Frequency Data Compression for Public Transportation Network Algorithms

Hannah Bast and **Sabine Storandt**
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
79110 Freiburg, Germany
bast,storandt@informatik.uni-freiburg.de

Abstract

Timetable information in public transportation networks exhibit a large degree of redundancy; e.g. consider a bus going from station A to station B at 6:00, 6:15, 6:30, 6:45, 7:00, 7:15, 7:30, ..., 20:00, the very same data can be provided by a frequency-based representation as '6:00-20:00, every 15 minutes' in considerably less space. Nevertheless a common graph model for routing in public transportation networks is the time-expanded representation where for each arrival/departure event a single node is created. We will introduce a frequency-based graph model which allows for a significantly more compact representation of the network, resulting also in a speed-up for station-to-station queries. Moreover we will describe a new variant of Dijkstra's algorithm, where also the labels are frequency-based. This approach allows for accelerating profile queries in public transportation networks.

Introduction

Buses, trains, subways etc. often depart at fixed intervals from the respective stations. Hence the compressibility of timetable information is high as we can represent a list of such departure events, e.g. 7:00, 7:05, 7:10, ..., 12:00, also as triple (7:00, 12:00, 5min) containing the start and end time of the service and the frequency. In the following we will describe ways to improve the computation of fastest routes in public transportation networks based on this compressed representation.

Modelling a Frequency-Based Graph

A standard way to model timetable information into a graph is the *time-expanded* representation (as used e.g. in (Bast et al. 2010)). Here for every arrival/departure event at a station a node needs to be created, which results in a huge amount of data. In the *time-dependent* model the number of nodes/edges is significantly smaller, but the complexity of the departure events is now shifted into edge cost functions, which are costly to store and evaluate. Therefore both models are comparable in terms of space consumption and query times. We will now introduce an alternative graph model based on frequency-compression. Here we only create a node for each route per station (e.g. Bus 80 at main

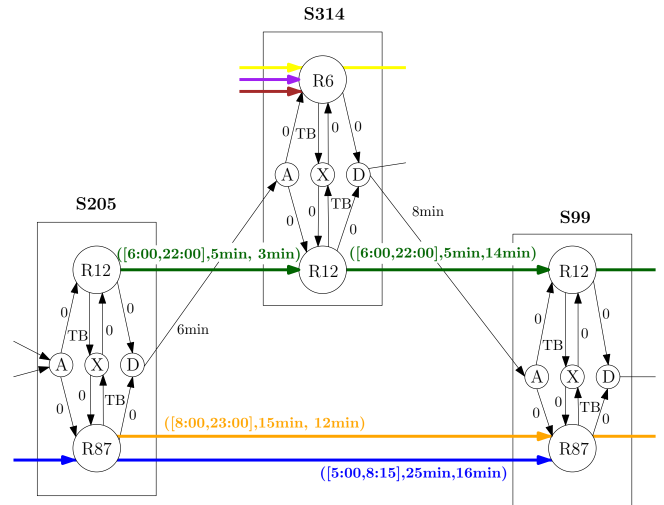


Figure 1: Cut-out of a frequency-based graph. 'A' and 'D' are walking (arrival/departure) nodes, 'X' are transfer nodes allowing to change between routes.

station) and connect them via frequency edges (FEs). Every FE (S, S') contains a quadruple $([a, b], f, c)$ with $[a, b]$ being the feasible interval in which all f seconds a certain vehicle departs, taking time c to get from station S to S'. The evaluation of such an edge for a certain time $t@S$ can be performed like this:

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

Our model also allows for including walking between stations and transfer buffers (TB), see Figure 1.

Timetable information is typically specified in a GTFS feed[1]. Here the possibility to provide frequencies is given, but not always used. So constructing our graph we are often faced with the following problem: Given a set of departure events $T = \{t_1, \cdots, t_k\}$ at a route node, compute the minimum set of frequency-edges to represent them all. This problem is known as *cover by arithmetic progressions* and was proven to be NP-complete (Heath 1990). Therefore

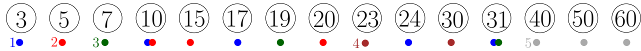
Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] https://developers.google.com/transit/gtfs/reference

	Manhattan	Toronto
#stations	1831	10891
#departure events	278920	1599746
#freqeuncy-edges	87505	315300
compression factor	3.2	5.1

Table 1: Heuristic frequency-graph construction.

we propose a heuristic strategy: At first, we sort T increasingly. Then we consider the first element t_1 and compute the longest possible arithmetic progression (AP) starting with t_1. We add this AP to our solution and mark all other elements in T covered by it. Now we repeat the approach with the next unmarked element. We do not exclude already marked ones from the set, but we give preference to the AP which covers the most unmarked elements. Example:



The resulting reduction of FEs can be found for two real-world examples in Table 1. Because a FE can be evaluated in constant time, the compression factor translates into the speed-up for station-to-station queries.

