A trip is a sequence of stop events served by the same vehicle. Footpaths model walking transfers between stops.

A journey planning algorithm uses the aperiodic timetable in order to answer queries which consist of a departure stop, a target stop, and desired departure time at the departure stop. The answer to such a query is a set of journeys, each defined as an alternating sequence of trips and footpaths. A journey can be evaluated by several criteria, such as arrival time, number of transfers, total walking time, cost, etc. A journey j_1 dominates another journey j_2

with respect to a given set of criteria if j_1 is not worse than j_2 in any evaluated criteria and strictly better than j_2 for at least one criteria. A set of non-dominated journeys, with ties broken arbitrarily, is a *Pareto set*. In this work, we are interested in finding the Pareto set of journeys with arrival time and number of transfers as criteria. Note that Pareto-optimization is hard in general [21] but has long been known to be feasible in practice for public transit networks for these and other sets of well-behaved criteria [28].

RAPTOR

The Round-bAsed Public Transit Optimized Router (RAPTOR) is an algorithm explicitly designed for multi-criteria search in public transit networks [16]. While its basic variant optimizes arrival time and number of transfers, the algorithm can be adapted to incorporate further criteria as well. It organizes the input as a few simple arrays of trips and routes, where a route is a set of trips following the same sequence of stops (at different times) without overtaking each other. RAPTOR is now essentially a dynamic program operating in rounds, one per transfer. For a given source stop p_s, round i computes earliest arrival times at all stops of the network for journeys involving exactly i transfers. Each round takes as input the stops whose earliest arrival improved in the previous round (in round 0 this is the source stop only). Then, all routes served by these stops are scanned, which is done by traversing the stops on a route in order of travel, keeping track of the earliest possible trip that can be taken, subject to the arrival times from the previous round. This trip may improve the tentative arrival times along the the stops of the route.

RAPTOR can be extended to rRAPTOR, which computes *profile queries* resulting in all optimal journeys in a given time range. Essentially, the algorithm runs an individual RAPTOR query for each departure in the given range in order of decreasing time, however, it keeps the labels between runs for the purpose of pruning provably suboptimal journeys [16].

3 Partitioning Public Transit Networks

In this section, we discuss two possible partitioning approaches that may be exploited by the RAPTOR algorithm.

The most commonly-used approach [32, 4] of partitioning a public transit network is what we call the *stop partition approach*. It partitions the *stops* of the network into k cells $c_1, \ldots, c_k$. The partition is computed on the undirected station graph $G = (S, E)$, which is derived from the timetable as follows. The set of vertices corresponds to the set of stops S, and there is an edge $p_i, p_j \in E$ if and only if p_i and p_j appear on at least one route in consecutive order. One may assign edge weights corresponding to the number of contributing routes. The objective of the partitioning algorithm is to compute k (preferably balanced) cells while minimizing the number of cut edges between cells. Note that in the context of RAPTOR, every cut edge corresponds to a set of routes (those represented by that edge). Two connected (by a cut edge) boundary stops are therefore separated by a set of routes.

The elementary paradigm of the RAPTOR algorithm is to scan routes in their entirety. However, the stop partitioning approach separates stops by cutting routes, thus making it not the most natural fit for accelerating RAPTOR.

We therefore propose a different way to partition public transit networks, called the *route partition approach*. In this approach, we partition the *routes* (instead of the stops) of the network into k cells $c_1, \ldots, c_k$. Think of different (local or metropolitan) transit agencies that operate (densely interconnected) sets of routes that are sparsely connected with the routes of neighboring agencies in comparison. In order to obtain a route partition, we first

build an undirected route *hypergraph* $G = (R, E)$. Here, the set of vertices corresponds to the set of routes R. For every stop $p \in S$, we add a *hyperedge* $e \subset R$ to E, where e is exactly the set of routes that contain p. If several stops have the same set of incident routes, the corresponding hyperedge is only added once. As with the stop partition approach, we might also weigh the vertices and hyperedges in the graph to influence the balance of the resulting partition.

