Frequency-Based Search for Public Transit

Hannah Bast
University of Freiburg
79110 Freiburg, Germany
bast@informatik.uni-freiburg.de

Sabine Storandt
University of Freiburg
79110 Freiburg, Germany
storandt@informatik.uni-freiburg.de

ABSTRACT

We consider the application of route planning in large public-transportation networks (buses, trains, subways, etc). Many connections in such networks are operated at periodic time intervals. When a set of connections has sufficient periodicity, it becomes more efficient to store the time range and frequency (e.g., every 15 minutes from 8:00am - 6:00pm) instead of storing each of the time events separately. Identifying an optimal frequency-compression is NP-hard, so we present a time- and space-efficient heuristic.

We show how we can use this compression to not only save space but also query time. We particularly consider profile queries, which ask for all optimal routes with departure times in a given interval (e.g., a whole day). In particular, we design a new version of Dijkstra's algorithm that works with frequency-based labels and is suitable for profile queries. We evaluate the savings of our approach on two metropolitan and three country-wide public-transportation networks. On our largest network, we simultaneously achieve a better space consumption than all previous methods as well as profile query times that are about 5 times faster than the best previous method. We also improve Transfer Patterns, a state-of-the-art technique for fully realistic route planning in large public-transportation networks. In particular, we accelerate the expensive preprocessing by a factor of 60 compared to the original publication.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and Networks; E.4 [**Coding and Information Theory**]: Data compaction and compression

General Terms

Algorithms

Keywords

Route Planning, Public Transit Networks

SIGSPATIAL '14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA
Copyright 2014 ACM 978-1-4503-3131-9/14/11...$15.00
http://dx.doi.org/10.1145/2666310.2666405

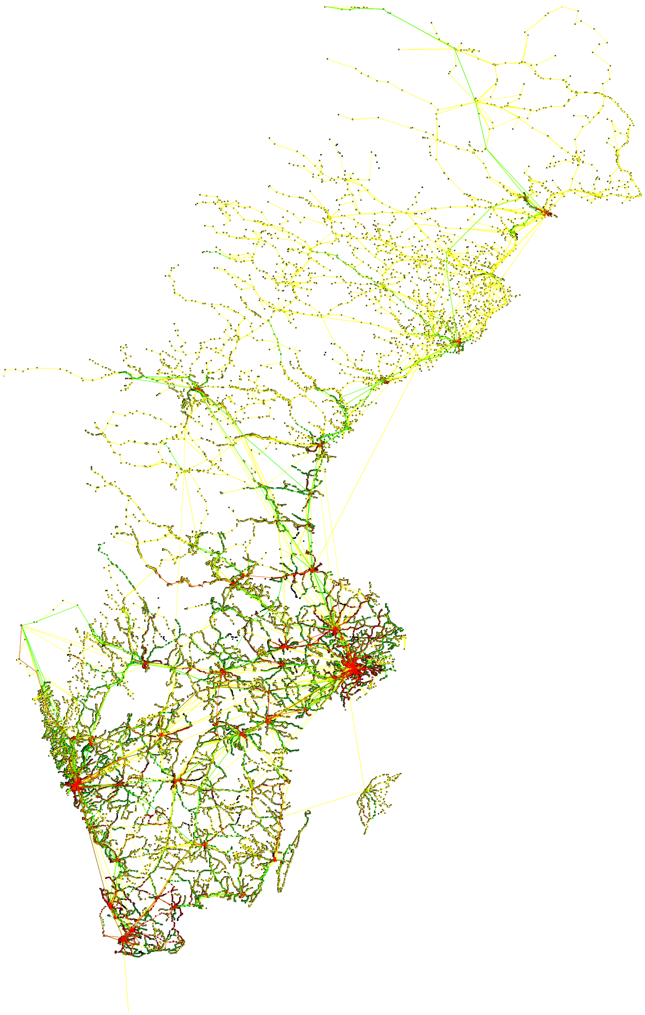

Figure 1: The public transit network of Sweden. Stations are indicated by black squares (visible when zoomed in). Lines are colored according to their service frequency. Yellow indicates a low frequency, green indicates a medium frequency (at least every two hours during day time), and red indicates a high frequency (at least every 30 minutes during day time).

1. INTRODUCTION

Finding optimal routes in large public transportation networks is challenging already due to the sheer amount of timetable data that has to be handled in the process. In a metropolis like New York, over 3 million times per day a vehicle departs from a station. In the public transit network of Germany, the number of departure events is about 14 million. In this paper, we try to compress timetable data by making use of the periodicity hidden in the schedules. For example, consider a bus which leaves at a particular station at 8:00, 8:15, 8:30, ..., 18:00. If we store each of these times explicitly, we need 41 entries. But we can provide the exact same information with the tuple (8:00, 15min, 41). The last item is the "frequency", which, throughout this paper, always means "periodic frequency", that is, the number of repetitions at some fixed time interval.

High-frequency connections are prevalent especially within large cities and metropolitan areas. But also the connections between such areas tend to be (periodically) frequent. Figure 1 gives an impression of the frequency distribution in the transit network of Sweden.

1.1 Contribution and Overview

In this paper, we contribute the following new algorithms and results. Figure 2 illustrates the main ideas in a nutshell.

- We present a heuristic to decompose sets of departure events into a small number of frequency-based tuples, in order to achieve high compression. See Section 2.
- We show how to efficiently process (profile) queries on a thus represented network. In particular, we show how to efficiently merge two sets of frequency tuples into a new such set, maintaing good compression. See Section 3.
- We compare the space consumption and profile-query time of our approach against two state-of-the-art methods (CSA and RAPTOR, see below) and two baseline methods (TD and TD$^+$, see below). In our most realistic setting, we simultaneously achieve the smallest space consumption as well as query times that are about 5 times faster than the best of these methods. See Section 4.
- We improve Transfer Patterns (see below). In particular, we accelerate the expensive preprocessing by a factor of 60 compared to the original publication. We also present query times (single-departure and profile) for a new large dataset (Germany, 13.9M connections). See Section 5.

1.2 Existing Route-Planning Approaches

Public-transportation route planning is a well-researched topic. We now briefly discuss two baseline approaches (TE and TD) and three state-of-the art approaches (CSA , RAPTOR, and Transfer Patterns). All of them can process multi-criteria profile queries (as a minimum: travel time and number of transfers) in large networks efficiently.

