Karlsruhe Institute of Technology

Delay-Robust Stochastic Routing In Timetable Networks

Diploma Thesis of

Ben Strasser

At the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Dorothea Wagner
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dipl.-Inform. Julian Dibbelt
Second advisor:	Dipl.-Inform. Thomas Pajor

Duration: 1.2.2012 – 31.7.2012

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Ben Strasser
Karlsruhe, 31st of July 2012

Contents

1. Introduction

Context

Computing optimal routes in various types of networks has been a very active research topic in the past decade. Routing in road networks and in public transit networks are two of the most prominent types [1]. Many algorithms have been proposed to solve the problem on roads and some even manage to answer queries within less than a micro second [2]. One has not been able to achieve similar query times on train networks [3, 4]. The problem of computing a fastest route has been the focus of many papers but also extensions have been considered. Profile queries [5] are one of them and consist of not only computing an optimal route for one departure time but for a whole departure interval. The result is a function that maps a departure time onto the travel time or the arrival time. Multi-criteria routing [6, 7] is another extension and consists of not only computing a single route that is optimal with respect to one criterion but to compute a set of different routes such that no route in the set is superior to another route with respect to every criterion. The routes in this set are called *pareto-optimal*. The considered criteria include minimizing the train (or buses, ferries, etc) changes, minimizing the train ticket prices and maximizing the reliability [7] of journeys within public transit networks. Trains in public transit networks are inevitably delayed some times. The reliability is a measure of how likely it is that every transfer in a journey works.

Motivation

Reliability is the main topic of this thesis. Most of the previous work (with [8] and [9] being two notable exceptions) considers a journey to be a sequence of trains connected by train transfers. A journey is reliable if no transfer is likely to fail. A major shortcoming of this approach is that all bets are off if a transfer does not work. Indeed, it is possible that no good alternative trains depart at that stop resulting in a major delay at the journey's arrival. Even a transfer with a high success rate may not be useful, if the costs that occur when it does fail are excessive. Another downside of the approach is that it might miss good journeys that have low reliability but have a lot of redundant alternative trains. For example trams within cities may have relatively high delays and thus the reliability of one specific sequence of trams may be low, but missing one is rarely an issue because the next tram of the same line departs only a few minutes afterwards. Journeys exist that are nearly certain to not succeed as originally planned but that contain enough redundancy to make it unlikely that the delay at the user's target stop is significant. A further problem

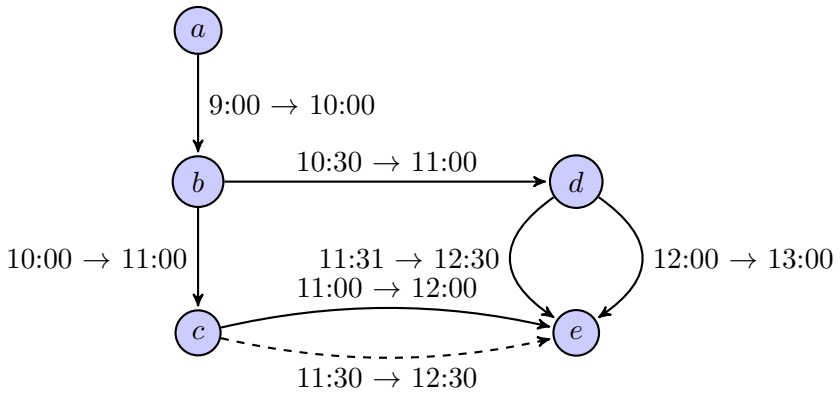


Figure 1.1: There are five stops: a, b, c, d and e. The arcs represent trains with their departure and arrival times. The user is at 9:00 at a and wants to get to e. The success probability of a change depends on the waiting time of the user. If he has no waiting time then the transfer works with a probability of 0.5, one minute waiting time has a 0.6 success rate and 30 min is always enough (i.e. has a success rate of 1.0). The solid lines are the trains inside some pareto-optimal path. The criteria considered are earliest arrival time and maximum reliability. Note that there is no path in the set that contains an alternative train if the change at c fails.

of journeys that consist of only a sequence is that, without access to a train schedule, they leave the user without any hints as to what train to take if a transfer fails.

Multi-criteria routing mediates the problems somewhat as it presents the user with alternatives. However, he only has a choice at the departure. It is possible that the computed journeys contain enough information to allow the user to recover from a failed transfer but this is not an optimization criterion. Figure 1.1 illustrates a situation in which the user can not always recover from a transfer because the set of routes computed does not contain the train he needs.

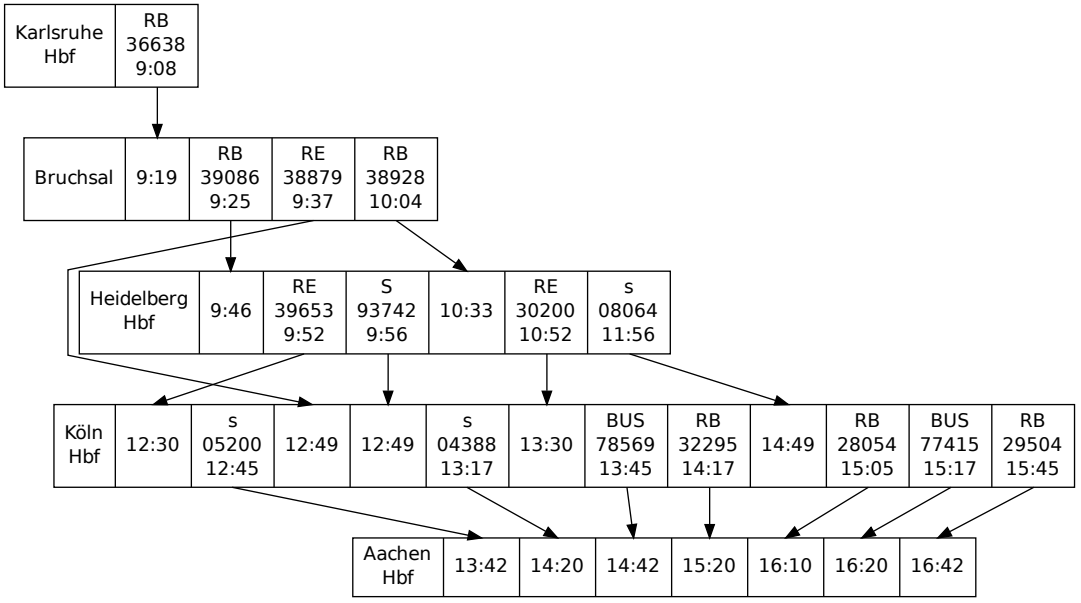
Problem Settings Considered

We propose to compute decision graphs that guide the user from a start stop and start time to a target stop. An example of such a graph is shown in Figure 1.2. For every stop we list a number of possible connecting trains. The probability that the user gets a specific one may be low but it is very likely that the user will get some connecting train. It is possible to define success probabilities for every transfer and this allows us to define an expected arrival time given a decision graph. We consider the problem of computing a decision graph with a minimum expected arrival time (M.E.A.T.) in public transit networks. The objective is to compute decision graphs that represent journeys that are robust to delay induced perturbations of the timetable and can be fitted onto a sheet of paper that can be handed to the user before he starts his journey.

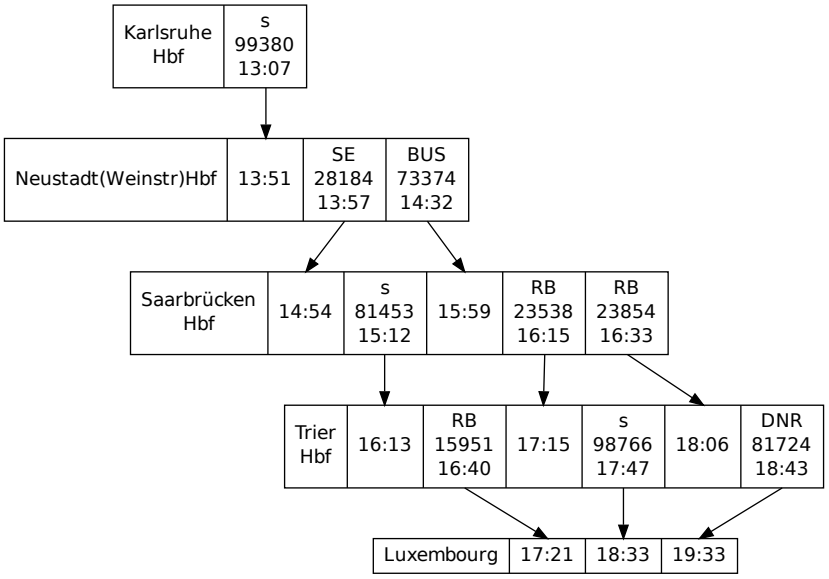
The algorithm we developed can also be applied to routing without delays and therefore the earliest arrival problem and the profile problem are also extensively considered in this thesis.

Related Work

The problem of computing a shortest path within a graph with random edge weights has been studied in [10, 11, 12]. In [8] the concept is expanded and an approach very similar



(a)



(b)

Figure 1.2: Two example route routes departing at Karlsruhe. For each stop the user is presented with a number of possible connections. The connections are ordered by their arrival and departure times. The user should take the first connection that he can get. As long as no delay is excessive the user is certain to get to his destination within reasonable time.

to ours is explored to determine hyper paths (i.e. paths with branches, very similar to our decision graphs) with a minimum expected length on graphs. The main difference to their work is that we exclusively focus on public transit networks and propose more detailed stochastic models that, we hope, better capture the delays of trains.

Another approach is presented in [9, 13]. In these works no decision graph is computed but a complete strategy. A strategy maps a location in space and time onto the next action that the user should take. They compute a complete strategy that also considers the locations that the user can not possibly get to from his starting location. However, a strategy that is only partially computed is very similar to our decision graphs. A major difference from our approach is that they do not minimize an expected path length but maximize the probability of the user arriving before a given desired arrival time. If the user wants to arrive early then this results in very risky journeys but if he has time then the algorithm distributes this time across the journey to maximize the reliability. Note that the algorithms developed are not tailored for public transit networks.

There have been several approaches to the earliest arrival problem that consist of reducing the public transit network to a graph. In the time-dependent approach the edge weights are allowed to be time-dependent and in the time-expanded approach they must be constant. The most common transformations are described in [14, 5]. Once the network is transformed either Dikstra's algorithm [15] or a time-dependent adaptation [16, 17] is applied to the graph. There have been other approaches that directly operate on the network structure [6] and some even make use of extensive preprocessing to speedup queries [18, 3, 4, 19, 20]. Profile queries has been investigated in [5, 6] resulting the PSPCS algorithm that allows to quickly compute profile function and makes efficient use of multiple cores.

Our Contribution

We show that it is possible to reduce earliest arrival routing in periodic networks onto aperiodic networks. We present an algorithm to group trips into a minimum number of routes and heuristic improvements to it. We introduce various relaxation operations that operate directly on the public transit network (instead of first constructing a graph) to solve some of the various variants of the earliest arrival problem. We present algorithms to solve the earliest arrival and the profile problem with and without minimum change times and with and without footpaths and the combination of both. We show that our profile algorithm with minimum change times but without footpaths has a worst case running time linear in the number of connections. A distinctive property of our algorithms is that they do not use a priority queue. We present a number of delay models and present an algorithm to efficiently compute delay robust journeys for one of them.

Outline

Chapter 2 focuses on network definition and transformations. In that chapter we describe how to compute a minimum number of routes given a network and how to reduce periodic networks onto aperiodic networks. In Chapter 3 we introduce the Connection-Scan algorithm for the earliest arrival problem and propose a number of optimizations. The next Chapter 4 focuses on solving the profile problem using the approach introduced in the previous chapter. Chapter 5 focuses on modeling delays using stochastic methods. We introduce several models that vary in their degree of realism. Chapter 6 focuses on efficiently computing journeys in one of the models introduced in the previous chapter. Section 6.1 sums up the precise problem setting discussed in that chapter. If one is only interested in the algorithm then Section 6.1 is probably a better starting point than Chapter 5. Finally, in Chapter 7 we evaluate the various algorithms introduced on real world data.

2. Timetable Networks

In this section we first precisely define the timetable networks used in this thesis. Afterwards we formally define journeys. Finally, we work out some of the differences to shortest path problems on weighted graphs. The timetable can model the departures and arrivals of any vehicle that behaves similar to trains (for example buses, ferries, etc). For simplicity we refer to all of them as trains.

2.1 Formal Definition

A *timetable network* is a tuple (S, C, F) where S is the set of *stops,* C the set of *connections* and F the set of *footpaths.* A stop describes some place where trains halt. For every stop the timetable contains a positive integer the *minimum change time* $s_{\text{minchange}}$. This is the time that the user needs to change trains at this stop. A connection defines a train that goes from one stop to another without halting at any intermediate stop. For every connection c the timetable contains a $(c_{\text{depstop}}, c_{\text{deptime}}, c_{\text{arrstop}}, c_{\text{arrtime}}, c_{\text{prev}}, c_{\text{next}}, c_{\text{maxdelay}})$-tuple where c_{depstop} is the *departure stop*, c_{deptime} the *departure time*, c_{arrstop} the *arrival stop* and c_{arrtime} the *arrival time.* The time values must be integers. Further we require without loose of generality that they must be positive. We require $c_{\text{deptime}} < c_{\text{arrtime}}$ and $c_{\text{depstop}} \neq c_{\text{arrstop}}$ for the timetable to be valid. Further the connections severed by the same train are interlinked using the *previous connection* c_{prev} and the *next connection* c_{next}. (Think of this construct as a linked list.) This list is called a *trip.* We denote the connections in a trip tr of length n as $c_1 \dots c_n$. Formally the following conditions have to be met for every connection c:

- The trip is a sequence, i.e., $c_{\text{next}} = \bot$ or $(c_{\text{next}})_{\text{prev}} = c$ and $c_{\text{prev}} = \bot$ or $(c_{\text{prev}})_{\text{next}} = c$
- No trip arrives earlier than it departs, i.e., $c_{\text{next}} = \bot$ or ($c_{\text{arrstop}} = (c_{\text{next}})_{\text{depstop}}$ and $c_{\text{arrtime}} < (c_{\text{next}})_{\text{deptime}}$)
- Every trips starts somewhere, i.e. every trip contains a connection with $c_{\text{prev}} = \bot$.

The timetable also contains a maximum delay value c_{maxdelay}. We parametrize our synthesized delay distributions solely on this value. In Section 2.4 we describe how this value is used. Trips can be grouped in *routes.* All trips in a route serve the same stops in the same order and fulfill the FIFO-property. Formally, this means that for two trips $x_1 \dots x_n$ and $y_1 \dots y_m$ within the same route the following must hold:

- They must have the same length, i.e., $n = m$

- They serve the same departure stops, i.e., $\forall i : (x_i)_{\text{depstop}} = (y_i)_{\text{depstop}}$
- They serve the same arrival stops, i.e., $\forall i : (x_i)_{\text{arrstop}} = (y_i)_{\text{arrstop}}$
- The FIFO-property, which basically states that the user never arrives earlier by taking a later trip within the same route, can be formalized as partial order relation $\preccurlyeq$. Formally, it is defined by

$$x \preccurlyeq y \iff \forall i : (x_i)_{\text{deptime}} \leq (y_i)_{\text{deptime}} \wedge (x_i)_{\text{arrtime}} \leq (y_i)_{\text{arrtime}} .$$

Finally, a train network also contains footpaths. These are time independent paths that the user may walk to get from one stop to another. For every footpath f the timetable contains a $(f_{\text{depstop}}, f_{\text{arrstop}}, f_{\text{dur}})$-tuple where f_{depstop} is the departure stop and f_{arrstop} is the arrival stop and f_{dur} is the integer time that the user needs to walk the path. We require that $f_{\text{depstop}} \neq f_{\text{arrstop}}$ and $0 < f_{\text{dur}}$. We require that there is at most one footpath between two stops. By definition it is never sensible to walk two footpaths after another. If this is not the case in the actual data then the transitive closure must be considered.

2.2 Modeling Big Stations

In real world train networks there exist large train stations with lots of tracks. The tracks are often very close to each other but queries are not realistic if no change time for the user is taken into account. The time needed varies depending on what track the user arrives on and where his next train departs. We consider two ways to model this that are very similar to the approached used by Pyrga et al. [14].

The first easy way consists of modeling the station as a single stop with a minimum change time that is sufficiently high to account for any possible track change. The advantage of this model is that it is small and efficient. The drawback is that change times in the query results may be bigger than necessary.

The second way is to model every track as a stop with a minimum change time of zero. For every pair of tracks in a station a footpath is added. Note that this approach results in a directed clique per station in the footpath graph. In the previous modeling the minimum change time must be large enough to handle the worst case train change. This can be larger than what is needed for a specific change. Footpaths allow a more detailed model and therefore do not have this problem. The drawback is that the network instances get a lot bigger. Further, such precise data about the change times does not exist for every network. Also, there are companies that do not plan on what track a train arrives but just pick one that happens to be free and announce it some time before the train arrives. In this case the second, finer model obviously does not work.

2.3 Minimum Number of Routes

In this section we propose an algorithm to compute a minimum number of routes given a set of trips. We further provide heuristic improvements. Our algorithm works in two phases. In the first it partitions the trips according to their stop sequence and ignores the actual departure and arrival times. In the second phase trips with the same stop sequence are distributed to a minimum number of routes such that the FIFO-property is met.