Note that a route partition with k cells can be transformed into a stop partition with $k+1$ cells as follows. Since each noncut stop has routes assigned to the same cell c_i, we can simply assign these stops to their corresponding cell c_i in the stop partition. The remaining (cut) stops are assigned to the extra cell c_{k+1}.

4 Main Algorithm

We now introduce HypRAPTOR, our new partition-based speedup technique for RAPTOR. As mentioned in the previous section, the route partition approach harmonizes more naturally with RAPTOR, which is why we base our algorithm on it. In what follows, we first go into more detail about the partitioning itself, then explain how—based on the partition—we compute a set of fill-in routes that are important for travel across cells, and finally discuss our accelerated query algorithm.

4.1 Partitioning

We propose two different approaches of computing a route partition: the first builds a partition by greedily merging adjacent cells, while the second one uses a hypergraph partitioning algorithm, such as h-metis [25], as a black box.

4.1.1 Greedy Merging

Our greedy approach iteratively merges (smaller) cells, and stops once the desired number of cells is reached. It starts with a trivial partition by making each route its own cell. Starting from this partition, it continues by merging cells greedily as follows. Two cells are considered as possible candidates for merging, if they contain at least one common stop (i. e., a stop that is contained in routes from both cells). For each pair of candidate cells, we first evaluate the *gain* of merging the cells. We define the gain in terms of the reduction of the fill-in size resulting from merging the cells (see Section 4.2 where we discuss the fill-in computation). The greedy algorithm maintains a priority queue containing all pairs of cells with the gain as key. In each iteration, the algorithm merges the pair of cells with minimum key (largest reduction of the fill-in size) and then updates the key of all adjacent pairs of cells by rerunning the fill-in computation on them.

Note that the greedy algorithm may be run from any given initial partition. If available, auxiliary data regarding the structure of the public transit network can be used to obtain such an initial partition. For example, one usually expects routes operated by the same agency to be more densely connected with each other than routes of different agencies. We may therefore initialize the algorithm with a partition, where each cell is exactly formed of the routes operated by one agency.

4.1.2 Hypergraphs

Our second approach is based on hypergraph partitioning. For this, we first construct an appropriate hypergraph from the public transit network and may then apply any hypergraph

partitioning algorithm that computes a vertex partition and minimizes the hyperedge cut. As already mentioned in Section 3, the vertices of the hypergraph correspond to the routes of the public transit network. Furthermore, we create one additional vertex for every footpath in the network. Finally, we create one hyperedge for every stop, such that the edge corresponding to stop $p \in S$ connects all vertices representing routes or footpaths that are incident to p.

In this work, we partition the resulting hypergraph with the well-known partitioning algorithm h-metis [25]. The general objective of the partitioning algorithm is to find cells of roughly equal size while minimizing the number of cut hyperedges that are incident to multiple cells. We can influence the balance of the partition by introducing weights for the vertices and edges in the hypergraph. In the weighted scenario, the objective of the partitioning algorithm is to find cells of equal weight while minimizing the sum of the weights of all cut edges.

For the vertices (routes) of the hypergraph, we propose three possible weighting schemes. Our first vertex weight is the number of stops contained in the corresponding route. Since scanning a route in RAPTOR requires some computation for every stop of the route, this weight directly measures the cost of scanning a route. Also, the cost of scanning a route depends on the number of trips of a route, so we propose to use the number of trips in the route as vertex weight. Finally, we can use the number of stop events of a route as the vertex weight, since this reflects both the number of stops and trips of a route.

Similar to the vertex weights, we want to design edge weights such that the size of a vertex cut corresponds to the induced complexity of the query algorithm. We propose two types of edge weights beyond unit weights. An obvious choice is to use the number of stop events at the stop as edge weight. However, this weight is quite unevenly distributed and cannot distinguish well between stops with many incident routes and those with only a few. To overcome this, we propose the logarithm of the number of stop events as a second edge weight.