The two baseline approaches are variants of Dijkstra's algorithm. They differ in how the network is represented. In the basic *time-expanded (TE)* model, a node is introduced for each departure, arrival and transfer event. Elementary connections (all vehicle movements without intermediate stops) are modeled as arcs between these nodes. In the basic *time-dependent (TD)* model, there is a node for each station and an arc represents a set of elementary connections between two stations. Depending on the implementation, there can be a single node for the whole station, or separate nodes for different lines.

There are many variations of both TE and TD, and their efficiency strongly depends on the model details and their implementation. For an experimental evaluations and comparisons, see [12] [7] [13], as well as Sections 4 and 5. In a nutshell, TD is much faster (factor 5) than TE in simple settings (e.g., when optimizing only travel time and with few footpaths), but only slightly faster (factor 1.5) in realistic settings (e.g., when optimizing both travel time and number of transfers and with many footpaths). A particularly efficient variant of TD is TD$^+$ [4], which precomputes paths between selected stations. In Section 4, we compare our approach with both TD and TD$^+$.

CSA [6] stores all connections in a single large 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 journeys to the given target station are found. The number of scanned connections is usually very large, yet this algorithm is fast because of its ideal data locality. CSA can be viewed as an efficient realization of the TE baseline.

RAPTOR [5] takes advantage of the fact, that optimal journeys typically involve only a few transfers. It operates in rounds, sweeping over all relevant direct connections in each iteration and therefore identifying in round i all optimal journeys with $i-1$ transfers. RAPTOR can be viewed as an efficient realization of the TD baseline.

Transfer Patterns (TP) [1] is a precomputation-heavy approach that yields very fast query times also on very large networks. It is the algorithm behind public-transportation route planning on Google Maps. The main idea behind TP is to precompute and store in a compact way the so-called transfer patterns of all optimal paths at all times in the network. A transfer pattern of a path is simply the sequence of stations, where a transfer to another vehicle happens (including the very first and the very last station of the path). All temporal information as well as all information about intermediate stops (where no change of vehicle occurs) are thus factored out. The missing information can then be easily and efficiently re-inserted at query time; see Section 5.

1.3 Previous Compression Schemes

Periodicity-aware compression of timetable data has already been considered in previous work.

In [3], an algorithm for encoding periodic time sets was introduced. Their goal is to minimize the period, whereas our goal is minimize overall space consumption. For example, consider the time sequence: 6:05, 6:06, 6:07, 7:05, 7:06, 7:07, 8:05, 8:06, 8:07, 9:05, 9:06, 9:07. Minimizing the period results in four labels: 6:05-6:07 every 1 min, 7:05-7:07 every 1 min, 8:05-8:07 every 1 min, 9:05-9:07 every 1min. Our heuristic computes only three labels: 6:05-9:05 every 60 min, 6:06-9:06 every 60 min, 6:07-9:07 every 60 min.

In [10], repeating trips are represented as a set of labeled trees (so called multislices) which describe regularities and exceptions. In that work, the main contribution is to encode single trips as efficiently as possible (regarding operation days, holidays etc.) rather than extracting periodicities in the first place.

In [11], the encoding of complex temporal information like "every first Sunday of each month in autumn" is studied. A

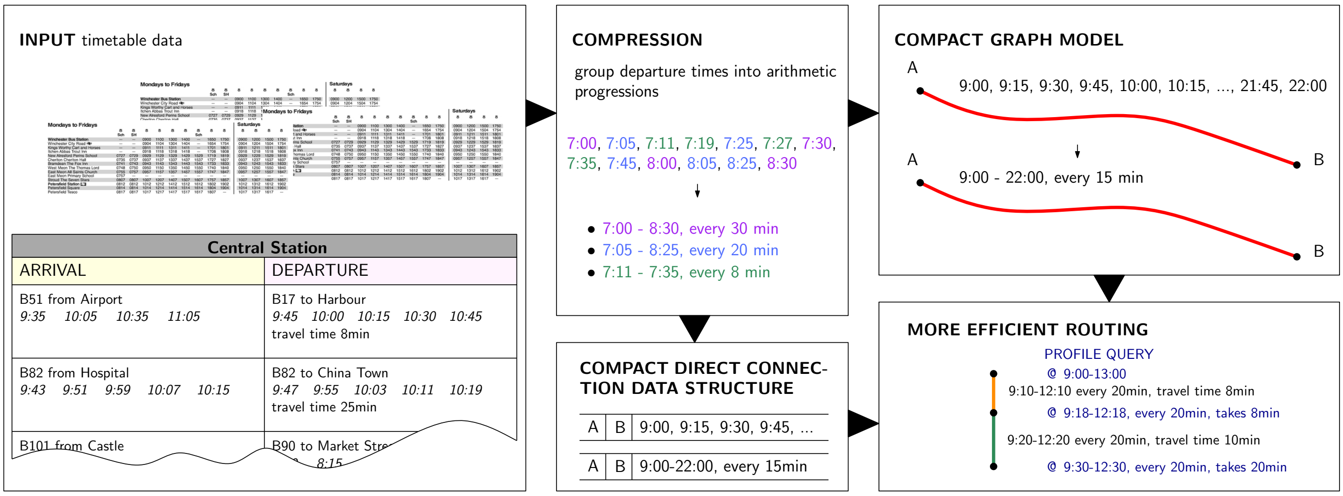

Figure 2: Overview of the main steps behind our frequency-based graph model and routing. Most of the technical challenge lies behind the box in the lower right (more efficient routing). In the simple example in that box, one arithmetic progression is enough to represent the intermediate result at each station. Usually, multiple arithmetic progressions with incompatible periodicities and ranges have to be merged. This poses a number of theoretical and implementation challenges.

symbolized representation is used to respect the natural data hierarchy (…, hours,day, week, month, year, …). Again, the idea is rather to encode given bits of information instead of searching for a clever partitioning of the data such that the encoding yields a concise overall representation.

2. FREQUENCY-BASED MODELING

In this section, we present our heuristic to decompose sets of departure events into a small number of frequency-based tuples, in order to achieve high compression.

2.1 Compression of Frequency-Data

Some timetable data storage formats support an explicit frequency-based specification. For example, in the widely used GTFS, for a periodic bus line, instead of explicitly specifying each trip at each time of the day, one can also specify only the schedule of the first trip together with a period and a time range. However, only few feeds make use of this, and even for those that do, it remains unclear if this is the best way to compress the data. Therefore, we first devise algorithms which take arbitrary timetables as an input and convert the data into a frequency-based representation.