In the first phase we partition the trips into sets such that all trips in a set serve the same stops in the same order. It follows directly from the definition that no route can exist that contains trips from two different sets. In the second phase we consider all these sets independently. We can compare all the trips in one set using the strict FIFO-order

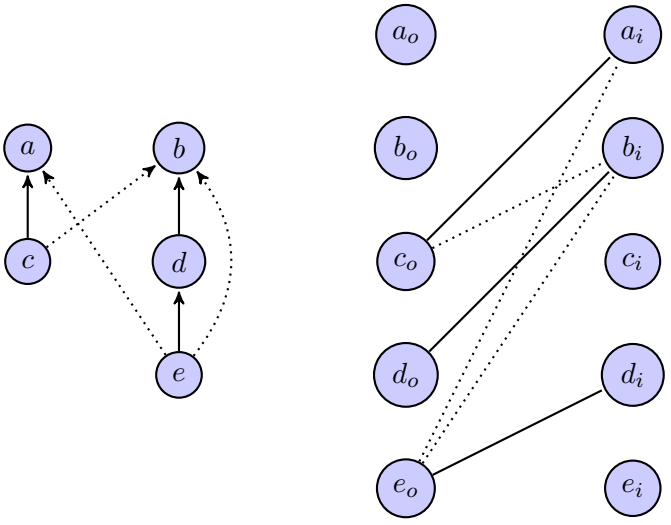


Figure 2.1: A path cover problem reduced to a bipartite matching. Dotted lines represent arc/edges not in the matching.

relation $\prec$[1]. Every strict partial order yields a directed acyclic graph[2]. Valid routes correspond to paths within this graph. We want to compute a minimum number of paths such that every node is contained in exactly one path. Note that paths with only one node are valid. This problem is known as the minimum path cover problem. On undirected graphs this problem is NP-hard as it can be used to determine if a Hamiltonian path exists which is already NP-hard [21]. However on directed graphs the problem can be solved using a reduction to maximum bipartite matching [22] in polynomial time. The basic observation is that every node must have at most one successor and at most one predecessor. Another way of thinking of this is that every node v has one in-slot v_i where a path may enter and one out-slot v_o where a path can leave. The goal is to match as many in-slots to out-slots as possible. We construct a bipartite matching problem whose two node sets are formed by the set of all in-slots and the set of all out-slots. See figure 2.1 for an example. For every two trips u and v with $u \prec v$ we add an edge $\{u_o, v_i\}$ to the matching problem. Given a matching we can construct a path cover as follows. Start with the path cover where every node is contained in a path that only contains this one node. For every edge in the matching concat the two corresponding paths. With every matched edge considered the number of paths decreases thus by one. It therefore holds that $n - m$ paths are constructed where n is the number of trips and m the number of matched edges. Analogously given a path cover with r paths it is possible to construct a matching with $n - r$ edges. This shows that a maximum matching corresponds to a minimum path cover. A bipartite matching problem can be solved within $O(n^3)$ running time using a push-relabel flow algorithm [23]. As it dominates the running time of our algorithm this is also its worst case running time.

In realistic networks the number of routes is often significantly smaller than the number of trips. Frequently the trips with the same stop sequence even form a single route. This can be exploited to construct heuristics. We first sort the trips by their departure time at the first stop and obtain a list $\mathrm{tr}_1 \dots \mathrm{tr}_n$. We check whether this order already is a single route. If it is we have guessed an optimal solution and can skip the computation of the bipartite matching. If this order does not yield an optimal solution it provides a good initial matching. If $\mathrm{tr}_p \prec \mathrm{tr}_{p+1}$ holds then we add $\{(\mathrm{tr}_p)_o, (\mathrm{tr}_{p+1})_i\}$ to the initial matching.

[1] We define $u \prec v$ formally as ($u \preceq v$ and not $v \preceq u$).
[2] The graph is also transitive and dense but we do not need this properties.

2.4 Modeling Delays

A crucial part of a stochastic network model are the random variables that model the delays. Unfortunately we do not have any real world data at our disposition. We therefor determine c_{maxdelay} using a heuristic and then evaluate our implementations using a number of distributions that are solely parametrized on c_{maxdelay}. We describe the exact distribution when describing the implementations as they vary. See Section 6.3 for precise details about how c_{maxdelay} is interpreted. In this section we are going to describe how we determine c_{maxdelay}.

We first fix an upper bound x for the maximum delay in the train network. We use 30 minutes but other values may also be good. Recall that we do not try to model large delays that are a symptom of perturbations that seem to be too large to model using stochastic approaches and therefore larger values for this upper bound are not a good choice. Next we compute the routes in a train network. For every connection c we then determine the waiting time y until the next connection in the same route departs. More formally y is the time between c_{arrtime} and the arrival time of the corresponding connection in the next trip of the same route. If the connection is part of the latest trip then we set $y = +\infty$. We define $c_{\text{maxdelay}} = \min\{x, y\}$. The intuition for this choice is that routes with a high frequency (i.e. a small y) normally have a smaller delay and therefore should have a smaller maximum delay. Note that on intercity train networks for most connections $c_{\text{maxdelay}} = x$ holds and for must intracity train networks $c_{\text{maxdelay}} = y$. Our heuristic is sufficiently robust to handle mixed networks.

2.5 Common Problems Settings in Timetable Networks

In this section we formally define journeys and describe a number of commonly considered problems involving them.

A *journey* in a timetable network represents a way that a user may get from a start stop to a target stop. Formally, it is a sequence $p_1 \dots p_n$ of connections and footpaths such that for any two consecutive elements p and p' the user can get from the arrival of p_i to the departure of p_{i+1} or both are connections and belong to the same trip, i.e., ($p_{\text{arrstop}} = p'_{\text{depstop}}$ and $p'_{\text{deptime}} - p_{\text{arrtime}} \geq (p_{\text{arrstop}})_{\text{minchange}}$) or $p_{\text{next}} = p'$. Further we require that no two consecutive elements are footpaths. This definition avoids having the time independent shortest path problem as a subproblem. Recall that, as already described in Section 2.1, if consecutive footpaths are needed then the transitive closure must be computed first. The departure stop of a journey is $(p_1)_{\text{depstop}}$, its arrival stop is $(p_n)_{\text{arrstop}}$, its departure time is $(p_1)_{\text{deptime}}$ and its arrival time is $(p_n)_{\text{arrtime}}$.

Common journey problems considered are the earliest arrival problem and the profile query problem. The earliest arrival problem comes in two versions: the one-to-one problem and the one-to-all problem. In the one-to-all setting a start stop $s_{\text{startstop}}$ and a start time $t_{\text{starttime}}$ is given and the problem consists of finding a function d that maps every stop s onto the minimum arrival time over all journeys departing at the start stop after $t_{\text{starttime}}$ arriving at the stop s. In the one-to-one setting a target stop $s_{\text{targetstop}}$ is further given and the problem consists of only computing $d(s_{\text{targetstop}})$.

The profile query problem comes in three variations: all-to-one, one-to-one and one-to-all. In the one-to-one setting a start stop and a target stop is given. The problem consists of computing a profile function $f : \mathbb{R} \to \mathbb{R} \cup \{+\infty\}$ that maps a departure time at the start stop onto the earliest arrival time at the target stop or $+\infty$ if no such journey exists. In the all-to-one setting only a target stop is given and the problem consists of computing a function that maps a stop s onto the profile function from s to the target stop. Analogously only that start stop is given in the one-to-all variant.

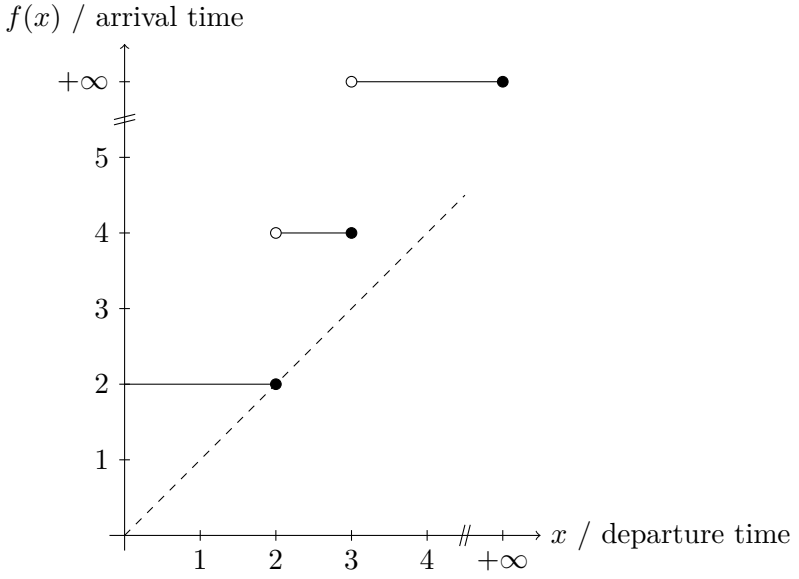


Figure 2.2: The plot of an example profile function. The black points are the jumps. The jump sequence is $(2,2)(3,4)(+\infty,+\infty)$.

Note that a profile function is always a non decreasing step function. We refer to the moments at which it changes as (d,a)-jumps. We call d the departure time and a the arrival time. A profile function can be represented by a sequence of jumps $(d_1,a_1)\dots(d_n,a_n)$. We require that $d_i < d_{i+1}$ holds. As the function is non decreasing $a_i < a_{i+1}$ also holds. Evaluating $f(x)$ consists of searching for the jump with the minimum departure time that is still greater than x and returning the corresponding arrival time, i.e., find the minimum i such that $x \leq d_i$ and set $f(x) = a_i$. Note that on finite networks $f(+\infty) = +\infty$ must hold and therefore a $(+\infty,+\infty)$-jump must exist. Further the arrival time may never be smaller than the departure time and therefore we only consider functions that satisfy $f(x) \geq x$. Figure 2.2 shows an example of a profile function.

In Chapter 5 we further discuss a number of new problem settings that consider random delays.

2.6 Eliminating Minimum Change Times And Footpaths

Most algorithms are simpler when only operating on simple networks with no minimum change times or footpaths. To solve the problems on general networks either the algorithms must be adapted or the networks transformed. In this section we consider network transformations. Note, however, that adapting the algorithms generally results in better performance.

We first discuss eliminating minimum change times and assume that there are no footpaths. The core idea is to expand every trip of length n into $O(n^2)$ many connections. The transformed network contains the same stops but all have a minimum change time of 0. For every trip $c_1c_2\dots c_n$ the transformed network contains for every $1 \leq i \leq j \leq n$ the connection c'_{ij} starting at $(c_i)_{\text{depstop}}$ departing at $(c_i)_{\text{deptime}} - ((c_i)_{\text{depstop}})_{\text{minchange}}$ to $(c_j)_{\text{arrstop}}$ arriving at $(c_j)_{\text{arrtime}}$. We show that there is a one-to-one correspondence between the journeys in the original network and the transformed one. Consider some journey $c'_{1,i}c'_{i+1,j}\dots c'_{k,n}$ in the transformed network. Replacing all the connections with the corresponding connection sequence in the original network yields a corresponding journey $c_1\dots c_i\dots c_j\dots c_k\dots c_n$ in the original network with the same end stops and times. Now

consider some journey $c_1 c_2 \dots c_n$ in the original network. We group the connections that belong to the same trip and obtain a sequence of subsequences

$$(c_1 c_2 \dots c_p) \dots (\dots c_{i-1})(c_i c_{i+1} \dots c_j)(c_{j+1} \dots) \dots (c_q \dots c_n)\,.$$

For every trip $c_i c_{i+1} \dots c_j$ the transformed network contains a connection c'_{ij}. The journey corresponds to the $c'_{1p} \dots c'_{ij} \dots c_{qn}$ journey in the transformed network. As these two operations are inverse there is a one-to-one correspondence.

A similar one-to-one correspondence between networks that allow both minimum change times and footpaths and networks with only minimum change times can not exist. The reason is that we allow journeys that only consist of a single footpath and therefore are time independent. Consider some footpath f then for any start time t a journey exists that departs at f_{depstop} at t and arrives at f_{arrstop} at $t + f_{\text{dur}}$. These are infinitely many arrival times as there are infinitely many t. If a one-to-one correspondence existed then there would be for every journey using only f a connection-only sequence in the transformed network that would also arrive at precisely $t + f_{\text{dur}}$. As a consequence there would have to be an infinite number of incoming connections at f_{arrstop} as the arrival time is given by the arrival time of the last connection in the journey. This is not possible in a finite network.

There are several ways to fix this problem. The key ingredient is to forbid single footpath journeys and handle them in a post- or preprocessing step. To get a one-to-one correspondence we forbid all journeys that start with a footpath, which is more then just forbidding single footpath journeys. The connections c'_{ijf} in the transformed network do not only represent a sequence of connections $c_i \dots c_j$ in the original network but also optionally a subsequent footpath f. We use the $\perp$ symbol to indicate no footpath. The transformed network contains for every connections c_i and c_j in the same trip a connection $c'_{ij\perp}$. Further for every footpath that departs at $(c_j)_{\text{arrstop}}$ there is also a connection c'_{ijf}. Similar to the previous construction it is possible to expand every journey in the transformed network into one in the original network. The elements of every journey that does not start with a footpath in the original journey can be grouped in a unique way to form a journey in the transformed network. These operations are inverse and therefore a one-to-one correspondence exists. A similar construction is also possible that forbids terminal footpaths.

2.7 Periodic Networks

Up to now we only considered finite networks and this will remain the main focus of this thesis. However, in this section we briefly consider periodic networks with an infinite but recurring set of connections. This concept can be useful to model timetables that for example behave exactly the same way on each day. There are finitely many connections in one day but as it is repeated indefinitely the network has infinitely many connections. We show that it is possible in some cases to reduce periodic networks onto finite networks. It is very important to note that this reduction only works for earliest arrival problems. The problems introduced in Chapter 5 can not be reduced in this way as Section 5.3 shows. Given a periodic infinite network we construct a finite network such that all earliest arrival queries with a start time in the first period produce the same result. As the infinite network is periodic it is possible to reduce every query to a query in the first period.

In a periodic networks all of the connection recur every period, i.e., if Π is the period and a connection c departs at c_{deptime} then another connection will also depart at $c_{\text{deptime}} + k\Pi$ for every $k \in \mathbb{Z}$ and analogously for the arrival times. The set of footpaths and the set of stops is finite. The set of connections is infinite but can be represented by the finite set of

connection departing in the first period. Given an infinite periodic network with period Π the goal is to compute a finite connection subset such that every earliest arrival query with a departure time in $[0, \Pi)$ results in the same journey no matter if it is computed in the original network or in in the transformed one.

The connection subset we construct contains all the connections that depart in the first period, i.e., $[0, \Pi)$ and additionally all the connections needed to answer a finite set Q of one-to-all earliest arrival problems. Given a one-to-all query it is sufficient to include for every target stop all the connections used by some earliest arrival journey to this target stop. As every journey is finite and the number of stops is finite only finitely many connections need to be included.

Consider a fixed target stop $s_{\text{targetstop}}$ and two start stops $s_{\text{startstop}}$ and $s'_{\text{startstop}}$ with departure times $t_{\text{starttime}}$ and $t'_{\text{starttime}}$. Suppose that an earliest arrival journey j from $s_{\text{startstop}}$ needs to wait at $s'_{\text{startstop}}$ at $t'_{\text{starttime}}$ then the tail of j is an earliest arrival journey from $s'_{\text{startstop}}$. If this was not the case then j would not have been an earliest arrival journey as it can be improved by replacing its tail. We use this propriety to construct a wall Q of one-to-all instances at the end of the first period that every earliest arrival journey must pass. There are 3 ways that a journey may pass the period border as illustrated in Figure 2.3.

1. The first way consists of the user waiting at a stop. To handle this situation Q includes all (s, Π) queries.
2. The second way consists of the user sitting in a train, that is driving at the moment that the period border is crossed. In this case the journey must include a connection c with $c_{\text{deptime}} < \Pi$ and $c_{\text{arrtime}} > \Pi$. For every place where the user may exit this train we add a query to Q. More formally we add all the $(d_{\text{arrstop}}, d_{\text{arrtime}})$ queries where d is a connection in the same trip as c and comes after c.
3. The third way consists of the user sitting in a train, that is waiting at a stop. In this case the journey must contain a connection c that departs after the period border and that has a previous connection that arrives before the border, i.e., $c_{\text{deptime}} \geq \Pi \wedge c_{\text{prev}} \neq \perp \wedge (c_{\text{prev}})_{\text{arrtime}} < \Pi$ must hold. As in the previous case we add a query for every place where the user may exit the train.

2.8 Time-Expanded Approach

In this section we describe how the earliest arrival problem can be reduced to a shortest path problem on weighted directed graphs. The basic idea is to transform the network instance into a graph instance and then to use a shortest path algorithm. This is called the time-expanded approach has been surveyed by Pyrga et al. [14]. We do not directly use this transformation but there are some similarities to our algorithms as discussed at the end of Section 2.9. We only describe the basic variant that assumes that no minimum change times nor footpaths exist. Refer to the paper for transformations that takes these into account.