Finally, we remark that the construction of the hypergraph may lead to multi-edges. This may happen when two stops are incident to the exact same set of routes and footpaths. We can reduce these multi-edges to one edge by summing up there respective edge weights. This does not influence the quality of the partition since either all or none of the edges from a set of parallel edges will be cut.

4.2 Fill-In Computation

In our query algorithm we would ideally restrict the RAPTOR computation to the cells of the source and target stop, which we call *active* cells. Unfortunately this only yields the correct output, if the optimal journey does not use routes outside these active cells. We therefore need to compute information on the optimal way of traveling across the other cells. We solve this problem by precomputing a set of routes, called *fill-in*, such that an optimal p_s-p_t-journey can always be found using only routes from the source cell, the target cell, and the fill-in. If an optimal journey requires routes from the fill-in, then it has to enter and leave the fill-in by transferring between trips at cut stops. This observation can be exploited for the fill-in computation itself. Consider an optimal journey with two transfers at stops p_u and p_v. In this case, the sub-journey from p_u to p_v is either an optimal journey by itself or can be replaced with an optimal one, without impairing the original journey. Therefore, it suffices for the fill-in computation to compute all optimal journeys between cut stops and add the involved routes to the fill-in.

A straightforward implementation of the fill-in computation performs an unrestricted rRAPTOR query on the entire public transit network from every cut stop of the partition.

However, this may be too time consuming in practice. In what follows we propose several techniques that exploit the partition of the public transit network in order to speed up the fill-in computation. Our first variant of the fill-in computation performs one rRAPTOR query from every cut stop p_u, restricted to the cells that p_u is part of. Essentially, we are using our accelerated query that only uses the source and target cells as active cells, in order to accelerate the fill-in computation itself. Since at that time the fill-in is not yet fully computed, the result of these queries may not be optimal (since optimal journeys may require routes that are not part of the active cell), causing the algorithm to add superfluous routes to the fill-in. Note that this has no effect on the optimality of the query algorithm.

In the second variant of the fill-in computation we go a step further and restrict the rRAPTOR queries to one cell at a time. This requires us to run rRAPTOR multiple times from every cut stop, once for every cell containing a route the cut stop is a part of. As before, this restriction can introduce unnecessary routes to the fill-in, at the benefit of reducing the computation time.

In order to reduce the overhead of the preprocessing even further, we have to analyze which routes have to be part of the fill-in. We already know that an optimal p_s-p_t-journey can contain routes that are not contained in the cells of p_s and p_t. Furthermore, we know that the first route of the journey has to be contained in one of the cells of p_s. This is because by definition the cells containing p_s also contain all routes reachable from p_s. Therefore, it suffices to add a route to the fill-in, if they are part of an optimal journey that first uses a route from an active cell and then connects two border stops. We now show how we can exploit this fact during the fill-in computation. Normally, rRAPTOR starts with an initialization phase, where all departure times of trips at the source stop are collected [16]. We modify the collection phase in order to reflect the fact that another trip has to be used prior to the fill-in. When computing the fill-in for a cut stop p_u restricted to cell c, we collect all arrival times of trips that are not part of routes in c. Afterwards we proceed with a standard rRAPTOR execution.

4.3 Queries

Finally, we introduce our query algorithm, a variant of RAPTOR that is restricted to the cells of the source and target cells, as well as the fill-in. We present three different variants of the algorithm that differ in their representation of the fill-in. These variants implement different trade-offs between memory usage and query performance.

The simplest representation of the fill-in requires one boolean flag for every route and every stop event in the public transit network. The flag of a stop event indicates whether the stop event is part of the fill-in, i.e., it is required to travel between border stops of the partition or not. If any flag of a stop event belonging to a route is set, then the flag of this route is also set.