Formally, we are confronted with the following problem: We are given a set of connections between two stations specified by departure times $T = \{t_1, t_2, \cdots, t_l\}$, with all the connections bearing the same travel costs. We aim for the minimum set of tuples (t, p, f) with t being the start time, p the period and f the frequency such that for all $t_i \in T$ it exists a tuple with $t + np = t_i \leq t + fp$ for some $n \in \mathbb{N}$. So we want to cover the set of departure times by tuples, explicitly allowing overlaps, i.e. there can exists more than one encoding tuple per departure. This problem is better known as *cover by arithmetic progressions (CAP)* and was proven to be NP-complete [9]. In our application the set of departure times can contain hundreds or even thousands of elements. As we have to solve this problem multiple times for each network, we seek for an efficient way to retrieve small cover sets.

2.1.1 *Reduction and Heuristic Solution*

The CAP problem can easily be reduced to SetCover in polynomial time, hence approximation algorithms and heuristics for SetCover carry over to CAP. The reduction works as follows. The universe of elements to cover is T. From now on we consider the elements in T sorted increasingly. The collection of subsets of T is $\mathcal{S} = \{S_{ij} | 1 \leq i < j \leq l\}$ with S_{ij} containing t_i, t_j and $t_i + n(t_j - t_i)$ for $n = 2, 3, \ldots$ as long as they are present in T without any gaps. So the set S_{ij} is simply an expansion of the arithmetic progression induced by start time t_i and period $p = t_j - t_i$. Obviously choosing a minimum subset of $\mathcal{S}$ to cover T solves the CAP problem. So now we can e.g. apply the standard greedy SetCover algorithm to guarantee an $\ln(|T|)$ approximation in polynomial time [8] for CAP. Unfortunately, this bound is not tight enough for practical purposes. Moreover constructing the set system explicitly requires cubic time and quadratic space. Therefore we propose a different algorithm which also works in a greedily manner but is more aware of the structure of the sets: We start with the smallest departure t_1 and search for the longest arithmetic progression (AP) in T starting with t_1. We add this AP to our solution and mark all elements covered by the AP. Then we repeat this approach with the next unmarked element $t_{i \geq 2}$ as start time. We do not exclude already marked elements from the set but do not allow them as suffix of a new AP and give preference to the AP which covers most unmarked elements. For example, consider

$$T = \{3, 5, 7, 10, 15, 17, 19, 20, 23, 24, 30, 31, 40, 50, 60\},$$

our algorithm would produce five APs

$$\{3, 10, 17, 24, 31\},$$

$$\{5, 10, 15, 20\},$$

$$\{7, 19\},$$

$$\{23, 30\},$$

$$\{40, 50, 60\}$$

with 10 appearing in two APs. The runtime of this greedy approach is in $\mathcal{O}(|T|^3)$ which in theory is the same as for the reduction of CAP to SetCover plus the standard greedy execution time. But in practice our approach considers only a small subset of all possible APs and moreover only requires linear space.

Further improvements can be made by running our greedy approach in multiple rounds adding a minimum AP length constraint K which gets reduced iteratively. So e.g. starting with $K = 6$ we would find in the first round the AP

$$\{10, 20, 30, 40, 50, 60\}$$

for our example instance. Proceeding with $K = 5$ in the next round we would discover

$$\{3, 10, 17, 24, 31\},$$

nothing for $K = 4$ but

$$\{15, 19, 23\}$$

for $K = 3$ and finally

$$\{5, 7\}$$

for $K = 2$. So we end up with four cover APs instead of five. For reasonable initial values of K the runtime increase is insignificant, as the reduction of possible start elements due to long APs at the beginning saves time in later rounds.

2.2 Stable Covers and Trip Covers

Typically, we are not only interested in connections belonging to a single operation day but in those of a whole week or in the complete period of validity of the timetable (e.g., about a year for German rail network data). Several operation days are important when e.g. an optimal journeys goes overnight, one is interested in outward and return journeys on different days or one wants to compare optimal journeys between fixed start and destination on several dates (e.g. on work days and on the weekend). Considering all departures between two stations over e.g. a week as one set of increasing departures is not beneficial, as there is typically a service gap at night which prohibits tuples covering connections of several operation days. So instead we assign operation days to tuples and compress the timetable data such that we have *stable* tuples, i.e. tuples valid for many operation days. To incorporate this in our greedy algorithm, we search then for the longest AP with longest denoting the number of covered departures of the whole considered period. So for example if the AP {8:00, 8:15, 8:30, 8:45} is only valid on Monday but {8:00, 8:15, 8:30} on Monday and Tuesday we favour the latter (as it covers six instead of four departures).

Another abstraction is to consider connections not on their own but in the context of the trips they are embedded in. So we compress trips by providing the sequence of stops, an initial set of departures and arrivals for each stop and then a period and a frequency indicating the repetitions. A trip-based compression is advantageous e.g. for RAPTOR where whole trips are evaluated in each round. So the compression can help to store this data more compactly and might be beneficial for profile runs as well.

3. FREQUENCY-BASED PROFILE SEARCH

In a profile query, we are given a set of departure times T in a certain time interval (e.g. all departures over a day) at a station S. The goal is to compute the optimal journeys from S to another station S' (or all other stations) departing in T. Profile queries are the key procedure for Transfer Patterns construction but are also important on their own e.g. if a user is interested in a set of options for departing in a certain time interval. Of course, profile queries can be answered by running $|T|$ conventional queries (one for each $t \in T$) and subsequently filtering the results (e.g. if one starts at 9:45 and arrives at 10:15 or starts at 9:47 and arrives at 10:15 as well, only the latter is kept). If the single departures in T are processed in reverse chronological order and the results of the last run are always remembered, one can already stop computations for earlier departures at the moment they do not improve the results for later ones. This is the approach e.g. used for profile queries with RAPTOR. For profile queries in the time-expanded graph (as used in the original Transfer Pattern paper [1]) it suffices to initialize the set of nodes corresponding to the departure events in T and then perform a single Dijkstra run. For all other approaches a single run also produces the desired result set if departure time is used as an additional Pareto-criterion.