An *event* is a (t, s)-pair where t is a moment in time and s a stop such that a connection exists that arrives at or departs from moment t. These pairs form the nodes of the transformed graph. The arcs of the transformed graph are made up of so called *stay arcs* and *train arcs*. Stay arcs model the user standing at a stop and waiting. For every stop s there is a stay arc between every (t, s)-pair and (t', s)-pair if no event (t'', s) exists that is between them, i.e. $t < t'' < t'$. A stay arc is weighted by the time difference $t' - t$. For every connection c a train arc is constructed between the events $(c_{\text{deptime}}, c_{\text{depstop}})$ and $(c_{\text{arrtime}}, c_{\text{arrstop}})$ weighted with $c_{\text{arrtime}} - c_{\text{deptime}}$. Figure 2.4 illustrates this graph

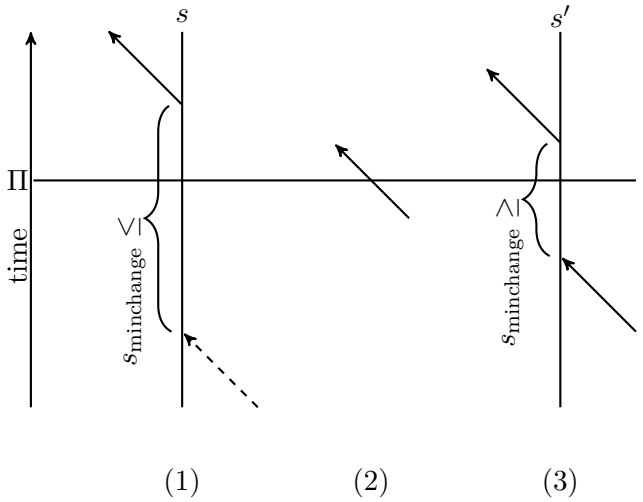


Figure 2.3: The three ways a journey may cross the period border. Arrows are connections. The vertical axis is the time. The horizontal axis represents the stops. Dashed and solid arrows may belong to different trips.

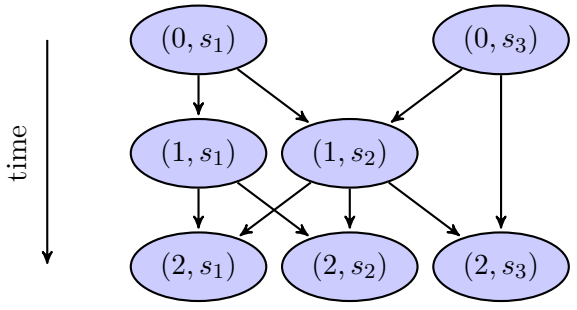


Figure 2.4: An expanded graph for a network with three stops s_1, s_2 and s_3 and five connections $c_1 \dots c_5$.

Algorithm 2.1 one-to-all variants of Dijkstra's algorithm for graphs and timetables

Input: graph and source node s;
Output: one-to-all shortest path length d;
$d(v) \leftarrow +\infty$ for all nodes v;
$d(s) \leftarrow 0$;
$q(v) \leftarrow false$ for all nodes v;
clear queue;
push s with key $d(s)$;
while *queue not empty* **do**
  pop min-key node x;
  $q(x) \leftarrow true$;
  for *all out-arcs* (x, y) **do**
    relax arc (x, y);
    if $q(y) = false$ **then**
      push y with key $d(y)$ or decrease it's key;

Input: network, start stop $s_{\text{startstop}}$ and start time $t_{\text{starttime}}$;
Output: one-to-all earliest arrival d;
$d(v) \leftarrow +\infty$ for every stop v;
$d(s_{\text{startstop}}) \leftarrow t_{\text{starttime}}$;
$q(v) \leftarrow false$ for all nodes v;
clear queue;
push $s_{\text{startstop}}$ with key $d(s)$;
while *queue not empty* **do**
  pop min-key stop x;
  $q(x) \leftarrow true$;
  for *all* $c \in C$ *with* $c_{\text{depstop}} = x$ **do**
    relax connection c if relaxable;
    if $q(c_{\text{arrstop}}) = false$ **then**
      push c_{arrstop} with key $d(c_{\text{arrstop}})$ or decrease key;

construction. Note that by construction every path from a a node (t_1, s_1) to a node (t_2, s_2) has weight $t_2 - t_1$.

Given a start stop $s_{\text{startstop}}$, a start time $t_{\text{starttime}}$ and a target stop $s_{\text{targetstop}}$ an earliest arrival journey can be computed using Dijkstra's algorithm [15] on the transformed graph. First, find the next event $(t, s_{\text{startstop}})$ after the start time, i.e. with minimum t such that $t \geq s_{\text{starttime}}$. Then start the search from this node. Abort once a node $(t', s_{\text{targetstop}})$ is settled. The earliest arrival time is t' as no node $(t'', s_{\text{targetstop}})$ with $t'' < t'$is reachable. If it was then the corresponding path would be shorter than the one to $(t', s_{\text{targetstop}})$ which can not be the case because the first path found by Dijkstra's algorithm is guaranteed to be the shortest.

2.9 Comparison of Relaxation in Graphs and Timetable Networks

It seems intuitive that there is some correspondence between the earliest arrival problem in timetable networks and the shortest path problem in directed weighted graphs. In this subsection we try to identify a set of basic building blocks that make up many shortest path algorithms and to find counterparts for these blocks in the network setting. We then take a specific shortest path algorithm and swap the basic build blocks and investigate whether the resulting algorithm can be used as a basis to develop an algorithm with similar properties that operates directly on networks. It is very important to note that the idea is not to add a problem instance transformation preprocessing phase to existing algorithms. This is the main difference to the time-expanded approach. We do not transform the network into a graph. We operate directly on the network. We transform the algorithms. The main goal of this section is to give a basic intuition about what constructs can be ported from graphs onto timetable networks.

In the graph setting there are nodes, arcs and edge weights. In the timetable setting there are stops, connection and departure and arrival times[3]. The nodes correspond to stops and the arcs to connections. The counterpart of an arc weight is the difference between

[3]We ignore footpaths and minimum change times for the scope of this section as they are not crucial for the intuition. In the next Chapter we describe how to handle them.

Algorithm 2.2 one-to-all shortest path algorithm

Input: directed acyclic graph $G = (V, E)$ and source node s;
Output: one-to-all shortest path length d;
$d(v) \leftarrow +\infty$ for every node v;
$d(s) \leftarrow 0$;
$E' \leftarrow$ topological sort of E;
for *all arcs* $(u, v) \in E'$ *with weight* w **do**
  $d(v) \leftarrow \min\{d(v), d(u) + w\}$;

the arrival time and the departure time of the corresponding connection. The departure time itself has no direct counterpart. A path in a graph is a sequence of arcs just as a journey is a sequence of connections. An earliest arrival journey minimizes the arrival time at the target stop. If we interpret the arc weights as the time needed to traverse the arc then a shortest path also minimizes the arrival time at its target node. In the graph setting shortest path queries are only composed of a source node and a target node. In the network world there exists besides the start stop and the target stop also a start time. This seems at first like a major difference but it is possible to interpret a graph as a periodic network with period $\Pi = 1$. The basic idea is that at every moment every connection departure recurs and therefore the start time does not matter and we can set it to 0 without loss of generality.

The foundation of many shortest path algorithms in the graph setting (such as Dijkstra [15] and Bellman-Ford [23, 24, 25]) is a basic operation called arc relaxation. To answer a one-to-all shortest path query first a function d is initialized with $+\infty$ for every node except the source which is set to 0. The next step consist of performing arc relaxations until d can not be changed anymore. At the end d maps every node onto the length of the shortest path to it. Let (u, v) be an arc with weight w then the arc relaxation operation is given by

$$d(v) \leftarrow \min\{d(v), d(u) + w\} .$$

A similar construct is possible in timetable networks. To answer a one-to-all earliest arrival query one can first initialize a function d for every stop to $+\infty$. At the start stop it is not initialized with 0 but with the start time. Next, connection relaxations are performed until d can not be changed anymore. At the end d maps every stop onto the arrival time of the earliest arrival journey to it. Relaxing a connection c is given by the following operation:

$$d(c_{\text{arrstop}}) \leftarrow \min\{d(c_{\text{arrstop}}), c_{\text{arrtime}}\}$$

However a connection may only be relaxed if the user arrives early enough at the departure stop, i.e.,

$$d(c_{\text{depstop}}) \leq c_{\text{deptime}} .$$

Given this counterpart for arc relaxation it is possible to adapt Dijkstra's algorithm with only mechanical term substitutions as the pseudocode comparison in 2.1 illustrates. The same can be done for the Bellman-Ford algorithm. However, the requirement on the connection relaxation can be used show a form of acyclicity. This requirement together with the fact that $c_{\text{deptime}} < c_{\text{arrtime}}$ holds for every connection c by definition prohibits cyclic journeys. There may not exist any journey $c_1 \dots c_i \dots c_j \dots c_n$ with $c_i = c_j$. Suppose that this was the case then following must hold:

$$(c_i)_{\text{arrtime}} \leq \dots \leq (c_j)_{\text{deptime}} < (c_j)_{\text{arrtime}} = (c_i)_{\text{arrtime}}$$

which is a contradiction. General graphs on the other hand allow cyclic paths and therefore timetable networks are in some sense more similar to directed acyclic graphs that do not

allow such paths. On directed acyclic graphs it is possible to solve the shortest path problem by first sorting the arcs topologically[4] by their target node and then relaxing them in that order. Note that this reordering is independent of the shortest path query and may therefore be computed in a preprocessing phase. Algorithm 2.2 illustrates this using pseudocode. The algorithm is correct because an arc (u, v) is only relaxed if the final value of $d(u)$ is known. This simple and elegant algorithm can be ported to timetable networks. Ordering the connections by departure time results in a topological ordering and before performing a relaxation one must check whether the relaxation is allowed. In chapter 3 we show that is possible to use this algorithm as basis to construct simple and fast algorithms that handle minimum change times and footpaths and solve a variety of earliest arrival problems including the profile problem. Note that the structure of the basic variant is nearly the same just with swapped basic building blocks.

[4]We say that an arc (u, v) is topologically ordered before (u', v') if a path from v to u' exists, i.e., they are topologically ordered by their target nodes.

3. Connection-Scan-Algorithm

At the end of the last section it was suggested to port a shortest path algorithm that works on directed acyclic graphs to work on timetables. In this chapter, we elaborate the details of this approach and develop several of algorithms to solve the earliest arrival and profile query problems. We call these algorithms *connection-scan-algorithms*. These have a simple structure and are therefore relatively easy to implement but they are also fast because they make good use of cache effects.

3.1 Basic Algorithm and Correctness

In this section, we describe the basic variants of our algorithms that solve the one-to-all earliest arrival problem. We first recall from the last chapter how connection relaxation is performed on simple networks with no minimum change time and use it to construct an easy algorithm. After that we adapt the relaxation to also take minimum change times into account. Finally, we modify this algorithm to support footpaths.

Let us recall that the one-to-all earliest arrival problem consists of computing for a given start stop and a given start time a function d that maps every stop s onto the earliest arrival over all journeys that end at s departing at the start stop after the start time. To solve this problem we first sort the connections by their departure time. We then relax every relaxable connection in this order. Recall that we defined the connection relaxation with no minimum change times for a connection c as

$$d(c_{\text{arrstop}}) \leftarrow \min \{d(c_{\text{arrstop}}), c_{\text{arrtime}}\}$$

and we require that it is only applied to connections that fulfill

$$d(c_{\text{depstop}}) \leq c_{\text{deptime}} .$$

Pseudocode for our basic algorithm is given in 3.1. The connections can be ordered in a preprocessing step that is independent of the query. Given an ordered connection list it is obvious that the running time of our algorithm is linear in the number of connections. Further it is possible to arrange the connections in a single array resulting in an algorithm whose main part consists of a scan over a continuous memory block. This is a very cache friendly operation and therefore the algorithm should perform well on modern hardware. It remains to show that the algorithm is correct.

Algorithm 3.1 basic connection-scan-algorithm

$d(s) \leftarrow +\infty$ for every stop s;
$d(s_{\text{startstop}}) \leftarrow t_{\text{starttime}}$;
for *all connections c ascending by departure time* **do**
  if $d(c_{\text{depstop}}) \leq c_{\text{deptime}}$ **then**
    $d(c_{\text{arrstop}}) \leftarrow \min\{d(c_{\text{arrstop}}), c_{\text{arrtime}}\}$;

Theorem 1. *The connection-scan algorithm solves the earliest arrival problem.*

Proof. Our proof consists of three parts.

1. We first show that modifying d only through relaxation operations never leads to a state where a stop exists that has an arrival time in d that does not correspond to a journey.
2. We then show that if no relaxation that would modify d can be applied then d contains only earliest arrival times.
3. Finally, we show that after the execution of the loop no connection that would modify d can be relaxed anymore.

We show the first statement by induction. The induction hypothesis is that a journey from the start exists for every stop s for which $d(s) \neq +\infty$ with the arrival time $d(s)$. Initially only $d(s_{\text{startstop}}) \neq +\infty$ and therefore it is trivially fulfilled. Suppose that a connection c' is being relaxed. For this to happen c' has be relaxable, i.e., $d(c'_{\text{depstop}}) \leq c'_{\text{deptime}} < +\infty$ must hold. The induction hypothesis therefore tells us that a journey $c_1 c_2 \dots c_n$ with $(c_n)_{\text{arrstop}} = c'_{\text{depstop}}$ and $(c_n)_{\text{arrtime}} = d((c_n)_{\text{arrstop}})$ must exist. As

$$(c_n)_{\text{arrtime}} = d((c_n)_{\text{arrstop}}) = d(c'_{\text{depstop}}) \leq c'_{\text{deptime}}$$

the journey $c_1 c_2 \dots c_n c'$ is valid and therefore we can set $d(c'_{\text{arrstop}})$ to c'_{arrtime}.

The second statement can by shown by contradiction. Suppose that no relaxable connection exists that changes d and that there is a stop s such that $d(s)$ is the not earliest arrival time. Then there exists a journey $c_1 c_2 \dots c_n$ that ends at s with $(c_n)_{\text{arrtime}} < d(s)$. Further we know that $d(c_1) = t_{\text{starttime}} \leq (c_1)_{\text{deptime}}$ and therefore c_1 can be relaxed. If relaxing it would change d then we have a contradiction. This means that this relaxation operation can not change d. As a consequence c_2 must be relaxable. Using the same reasoning we can show that every c_i is relaxable but may not change d. As relaxing c_n does not change d we can derive $(c_n)_{\text{arrtime}} = d(s)$ which is a contradiction.

We show the third statement also by contradiction. Suppose that after the execution of the loop a relaxable connection c that changes d would exist. If c departs at the start stop then it would have been relaxed as $d(s_{\text{startstop}})$ is constant during the whole execution. If c departs at another stop then another connection c' must exist that arrives at that stop and sets the earliest arrival time. As c is relaxable we know that

$$c'_{\text{deptime}} < c'_{\text{arrtime}} = d(c_{\text{depstop}}) \leq c_{\text{deptime}}\,.$$

However this also allows us to derive that $c'_{\text{deptime}} < c_{\text{deptime}}$ and therefore c' would have been relaxed before c. This concludes the contradiction. □

The next step is to add minimum change times to the algorithm. A sequence of connections is a valid journey if every two subsequent connections c and c' are either part of the same

Algorithm 3.2 basic connection-scan-algorithm with minimum change times

$d(s) \leftarrow +\infty$ for every stop s;
$d(s_{\text{startstop}}) \leftarrow t_{\text{starttime}} - (s_{\text{startstop}})_{\text{minchange}}$;
$g(c) \leftarrow false$ for every connection c;
for *all connections c ascending by departure time* **do**
  if $d(c_{\text{depstop}}) + (c_{\text{depstop}})_{\text{minchange}} \leq c_{\text{deptime}}$ *or* ($c_{\text{prev}} \neq \perp$ *and* $g(c_{\text{prev}})$) **then**
    $d(c_{\text{arrstop}}) \leftarrow \min\{d(c_{\text{arrstop}}), c_{\text{arrtime}}\}$;
    $g(c) \leftarrow true$;
$d(s_{\text{startstop}}) \leftarrow d(s_{\text{startstop}}) + (s_{\text{startstop}})_{\text{minchange}}$;

trip (i.e. $c_{\text{next}} = c'$) or leave enough change time (i.e. $c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} \leq c'_{\text{deptime}}$). This leads to the same relaxation operation as before, i.e.,

$$d(c_{\text{arrstop}}) \leftarrow \min\{d(c_{\text{arrstop}}), c_{\text{arrtime}}\}$$

but the condition when it can be applied must be modified. There are two conditions and if at least one is true then the connection can be relaxed.

1. The user can get a connection if he is early enough at the departure stop.
2. The user can get a connection if he can get the previous connection in the same trip.

It remains to show that these two conditions can efficiently be tested. The first condition can be tested using the information stored in d. The precise formulation is

$$d(c_{\text{depstop}}) + (c_{\text{depstop}})_{\text{minchange}} \leq c_{\text{deptime}}\,.$$

The second condition is more tricky. A connection can be relaxed if its previous connection can be relaxed. Unfortunately, this is a recursive definition. Fortunately, one can observe that in our algorithm at the moment a connection is being tested for relaxation the previous connection has already been tested because $(c_{\text{prev}})_{\text{deptime}} < c_{\text{deptime}}$ holds. This means that relaxable and has been relaxed are equivalent. Checking if a connection c_{prev} has been relaxed can be done by raising a flag $g(c_{\text{prev}})$ when relaxing c_{prev} and then testing this flag.

Every relaxation extends an existing journey by first waiting for the minimum change time at the departure stop and then adding a connection. This is nearly always what is wanted except for the start stop where no minimum change time is needed. This can be fixed by adjusting the start time. The pseudocode for this algorithm is given in 3.2. The correctness proof is in essence the same one as in the basic version without minimum change times.

Another thing to add to the algorithm are footpaths. The relaxation operation in essence only extends an existing journey. Using the variant presented below one relaxation extends a journey by one connection followed optionally by a single footpath. If we initially only set $d(s_{\text{startstop}})$ to a value different from $+\infty$ then only journeys without an initial footpath are considered. To correct this error we also set $d(f_{\text{arrstop}}) = t_{\text{starttime}} + f_{\text{dur}}$ for every footpath f with $f_{\text{depstop}} = s_{\text{startstop}}$.