Our first variant of the accelerated query completely ignores stop events if they are not flagged. For the most parts, the search is identical to standard RAPTOR. However, if the flag of a route is not set and the route is not part of the source or target cell, then the algorithm does not scan the route at all. Additionally, routes that are not part of the source or target cell, are only scanned partially. During the scan of these routes, stop events that are not marked are ignored. In particular, this means that scanning an unmarked stop event neither updates the arrival time of the associated stop, nor changes the trip that is scanned. Unfortunately, unmarked stop events still need to be scanned in order to determine if they are flagged or not, which is time-consuming.

Our second variant improves on this by introducing skip-lists. The basic idea of the skip-list is to provide for every stop event an offset that indicates the number of stop events

that can be skipped to reach the next marked stop event. In order to reduce the memory usage of the skip lists, we do not use one offset value per stop event. Instead, we use one offset value for every stop and every trip that is part of a route. The skip-list entry for stop p_u regarding route r specifies the index of the next stop p_v in r, such that a trip exists where stop which corresponds to p_v is marked. Similarly, the skip-list entry for a trip t of route r specifies the index of the latest trip t' before t, such that t' contains at least one marked stop event. The skip-lists are used when scanning the stop events of a route, allowing to skip entire parts of the route. While this approach scans less data than our first approach, it is still not very cache-friendly.

Finally, we present an even more efficient way of scanning fill-in routes, at the expense of an increased memory usage. For this variant we simply copy all trips that contain at least one marked stop event. From the copied trips we remove all stop events that are not marked, resulting in pure fill-in trips. Pure fill-in trips that serve the same sequence of stops are again aggregated like for normal RAPTOR into pure fill-in routes. Afterwards, our query algorithm only needs to scan routes that are either part of the source or target cell, or are pure fill-in routes.

A special case of the query algorithm occurs if the source or target stop are cut stops of the partition. In this case the cell of source or target stop is not defined. A straightforward solution for this problem is to combine all cells adjacent to the source or target stop. The algorithm can then proceed with the combined cells as source or target cell. However, we do not need to scan the source (or target) cell in case we use standard rRAPTOR queries to compute the fill-in. In this case we already know that all stop events are marked that are required to travel from the cut stop to another cell. Note that cut stops most probably are important stops, and queries between important stops are thus accelerated even more by our approach.

Restricted Walking

In the original publication of RAPTOR [16], the footpaths are assumed to be transitively closed, forming clusters of full cliques between stops. This enables RAPTOR to scan them as part of the same round, possibly improving the earliest arrival times associated in the respective round.

On realistic inputs, however, computing the transitive closure of all footpaths is often not feasible, as this would result in too many footpaths. Therefore, we propose to restrict walking between trips as follows. After scanning the routes in round i, we introduce an extra intermediate round i' (with its own dedicated set of earliest arrival time labels). Round i' scans all footpaths incident to all stops whose earliest arrival improved in round i in arbitrary order by reading labels from round i and writing labels to round i'. The subsequent regular round $i+1$ then reads its labels from round i' instead of i. The domination rule for target pruning [16] is adjusted, accordingly.

5 Experiments

We implemented our algorithms in C++ using LLVM 8.1 with full optimization. All experiments were conducted on a 2015 15-inch MacBook Pro with a quad core Intel Core i7 CPU and 16 GiB of 1600 MHz DDR-3 RAM running macOS 10.12.5. We use h-metis 1.5 [25] as hypergraph partitioner. All runs are *sequential*.