We will now introduce a new type of profile search which incorporates the frequency-based compression directly. For this purpose, we consider the time-dependent graph model. Conventionally, the arcs in this model are augmented with sets of departure times each paired with a travel cost. We now replace these sets by our frequency-based representation. For every tuple constructed to cover departure times with the same associated travel costs, we insert a frequency-label $([a, b], p, c)$ consisting of the start time a, the end time b, the period p and the travel cost c. Note that this differs slightly from explicitly stating the frequency f which is now encoded by $(b - a)/p + 1$. But this alternative notation will be beneficial for the algorithm description later on. To evaluate such a frequency-label for a certain point in time t the following formula applies:

$$cost(t) = \begin{cases} a - t + c & if\ t < a \\ a + \lceil (t-a)/p \rceil \cdot p - t + c & if\ t \in [a, b] \\ \infty & if\ t > b \end{cases}$$

So obviously a frequency-based label can be evaluated in constant time, while a set of departure events with travel costs in the standard time-dependent model has to be parsed cleverly to come close to that, e.g. via binary search or specific look-ups [12], [13]. So if we start a single Dijkstra for each $t \in T$ in our frequency-based model the runtime should be improved compared to the standard time-dependent approach (if the compression step produced a small numbers of frequency-labels). Still, if the basic data exhibits synchronized departure times, we repeat the same set of operations again and again in a profile run (only with a time shift). Hence we would like a single Dijkstra computation to handle all these departures at once. We realize that by introducing frequency-labels not only for the edges but also as node labels (instead of scalar values) in the Dijkstra run.

3.1 Frequency-Dijkstra

In a frequency-Dijkstra run, we assign quadruples $([a,b],p,c)$ to the nodes, with $[a,b]$ marking the interval of arrival times with period p, and c being the summed-up costs since the departure from S. So the initialization for a profile query over a day is simply the label *([0:00,24:00], 1 min, 0 min)* assigned to the start station. Of course any other time span can be plugged in easily. The priority queue in the Dijkstra run then sorts such elements increasingly by first arrival time, using the cost value as tie breaker.

3.1.1 Edge Relaxation

The crucial task is now to adapt the edge relaxation step to this new setting. So given a label $l = ([l_a,l_b],l_p,l_c)$ at node $u \in V$ and an edge $e = (u,v) \in E$ with $([e_a,e_b],e_p,e_c)$, the goal is to compute the respective label(s) at node v. We proceed in five steps (see Figure 3 for an example):

1. Compute $lcm = lcm(l_p, e_p)$ to get the lowest common period.

2. Compute the first relevant start time $start$ at u. If $l_a \geq e_a$, it yields $start = l_a$ (if $l_a > e_b$ the edge must no be considered at all). Otherwise if $l_a < e_a$ then $start = l_a + \lfloor (e_a - l_a)/l_p \rfloor \cdot l_p$. If this would result in the $start$ value exceeding l_b we reset it to l_b.

3. Compute for the first $steps = lcm/l_p$ departure times $\{start, start+l_p, \cdots, start+(steps-1)\cdot l_p\}$ (restricted to values $\leq l_b$) the explicit edge costs $cost$ and arrival times arr at v. Store these values in a vector V.

4. Scan through V and remove multiple occurring arrival times (of course keeping the one with lowest cost); but be careful, that the last connection is not among the pruned ones (if so, add this single connection manually).

5. For every remaining item in V, create a new label l' at node v, with $l'_a = arr, l'_b = l'_a + \lfloor (\min(l_b, e_b) - l'_a + cost)/lcm \rfloor \cdot lcm, l'_p = lcm$ and $l'_c = l_c + cost$.

So the runtime of an edge relaxation is in $\mathcal{O}(\max(l_p, e_p))$, and at most lcm/l_p new labels are created at v.

Of course, not all labels created at the target node this way necessarily represent (temporary) optimal connections. Hence the goal is to prune the labels efficiently and join labels if possible to reduce space consumption and the number of subsequent operations.

3.1.2 Full Domination

A single connection departing at time t from station S and arriving at time a at station S' dominates another connection with parameters t', a', if $t \geq t'$ and $a \leq a'$ (with inequality holding at least once). We say that a label l fully dominates another label l', if for every connection implied by l' a dominating one implied by l exists. Of course, we do not want to break l' down into single connections and check each connection individually as this would take too much time. Instead, we present two criteria for full domination which can be used on the level of complete labels.

The first criterion is based on the idea, that all departures encoded in l' are also contained in l and the costs are not higher for the latter:

$$[l_a,l_b] \supseteq [l'_a,l'_b] \,\text{and}\, l'_a = l_a + k\cdot l_p, k \in \mathbb{N} \,\text{and}\, l_p \perp l'_p \,\text{and}\, l_c \leq l'_c$$

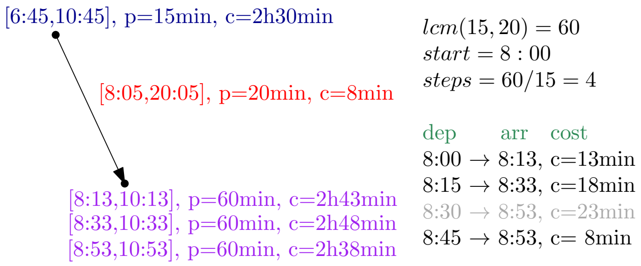
[6:45,10:45], p=15min, c=2h30min
[8:05,20:05], p=20min, c=8min
[8:13,10:13], p=60min, c=2h43min
[8:33,10:33], p=60min, c=2h48min
[8:53,10:53], p=60min, c=2h38min
lcm(15, 20) = 60
start = 8 : 00
steps = 60/15 = 4
dep arr cost
8:00 → 8:13, c=13min
8:15 → 8:33, c=18min
8:30 → 8:53, c=23min
8:45 → 8:53, c= 8min

Figure 3: Frequency-based edge relaxation example. The blue label is the frequency-based node label, which encodes that arrivals happen between $6:45$ and $10:45$ every 15 minutes, and that for each of these arrival events the travel time from the source to this node is 2 hours and 30 minutes. The red label tells that a vehicle departs from this node between $8:05$ and $20:05$ every 20 minutes, and that it takes 8 minutes to reach the next station. On the right side the steps according to the edge relaxation procedure are illustrated. These steps lead to the three violet labels that encode the complete set of arrivals at the next node.

Our second criterion implies that for every connection in l', it is worth to wait for the next departure enclosed in l as the summed costs of waiting and travel time are still below the travel time via l':

$$l'_a \geq l_a \quad \text{and} \quad l'_b - l'_c \leq l_b \quad \text{and} \quad l'_c > l_p + l_c - 1$$

In both cases, we can check in constant time whether l' is dominated and hence can be pruned at the respective node.