The conditions on the relaxation are the same as for the algorithm with the minimum change time. The actual relaxation has to expanded. Besides the already introduced change of d at the arrival stop

$$d(c_{\text{arrstop}}) \quad \leftarrow \quad \min\{d(c_{\text{arrstop}}), c_{\text{arrtime}}\}$$

Algorithm 3.3 basic connection-scan-algo with minimum change times and footpaths

$d(s) \leftarrow +\infty$ for every stop s;
$d(s_{\text{startstop}}) \leftarrow t_{\text{starttime}} - (s_{\text{startstop}})_{\text{minchange}}$;
for *all footpaths* f *with* $s_{\text{startstop}} = f_{\text{depstop}}$ **do**
  $d(f_{\text{arrstop}}) \leftarrow t_{\text{starttime}} + f_{\text{dur}}$;
$g(c) \leftarrow false$ for every connection c;
for *all connections* c *ascending by departure time* **do**
  if $d(c_{\text{depstop}}) + (c_{\text{depstop}})_{\text{minchange}} \leq c_{\text{deptime}}$ *or* $(c_{\text{prev}} \neq \perp$ *and* $g(c_{\text{prev}}))$ **then**
    $d(c_{\text{arrstop}}) \leftarrow \min\{d(c_{\text{arrstop}}), c_{\text{arrtime}}\}$;
    for *all footpaths* f *with* $c_{\text{arrstop}} = f_{\text{depstop}}$ **do**
      $d(f_{\text{arrstop}}) \leftarrow \min\{d(f_{\text{arrstop}}), c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} + f_{\text{dur}}\}$;
    $g(c) \leftarrow true$;
$d(s_{\text{startstop}}) \leftarrow d(s_{\text{startstop}}) + (s_{\text{startstop}})_{\text{minchange}}$;

we also consider all outgoing footpaths f (i.e. all footpath with $f_{\text{depstop}} = c_{\text{depstop}}$) and set

$$d(f_{\text{arrstop}}) \leftarrow \min\{d(f_{\text{arrstop}}), c_{\text{arrtime}} + f_{\text{dur}} + (c_{\text{arrstop}})_{\text{minchange}}\}$$

which leads to the algorithm illustrated in 3.3.

3.2 Comparison with Time-Expanded Approach

Our earliest arrival algorithm has some similarities with the time-expanded approach described in Section 2.8. In the time-expanded approach the train-arcs are relaxed ordered by the time of the corresponding departure event. This means that the order is equivalent to the order in which we relax the connections. A consequence is that many high level observations about the two approaches are similar. However, the implementation of the two approaches is vastly different. We do not need additional stay-arcs. Our algorithm can be implemented using a scan over a continuous memory block which allows better cache efficiency. The main advantage of our algorithm is that it needs no priority queue.

3.3 Optimizations for Earliest-Arrival Problem

In this section we propose several optimizations for the algorithms described in the previous section.

3.3.1 Reducing the Number of Arithmetic Operations

In this subsection, we propose a small optimization that reduces the number of arithmetic operations in the main loop. To accomplish this we transform the network slightly in a preprocessing step that is independent of the actual query. By looking at the pseudocode in 3.3 one can identify two situations in which the minimum change times are used in the main loop. The first occurrence is the relaxation condition

$$d(c_{\text{depstop}}) + (c_{\text{depstop}})_{\text{minchange}} \leq c_{\text{deptime}}$$

and the second is the relaxation of footpaths

$$d(f_{\text{arrstop}}) \leftarrow \min\{d(f_{\text{arrstop}}), c_{\text{arrtime}} + (c_{\text{depstop}})_{\text{minchange}} + f_{\text{dur}}\}\,.$$

The goal is to get rid of these two additions and the associated memory lookup for the minimum change times. To achieve this we change the meaning of the earliest arrival

times d. We require that the minimum change time for the next element in the journey is already added to d. In a postprocessing step we compensate for this by subtracting it again. (Note that the start stop is special as is explained at the end of this subsection.) This allows us to change the relaxation condition to

$$d(c_{\text{depstop}}) \leq c_{\text{deptime}}$$

but we have to update all places were the earliest arrival time d is modified. To avoid introducing new usages of the minimum change time in the main loop we transform the footpath durations and the arrival times of the connections in a preprocessing step. They should also take the minimum change time at the target stop into account. This can be performed by applying

$$\begin{aligned} c_{\text{arrtime}} &\leftarrow c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} \\ f_{\text{dur}} &\leftarrow f_{\text{dur}} + (f_{\text{arrstop}})_{\text{minchange}} \end{aligned}$$

for every connection c and every footpath f. This adds the minimum change time in every situation except the case where the earliest arrival time for the start stop is set, i.e.,

$$d(s_{\text{startstop}}) \leftarrow t_{\text{starttime}}$$

However, this assignment is correct as it is. By definition we only require of a journey that the user waits for the minimum change time between journey elements and not at the start of the journey. For this reason we also have to make sure that the minimum change time is not subtracted from the earliest arrival time $d(s_{\text{startstop}})$ at the start stop in the postprocessing step.

3.3.2 Start Criterion

No connection that departs before $t_{\text{starttime}}$ can ever be relaxed. We may therefore ignore the first elements of the connection array and start the scan at the first connection whose departure time is sufficiently large. As we know nothing about the distribution of $t_{\text{starttime}}$ a binary search is the best algorithm to find the first connection. This search can be performed directly on the connection array as it is sorted by departure time.

3.3.3 Stop Criteria

A stop criterion is a condition at which the main loop can be aborted because the earliest arrival times d no longer improve. In the one-to-one problem setting, one is only interested in the earliest arrival time at the target stop $d(s_{\text{targetstop}})$ and this allows for a very simple stop criterion. The loop can be aborted if

$$d(s_{\text{targetstop}}) \leq c_{\text{deptime}}$$

is met. Suppose that there was a connection c' that can be relaxed and that could improve $d(s_{\text{targetstop}})$. It would have to be ordered after c and it would have to set the earliest arrival time to c'_{arrtime} (or even more if footpaths are considered). Further it has to decrease $d(s_{\text{targetstop}})$ as otherwise it would not change it. Adding these statements together leads to

$$d(s_{\text{targetstop}}) \leq c_{\text{deptime}} \leq c'_{\text{deptime}} < c'_{\text{arrtime}} < d(s_{\text{targetstop}})$$

which is a contradiction. The value of $d(s_{\text{targetstop}})$ therefore can not improve anymore if the stop criterion is met.

A stop criterion for the one-to-all setting is more complicated. We have to make sure that no stop s exists whose arrival time $d(s)$ can still improve. Using an argument similar to the previous one it is possible to see that

$$\max_{s \in S} d(s) \leq c_{\text{deptime}}$$

is such a stop criterion. (Recall that S is the set of all stops.) The problem is that computing $\max_{s \in S} d(s)$ every iteration is not fast. One possibility is to evaluate it only every few iterations but there is a different method: We determine an upper bound for it and abort if c_{deptime} is bigger than this upper bound. If there is a stop that has not yet been reached then $\max_s d(s) = +\infty$ and otherwise it is a finite number. Therefore as long as a stop exists that has not been reached we may not abort. To determine this we keep track of the number of stops r that have been reached. Further we introduce another variable u that is an upper bound for the earliest arrival times of all the reached stops. We can abort if

$$r = |S| \wedge u \leq c_{\text{deptime}}$$

holds. We update u and r every time that d is modified. If the old value is $+\infty$ then r is increased. The variable u stores the maximum over all the values that have been assigned to some element in d. As $d(s)$ only decreases during the execution for a fixed s it is sufficient to only update u when the old value was $+\infty$.

3.3.4 Cache-Efficient Flags

At every iteration of the main loop $g(c_{\text{prev}})$ has to be evaluated. This often causes a cache miss because it is the only information about c_{prev} that is looked up and c and c_{prev} are not necessarily close in memory. There are basically two ways to relieve this problem. The first consists of storing g in its own array and make sure that every entry only occupies a single bit of information. This allows for a very compact memory representation and therefore large parts of the array fit into the cache. The other way is to use flags with a different meaning. The flag $g(c)$ in the basic algorithm signals if the connection c has been relaxed. We change this meaning and raise $g(c)$ if c can be relaxed. The relaxation condition must therefore be changed from

$$(c_{\text{prev}} \neq \perp \wedge g(c_{\text{prev}})) \wedge d(c_{\text{depstop}}) \leq c_{\text{deptime}}$$

to

$$g(c) \wedge d(c_{\text{depstop}}) \leq c_{\text{deptime}}$$

and the point at which the flag is raised must be changed from

$$g(c) \leftarrow \mathit{true}$$

to

$$g(c_{\text{next}}) \leftarrow \mathit{true} \,\text{if}\, c_{\text{next}} \neq \perp .$$

The advantage is that in the relaxation condition only information about c is being checked. As the data about it is loaded anyhow no additional cache miss is caused at this point. Unfortunately this is extra cache miss can now occur at the place where the flag is raised. However this code is only executed if the connection is relaxed instead of every iteration.

3.3.5 Combining the Optimizations

In the previous subsections we have described a number of small optimizations. It might not be obvious that they can all work together well but it is possible. The pseudo code is given in 3.4.

Algorithm 3.4 optimized connection-scan-algorithm for one-to-all earliest arrival

for *all footpaths* f **do**
  $f_{\text{dur}} \leftarrow f_{\text{dur}} + (f_{\text{arrstop}})_{\text{minchange}}$;
for *all connections* c **do**
  $c_{\text{arrtime}} \leftarrow c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}}$;
Sort the connections by departure time;
`// The code above can be executed in a preprocessing step. The code below has to be run for every query.`
$d(s) \leftarrow +\infty$ for every stop s;
$d(s_{\text{startstop}}) \leftarrow t_{\text{starttime}}$;
$r \leftarrow 1$;
$u \leftarrow t_{\text{starttime}}$;
for *all footpaths* f *with* $s_{\text{startstop}} = f_{\text{depstop}}$ **do**
  settime$(f_{\text{arrstop}}, t_{\text{starttime}} + f_{\text{dur}})$;
$g(c) \leftarrow false$ for every connection c;
Find the first connection c_0 after $t_{\text{starttime}}$ using a binary search;
for *all connections* c *ascending by their departure time after* c_0 **do**
  if $r = |S| \wedge u < c_{\text{deptime}}$ **then**
    break out of loop;
  if $d(c_{\text{depstop}}) \leq c_{\text{deptime}} \vee g(c)$ **then**
    if $c_{\text{next}} \neq \perp$ **then**
      $g(c_{\text{next}}) \leftarrow true$;
    settime$(c_{\text{arrstop}}, c_{\text{arrtime}})$;
    for *all footpaths* f *with* $c_{\text{arrstop}} = f_{\text{depstop}}$ **do**
      settime$(f_{\text{arrstop}}, c_{\text{arrtime}} + f_{\text{dur}})$;
for *all stops* s *except* $s_{\text{startstop}}$ **do**
  $d(s) \leftarrow d(s) - s_{\text{minchange}}$;

---

if $d(s) = +\infty$ **then**
  $r \leftarrow r + 1$;
  $u \leftarrow \max\{u, t\}$;
  $d(s) \leftarrow t$;
else if $d(s) > t$ **then**
  $d(s) \leftarrow t$;

Procedure `settime(`s`,`t`)`

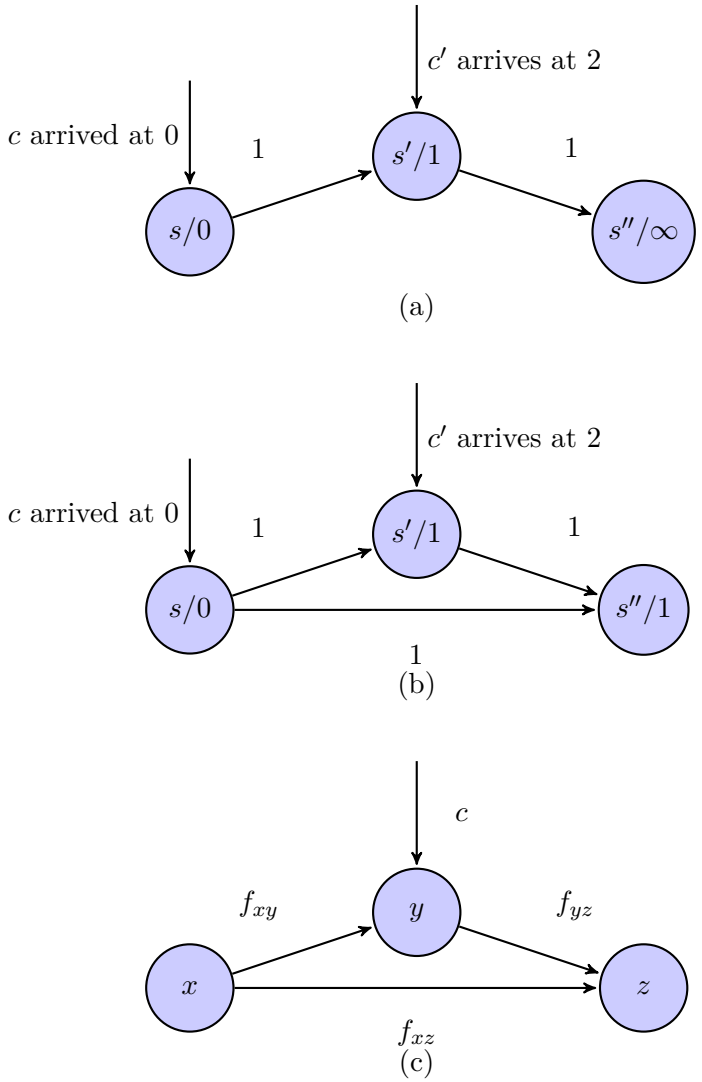


Figure 3.1: Every circle represents a stop s and contains the current $d(s)$. The arcs represent footpaths with their duration and connections with their arrival time. The connection c has already arrived whereas the connection c' is about to arrive.

3.3.6 Exploiting Transitive Footpath Closure

Recall that in some circumstances instances have transitive footpaths (that also fulfill the triangle inequality). Further the instances have no minimum change times. These properties can be exploited to improve performance. The key observation is that it is sufficient to only iterate over the outgoing footpaths if a relaxation improves the arrival time at the arrival stop. Formally the optimization presented in this subsection can only be applied if the following holds: For all footpaths f and f' with $f_{\text{arrstop}} = f'_{\text{depstop}}$ and $f_{\text{depstop}} \neq f'_{\text{arrstop}}$ a footpath f'' exists such that $f_{\text{depstop}} = f''_{\text{depstop}}$, $f'_{\text{arrstop}} = f''_{\text{arrstop}}$ and $f''_{\text{dur}} \leq f_{\text{dur}} + f'_{\text{dur}}$. (Note that the transitivity is slightly adjusted to not require loops on every stop.)

Consider the situation in Figure 3.1. In (a) the footpaths are not transitive and in (b) they are. In both c' can not improve the arrival time at $d(s')$ but in (a) it can improve $d(s'')$ to 3. We proof that this can not be the case if the footpaths are transitive as in (b).

Consider the more general situation in (c). We show that if c does not improve $d(y)$ it can not improve $d(z)$, i.e., that $d(z) < c_{\text{arrtime}} + (f_{yz})_{\text{dur}}$. As $d(y) \neq +\infty$ a connection c' must have set the value and it must have arrived before c (i.e. $c'_{\text{arrtime}} < c_{\text{arrtime}}$). Either it arrived at y or at x.

1. The connection c' arrived at y. In this case $d(z) = c'_{\text{arrtime}} + (f_{yz})_{\text{dur}}$ must hold and $d(z) < c_{\text{arrtime}} + (f_{yz})_{\text{dur}}$ directly follows as $c'_{\text{arrtime}} < c_{\text{arrtime}}$.
2. The connection c' arrived at x. In this case $d(z) \leq c'_{\text{arrtime}} + (f_{xz})_{\text{dur}}$ and $d(y) = c'_{\text{arrtime}} + (f_{xy})_{\text{dur}}$ must hold. Using $(f_{xz})_{\text{dur}} \leq (f_{xy})_{\text{dur}} + (f_{yz})_{\text{dur}}$ one can show that $d(z) - (f_{xy})_{\text{dur}} \leq d(z) \leq c'_{\text{arrtime}} + (f_{yz})_{\text{dur}}$. Again using $c'_{\text{arrtime}} < c_{\text{arrtime}}$ one can show that $d(z) < c_{\text{arrtime}} + (f_{yz})_{\text{dur}}$.

We showed that if the footpaths are transitive then the outgoing footpaths only have to be considered if the arrival time at the target stop is improved.

4. Connection-Scan-Algorithm for Profile Queries

In this section we propose an algorithm based upon connection scanning and relaxation to solve profile queries in timetable networks. There are two ways to realize it: Either as an one-to-all or as an all-to-one query. We describe the all-to-one variant because it serves as the basis for the algorithm introduced in Section 6. The algorithm is described in two parts. The first part is called the *main algorithm* and uses an efficient blackbox implementation of step functions exists. The second part consists of a fine tailored implementation of this blackbox. We first describe the operations that the blackbox must support efficiently. We then describe the main algorithm as it determines the access patterns to the step function data structures. Using this knowledge, it is possible to construct a datastructure optimized for this specific use case.

4.1 Required Step Function Operations

The step function data structure f must support the following operations.

1. It must be possible to initialize f with the function $\{x \mapsto +\infty\}$.
2. It must be possible to evaluate f for every x, i.e., $f(x)$ must be computable.
3. New jumps must be insertable in f. Given a (x, y)-jump we define the insertion as

$$(f \diamond (x, y))(z) = \begin{cases} f(z) & z > x \\ \min\{y, f(z)\} & z \leq x \end{cases}$$

4. It must be possible to compute the pointwise minimum of f and g, i.e.,

$$\min\{f, g\}(x) = \min\{f(x), g(x)\}$$

5. It must be possible to add a constant to a step function, i.e., for a function f and a constant c the function $f + c$ must be computable and is defined by

$$(f + c)(x) = f(x) + c\,.$$

4.2 Main Algorithm

In this subsection we describe the algorithm in terms of a step function blackbox. The basic approach is the same as before. We start with a simple network without minimum change times and footpaths and define a relaxation operations. Once this is done we extend the algorithm.