We consider two realistic inputs of country size: the national networks of Netherlands (`datahub.io/dataset/gtfs-nl`) and Switzerland (`gtfs.geops.ch`). Both networks

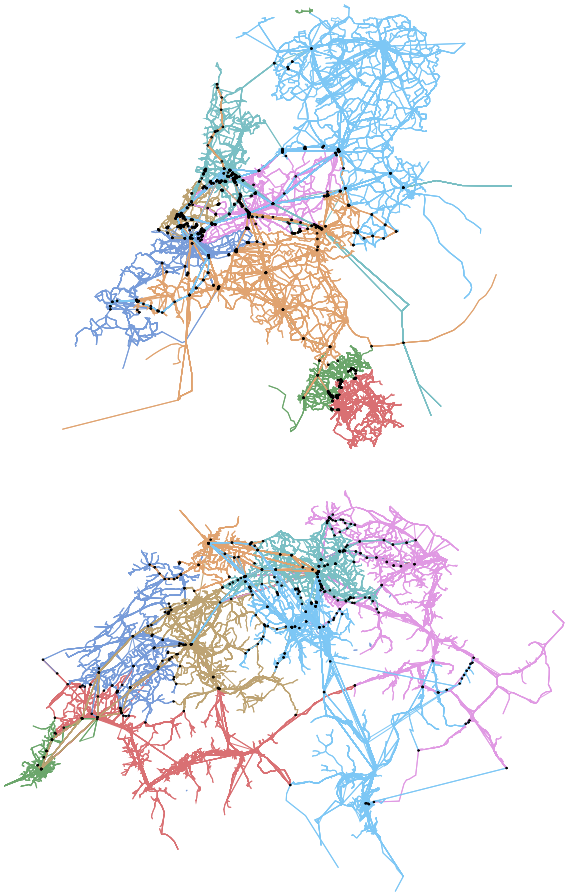

Figure 1 The routes of the Netherlands (left) and Switzerland (right) networks partitioned into 8 cells, each. Each color represents a different cell of the partition and the black circles indicate cut stops between routes of different cells.

contain local and long-distance transport. We extract the timetable of one week between June 4, 2016 and June 9, 2016. The footpath data for these two inputs is incomplete. Hence, we also artificially generate footpaths by connecting all pairs of stops that are closer than 200 m by straight-line distance and a walking speed of 3.6 kph. Note that in contrast to some prior work [16, 18, 12], our footpaths are not required to form clusters of full cliques. The resulting network of Netherlands has 54,500 stops, 618,961 trips, 12,989 routes, and 13,231,954 stop events. The network of Switzerland has 25,607 stops, 1,076,662 trips, 16,122 routes, and 12,733,856 stop events.

Hypergraph Partitioning

The first stage of our preprocessing is to compute a partition on the routes hypergraph (cf. Section 4.1.2). The hypergraphs resulting from our networks have 78,007 vertices and 54,500 hyperedges (Netherlands), and 30,839 vertices and 25,607 hyperedges (Switzerland). In addition, we use the number of stop events in a route as weight for the corresponding vertex in the hypergraph, vertices corresponding to a transfer have a constant weight of 0. Hyperedges (stops) are weighted with the logarithm of the number of stop events at the stop. We chose those weights because they yielded the best partitions during preliminary experiments. We then

Table 1 Figures for the partitioning and fill-in computation stages on 2, 4, 8, and 16 cells. We report the number of cut stops (# cut), the balance ratio of the largest to the smallest cell in terms of stop events (bal.), the number of routes in the fill in (% fn. rts), the number of stop events in the fill-in (% fn. ses), and the total running time of hmetis and the fill-in computation ([m:s]).

	Netherlands					Switzerland				
# cells	# cut	bal.	% fn. rts	% fn. se	[m:s]	# cut	bal.	% fn. rts	% fn. se	[m:s]
2	365	1.7	31.5	5.3	67:32	155	1.7	19.1	1.5	13:02
4	589	2.1	40.7	7.3	82:53	345	2.0	32.0	3.5	20:58
8	1,072	3.7	54.7	13.0	113:45	545	3.6	42.6	6.1	27:19
16	1,980	6.2	68.2	22.1	203:34	907	6.1	52.5	14.4	36:51

run h-metis [25] on these hypergraphs with the following parameters: balancing (`UBfactor`) set to 15, number of runs (`Nruns`) set to 20, hybrid first choice (HFC) vertex grouping scheme (`CType`), Fiduccia-Mattheyses (FM) refinement scheme (`RType`), and V-cycle refinement on the final solution of each bisection step (`VCycle`), see [25] for a detailed explanation of the parameters.