3.1.3 Partial Domination

Considering two or more labels, some of the implicitly contained connections might be non-optimal among all of the connections represented by the labels, while there does not need to be a label fully dominating another one. To check this, we can proceed similarly to the edge relaxation approach: we first compute the lcm of the two periods and then expand each label from a common start point as often as the lcm divided by the period implies. To identify dominated connections efficiently, we first merge the connections derived by the two labels according to their departure times. In this joined list, a dominating connection must follow the dominated one immediately. Therefore this pruning step can be performed in time linear in the list length, that is $\mathcal{O}(l_p + l'_p) = \mathcal{O}(\max(l_p, l'_p))$. If a connection gets pruned, also all repetitions with a time shift of a multiple of the lcm will be non-optimal (except when it is the last connection). Therefore, all of these connections get pruned, which might require to split the label in several sub-labels. Unfortunately, this means that, for example, a label with every seventh connection being non-optimal, has to be split into six new labels. Therefore, in terms of storage and run time it might be beneficial to keep some non-optimal connections. To avoid this problem of label increase (and possibly expanding lcm divided by the period labels every time) in practice without giving up the idea of using partial domination completely, we restrict ourselves to check if prefixes or suffixes of the implicitly contained list of connections can be pruned. If that is the case, we can shift the start/ end time of the label, receiving a smaller feasible interval (which finally might result in label deletion), but we will never end up with the creation of additional labels with this approach.

3.1.4 *Label Joining*

To reduce space consumption and save query time, we would like to join several labels into a single one whenever its possible. Two labels $l = ([a, b], p, c)$ and $l' = ([a', b'], p', c')$ can be joined under the following circumstances to form a new label l_j:

- $c = c'$ and $p = p'$ and $b + p = a' + k \cdot p \leq b', k \in \mathbb{N}$ $\Rightarrow \; l_j = ([a, b'], p, c)$
- $c = c'$ and $p = p'$ and $a' = a + p/2$ and $b' = b + p/2$ $\Rightarrow \; l_j = ([a, b'], p/2, c)$
- $c = c'$ and $a = b$ and $b + p' = a'$ $\Rightarrow \; l_j = ([a, b'], p', c)$
- $c = c'$ and $a' = b'$ and $b + p = a'$ $\Rightarrow \; l_j = ([a, b'], p, c)$
- $c = c'$ and $a = b$ and $a' = b'$ $\Rightarrow \; l_j = ([a, b'], a' - b, c)$

The first criterion implies that the labels are consecutive (with possible overlap), i.e. the second one takes over then the first one ends, with equal periods and costs. The second criterion describes the case, where the periods are equal, but also shifted by $p/2$; allowing the shift to become the new frequency. This criterion can be generalized to i labels shifted by p/i. The last three criteria describe how single connection can be included into existing labels or how two single connection can be combined respectively.

Determining the optimal set of join operations is hard, as the correct clustering of single connections again is an instance of CAP. Hence we proceed as follows: We keep the labels at a node representing not only a single connection sorted by costs, periods and a-value. If a new label of this type is inserted, we try to apply the first two criteria considering neighbouring labels in this list. Then we try to add single connections (which are stored separately) to this new label as implied by the criteria in lines 3 and 4 above. If the new label is a single connection, we start by using these same two criteria. If both of them do not apply, we augment the set of single connections with the new label and then invoke our heuristic greedy strategy to find a small frequency-based representation for them. Note, that here we only join connections into APs with size at least three and leave the remaining ones on their own.

3.2 Multi-Criteria Search

So far, we were only concerned with finding connections that minimize the travel time. But in a more realistic setting, we also want to consider other criteria (like the number of transfers). Note, that we can easily extend our frequency-labels by additional values for this purpose. The edge relaxation process then stays almost unmodified. For domination (partial and full) we now have the additional requirement that the additional cost value of the dominator is not higher than the respective value of the other label, and we join only labels with the same additional cost value(s).

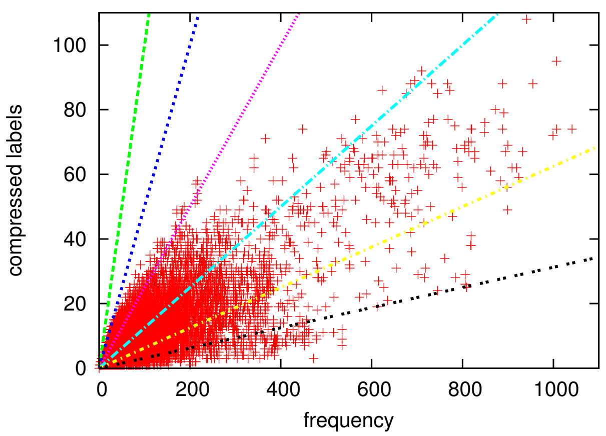

Figure 4: Frequency-compression for Sweden. Each data point is related to a set of departure times, with the set size given by the x-axis. The y-axis shows how many frequency labels were created for this input. The coloured lines indicate compression factors: green - no compression, blue - compression factor of 2, pink - factor 4, light blue - factor 8, yellow - factor 16, black - factor 32.

4. EXPERIMENTAL EVALUATION

To evaluate the impact of our algorithms on real-world data, we implemented our compression technique and our profile-search algorithm, along with five previous approaches, in C++. Run times were measured on a single core of an Intel i5-3360M CPU with 2.80GHz and 16GB RAM.

4.1 Data Sets and Baseline Algorithms

We conducted our experiments on several public transportation networks, with the underlying data being either extracted from public GTFS feeds or provided by Deutsche Bahn (DB). The characteristics of our test instances can be found in Table 1. We selected a Monday for our 1-day experiments and a complete week from Monday to Sunday for our 7-day experiments. For footpath integration we either used the transfer data contained in the DB data set (*few* in the table) or combined our feeds with the footpath network extracted from OpenStreetMap data[1] and inserted footpaths for a small walking radius of 5 minutes (*few*) or 15 minutes (*many*). The optimality criteria for journeys considered in the evaluation are arrival time and number of transfers. A transfer buffer of three minutes for changing vehicles is used. We compare to five of the six algorithms from previous work that we described in Section 1.2: CSA, RAPTOR, Transfer Patterns (TP), the conventional time-dependent approach (TD) and the improved version with transfer stations (TD$^+$, for earliest arrival time only). We refer to our frequency-based approach as FREQ. We omitted the time-expanded approach (TE), since CSA can be interpreted as an improved variant thereof; see Section 1.2. We also omitted ACSA [14], an accelerated version of CSA, because it requires that footpaths have to form cliques (which is not fulfilled in most of our networks), and because it does not compute Pareto-optimal solutions.