Note that in this problem setting, the distance function d does not not map a stop onto a moment in time as was the case in the earliest arrival setting. Here, the distance function d maps a stop onto another function that maps a departure time onto an arrival time. In nearly all cases functions onto which d maps are a step functions. If no footpaths are considered then the target stop is the only stop whose profile function is not a step function but the identity function. As the blackbox only supports strict step functions it can not represent this function. We add checks to the code to circumnavigate the problem. If footpaths are considered then the situation is more complicated as detailed below.

The relaxation operation for a connection c in a network with no minimum change time and no footpaths can be written as

$$d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, d(c_{\text{arrstop}})(c_{\text{arrtime}}))$$

if it does not depart at the target stop. If it does, then the relaxation operation does nothing as the identity function can not be improved. The relaxation operation has no attached condition and therefore can always be applied. Initially we set $d(s) = \{x \mapsto +\infty\}$ for every stop s except for the target stop we set $d(s_{\text{targetstop}}) = \{x \mapsto x\}$. The algorithm is a backward search and therefore we relax the connections by decreasing departure time instead of increasing departure time.

Theorem 2. *The all-to-one profile algorithm is correct.*

Proof. To show the correctness one has to show that relaxations do no introduce departure times that do not correspond to any journey. Further one has to show that if no relaxation can improve d anymore then d is the optimal solution. Finally one has to show that after relaxing the connections decreasing by their departure time, as done in our algorithm, no connection can be relaxed anymore that would modify d.

The first statement can be shown by induction similarly as in the previous sections. The idea is that every jump corresponds to a journey to the target. Every new jump corresponds to an old jump whose journey is extended by prepending another connection.

The second statement can be shown by contradiction. Suppose that no relaxation can be applied that would modify d but $d(s)(x)$ is not optimal for some stop s and some departure time x. In this case a journey must exist that departs at s no earlier than x and arrives at the target stop strictly before $d(s)(x)$. We can relax the connections of this journey in reverse journey order and propagate the better arrival time to s. This modifies d and therefore a relaxation that modified d must have been applied. This is a contradiction.

The third statement can also be shown by contradiction. Suppose that after the execution of the algorithm a connection c existed that would change d. As adding the same jump twice changes nothing the jump induced by c must be different from the first time it was relaxed. This means that $d(c_{\text{arrstop}})(c_{\text{arrtime}})$ must have changed. Adding a jump may only modify profile function times below its own departure time. All the connections that have been relaxed after c must not have a departure time greater than c and therefore can not have modified $d(c_{\text{arrstop}})(c_{\text{arrtime}})$. This is a contradiction as no other connections were relaxed and we can derive that the value must not have been modified. □

Algorithm 4.1 basic connection-scan-backward-profile-algorithm

$d(s) \leftarrow \{x \mapsto +\infty\}$ for every stop s;
$d(s_{\text{targetstop}}) \leftarrow \{x \mapsto x\}$;
for *all connections c descending by departure time* **do**
  if $c_{\text{depstop}} \neq s_{\text{targetstop}}$ **then**
    if $c_{\text{arrstop}} \neq s_{\text{targetstop}}$ **then**
      $d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, d(c_{\text{arrstop}})(c_{\text{arrtime}}))$;
    else
      $d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, c_{\text{arrtime}})$;

The pseudo code for the basic algorithm is given in Algorithm 4.1. Note that if the connections are relaxed decreasing by departure time then at the moment that c is relaxed the arrival time at the target stop $d(c_{\text{arrstop}})(c_{\text{arrtime}})$ has already been correctly computed.

4.2.1 Minimum Change Times

Supporting minimum change times can be realized in a similar way as in the earliest arrival algorithm. The main difference is that the journeys are no longer extended at the end but prefixed with new connections. A relaxation operation prepends a journey with a connection and the minimum change time at the arrival stop of this connection. Note that if the arrival stop is the target stop then this minimum change time must not be accounted for. In the earliest arrival setting we used flags $g(c)$ to determine what connections the user can reach. This simple approach is no longer sufficient as nearly every connection is relevant for some journey in the profile. For this reason the flags $g(c)$ have to be replaced with arrival times at the target stop. In the profile algorithm $g(c)$ is the earliest arrival time at the target stop over all journeys that start with c.

The relaxation operation again has no conditions but is more complex as it needs to take care of all the different cases. If the connection departs at the target stop then nothing is done. Otherwise it first computes $g(c)$ and then adds a jump at the departure stop using

$$d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, g(c)) \ .$$

Recall that $g(c)$ is the earliest arrival time if the user starts his journey sitting in the train at the connection c. Computing $g(c)$ is more complicated because of the different ways in which the user can continue his journey. We compute the arrival time in each case and use the earliest of these arrival times. The different ways to continue his journey are:

1. If the connection arrives at the target stop then the user can end his journey. The arrival time is c_{arrtime}. Note that if this option is available then it is always superior to the two below.
2. The user can exit the train and take another connecting train. The arrival time can be computed as $d(c_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}})$.
3. If the connection has a next connection then the user can remain seated. The arrival time then is $g(c_{\text{next}})$.

The pseudo code is illustrated in 4.2.

4.2.2 Footpaths

Footpaths may be handled by modifying the relaxation operation. The user now has another way of continuing his journey.

Algorithm 4.2 basic connection-scan-backward-profile-algo with minimum change times

$d(s) \leftarrow \{x \mapsto +\infty\}$ for every stop s;
$d(s_{\text{targetstop}}) \leftarrow \{x \mapsto x\}$;
for *all connections c descending by departure time* **do**
  if $c_{\text{depstop}} \neq s_{\text{targetstop}}$ **then**
    if $c_{\text{arrstop}} = s_{\text{targetstop}}$ **then**
      $g(c) \leftarrow c_{\text{arrtime}}$;
    else
      $g(c) \leftarrow d(c_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}})$;
      if $c_{\text{next}} \neq \perp$ **then**
        $g(c) \leftarrow \min\{g(c), g(c_{\text{next}})\}$;
    $d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, g(c))$;

4. If a footpath f exists that connects c_{arrstop} with the target stop then the user can walk there and end his journey. The arrival time is

$$c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} + f_{\text{dur}} .$$

   Note that this option dominates the option below if it is available.

5. The user can exit the train and walk to another stop and then continue his journey. The arrival time is

$$d(f_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} + f_{\text{dur}} + (f_{\text{arrstop}})_{\text{minchange}}) .$$

However, this relaxation operation does not consider journeys that start with an initial footpath. This can be fixed by introducing a postprocessing step. We first consider footpaths f that do not end at the target stop. For every one of these the following operation has to be applied

$$d(f_{\text{depstop}}) \leftarrow \min\{d(f_{\text{depstop}}), d(f_{\text{arrstop}}) + f_{\text{dur}} + (f_{\text{arrstop}})_{\text{minchange}}\}$$

which basically consists of prefixing journeys with a footpath and the minimum change time at the arrival stop of the footpath.

Now consider footpaths f that end at the target stop. It is never a sensible idea to add another connection after these in a journey because the resulting journey is obviously not an earliest arrival journey. As a consequence the journey consisting of only f is the only journey that starts with a footpath (and therefore has not yet been computed) and might be an earliest arrival journey. This property allows us to compute these journeys in a postprocessing step. These journeys are time independent and therefore produce (partially) linear profile function even at different stops than the target stop. However, this is no real problem as no other computations are based upon these functions. We simply store a strict step function and a strict linear function and at each evaluation we evaluate both and return the minimum. The operation that has to be performed for f is

$$d(f_{\text{depstop}}) \leftarrow \min\{d(f_{\text{depstop}}), \{x \mapsto x + f_{\text{dur}}\}\} .$$

The pseudo-code is illustrated in 4.3.

4.3 Optimizations for the Main Algorithm

In this section, we propose a number of optimization for the main algorithm.

Algorithm 4.3 basic conn-scan-back-profile-algo with min change time and footpaths

$d(s) \leftarrow \{x \mapsto +\infty\}$ for every stop s;
$d(s_{\text{targetstop}}) \leftarrow \{x \mapsto x\}$;
for *all connections c descending by departure time* **do**
  if $c_{\text{depstop}} \neq s_{\text{targetstop}}$ **then**
    if $c_{\text{arrstop}} = s_{\text{targetstop}}$ **then**
      $g(c) \leftarrow c_{\text{arrtime}}$;
    else
      $g(c) \leftarrow d(c_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}})$;
      if $c_{\text{next}} \neq \perp$ **then**
        $g(c) \leftarrow \min\{g(c), g(c_{\text{next}})\}$;
      for *all footpaths f with* $c_{\text{arrstop}} = f_{\text{depstop}}$ **do**
        if $f_{\text{arrstop}} = s_{\text{targetstop}}$ **then**
          $g(c) \leftarrow \min\{g(c), c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} + f_{\text{dur}}\}$;
        else
          $g(c) \leftarrow \min\{g(c), d(f_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} + f_{\text{dur}} +$
          $(f_{\text{arrstop}})_{\text{minchange}})\}$;
    $d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, g(c))$;
for *all footpaths f with* $f_{\text{arrstop}} \neq s_{\text{targetstop}}$ **do**
  $d(f_{\text{depstop}}) \leftarrow \min\{d(f_{\text{depstop}}), d(f_{\text{arrstop}}) + f_{\text{dur}} + (f_{\text{arrstop}})_{\text{minchange}}\}$;
`// The code below may produce linear profile functions at stops different from the target stop`
for *all footpaths f with* $f_{\text{arrstop}} = s_{\text{targetstop}}$ **do**
  $d(f_{\text{depstop}}) \leftarrow \{x \mapsto \min\{d(f_{\text{depstop}})(x), x + f_{\text{dur}}\}\}$;

Algorithm 4.4 one-to-one backward profile query with pruning

$p \leftarrow$ one-to-all earliest arrival solution;
$d(s) \leftarrow \{x \mapsto +\infty\}$ for every stop s;
$d(s_{\text{targetstop}}) \leftarrow \{x \mapsto x\}$;
for *all connections c descending by departure time* **do**
  if $c_{\text{deptime}} < t_{\text{earlydep}}$ **then**
    break;
  if $c_{\text{arrtime}} > t_{\text{latestarr}}$ **then**
    continue;
  if $p(c_{\text{depstop}}) > c_{\text{deptime}}$ **then**
    continue;
  if $c_{\text{depstop}} \neq s_{\text{startstop}}$ **then**
    if $c_{\text{arrstop}} \neq s_{\text{startstop}}$ **then**
      $d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, d(c_{\text{arrstop}})(c_{\text{arrtime}}))$;
    else
      $d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, c_{\text{arrtime}})$;

4.3.1 Source and Target Pruning

The algorithms described so far compute the solution for every moment in time and for every start stop. This is what the problem statement formally requests but often this is not what we actually want to compute. Suppose that we actually had a fixed start stop $s_{\text{startstop}}$ and an earliest departure time t_{earlydep}. The first optimization is to to abort the loop once the departure time drops below the start time. The second optimization is to first compute a one-to-all earliest arrival query with $s_{\text{startstop}}$ as the start stop and t_{earlydep} as the start time and use the result to prune some jumps in the profile query. The idea is that no jump can be relevant if there is no path from the start to get there. Algorithm 4.4 shows how this can be done in a network without minimum change times nor footpaths.

Another notable performance gain can be obtained by fixing a latest arrival time $t_{\text{latestarr}}$. This allows us to skip the first iterations of the loop as we must ignore every connection that arrives after the latest arrival time. Sometimes we do not have a latest arrival time at our disposition but of a latest departure time. For example if we wanted to compute the profile between two stops for a given time window at the start. We can transform the latest departure time at the start stop by performing a one-to-one earliest arrival query from the start to the target stop starting at the latest departure time. The resulting arrival time can then be used as the latest arrival time at the target. To see that this is correct suppose that a relevant path existed that departs before the latest departure time but arrives after the computed latest arrival time. In this case it could be replaced by the computed path and thus the solution would have been suboptimal. Such a path may therefore not exist and therefore this optimization is correct.

4.3.2 Reducing the Number of Arithmetic Operations

It is possible to eliminate the explicit usage of the minimum change times by transforming c_{arrtime} and f_{dur}. By looking at the pseudo-code in 4.3 it is possible to observe that every usage of c_{arrtime} is accompanied by an addition of $(c_{\text{arrstop}})_{\text{minchange}}$ exactly if c_{arrstop} is different from the target stop. It is therefore possible to eliminate this addition by increasing c_{arrtime} for all connections and compensating for this by subtracting $(s_{\text{targetstop}})_{\text{minchange}}$ at the right places. Analogously every f_{dur} is accompanied by an addition of $(f_{\text{arrstop}})_{\text{minchange}}$ if f_{arrstop} is not the target stop. The pseudo-code implementing this optimization is given in 4.5.

Algorithm 4.5 connection-scan-back-profile-algorithm with transformed arrival times

for *all footpaths* f **do**
  $f_{\text{dur}} \leftarrow f_{\text{dur}} + (f_{\text{arrstop}})_{\text{minchange}}$;
for *all connections* c **do**
  $c_{\text{arrtime}} \leftarrow c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}}$;
Sort the connections by departure time;
`// The code above can be executed in a preprocessing step. The code below has to be run for every query.`
$d(s) \leftarrow \{x \mapsto +\infty\}$ for every stop s;
$d(s_{\text{targetstop}}) \leftarrow \{x \mapsto x\}$;
$m \leftarrow (s_{\text{targetstop}})_{\text{minchange}}$;
for *all connections* c *descending by departure time* **do**
  if $c_{\text{depstop}} \neq s_{\text{targetstop}}$ **then**
    if $c_{\text{arrstop}} = s_{\text{targetstop}}$ **then**
      $g(c) \leftarrow c_{\text{arrtime}} - m$;
    else
      $g(c) \leftarrow d(c_{\text{arrstop}})(c_{\text{arrtime}})$;
      if $c_{\text{next}} \neq \perp$ **then**
        $g(c) \leftarrow \min\{g(c), g(c_{\text{next}})\}$;
      for *all footpaths* f *with* $c_{\text{arrstop}} = f_{\text{depstop}}$ **do**
        if $f_{\text{arrstop}} = s_{\text{targetstop}}$ **then**
          $g(c) \leftarrow \min\{g(c), c_{\text{arrtime}} + f_{\text{dur}} - m\}$;
        else
          $g(c) \leftarrow \min\{g(c), d(f_{\text{arrstop}})(c_{\text{arrtime}} + f_{\text{dur}})\}$;
  $d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, g(c))$;
for *all footpaths* f *with* $f_{\text{arrstop}} \neq s_{\text{targetstop}}$ **do**
  $d(f_{\text{depstop}}) \leftarrow \min\{d(f_{\text{depstop}}), d(f_{\text{arrstop}}) + f_{\text{dur}}\}$;
`// The code below may produce linear profile functions at stops different from the target stop`
for *all footpaths* f *with* $f_{\text{arrstop}} = s_{\text{targetstop}}$ **do**
  $d(f_{\text{depstop}}) \leftarrow \{x \mapsto \min\{d(f_{\text{depstop}})(x), x + f_{\text{dur}} - m\}\}$;

4.4 Tailored Step Function Implementation

In this subsection we describe how to implement the blackbox step functions used in the main algorithm. We store a function f as an ordered array of dominated jumps $f_J = (d_1, a_1) \dots (d_n, a_n)$. We sort the jumps by departure time (i.e. $d_i < d_{i+1}$) and as f is non decreasing also the arrival times must be ascending (i.e. $a_i < a_{i+1}$). In Section 4.1 five operations were introduced that the blackbox must support. Recall that the last jump must be at infinity (i.e. $(d_n, a_n) = (+\infty, +\infty)$). We first describe how the operations can be implemented and then how the access pattern of the main algorithm can be used to optimize them.

1. *It must be possible to initialize a function with* $\{x \mapsto +\infty\}$. This can be realized by adding only the $(+\infty, +\infty)$-jump to the array.
2. I*t must be possible to evaluate a function at every* x. This can be implemented by searching for the smallest i such that $x \leq d_i$ and returning a_i. As $d_n = +\infty$ such an i must always exist.
3. *New jumps must be insertable.* Recall that the jump insertion operation was defined as
$$(f \diamond (x, y))(z) = \begin{cases} f(z) & z > x \\ \min\{y, f(z)\} & z \leq x \end{cases} .$$
Suppose that one wants to add a new (x, y)-jump. Two cases must be distinguished. If a jump with $x \leq d_i$ exists and $y \geq a_i$ then the jump list does not have to be changed. Otherwise the jump has to inserted into the array after the jump with the biggest i such that $d_i < x$ still holds. All the jumps j with $j \leq i$ and $a_j \geq y$ have to be removed from the array.
4. *It must be possible to construct the pointwise minimum* h *of two step functions* f *and* g. This operation can be realized by simultaneously scanning the jump lists of both f and g and copying the jumps that are not dominated into h. The pseudo-code in 4.6 illustrates the details.
5. *It must be possible to add a constant* c *to a step function.* This operation consists of adding c to every arrival time a_i.

Operation 1 is always fast. Operations 4 and 5 are only needed in the postprocessing step, that is only required to handle footpaths, and therefore are only rarely used. As they are already linear in the number of jumps, optimizing the remaining Operations 2 and 3 has priority. We first show a short but useful lemma.