Table 1 shows figures of the partitioning stage for 2, 4, 8, and 16 cells (Figure 1 shows the partition into 8 cells for both our inputs). For each instance, we report the number of cut hyperedges (# cut) and the balance of the partition (bal.) as the ratio between the largest and the smallest cell in terms of stop events. The other columns in the table are related to the fill-in computation and are discussed later.

We observe that by increasing the number of cells, the size of the cut as well as the imbalance between the cells grows. While the former is natural, the latter may be surprising at first. However, the sizes of metropolitan areas (which are typically put in different cells) differ substantially.

Fill-in Computation

The second stage of our preprocessing involves the computation of the fill-in based on the output of the partitioning stage. In our experiment we use the following fill-in algorithm from Section 4.2: for each cut stop, we run an rRAPTOR query, restricted to the cells that are incident to the respective cut stop.

Table 1 shows figures on both our networks and for partitions of size 2, 4, 8, and 16 cells. We report the number of routes in the fill-in as a percentage of the total number of routes in the network (%,fn. rts). Recall that for correctness, not necessarily all trips of each route are required to be part of the fill-in. We therefore also report the number of stop events (again as a percentage) of the fill-in (% fn. se). Finally, we also report the running time for the entire preprocessing (partitioning and fill-in computation) in minutes and seconds. Note that the partitioning stage takes only a small fraction of the total reported time (less than a couple of minutes).

We observe that with increasing number of cells, the fraction of routes in the fill-in also grows, which is expected. However, the fraction of required stop events is much lower, a factor 4.2 on Netherlands and even a factor 7 on Switzerland (both on 8 cells). This confirms that only a small subset of the trips on the selected routes is actually important for the fill-in. Regarding running time, our preprocessing takes between 67 and 203 minutes on Netherlands, and between 13 minutes and 36 minutes on Switzerland. Note that while our reported running times are sequential, we would expect excellent speedups in a parallel

Table 2 Query performance figures of HypRAPTOR and RAPTOR for varying number of cells. We report the number rounds (# rnd), the number of scanned routes (# rts), the percentage of scanned routes in the fill-in (% fn. rts), and the average running time in milliseconds ([ms]). Each figure is obtained by taking the average over 10,000 queries with origin and destination stops picked uniformly at random.

		Netherlands				Switzerland			
Algorithm	# cells	# rnd	# rts	% fn. rts	[ms]	# rnd	# rts	# fn. rts	[ms]
RAPTOR	—	10.0	28,021	—	29.3	9.1	29,090	—	19.3
HypRAPTOR-f	2	9.8	24,592	7.5	25.1	9.1	25,267	4.2	16.7
HypRAPTOR-sk	2	9.8	24,592	7.5	25.2	9.1	25,267	4.2	16.7
HypRAPTOR-cr	2	9.8	24,666	7.8	25.0	9.1	25,306	4.4	16.8
HypRAPTOR-f	4	9.6	21,053	29.6	21.4	8.9	19,474	23.4	12.9
HypRAPTOR-sk	4	9.6	21,053	29.6	21.3	8.9	19,474	23.4	13.1
HypRAPTOR-cr	4	9.6	21,313	30.4	19.3	8.9	19,654	24.1	11.8
HypRAPTOR-f	8	9.7	19,821	56.4	21.0	8.7	17,056	48.1	12.0
HypRAPTOR-sk	8	9.7	19,821	56.4	21.6	8.7	17,056	48.1	11.9
HypRAPTOR-cr	8	9.7	20,278	57.3	17.5	8.8	17,405	49.1	9.3
HypRAPTOR-f	16	9.8	20,521	76.7	22.4	8.7	17,294	72.3	13.7
HypRAPTOR-sk	16	9.8	20,521	76.7	23.9	8.7	17,294	72.3	14.5
HypRAPTOR-cr	16	9.9	21,085	77.3	18.2	8.7	17,799	73.0	10.1

implementation, as the individual rRAPTOR queries are independent and could be easily run in parallel, or even distributed.