[1] http://www.openstreetmap.org

data	(abb.)	source	modi	#stops	# connections		# footpaths	
					one day	one week	few	many
New York City	(NY)	GTFS	ALL	16,450	3.44M	23.70M	238K	1,219K
Sweden	(SW)	GTFS	ALL	50,855	4.23M	27.31M	59K	306K
Weser-Ems-Bus	(WEB)	DB	BUS	32,683	1.25M	8.44M	76K	191K
Germany, trains	(GR-TRAIN)	DB	TRAIN	6,646	0.62M	4.26M	1K	2K
Germany, all	(GR-ALL)	DB	ALL	248,410	13.94M	90.42M	394K	1,262K

Table 1: Basic measurements of the five public-transportation networks from our experiments.

range	1 day				7 days	
	# tuples (%)	space consumption			# tuples (%)	space factor
		orig.	compr.	factor		
NY	911K (26)	78.5MB	27.8MB	2.8	1350K (6)	13.2
SW	1053K (25)	96.8MB	32.2MB	3.0	1281K (5)	16.0
WEB	560K (44)	28.7MB	17.1MB	1.7	748K (9)	8.5
GR-TRAIN	150K (24)	14.2MB	4.6MB	3.1	424K (10)	7.5
GR-ALL	5070K (36)	318.9MB	154.7MB	2.1	9821K (11)	7.0

Table 2: Compressing timetable data with frequency-based labels. Single connections are represented with six integer values (source, target, departure, arrival, tripID, serviceDays), frequency-labels with eight (source, target, a, b, p, c, tripID, serviceDays).

4.2 Preprocessing/Compression

We applied frequency-based compression to all our inputs, computing heuristic CAP solutions for the connections of a single day and also stable covers for a whole week. The respective results can be found in Table 2. We used $K = 10$ as initial minimum AP size for the results reported here, but found only little differences for all values between 5 and 15. For smaller initial values the solution quality got worse, for larger values the runtime increased. For $K = 10$, the computation time stayed well below 20 minutes for all inputs and settings. We observe that train networks compress much better than bus networks, reflecting the better synchronization of departures and arrivals for trains. In Figure 4, we analyze the compression of the Sweden network (considering a single Monday) more thoroughly. The plot shows that practically for all sets of departure times we fed in our frequency-label construction algorithm, the number of resulting labels is significantly smaller than the input size. Especially for high-frequency connections we achieve compression factors of 8 or better. The reason why the total compression factor is only 3 for Sweden is the large number of inputs that consist only of a single departure time. For these inputs obviously no compression can be achieved, and even worse, a frequency-label requires more space than the original representation. Therefore a mixed model that allows original and frequency-labels might be beneficial for an even higher overall compression.

Nevertheless, for all considered inputs, the compression factor validates our basic approach. For longer intervals, as e.g. the full week considered here, the number of elements is an order of magnitude smaller than originally.

The effect of the compression becomes even clearer when comparing the space of the auxiliary data necessary for different route planning schemes. In Table 3 we observe that especially for long time periods most of the baseline approaches get really space-consuming, while FREQ – based on the compressed representation – keeps a small memory footprint and seems to be applicable to represent even longer intervals without difficulty.

	memory consumption			
	1+few	1+many	7+few	7+many
CSA	753MB	1.7GB	4.8GB	10.9GB
RAPTOR	430MB	442MB	1.4GB	1.4GB
TP	2.3GB	2.4GB	3.2GB	3.3GB
TD	365MB	377MB	602MB	614MB
TD^+	784MB	796MB	2.2GB	2.2GB
FREQ	336MB	348MB	409MB	421MB

Table 3: Necessary space for computations on the German transit network (GR-ALL). Single connections are represented as before with six integers, footpaths with three (source, target, duration). 1/7+few/many denotes measurements for a single day/week and few/many footpaths.

4.3 Profile Queries

To show that compressing connections into frequency-based labels does not come at the cost of longer access and query times in the time-dependent model, we performed profile searches over the range of a whole week and summarized the results in Table 4. We observe that CSA is faster than RAPTOR for few footpaths, but RAPTOR performs better on more footpaths and criteria in our implementation. Note, though, that RAPTOR proceeds in the same way for EAT as for EAT+TR because it was designed to find the set of Pareto-optimal solutions considering travel time and number of transfers (from this set the single criterion solutions are then deduced). TD is always the slowest algorithm. TD^+ performs better but is still slower than CSA and RAPTOR. This might change in a multi-core setting, for which TD^+ was optimized [4]. In all considered scenarios, FREQ is the fastest algorithm, by a factor of 3-10. Note that a speed-up not larger than 7 for a week implies that for shorter intervals the other approaches can outperform FREQ. This is due to a single edge relaxation being costly in our model and so compression pays off only if the reduction of operations is huge; which is indeed the case when considering

	GR-TRAIN		GR-ALL							
	EAT + TR + few		EAT + few		EAT + many		EAT + TR + few		EAT + TR + many	
	time (s)	#ops	time (s)	#ops	time (s)	#ops	time (s)	#ops	time (s)	#ops
CSA	0.805	6.6M	25.97	363M	66.75	952M	42.71	363M	84.25	952M
RAPTOR	1.047	6.2M	38.50	314M	62.31	746M	38.50	314M	62.31	746M
TD	3.133	9.8M	69.19	204M	77.98	398M	125.44	512M	138.53	855M
TD$^+$	–	–	44.78	182M	68.70	352M	–	–	–	–
FREQ	0.482	1.4M	7.84	21M	9.86	34M	8.20	28M	14.22	64M

Table 4: One-to-all profile queries over a week for earliest arrival time (EAT) and where indicated #transfers (TR) plus few or many footpaths. *#ops* denotes the number of basic operations (scan or poll). Values are averaged over 1000 random queries.

a week, because lots of stable tuples (valid for many operation days) could be identified. So the speed-up achieved with FREQ is significant and likely to grow with the size of the considered interval – and that with a lower memory footprint than all the other approaches. One could argue, that the other approaches are inherently parallelizable and with the workload being distributed to several cores the run times would be better than for FREQ. But on one hand, a multi-core implementation again would cost more memory and moreover in a client/server-architecture it is often not favourable to let one user fully load several cores. Therefore the implicit parallelization of FREQ by compressing many connections into a single label is beneficial in this context. On the other hand, we could turn FREQ in an explicitly parallelizable algorithm by splitting the input interval into several time frames, e.g. 0:00-8:00, 8:00-18:00, 18:00-0:00 each valid for all considered operation days (in contrast to the other algorithms using e.g. a core per day).