Lemma 3. *The main algorithm generates new jumps decreasing by departure time.*

Proof. By investigating the pseudo-code in 4.3 one can see that there is only one operation that can generate new jumps. It is
$$d(c_{\text{depstop}}) \leftarrow d(c_{\text{depstop}}) \diamond (c_{\text{deptime}}, g(c)) .$$
As the connections are scanned decreasing by departure time the jumps are also generated in this order. □

By using this lemma we can modify operation 3. Adding a (x, y)-jump to a jump list $(d_1, a_1), \dots, (d_n, a_n)$ can be done as following:

- If $a' \geq a_1$ do nothing.
- If $a' < a_1$ and $d' = d_1$ set $a_1 \leftarrow a'$.

Algorithm 4.6 pointwise minimum of step functions

Input: jump lists $f_J = (d_1^f, a_1^f) \dots (d_n^f, a_n^f)$ and $g_J = (d_1^g, a_1^g) \dots (d_m^g, a_m^g)$;
Output: pointwise minimum jump list $h_J = (d_1^h, a_1^h) \dots (d_p^h, a_p^h)$;

```
i ← n;
j ← m;
y ← +∞;
h_J = (+∞, +∞);
while i ≠ 0 ∧ j ≠ 0 do
    if d_i^f ≥ d_j^g then
        if a_i^f < y then
            add (d_i^f, a_i^f) jump to the front of h_J;
            y ← a_i^f;
        i ← i − 1;
    else
        if a_j^g < y then
            add (d_j^g, a_j^g) jump to the front of h_J;
            y ← a_j^g;
        j ← j − 1;
while i ≠ 0 do
    if a_i^f < y then
        add (d_i^f, a_i^f) jump to the front of h_J;
    i ← i − 1;
while j ≠ 0 do
    if a_j^g < y then
        add (d_j^g, a_j^g) jump to the front of h_J;
    j ← j − 1;
```

Algorithm 4.7 step function operations with min change times and footpaths

Recall that for every step function an array of jumps $(d_1, a_1) \dots (d_n, a_n)$ is stored and that enough memory has been allocated to extend it at the beginning.

```
// assert:  x ≤ d_1
if y < a_1 then
    if x < d_1 then
        insert (x,y) as first jump;
    else
        a_1 ← y;
```

Procedure `insert-jump(`x`,`y`)`

```
for i ← 1 to n do
    if x ≤ d_i then
        return a_i;
```

Procedure `evaluate(`x`)`

- If $a' < a_1$ and $d' < d_1$ add that (d', a')-jump as the first jump in the array.
- The case $a' < a_1$ and $d' > d_1$ can not occur because of the previous lemma.

As already observed in [5], if no jumps are dominated and no footpaths exist then every profile function will contain one jump more (the infinity jump) than the corresponding stop has outgoing connections. If footpaths exist then this property holds until the start of the postprocessing phase as initial footpaths are forbidden. Using this property we can allocate the necessary memory before starting the main algorithm and still have a memory consumption linear in the number of connections. With this optimization Operation 3 can be performed in constant time.

It remains to show that Operation 2 can be realized in a fast way. If footpaths are considered we only know of a heuristic. There are two places in the algorithm where step functions are evaluated. The first is

$$d(f_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}})$$

and the second is

$$d(f_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}} + f_{\text{dur}} + (f_{\text{arrstop}})_{\text{minchange}}) \,.$$

Consider $\epsilon = c_{\text{arrtime}} - c_{\text{deptime}} + (c_{\text{arrstop}})_{\text{minchange}} + \dots$. When relaxing the connection c a step function is evaluated at a moment $c_{\text{deptime}} + \epsilon$. The idea is that ϵ has to be small compared to a whole period as connections only travel for a small amount of time. Further we know because of the previous lemma for certain that $c_{\text{deptime}} \leq d_1$. We can conclude that $c_{\text{deptime}} + \epsilon \lessapprox d_1$ and therefore a linear scan over the jump list starting at d_1 is more efficient than a binary search. Pseudo-code is given in 4.7.

However, if no footpaths are considered then it is possible to implement the function evaluation in constant time. The key idea is to do the actual work in the jump insertion and to only lookup the result. Unfortunately, this leads to a non-constant jump insertion. Fortunately however, using an amortized analysis it is possible to show that the running time of all insertions is linear in the number of connections.

By analyzing the pseudo-code in 4.2 one can observe that the step functions are only evaluated in the following line.

$$g(c) \leftarrow d(c_{\text{arrstop}})(c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}}$$

Algorithm 4.8 step function operations for networks with only min change times

Recall that for every step function an array of jumps $(d_1, a_1) \dots (d_n, a_n)$ is stored and that enough memory has been allocated to extend it at the beginning. Further, there is a list of incoming connections sorted increasing by arrival time $b_1 \dots b_m$. Finally, also an index p to the first incoming connection with $(b_p)_{\text{arrtime}} + ((b_p)_{\text{arrstop}})_{\text{minchange}} \leq d_1$ is stored.

```
// assert:  x ≤ d_1
if y < a_1 then
    t ← a_1;
    if x < d_1 then
        insert (x,y) as first jump;
    else
        a_1 ← y;
    while p ≥ 1 ∧ (b_p)_arrtime + ((b_p)_arrstop)_minchange > x do
        h(b_p) ← t;
        p ← p − 1;
```

Procedure `insert-jump(`x,y`)`

```
if c_arrtime + (c_arrstop)_minchange ≤ d_1 then
    return a_1;
else
    // assert:  h(c) has been assigned a value in insert-jump.
    return h(c);
```

Procedure `evaluate(`c`)`

It is therefore sufficient to be able to evaluate a step function f at the moment $c_{\text{arrtime}} + (c_{\text{arrstop}})_{\text{minchange}}$ for every connection c that arrives at the stop corresponding to f. We will denote this moment by m_c. Further, by investigating the constant time jump insertion operation introduced above, one can see that only the first jump (d_1, a_1) in the jump list $(d_1, a_1) \dots (d_n, a_n)$ of any function may be modified. All other jumps have already the value that they will have after the algorithm has terminated. This means that the evaluation at all moments $x > d_1$ can be done independently of all jumps that might still be inserted. As the function is only evaluated at moments that correspond to a connection c it is possible to attach the resulting arrival time $d(c_{\text{arrstop}})(m_c)$ directly onto the connection as $h(c)$. Suppose that all $h(c)$ have been computed for all connections with $c_{\text{deptime}} > d_1$ then it is possible to evaluate the arrival time as

$$d(c_{\text{arrstop}})(m_c) = \begin{cases} h(c) & \text{if } m_c > d_1 \\ a_1 & \text{if } m_c \leq d_1 \end{cases}$$

which is a constant time operation. It remains to show that the jump insertion can still be realized in amortized constant time. For this we sort the incoming connections $b_1 \dots b_n$ of every stop increasing by their m_{b_i}, which is the same as descending by arrival time, in a preprocessing stop. For every stop we store the largest index p over all incoming connections such that m_{b_p} does not exceed d_1 (i.e. $m_{b_p} \leq d_1$). When inserting a (x, y)-jump we decrease p and evaluate all the intermediate $h(b_p)$. These are always equal to y and therefore this can be done in constant time per connection. In total the sum of all p incrementations is equal to the number of connections and therefore the total time is linear in the number of connections. The pseudo-code for the operations is given in 4.8.

Note that the optimizations proposed for the inner algorithm are compatible with the arrival time transformations introduced in Section 4.3.2.

5. Stochastic Models

In this chapter, we consider models for delay-robust stochastic routing. In Section 5.1, we describe several different stochastic models that vary in their degree of realism. We note that there is a trade-off between realism and sufficiently fast computability. In Section 5.2, we discuss how to compute delay-robust routes based on these models. We do not expect that for the most realistic models queries can be solved within reasonable time. In Chapter 6 we illustrate how one of the models can be solved within reasonable time.

The key idea is to define *states* that model the user in different situations and *transitions* between these states. Every journey corresponds to a path in this state-transition graph. Examples of states are:

1. The user stands at a stop at a moment in time.
2. The user is sitting in a train entering a stop.
3. The user stands at a stop at a moment in time knowing the delays of the arriving trains with certainty.
4. The user is sitting in a train entering a stop and has decided at what stop to exit.
5. The user is sitting in a delayed train knowing that another train is delayed.

It is important to note that besides the local and temporal location the state must also encode the whole non-static knowledge of the user. Precise train delays are such knowledge. Therefore the user is in a different state if he knows when his connecting train will precisely arrive than if he is absolutely clueless about specific delays. This is what makes the difference between Examples 1 and 3, and the difference between 2, 4 or 5. Example 3 shows that knowledge encompasses data on which the user will base his next action. Example 4 shows that the decisions that the user has taken in the past are also part of his knowledge. If he enters a state that does not encode his decision he forgets what he decided. Example 5 illustrates that the user's knowledge also contains every information that the user has gathered so far on his journey. For example he might have learned while waiting at a stop that a train tr is delayed even though the user does not intend to take tr. It is possible that the user will never base a decision on this data and might just as well forget about it. However it is also possible that he misses a connecting train and therefore considers if the train tr is a good alternative. He might determine that if tr was on time he would not be able to catch it but as he knows that it is delayed he can get it. Note that even our most complex model does not allow modeling Example 5.

Every model further defines how the user transitions from one state into another one. We distinguish two types of states: *action-states* and *happen-states*. If the user is in an action state then he can choose his next state from a set of valid next states. He may base his decision only on information encoded inside of his current state. An example for an action state is the user standing at a stop and choosing what train to take. If the user is in a happen-state then he has no direct control over his next state. The model defines a set of valid next states and a probability distribution that defines how likely a specific next state is. This distribution is constant over time and is known by the user. For example if the user sits in a train with no delay he might transition into a state with delay or with no delay. As the user can not influence the delays he can not decide in what state he will end up. However he has a statistic about the previous delays and therefore knows how likely what next state is.

We consider the problem of guiding the user towards a target stop given a start stop and start time. The initial state the user is in is called the *start-state*. A state that the user wants to reach is a *target-state*. In every model there is for every moment in time a state that models the user standing at a specific stop at that time. This means that there is only one start-state. There is however for every moment in time a target-state. Note that every target state is associated with an *arrival time*. One way of guiding the user is to compute for every action-state the best choice that the user can make to get to his target. This is called a *strategy*. We can print out a list of action-states and the recommended actions and hand it to the user. The main problem with this approach is that the list is too large to be of use to the user. We therefore only compute the actions for the action-states that the user can reach from the start state.

More precisely we compute a *decision graph*. The nodes of this graph are formed by states. Every action-state has exactly one out-arc that represents the action that the user should take and that ends in the next state as defined by the model. Every happen-state has an out-arc to all of the next states as defined by the model and is associated with the probability of that transition. Note that the probabilities associated with the out-arcs of one happen-state must add up to 1. For convenience, we define that the out-arc of an action-state is associated with probability 1. The target-states are the only states that have no out-arcs. The start-state must be in the decision graph and every other node must be reachable from it. Note that without time discretization decision graphs have an infinite set of nodes and arcs as there is an infinite number of moments in time. Figure 5.1 illustrates an example decision graph with a very rough time discretization.

For every decision graph and every state q in it is possible to compute an *expected arrival time*. Let $\Pr(a)$ denote the probability associated with arc a. Further we denote by P the set of paths that start at q and end at a target-state. Denote by p_{arrtime} the arrival time of a path p and by A_p the set of its arcs. Given a fixed decision graph and a state q in it we define the expected arrival time as

$$f(q) \quad = \quad \begin{cases} q\text{'s arrival time} & \text{if } s \text{ is a target state} \\ +\infty & \text{if } P = \emptyset \\ \sum_{p \in P} \left(p_{\text{arrtime}} \cdot \prod_{y \in A_p} \Pr(y) \right) & \text{otherwise}\,. \end{cases}$$

Given a state q the goal is to compute a decision graph that has q as its start state and minimizes $f(q)$, i.e., to compute a decision graph with a *minimum expected arrival time* (M.E.A.T.). We refer to this as the *M.E.A.T.-problem*. We denote by A the set of all arcs in a specific decision graph. The first central observation is that $f(q)$ can be formulated

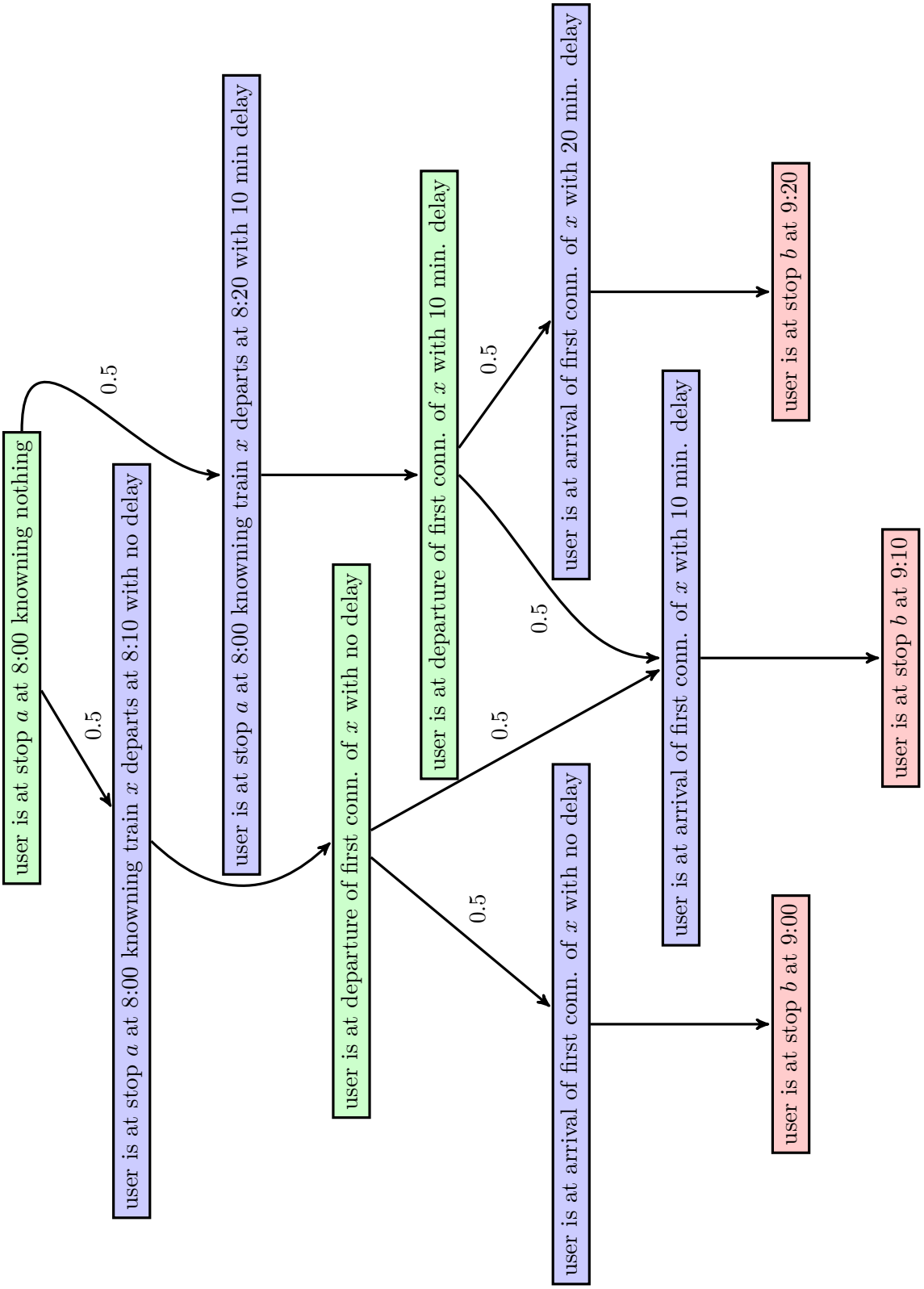


Figure 5.1: An example of a simplified decision graph with a start stop a, a target stop b and a trip x. The out-arcs of happen-states are labeled with their probability. The expected arrival time is $0.5^2 \cdot 9{:}00 + 2 \cdot 0.5^2 \cdot 9{:}10 + 0.5^2 \cdot 9{:}20 = 9{:}10$. Action states are in blue, happen states in green and target states in red.

recursively as

$$f(q) \quad = \quad \sum_{(q,q')\in E} \Pr(q,q')f(q')$$

and this formula suggests that is can be efficiently evaluated using dynamic programming, which is exactly what we do in Section 5.2.2. Denote by $g(q)$ the minimum $f(q)$ over all valid decision graphs starting at q and by Y the set all arcs allowed by the model. Suppose that $g(q)$ is known for every state except q then we can compute $g(q)$ as

$$g(q) = \sum_{(q,q')\in Y} \Pr(q,q')g(q')$$

if q is a happen-state and as

$$g(q) = \min_{(q,q')\in Y} g(q')$$

if x is an action-state.

5.1 Formal Definition

This is section is a very formal approach to the topic. A more informal illustration of the basic concepts specific to the problem setting our algorithm solves is given in Section 6.1. In this section We define the set of states and the set of next states for every action- and happen-state. For every possible transition we indicate a recursive formula to compute the corresponding minimum expected arrival time. All models share the same set of states. They differ in the allowed transitions. The more realistic models use more complex transitions. Every transition option belongs to a certain category. Choosing an option for every category yields a precise formal model definition. Given this precise definition we can consider the problem of determining a decision graph with a minimum expected arrival time.

The options listed here try to formalize the following aspects:

- Do the trains arrive delayed?
- Do the trains depart delayed?
- Are the delays at different connections in one trip stochastically independent?
- Can a train decrease its delay at a stop by waiting for a shorter time than planned?
- When does the user decide at what stop to exit the train? Is it when he enters the train or when the train enters the exit stop?