Regarding space consumption, the only information we need to store is the cell each route belongs to as well as one bit for each stop event. This is only a small fraction of the size of the input data.

Query Performance

Given the partition and fill-in, we now evaluate the query performance of our algorithms. Table 2 reports figures for the basic RAPTOR algorithm compared to three variants of HypRAPTOR: HypRAPTOR-f checks a flag associated with each stop event (of fill-in routes) to determine whether they are relevant; HypRAPTOR-sk uses skip lists to skip over ranges of irrelevant stop events more quickly; and HypRAPTOR-cr uses a compressed representation for the fill-in routes by completely rebuilding them based on the relevant stop events.

The figures in the table are obtained by running 10,000 queries between stops selected uniformly at random. We report the average number of rounds (# rnd), the average number of scanned routes (# rts), where applicable the percentage of the scanned routes that are in the fill-in (% fn. rts), and finally the average running time per query in milliseconds.

We observe that on both instances, HypRAPTOR scans significantly fewer routes in total, when compared to the basic RAPTOR implementation. For example, on 8 cells the amount decreases by 38 % on Netherlands and 67 % on Switzerland. Recall, that HypRAPTOR scans *all* routes in the cells containing the origin or destination stop as well as the routes contained in the fill-in. While for HypRAPTOR-f and HypRAPTOR-sk the number of scanned routes is the same by definition, HypRAPTOR-cr may scan slightly more. Building the compressed

routes in HypRAPTOR-cr may split some of the original routes, if the sequences of stops at which stop events are marked vary. This may result in a slightly higher number of total routes in the network.

In terms of running time, the best performance is achieved with HypRAPTOR-cr on 8 cells, resulting in speedups of 1.7 on Netherlands and 2.1 on Switzerland. Note that an upper bound on the expected speedup is roughly 4 when using a partition with 8 cells, as the algorithm must always look at the two origin and destination cells in full (plus the fill-in). When using only cut stops as origin and destination (not shown in the table), the speedups are 2.6 (Netherlands) and 6.3 (Switzerland). Partitions with higher number of cells than 8 do not seem to pay off on our instances, as they tend to cut through dense metropolitan areas, which leads to a significant increase of the fill-in size (cf. Table 1). However, on larger-scale instances with more metropolitan areas (such as the public transport network of multiple countries or a continent), we expect a higher number of cells to yield the best query performance.

Comparison

Comparing our results to existing ones from the literature is hard since there is no publicly available benchmark instance. Even worse, many large instances are actually proprietary [32, 2]. Comparing with the other two techniques that also employ partitioning, we see these tendencies: ACSA [32] has a speedup of a factor of 30 over regular CSA, but the evaluated instance is larger (the full network of Germany). More importantly, ACSA was only evaluated in a single-criterion scenario, which is less relevant for practical applications. Scalable Transfer Patterns [4] yields query times faster than ours (when scaling for network size) but preprocessing effort is higher, too. To summarize, our speedups are lower than those reported in the literature, but we focus on bi-criteria optimization and evaluate only smaller networks. As we already mentioned the best speedup we can achieve when using k cells is roughly $k/2$. On larger networks with more cells, we therefore expect to achieve greater speedups than the ones reported in our experiments. In fact, preliminary results on our own proprietary networks confirm this.

6 Conclusion

In this paper we presented a novel partitioning-based speedup technique for the RAPTOR algorithm. Unlike previous algorithms—which usually partition the set of stops—we instead compute a partition of the routes by employing a hypergraph partitioning algorithm on a carefully chosen hypergraph, where vertices correspond to routes and hyperedges to stops. We also showed how RAPTOR can be adapted to skip over trips that are neither necessary for travel inside the source and target cells nor for travel between cut stops. Our experiments on country-sized networks showed reasonable speedups, and we argued that by increasing the network size, the expected speedups would further grow.