Note, that we omitted timings for TP in Table 4. In fact, TP allows for much faster profile queries, on the order of milliseconds; see Table 5 in the next section. But TP is based on a precomputation that itself relies on fast profile queries. The whole next section is about improving TP using the fast profile searches from Table 4.

5. EFFECT ON TRANSFER PATTERNS

Profile queries are interesting by themselves, but they are also an important building block for advanced routing techniques. In particular, profile queries are the main ingredient in the preprocessing of Transfer Patterns (TP) [1]. We briefly described TP in Section 1.2 and will provide more details in this section.

Recall from Section 1.2 that the bulk of the preprocessing of TP consists of a profile query for each station of the network. From these profile queries, all transfer patterns can be easily computed by tracing back the shortest path trees. The preprocessing time is hence the number of stations in the network (see Table 1) multiplied with the average time for a profile search for a single station in the network (see Table 4). The additional time needed for the backtracking is negligible.

In [1], so-called *hub stations* are introduced as a means to reduce preprocessing time (in a nutshell, by computing transfer patterns only until hubs as well as between hubs). This blows up query times and adds a number of other challenges though. In particular, when using TP with hub stations, a very small fraction of queries may yield sub-optimal results; see [2]. However, with sufficiently fast profile queries, as we have them available through the work from this paper, hub stations are no longer necessary, not even for a network as large as GR-ALL. In particular, this improves query times, and all queries are guaranteed to yield optimal results.

5.1 Improved TP preprocessing times

In Table 5, we provide preprocessing times for TP for the GR-ALL instance for a variety of settings. Namely, we consider all eight combinations of: single-criteria (EAT) or multi-criteria (EAT+TR), few or many footpaths, and a 1-day or 7-day slice of the GR-ALL network. We compare the preprocessing using our FREQ approach with the best of the other approaches (CSA for EAT+few, RAPTOR for the other combinations).

The most realistic of the eight settings is EAT+TR+many (multi-criteria search with many footpaths) and 7 days. For that setting, our FREQ achieves a TP preprocessing time of 981 hours. The next-best method for this setting, RAPTOR, requires 4300 hours, which is more than 4 times longer. The baseline TD approach would take another factor of more than 2 longer and is thus almost 10 times slower than our FREQ; see the last column of Table 4.

Let us also compare these numbers to the numbers reported in [1], the original TP paper. They report 635 hours for the preprocessing of the public-transportation network of Switzerland without hub stations; see Table 3 in their paper. This is a relatively small network with 20.6K stations and 1.75M connections[2]; see Table 1 in their paper. This translates to 63.4 seconds per station per 1 million connections. In comparison, the 981 hours of FREQ from above, on 7 days of the GR-ALL network, translate to 1.0 seconds per station per 1 million connections. This is a dramatic improvement of more than a factor of 60.

The main reason for this improvement is that in [1], profile queries were run using a plain multi-criteria Dijkstra on a time-expanded graph (TE). This approach is simple and flexible, yet performance-wise comparable to, and even slightly worse than, our TD baseline[3]. And as discussed above, this TD baseline is almost 10 times slower than our FREQ in a realistic setting. Another factor is that the setting in [1] was slightly more complex, with a 14-day schedule and a continuous penalty as second criterion (for EAT+TR, we only consider the discrete number of transfers as second

[2] In [1], only the number of nodes is reported. However, in their time-expanded network, the connections correspond 1-1 to pairs of an arrival and a departure node. Thus, the number of connections is simply half their number of nodes.

[3] Recall from Section 1.2, that in realistic settings, TD is only slightly faster than TE.

	1 day		7 days	
EAT+few	CSA	FREQ	CSA	FREQ
preprocessing time	249h	451h	1792h	541h
query graph size (#nodes + #edges)	42+74		46+85	
query time / number of solutions	0.2ms / 0.91		0.4ms / 0.97	
profile query time / number of solutions	3.3ms / 16.44		22.0ms / 121.19	
EAT+many	RAPTOR	FREQ	RAPTOR	FREQ
preprocessing time	601h	612h	4300h	680h
query graph size (#nodes + #edges)	68+91		79+121	
query time / number of solutions	0.4ms / 1.00		0.9ms / 1.00	
profile query time / number of solutions	7.2ms / 22.51		51.7ms / 166.51	
EAT+TR+few	RAPTOR	FREQ	RAPTOR	FREQ
preprocessing time	372h	517h	2657h	566h
query graph size (#nodes + #edges)	57+81		66+98	
query time / number of solutions	0.3ms / 1.92		0.8ms / 1.95	
profile query time / number of solutions	5.0ms / 31.90		39.6ms / 225.97	
EAT+TR+many	RAPTOR	FREQ	RAPTOR	FREQ
preprocessing time	601h	817h	4300h	981h
query graph size (#nodes + #edges)	82+110		95+153	
query time / number of solutions	0.6ms / 1.68		1.5ms / 1.74	
profile query time / number of solutions	12.2ms / 34.14		95.3ms / 245.81	

Table 5: Experiments for Transfer Pattern construction and evaluation on the GR-ALL dataset with few/many footpaths. Computed paths are (Pareto-)optimal with respect to earliest arrival time (EAT) and where indicated also number of transfers (TR). The preprocessing time using FREQ is compared to the strongest competitor for the specific setting according to Table 4. Query graph sizes and query times are averaged over 1000 random queries.

criterion). Also, the numbers from [1] are about four years old, and processor speeds have likely about doubled since then.

Let us also briefly discuss the simplest setting, which is EAT+few (single-criteria with few footpaths) on a 1-day slice of the GR-ALL network. Here our TF-freq needs 451 hours for the TP preprocessing, which is about twice faster than for EAT+TR+many on the 7-day network. However, for this simple setting, CSA takes only 249 hours, which is yet another factor of 2 faster. This makes sense, because our FREQ pays an implementation overhead in order to take advantage of repetitiveness in the network. In comparison, CSA is an extremely simple algorithm with little implementation overhead. For a 1-day network, the repetitiveness in the data is not sufficient to outweigh the overhead. For a 7-day network, we saw above that there is already a significant performance gain. For even longer periods the gain becomes even larger: as the number of days increases, the cost of our FREQ grows only very slightly, whereas the cost of CSA and RAPTOR essentially grows linearly.