We start by formalizing the states. We group similar states into a class and describe them together. The list of classes is:

Stand-at-Stop. The user stands at a stop s at a moment t. He does not know the precise delays of any possible next train. He has not yet decided what train to take. This is a happen-state as the user gathers knowledge and can be represented by a (t, s)-pair.

Deciding-Next-Train. The user stands at a stop s at a moment t. He knows the delay of some possible next trains. We formalize this knowledge as a set Ne of (c, d)-pairs where c is a departing connection and d its delay. One can interpret Ne as a partial function. If $\text{Ne} = \emptyset$ then the user knows no precise delays. A state is described using a (t, s, Ne)-triple. Note that this is the only state where the tuple components are not only simple numbers. This is an action-state.

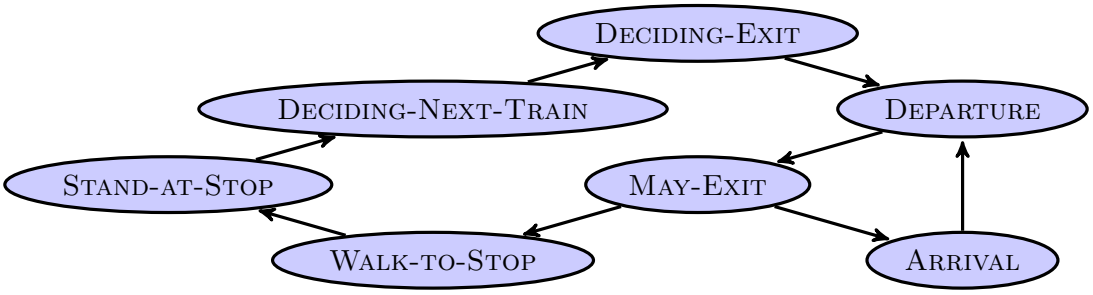


Figure 5.2: A graph containing the state classes as nodes. If a class A contains a state who has a next state in class B then the graph contains an arc (A, B).

Walk-to-Stop. The user has exited a train and stands at a stop s at a moment t. He may decide to walk along a footpath or remain at s. This is an action-state and can be represented by a (t, s)-pair.

Deciding-Exit. The user has just entered a train c delayed by d and must decide how he wants to exit it. A state is described using a (c, d)-pair. This is an action-state.

Departure. The user sits in a train c delayed by d that is about to leave a stop. The user may have fixed an exit connection e at the end of which he wants to exit the train. If not then e is a dummy connection, denoted by $\perp$. This state can be described using a (c, d, e)-triple. This is a happen-state.

May-Exit. The train c has just entered a stop. The user must decide whether he wants to exit it or remain seated. The train is delayed by d. Again the state also has an exit connection e. This state can be described using a (c, d, e)-triple. This is an action-state.

Arrival. The train c has just entered a stop. The user does not want to exit. The train is delayed by d. This state can be described using a (c, d, e)-triple. This is a happen-state.

The allowed transitions differ from model to model, however, all of them connect the same state classes. Figure 5.2 illustrates for every state class what the allowed direct next state classes are.

As already discussed briefly a time discretization is needed to construct finite decision graphs. We discretize the time differently for different state classes. Time components that describe a moment in time, such as those in Stand-at-Stop, Walk-to-Stop and Deciding-Next-Train, are sampled at a regular time interval. Time components that describe a delay, such as those in Deciding-Exit, Departure, May-Exit and Arrival, must be chosen from a finite set of valid delays D. We require that a delay of 0 is valid and that every delay must be positive. A small set of valid delays leads to fast running times but low accuracy. We do not require that the delays are sampled at regular intervals to allow a higher precision for small delays that generally are more relevant.

5.1.1 Notation

In the next subsections we specify how the M.E.A.T. of a state (always denoted by f) can be expressed in terms of the M.E.A.T. of its next states (denoted by g and h). Recall that C is the set of connections, F the set of footpaths, S the set of stops and D the set of valid delays. We denote by $C_s \subseteq C$ the connections that depart at the stop s and by $F_s \subseteq F$ the footpaths that start there.

5.1.2 Next States of Stand-at-Stop

A Stand-at-Stop state represents a user that is at a stop and decides what to do next. If he has arrived at the target stop then he finishes his journey, otherwise he gathers information about the precise delays of some trains. This gathering process is modeled using the transition into the Deciding-Next-Train state. Note that the minimum change time and footpaths do not have to be considered because they are handled by the transition from the Walk-to-Stop state into the Stand-at-Stop state.

The M.E.A.T. of a Stand-at-Stop state is $f(t, s)$ and the M.E.A.T. of a Deciding-Next-Train state is $g(t, s, \mathrm{Ne})$. If the user has arrived at his target then the M.E.A.T. can be computed using

$$f(t, s) = t$$

otherwise the next states are relevant. Now follows a list of the different options to define the valid next states.

Departure-on-Time. *The departures are always on time.* This is not very realistic but easy to implement. The only valid next state is the (t, s, Ne) Deciding-Next-Train state where Ne maps every connection onto 0.

$$f(t, s) = g\left(t, s, \{c \mapsto 0\}\right)$$

Departure-Delay-Unknown. *The user has no way of finding out when what train departs.* The only valid next state is $(t, s, \emptyset)$.

$$f(t, s) = g(t, s, \emptyset)$$

Departure-Delay-Random. *A departure c has a random departure delay.* We denote by $p_c(d)$ the probability that the connection c leaves with the precise delay d. The user gathers information about every connection that departs at s. The possible next states are the (t, s, Ne) states, where Ne contains an entry for every departing connection. Note that Ne can be interpreted as a function that maps a connection onto its precise delay. The possible delays are not restricted. A state (t, s, Ne) has a probability of $\Pi_{c \in C_s} p_c(\mathrm{Ne}(c))$. The M.E.A.T. can be calculated as following:

$$f(t, s) = \sum_{\mathrm{Ne}} g(t, s, \mathrm{Ne}) \prod_{c \in C_s} p_c(\mathrm{Ne}(c))$$

As a further extension one could consider dependencies between trains. For example it often happens that the last train in a route waits for other delayed trains. The delays are therefore no longer independent. A shortcoming of all the models presented here is that the user forgets everything he knows about the delays of other trains once he enters a train. However, dropping this simplification dramatically increases the number of states and therefore seems intractable using a dynamic programming approach. A result of this is that if he may get the same train at different stops the user supposes that their delays are independent and therefore might try to get the same train twice.

For example consider the case where the user is, according to the timetable, capable of catching the same train at two different stops. He tries to catch it at the first stop but it is delayed and therefore he takes another train. The user can derive from this that the train he originally intended to catch will also be delayed at the second stop. This, however, requires that he remembers that the train is delayed. The simplification introduced, however, supposes that he forgets about this information.

5.1.3 Next States of Deciding-Next-Train

The Deciding-Next-Train state represents a user that is at a stop and knows some exact delays. He must decide what train to take. This decision basically consists of choosing a departing connection. There are two groups of connections that depart at the stop s. The first group $C_{s,1}$ consists of the connections the user knows the precise delays of, i.e., those for which Ne has an image. The second group $C_{s,2}$ are the remaining ones.

If the user knows the exact delay of a connection (i.e. one in $C_{s,1}$) then he can reach one of the Deciding-Exit states with certainty. If this is not the case then for every possible delay a next state can be reached. The probability that a connection c is delayed by d is denoted by $p_c(d)$.

The M.E.A.T. of a Deciding-Next-Train state is $f(t, s, \mathrm{Ne})$ and the M.E.A.T. of a Deciding-Exit state is $g(c, d)$. We set $\min \emptyset = +\infty$.

$$f(s,t,\mathrm{Ne}) = \min \left\{ \min_{\substack{c \in C_{s,1} \\ t \le c_{\mathrm{deptime}} + \mathrm{Ne}(c)}} g(c, \mathrm{Ne}(c)), \min_{\substack{c \in C_{s,2} \\ t \le c_{\mathrm{deptime}}}} \sum_{d \in D} p_c(d) g(c,d) \right\}.$$

Consider a train that is scheduled to leave before the user could get it (i.e. $t > c_{\mathrm{deptime}}$). Normally the user would not be able to get it, but if it is delayed enough then it will still be considered as $t \le c_{\mathrm{deptime}} + \mathrm{Ne}(c)$ may hold. This only works if the user knows the precise delay. If he does not know it then there is a chance that he already missed the train. If this is the case then he would wait forever and therefore he always avoids this situation and never takes such a train.

5.1.4 Next States of Walk-to-Stop

The user has just exited a train and can walk to another stop. The M.E.A.T. of a Walk-to-Stop state is $f(t, s)$ and the M.E.A.T. of a Stand-at-Stop state is $g(t, s)$. Recall that the set of footpaths that start at a stop s is denoted by F_s. The formula is

$$f(t,s) = \min \left\{ \begin{array}{c} g(t + s_{\mathrm{minchange}}, s), \\ \min_{x \in F_s} g(t + s_{\mathrm{minchange}} + x_{\mathrm{dur}} + (x_{\mathrm{arrstop}})_{\mathrm{minchange}}, x_{\mathrm{arrstop}}) \end{array} \right\}.$$

5.1.5 Next States of Deciding-Exit and May-Exit

The Deciding-Exit and the May-Exit states are tightly coupled and therefore are both explained in one section. A Deciding-Exit state represent a user that entered a train and must decide where he exits it. A May-Exit state models a user in a train that enters a stop. The user may exit the train at this point. The action of the user in a May-Exit state depends on the decision in the Deciding-Exit state. The decision of the user is encoded in the state using the exit connection e tuple component.

We consider two options:

Exit-at-Train-Enter. *The user decides when entering a train where he exits it.* In this model the decision is only based on the precise delay of the train at the moment the user enters it. He does not know with certainty how the delay evolves.

Exit-at-Stop-Enter. *The user decides when entering a stop if he exits the train.* In this model the user may take the delay of the train at the moment it enters a stop into account.

These two options define the next states of the DECIDING-EXIT class and MAY-EXIT class. We first consider the DECIDING-EXIT class.

Next States of DECIDING-EXIT

The M.E.A.T. of a DECIDING-EXIT state is $f(c, d)$ and the M.E.A.T. of a DEPARTURE state is $g(c, d, e)$. If the EXIT-AT-STOP-ENTER option is used the user has nothing to decide. The exit connection is set to a dummy value, denoted by $\perp$, i.e.,

$$f(c, d) = g(c, d, \perp) .$$

If the EXIT-AT-TRAIN-ENTER option is used then the user may exit at any connection after c in the same trip, i.e.,

$$f(c, d) = \min_e g(c, d, e) .$$

Next States of MAY-EXIT

The M.E.A.T. of a MAY-EXIT state is $f(c, d, e)$, the M.E.A.T. of a WALK-TO-STOP state is $g(t, s)$ and the M.E.A.T. of an ARRIVAL state is $h(c, d, e)$. If c is the last connection in a trip, then the user must always exit the train, i.e.,

$$f(c, d, e) = g(c_{\text{arrtime}} + d, c_{\text{arrstop}}) .$$

Otherwise the behavior of the user depends on the option chosen. If the EXIT-AT-STOP-ENTER option is used then the user has two options to chose from, i.e.,

$$f(c, d, \perp) = \min \{g(c_{\text{arrtime}} + d, c_{\text{arrstop}}), h(c, d, e)\} .$$

If the EXIT-AT-TRAIN-ENTER option is used the the following formula defines his behavior.

$$\begin{aligned} f(c, d, e) &= h(c, d, e) && \text{if } c \neq e \\ f(c, d, c) &= g(c_{\text{arrtime}} + d, c_{\text{arrstop}}) \end{aligned}$$

5.1.6 Next States of DEPARTURE

A train transition from a DEPARTURE state into a MAY-EXIT state represents the train actually moving. It is possible that the train changes its delay while doing so. Recall that D is the set of valid delays. The M.E.A.T. of a DEPARTURE state is $f(c, d, e)$, the M.E.A.T. of a MAY-EXIT state is $g(c, d, e)$. We formalize the following options:

TRIP-DELAY-CONSTANT. *The delay of a train always remains constant, i.e.,*

$$f(c, d, e) = g(c, d, e) .$$

This is unrealistic but is easy to implement. Note that the train may have an initial delay. This is the difference to the model without any delays.

TRIP-DELAY-RANDOM. *The delay of a train changes randomly.* We denote by $p(c, d_{\text{dep}}, d_{\text{arr}})$ the probability that the train will have a delay of d_{arr} if it started with a delay of d_{dep}. It must be impossible for the user to arrive before he departs (i.e. $p(c, d_{\text{dep}}, d_{\text{arr}}) = 0$ for every departure d_{arr} such that $c_{\text{deptime}} + d_{\text{dep}} \geq c_{\text{arrtime}} + d_{\text{arr}}$). The functions relate as follows.

$$f(c, d_{\text{dep}}, e) = \sum_{d \in D} p(c, d_{\text{dep}}, d) g(c, d, e)$$

5.1.7 Next States of ARRIVAL

The transition from an ARRIVAL into a DEPARTURE state represents the train waiting at a stop. Note that this transition does not handle the user exiting the train or the trip ending. (This is what the MAY-EXIT state is for.) The train is scheduled to wait for a certain amount of time at the stop. By decreasing this waiting buffer time the train can decrease its delay. Recall that D is the set of valid delays. The M.E.A.T. of a ARRIVAL state is $f(c, d, e)$, the M.E.A.T. of a DEPARTURE state is $g(c, d, e)$. We formalize the following options:

STOP-DELAY-RESET. *The waiting buffer time is larger than every delay.* This is easy to implement and for small delays very accurate. The M.E.A.T. values relate as follows.

$$f(c, d, e) = g(c_{\text{next}}, 0, e)$$

STOP-DELAY-CONSTANT. *The waiting buffer time can not be decreased.* A (c, d, e) ARRIVAL state is followed by the (c_{next}, d, e) DEPARTURE state. The M.E.A.T. values relate as follows.

$$f(c, d, e) = g(c_{\text{next}}, d, e)$$

STOP-DELAY-MAX-REDUCE. *The waiting buffer time is decreased as much as possible.* The train arrives at $c_{\text{arrtime}} + d_{\text{old}}$ and departs at $(c_{\text{next}})_{\text{deptime}}$. The waiting buffer time is therefore $c_{\text{arrtime}} + d_{\text{old}} - (c_{\text{next}})_{\text{deptime}}$. This is the maximum amount by which the delay can be reduced. As this does not always result in a valid delay some rounding is necessary. Formally this can be expressed as

$$f(c, d_{\text{old}}, e) = g(c_{\text{next}}, \min\{d_{\text{new}} \in D \mid d_{\text{new}} \geq c_{\text{arrtime}} + d_{\text{old}} - (c_{\text{next}})_{\text{deptime}}\}, e)\,.$$

5.2 Computing the Minimum Expected Arrival Time

In this section we describe algorithms that compute the minimum expected arrival time for the various models introduced. We start with a recursive program that is derived from the minimum expected arrival time formulas introduced in the previous section and prove its termination. We then turn this recursive program into a dynamic one. (In Chapter 6 we construct an efficient algorithm for one specific model based on the dynamic program.) The programs introduced in this section work for all problems that use DEPARTURE-ON-TIME or do not use STOP-DELAY-RESET. However for many problems they are too slow to be of practical use. Especially the DEPARTURE-DELAY-RANDOM option seems impossible to handle because of the exponential number of states.

State Class Name	Tuple	Time Potential 1	Time Potential 2
Stand-at-Stop	(t, s)	t	t
Deciding-Next-Train	(t, s, Ne)	t	t
Walk-to-Stop	(t, s)	t	t
Deciding-Exit	(c, d)	$c_{\text{dep}} + d$	c_{dep}
Departure	(c, d, e)	$c_{\text{dep}} + d$	c_{dep}
May-Exit	(c, d, e)	$c_{\text{arr}} + d$	c_{arr}
Arrival	(c, d, e)	$c_{\text{arr}} + d$	c_{arr}

Table 5.1: Time potentials for problems without the Stop-Delay-Reset option. The departure time of a connection c is c_{dep} and its arrival time is c_{arr}.

All algorithms first compute the minimum expected arrival time for a fixed target stop and every possible start state. Afterwards, they use the M.E.A.T. to reconstruct an optimal decision graph. Constructing the decision graph is normally not the bottleneck and therefore we only consider the problem of computing the minimum expected arrival time. In the previous section we presented recursive formulas to compute the M.E.A.T. in the various states, which can be used to construct a recursive program. It remains to show that this program does not cause an endless recursion. We show that this is the case if Stop-Delay-Reset is not combined with Departure-Delay-Random or Departure-Delay-Unknown. The problem with these two combinations is that they lead to time travel as the following example illustrates:

Example 4. The user leaves a stop with a delay of 40 min at a planned departure time of 10:00. He arrives at the second stop at a planned arrival time of 10:10 with a delay of 40 min. He remains seated. The delay is reseted. The train will leave at the planned 10:20 on time and arrive at 10:30 on time at the next stop. The user effectively departed at 10:40 but arrived at 10:30. He thus traveled 10 min back in time.

We will only consider models that do not use Stop-Delay-Reset or use Departure-on-Time. An important tool in this section is the so called *Time Potential*. This is a function that maps every state onto a moment in time. Every state must have a time potential smaller or equal to the potentials of all its valid next states. Table 5.1 shows two choices for this function. We use Potential 1 if Stop-Delay-Reset is not used. Otherwise Departure-on-Time must be used and we use Potential 2. It is possible to verify that these choices are valid by looking at the definitions of the M.E.A.T. formulas in the previous section.