Profile Queries on Frequency Data

Profile queries play an important role e.g. in the construction of transfer patterns (Bast et al. 2010), a state-of-the-art speed-up technique for routing in transit networks. In a profile query, we are given a set of departure times T (e.g. all departures over the day) at a station S and want to know for each $t \in T$ the optimal connections to all other stations. Naively we could run a complete Dijkstra for every $t \in T$ from the 'A'-node of S, resulting in a runtime of $\mathcal{O}(|T|n\log(n) + |T|m)$. In practice this can be improved by considering the values in T from latest to earliest and storing along with every node the earliest arrival time assigned so far. If with an earlier departure time the earliest arrival time at some node v can not be improved, then v must not be further explored. But even in this scenario, if the basic data exhibits frequency-based departure times, we repeat the same set of operations again and again (only with a time shift). Hence we would like a single Dijkstra handling *all* these departure times at once. For this purpose we assign quadruples $([a, b], f, c)$ to the nodes instead of single values, with $[a, b]$ marking the interval of arrival times with frequency f, and c being the summed costs since the departure from S. The crucial task is adapting the edge relaxation to this new setting. So given a label $l = ([l_a, l_b], l_f, l_c)$ at node $u \in V$ and an edge $e = (u, v) \in E$ with $([e_a, e_b], e_f, e_c)$, the goal is to compute the respective label(s) at node v. We proceed in five steps (see Figure 2 for an example):

1. Compute $lcm = lcm(l_f, e_f)$ to get the lowest common frequency.
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 $start = l_a + \lfloor (e_a - l_a)/l_f \rfloor \cdot l_f$. If this would result in the $start$ value exceeding l_b we reset it to l_b.
3. Compute for the first $steps = lcm/l_f$ relevant departure times $\{start, start + l_f, \cdots, start + (steps - 1) \cdot l_f\}$ (restricted to values $\leq l_b$) the explicit edge costs $cost$ and arrival times arr at v. Store the respective values in a vector A.
4. Parse through A and remove arrival times, which occur more than once (of course keeping the one with the 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 A, create a new label l' at node v, with $l'_a = arr, l'_b = l'_a + \lfloor (\min(l_b, e_b) - l'_a + cost)/lcm \rfloor \cdot lcm, l'_f = lcm$ and $l'_c = l_c + cost$.

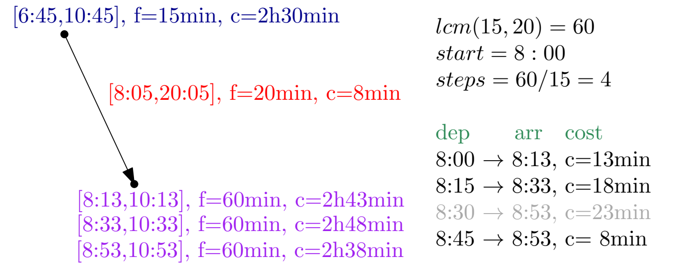


Figure 2: Example of a frequency-edge relaxation.

So the runtime of an edge relaxation is in $\mathcal{O}(\max(l_f, e_f))$, and at most lcm/l_f new labels are created at v (but obviously never more than the number of departure events at the source station). For walking edges and edges adjacent to transfer nodes (which can always be used immediately), it is not necessary to perform $lcm = lcm(l_f, e_f) = l_f$ expanding steps. Instead the relaxation leads to exactly one new label at v which can be determined directly as $l' = ([l_a + e_c, l_b + e_c], l_f, l_c + e_c)$.
Of course not all labels created at the target node must represent (temporary) optimal connections. Therefore the resulting labels have to be pruned properly and joined if possible to reduce the space consumption.

The respective runtimes of our approach for an implementation in C++ on an Intel i5-3360M CPU with 2.80GHz and 16GB RAM can be found in Table 2.

	Madrid	Artificial
#stations / #departures	4653 / 1.20M	750 / 0.23M
naive	26.83	28.05
frequency-based	5.98	0.49
speed-up	4.48	56.39

Table 2: Averaged runtime for 100 profile queries from random source stations in seconds.

References

Bast, H.; Carlsson, E.; Eigenwillig, A.; Geisberger, R.; Harrelson, C.; Raychev, V.; and Viger, F. 2010. Fast routing in very large public transportation networks using transfer patterns. In *Algorithms–ESA 2010*. Springer. 290–301.

Heath, L. S. 1990. Covering a set with arithmetic progressions is NP-complete. *Inf. Process. Lett.* 34(6):293–298.