5.2 Improved TP query times

In Table 5, we also provide query times for one-to-one queries using the preprocessed transfer patterns. We consider two kinds of queries: (1) single-time one-to-one queries, with the aim to compute all optimal paths for a single given departure time from a given source to a given target station; (2) profile one-to-one[4] queries, with the aim to compute all optimal paths for all departure times in a given time range from a given source to a given target station. Each figure in Table 5 was computed as the average of 1000 queries, where for each query the source and target station was chosen independently and uniformly at random from the set of all stations. For the 7-day results, a single day[5] was chosen at random for each query. For the single-time queries, the departure time was chosen at random from that day. For the profile queries, the whole day was taken as a (24-hour) time range.

The single-time one-to-one queries work just as explained in [1]. All transfer patterns from the given source to the given target station are overlaid to form the so-called query graph. A time-dependent Dijkstra computation is executed on that query graph. Each arc evaluation on that query graph asks for a direct connection between two stations. A direct connection is one which possibly has intermediate stops, but where no change of vehicle occurs. We use the simple data structure from [1] for the efficient processing of direct-connection queries. In a nutshell, this is simply a list of all direct connections (think: bus/train lines) for each station. A lookup in that data structure takes a few microseconds per station on average.

Table 5 shows that the query graphs are small on average: about 100-200 nodes and arcs, depending on the setting and the number of days. Correspondingly, query times are very fast, ranging from 0.2ms for our simplest setting (EAT+few, 1 day) to 1.5ms for our most realistic setting (EAT+TR+many, 7 days). The table also provides the average number of optimal paths per query. Note that in our simplest setting (EAT+few, 1 day), the number is slightly below one, because with few footpaths some queries don't have a solution. In the same setting with many footpaths

[4] In principle, TP can also be used to solve profile one-to-many queries. But we here focus on one-to-one queries, which are those of interest in a route-planning system.

[5] We considered as one day the 24-hour time period from 4:00am of one calendar day until 3:59am of the next day.

(EAT+many, 1 day), this does not happen. This gives a hint at the practical significance of a sufficient number of footpaths.

The profile one-to-one queries are also implemented as a time-dependent Dijkstra computation on the same kind of query graph. The difference is that now each arc evaluation asks for *all* direct connections between two stations. This can be done with the same direct-connection data structure as above. However, we now have to manipulate with a larger number of labels at each node of the query graph. Table 5 shows that query times are still fast, ranging from 3.3ms in our simplest setting (EAT+few, 1 day) to 95.3ms for our most realistic setting (EAT+TR+many, 7 days). Note that for profile queries, the number of solutions grows about linearly with the number of days. Hence, also the query time grows about linearly with the number of days.

5.3 Improved TP space consumption

The direct-connection data structure (used to evaluate the query graph) can also benefit from the frequency-based compression. Here (like for RAPTOR), connections are grouped by trips and trips are grouped by lines. Then for every station a list is maintained which contains the incident lines as well the position of the station in this line. This allows to access all departures from a station belonging to a certain line efficiently by parsing through the list of trip departures at the specified position. If we now compress those trips as described in Section 2.2, synchronized trips are combined and the access times as well as the space consumption for the direct connection data structure decrease. For the GR-ALL instance, we observed an improvement in terms of space by an order of magnitude.

6. CONCLUSIONS AND FUTURE WORK

We presented a compression scheme for timetable data of public transportation networks where synchronized departures were joined into single frequency-based labels. We observed good compression factors when considering timetable information for a whole week. We designed a new kind of profile search based on frequency-labels which outperforms previous approaches in both space and time consumption. We would expect even better results for transit networks with a high degree of synchronized trips, as e.g. Tokyo or Shanghai, for which unfortunately we had no data access. For future work it would be interesting to incorporate delays in the model. One way to do this would be to allow negations in the frequency-labels, so e.g. from 8:00 to 16:00 every 15 minutes but not at 9:30 or not between 10:00 and 11:00 (like considered in [10]). Such negations could be beneficial anyway, as they might lead to an even better compression (if only single departures are missing in long arithmetic progressions). Finally, our frequency-Dijkstra is not custom-tailored for frequency-labels based on timetable data, but might also be applicable to other scenarios, e.g. considering synchronized signal transmissions or flows over time.

7. ACKNOWLEDGEMENT

This work was partially supported by a Google Focused Research Award on Next-Generation Route Planning.

8. REFERENCES

[1] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *European Symposium on Algorithms (ESA)*, pages 290–301, 2010.

[2] Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-robustness of transfer patterns in public transportation route planning. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, pages 42–54, 2013.

[3] Claudio Bettini and Sergio Mascetti. An efficient algorithm for minimizing time granularity periodical representations. In *Symposium on Temporal Representation and Reasoning (TIME)*, pages 20–25, 2005.

[4] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics (JEA)*, 17:4–4, 2012.

[5] Daniel Delling, Thomas Pajor, and Renato Fonseca Werneck. Round-based public transit routing. In *Workshop on Algorithms Engineering and Experiments (ALENEX)*, pages 130–140, 2012.

[6] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In *Symposium of Experimental Algorithmics (SEA)*, pages 43–54, 2013.

[7] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *Workshop on Experimental Algorithms (WEA)*, pages 347–361, 2008.

[8] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[9] Lenwood S. Heath. Covering a set with arithmetic progressions is NP-complete. *Information Processing Letters (IPL)*, 34(6):293–298, 1990.

[10] Romans Kasperovics, MH Bohlen, and Johann Gamper. Representing public transport schedules as repeating trips. In *Symposium on Temporal Representation and Reasoning (TIME)*, pages 54–58, 2008.

[11] Marc Niezette and Jean-Marc Stevenne. An efficient symbolic representation of periodic time. In *Conference on Information and Knowledge Management (CIKM)*, pages 161–168, 1992.

[12] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12:2–4, 2008.

[13] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 92:3–15, 2004.

[14] Ben Strasser and Dorothea Wagner. Connection scan accelerated. In *Workshop on Algorithms Engineering and Experiments (ALENEX)*, pages 125–137, 2014.