Regarding future work, we are interested in computing better partitions, probably by an algorithm tailored to our scenario. In road networks, a multi-level partition boosts performance significantly and we are interested in seeing whether this is also true for public transit networks. Since our observations indicate that for country-sized networks only a small number of cells exhibits reasonable speedups, we expect multi-level partitions to be most useful for continental-sized networks. Finally, we also want to extend our approach beyond bi-criteria optimization.

References

1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In Leah Epstein and Paolo Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

2 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

3 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering – Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

4 Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable transfer patterns. In *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*, pages 15–29. SIAM, 2016.

5 Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-robustness of transfer patterns in public transportation route planning. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*, volume 33 of *OpenAccess Series in Informatics (OASIcs)*, pages 42–54. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. doi:10.4230/OASIcs.ATMOS.2013.42.

6 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

7 Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental study on speed-up techniques for timetable information systems. *Networks*, 57(1):38–52, January 2011.

8 Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, volume 12 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009. doi:10.4230/OASIcs.ATMOS.2009.2148.

9 Annabell Berger, Martin Grimmer, and Matthias Müller–Hannemann. Fully dynamic speed-up techniques for multi-criteria shortest path searches in time-dependent networks. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 35–46. Springer, May 2010.

10 Alessio Cionini, Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. Engineering graph-based models for dynamic timetable information systems. In Stefan Funke and Matúš Mihalák, editors, *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*, volume 42 of *OpenAccess Series in Informatics (OASIcs)*, pages 46–61. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/OASIcs.ATMOS.2014.46.

11 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing multimodal journeys in practice. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013.

12 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public transit labeling. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, Lecture Notes in Computer Science, pages 273–285. Springer, 2015. doi:10.1007/978-3-319-20086-6_21.

13 Daniel Delling, Kalliopi Giannakopoulou, Dorothea Wagner, and Christos Zaroliagis. Contracting timetable information networks. Technical Report 144, Arrival Technical Report, 2008.

14 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017. doi:10.1287/trsc.2014.0579.

15 Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-performance multi-level routing. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 73–92. American Mathematical Society, 2009.

16 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2015. doi:10.1287/trsc.2014.0534.

17 Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

18 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013.

19 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, April 2016. doi:10.1145/2886843.

20 Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.

21 Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company, 1979.

22 Robert Geisberger. Contraction of timetable networks with realistic transfers. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer, May 2010.

23 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

24 Sungwon Jung and Sakti Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, September 2002.

25 George Karypis. METIS – Family of Multilevel Partitioning Algorithms, 2007. URL: http://glaros.dtc.umn.edu/gkhome/views/metis.

26 Matthias Müller–Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria Pareto search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2007.

27 Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2007.

28 Matthias Müller–Hannemann and Karsten Weihe. Pareto shortest paths is often feasible in practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

29 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

30 Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

31 Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.

32 Ben Strasser and Dorothea Wagner. Connection scan accelerated. In Catherine C. McGeoch and Ulrich Meyer, editors, *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX'14)*, pages 125–137. SIAM, 2014.

33 Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 967–982. ACM Press, 2015. doi:10.1145/2723372.2749456.

34 Alexander Wirth. Algorithms for contraction hierarchies on public transit networks. Master's thesis, Karlsruhe Institute of Technology, 2015.

35 Sascha Witt. Trip-based public transit routing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, Lecture Notes in Computer Science, pages 1025–1036. Springer, 2015. Accepted for publication.

36 Sascha Witt. Trip-based public transit routing using condensed search trees. In Marc Goerigk and Renato F. Werneck, editors, *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'16)*, volume 54 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, August 2016. URL: https://arxiv.org/abs/1607.01299, doi:10.4230/OASIcs.ATMOS.2016.10.