For example using Stop-Delay-Constant the user can transition from an (c, d, e) Arrival state into a (c_{next}, d, e) Departure state. The Time Potential 1 of the first $c_{\text{arrtime}} + d$ and of the second is $(c_{\text{next}})_{\text{deptime}} + d$. As $c_{\text{arrtime}} + d \leq (c_{\text{next}})_{\text{deptime}} + d$ holds the potential is valid for this formula.

5.2.1 Recursive Program

The formulas given in the previous section lead to a recursive program. Based on the correctness of these formulas it is possible to see that if the program terminates it computes the correct value. It remains to show that the program terminates (i.e. no endless recursion occurs).

Theorem 5. *Consider a problem that does not use* Stop-Delay-Reset *or uses* Departure-on-Time. *Further consider a fixed target stop and a non-empty network. It is possible*

to compute the M.E.A.T. of any state in finitely many steps using the functions given in section 2.1.

Proof. We proof this by showing that:

1. *There exists a time potential.* We already showed that this is the case.
2. *There exists a constant $\epsilon > 0$ such that every state only indirectly depends on states of its own class with a time potential bigger by at least ϵ.* See below for a proof.
3. *There exists a minimum time potential $p_{\min}$ and a maximum time potential $p_{\max}$.* As there are only finitely many connections (and at least one exists) there are only finitely many reachable states (and at least one exists) and therefore the minimum and the maximum time potential exists.
4. *Every state has a finite number of possible next states.* This is the case because the time is discretized and the considered network is finite.

Using the first three statements it is easy to see that the maximum recursion depth is $\frac{p_{\max}-p_{\min}}{\epsilon}$. Adding the fourth one shows the theorem.

It remains to show that the required ϵ exists. Consider two states of the same class. As shown in figure 5.2 they may only indirectly depend on each other. In every case the user must transition through a DEPARTURE state. If the TRIP-DELAY-CONSTANT option is used then the time potential is increased when transitioning trough a DEPARTURE state by at least

$$\epsilon := \min_{c \in C} c_{\text{arrtime}} - c_{\text{deptime}}$$

where C is the set of all connections. The minimum exists because C is finite and non-empty. If the TRIP-DELAY-RANDOM-option is used then the choice for ϵ is more complicated:

$$\epsilon := \min_{\substack{c \in C \\ d_{\text{dep}} \in D \\ d_{\text{arr}} \in D \\ c_{\text{deptime}} + d_{\text{dep}} < c_{\text{arrtime}} + d_{\text{arr}}}} c_{\text{arrtime}} + d_{\text{arr}} - c_{\text{deptime}} - d_{\text{dep}}$$

where D is the set of valid delays. The set over which we compute the minimum is not empty because we can indicate an element in it (set $d_{\text{dep}} = 0$, $d_{\text{arr}} = 0$ and c to some element in C). Further it is finite because C and D are finite. Hence it has a minimum. □

Remark 6. The termination proof holds only for finite networks. See section 5.3 for an example where the recursive program would not terminate as the minimum decision graph is infinite.

5.2.2 Dynamic Program

In this section we turn the recursive program into a dynamic one. Let us recall that a query consists of a start stop, a start time and a target stop. Our dynamic program starts by enumerating all the possible states and then sorts them topologically. We use a special topological sorting that can be constructed using the time potential. For the given target stop we compute the M.E.A.T. to it from every other state. We first compute the M.E.A.T. of the states at the end of the topological sorting. These states do not depend on any other states. Next we compute the states before these. They only rely on the M.E.A.T. of states that have already been computed. This way we never have to make a recursive function call. Once all the M.E.A.T. are computed we construct and output the decision graph of the (start-stop, start-time)-STAND-AT-STOP-state. See 5.1 for an overview over the basic program. Note that this algorithm determines the M.E.A.T. for every decision graph that ends at the target stop. It is therefore a all-to-one algorithm.

Algorithm 5.1 Dynamic Program

```
enumerate all state ;
sort the states topologically ;
for all states s decreasingly do
    Compute the M.E.A.T. m ;
    store m ;
construct the start state decision graph using the stored M.E.A.T. ;
output this decision graph ;
```

5.2.3 Topological Sorting using the Time Potential

In this section we show how to topologically sort the states using the time potential. We want to assign an index to every state such that all valid next states have a higher index. We first sort the states ascending by their time potential. We then sort all states with the same time potential by their class. The order on the classes is given by MAY-EXIT < ARRIVAL < WALK-TO-STOP < STAND-AT-STOP < DECIDING-NEXT-TRAIN < DECIDING-EXIT < DEPARTURE. When removing the out-arc of the DEPARTURE-state, this order is basically a topological sort of the graph given in figure 5.2. The key idea is that the removed transition strictly increases the time potential and therefore no state (indirectly) depends on itself. We claim that the order of the states constructed in this way is a topological one.

Lemma 7. *First sorting by time potential and then by state class yields a topological sort of the states.*

Proof. Consider two states q and q' where q' is a (direct) next valid state of q. It is sufficient to show that q is ordered before q'. We denote by t and t' their time potentials. One of the following cases must be true:

1. $t < t'$: If this is the case then q is ordered before q' just as wanted.
2. $t = t'$ and q is not in the DEPARTURE-class: The time potentials are equal and therefore the order of q and q' is given by the order on the classes which sorts the states as needed.
3. $t = t'$ and q is in the DEPARTURE-class: This can not happen because all the valid next states of a DEPARTURE-state have time potential that is bigger by at least some constant $\epsilon > 0$. See the proof of termination theorem 5 for the exact value of ϵ.
4. $t > t'$: This can not happen because it violated the time potential definition.

□

5.3 Problems with Infinite Extensions

In this section we considers periodic networks as introduced in Section 2.7. The proof of termination Theorem 5 uses the fact that only finite networks are considered. This is certainly valid but seems more restrictive than needed. We show that it is not the case and that finiteness is a key ingredient. Note that the network transformations introduced in Section 2.7 do not apply here as we do not only consider earliest arrival journeys.

Consider the periodic network illustrated in figure 5.3. We show that it admits no finite optimal decision graph. It is obvious that every decision graph must first tell the user to take the $s \to x$ connection. After that the decision graphs differ. Discarding obviously suboptimal decision graphs there remain two candidates:

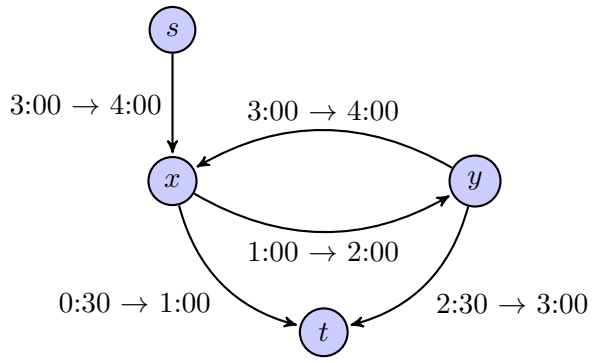


Figure 5.3: The user wants to get from s to t. The start time is 3:00. The time period is 4 hours. A train has a maximum delay of 40 min. The chance that the delay is less than 30 min is p. The trains always depart on time.

1. The user tries to get the $x \to t$ connection and succeeds with probability p. If this does not work he waits a period and then takes the $x \to t$. As the waiting time far exceeds the maximum delay he succeeds with certainty. The M.E.A.T. m_1 is

$$m_1 = p \cdot 5 + (1-p) \cdot 9 = -4p + 9$$

2. The user tries to get the $x \to t$ connection and succeeds with probability p. If this does not work he uses the $x \to y$ connection with certainty. Now he tries to get the $y \to t$ connection with probability p. If this again fails he uses the $y \to x$ connection. We can recursively define the M.E.A.T. m_2 as

$$m_2 = p \cdot 5 + p\,(1-p) \cdot 9 + (1-p)^2\, m_2$$

which can be solved for m_2:

$$m_2 = \frac{9p - 14}{p - 2}$$

The M.E.A.T. are the same for $p = 1$ and otherwise m_2 is always smaller. Consequently, the decision graph with the infinite number of nodes is optimal for $p \neq 1$.

It is possible to turn this decision graph into a finite cyclic one by reducing the moments in time stored in the states modulo the time period. However even this does not lead to any obvious algorithm to compute the M.E.A.T. nor the decision graph. We know of no algorithm capable of solving the M.E.A.T. problem for periodic networks.

6. Minimum Expected Arrival Time Algorithm

In this section, we propose an algorithm to efficiently solve the M.E.A.T.-problem for one of the simpler models introduced in the previous chapter. All connections have an independent random positive delay at their arrival. We suppose that the delay at the departure is negligible. The goal is to compute a decision graph that has a minimum expected arrival time at a given target stop. This corresponds to the usage of the Departure-on-Time, Exit-at-Train-Enter, Trip-Delay-Random and Stop-Delay-Reset options. We first illustrate a simplified variant of this problem setting to give an intuition about the problem. We then simplify the formulas introduced in the previous Chapter and construct an algorithm to solve this problem that is nearly identical to the backward profile algorithm introduced in Chapter 4. The only difference is the evaluation function of step functions. Finally, we describe how to generate the diagrams such as those shown in Figure 1.2.

6.1 Problem and Algorithm Illustration

The goal of this section is to give an intuition about the problem setting and the algorithm. For this reason we only consider a vastly simplified variant. Suppose that there are no footpaths, no minimum change times and all trips are composed of a single connection. We represent a connection c as illustrated in Figure 6.1. The horizontal axis is the time. Left is an early moment and right a later one. Every stop is identified using a horizontal line. The name of a stop is indicated at the right of it's line. The left end of a stop's line represents an early moment at that stop whereas the right end represents a later one.

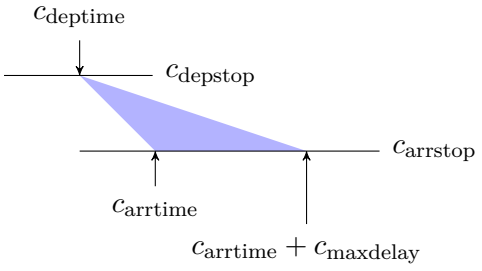


Figure 6.1: How a connection is represented.

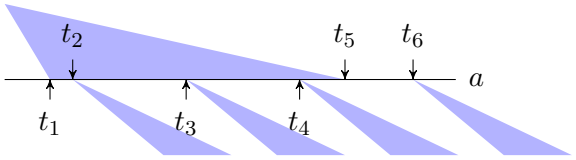


Figure 6.2: A stop with one incoming and four departing connections.

Connections are represented using triangles. They depart at their departure stop precisely at c_{deptime} and arrive somewhere between c_{arrtime} and $c_{\text{arrtime}} + c_{\text{maxdelay}}$ at the target stop. The top edge of the triangle represents the user being at c_{deptime} at the stop c_{depstop} and the bottom side him being at c_{arrstop} in some given interval. We do not only know that the user arrives within $[c_{\text{arrtime}}, c_{\text{arrtime}} + c_{\text{maxdelay}}]$ but we also know the probability of him having a specific delay even though the figure does not indicate this.

The problem variants considered in this Chapter are equivalent to selecting a subset of connections. The user is always supposed to take the next possible connection out of this subset. Given the delay distributions it is possible to compute an expected arrival time. Consider the situation illustrated in Figure 6.2. The user arrives at a stop a somewhere between t_1 and t_5 and there are four departing connections that he can take. If the first and the second departing connections are in the subset then the user takes the first if he arrives between t_1 and t_2 and the second one if he arrives between t_2 and t_3. If only the second one is in the subset then he takes that one if he arrives between t_1 and t_3. Note that the fourth departing connection must always be in the subset because otherwise the user would be not be able to depart from the stop if he arrived between t_4 and t_5. Suppose that the expected arrival time is know for each of the departing connections then it is possible to compute the expected arrival time for the incoming connection given a subset choice. Suppose that t is the arrival time of the user and e_1, e_2, e_3 and e_4 are the expected arrival times of the four departing connections. Further suppose that the first, third and the fourth departing connections are in the subset. Then the expected arrival time of the incoming connection is

$$P\left[t_1 \leq t \leq t_2\right] e_1 + P\left[t_2 < t \leq t_4\right] e_3 + P\left[t_4 < t \leq t_5\right] e_4 \,.$$

If only the second and the fourth connections were in the subset then the expected arrival time was

$$P\left[t_1 \leq t \leq t_3\right] e_2 + P\left[t_3 < t \leq t_5\right] e_4 \,.$$

Suppose that we did not select the fourth connection. In this case he could not leave the stop if he arrives after t_4. We defined the arrival time to be $+\infty$ in this case. As the chance of this happening is not zero (even though it might be very small) the expected arrival time would become $+\infty$. Even selecting every other departure results in $+\infty$.

$$P\left[t_1 \leq t \leq t_2\right] e_1 + P\left[t_2 < t \leq t_2\right] e_2 + P\left[t_3 < t \leq t_4\right] e_3 + P\left[t_4 < t \leq t_5\right] \infty = +\infty$$

This is the case regardless of how unlikely it is that the user arrives that late. There must always be a way for him to get away from the stop. Note that because all the connections depart on time it is possible to compute the subset that results in the minimum expected arrival time without iterating over all possible subsets. We already observed that the fourth connection must be in the subset. The third connection is in the subset exactly if $e_3 < e_4$, the second if $e_2 < \min\{e_3, e_4\}$ and the first if $e_1 < \min\{e_2, e_3, e_4\}$. Basically a connection is dominated if it is not better than another connection that departs later.

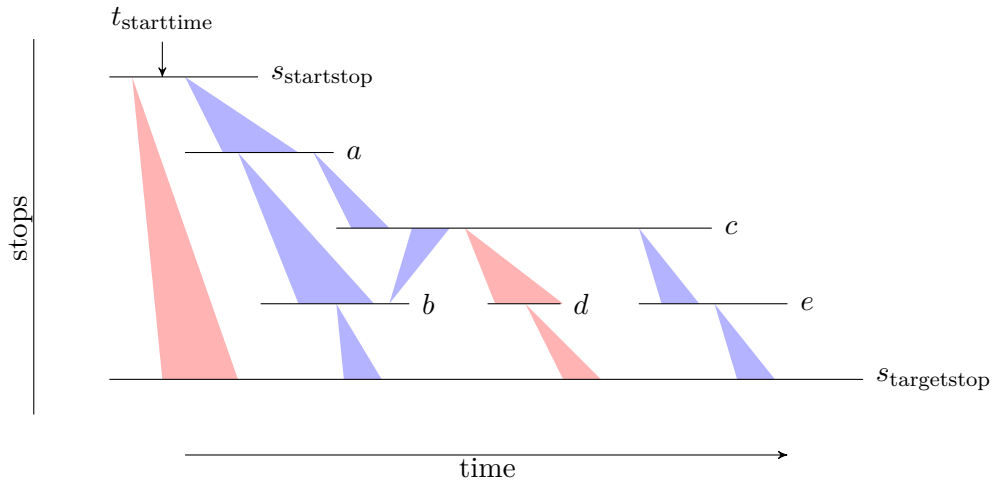


Figure 6.3: A timetable extract. The blue connections are in the decision graph. The red ones are not.

Our algorithm stores for every stop a dominated list of the departing connection along with their expected arrival time. This allows to quickly evaluate the expected arrival time of any incoming connection. Now consider the timetable extract in Figure 6.3. The user is at the start stop $s_{\text{startstop}}$ at the moment $t_{\text{starttime}}$ and wants to get to the target stop $s_{\text{targetstop}}$. We want to generate a decision graph for him. In the example above all connection in the decision graph are represented in blue whereas those that are not in it are red. The leftmost connection is not in the decision graph because it departs before the user arrives at the start stop. The journey along stop d has a high chance of being faster than the one along stop e but it also has a low chance of the user not being able to get away from d and therefore the journey along stop e is preferred.

6.2 Formula Simplification and Algorithm

First let us recall the relevant formulas. We denote by α the M.E.A.T. of a Stand-at-Stop state, by β the one of a Deciding-Next-Train state, by γ the one of a Deciding-Exit state, by δ the one of a Departure state, by ϵ the one of a May-Exit state, by ζ the one of a Arrival state and by η the one of a Walk-to-Stop state.

$$
\begin{aligned}
\alpha(t,s) &= \begin{cases} t & \text{if } s = s_{\text{targetstop}} \\ \beta\left(t, s, \{c \mapsto 0\}\right) & \text{otherwise} \end{cases} \\
\beta(s,t,\text{Ne}) &= \min \left\{ \min_{\substack{c \in C_{s,1} \\ t \le c_{\text{deptime}} + \text{Ne}(c)}} \{\gamma(c, \text{Ne}(c))\}, \min_{\substack{c \in C_{s,2} \\ t \le c_{\text{deptime}}}} \left\{ \sum_{d \in D} h_c(d)\gamma(c,d) \right\} \right\} \\
\gamma(c,d) &= \min_e \delta(c,d,e) \\
\delta(c, d_{\text{old}}, e) &= \sum_{d_{\text{new}} \in D} \Pr(c, d_{\text{old}}, d_{\text{new}}) \epsilon(c, d_{\text{new}}, e) \\
\epsilon(c,d,e) &= \zeta(c,d,e) \text{ if } c \neq e \\
\epsilon(c,d,e) &= \eta(c_{\text{arrtime}} + d, c_{\text{arrstop}}) \text{ if } c = e \\
\zeta(c,d,e) &= \delta(c_{\text{next}}, 0, e) \\
\eta(t,s) &= \min \left\{ \begin{array}{c} \alpha(t + s_{\text{minchange}}, s), \\ \min_{x \in F_s} \alpha(t + s_{\text{minchange}} + x_{\text{dur}} + (x_{\text{arrstop}})_{\text{minchange}}, x_{\text{arrstop}}) \end{array} \right\}
\end{aligned}
$$

There are a lot of constants in this system and therefore it can greatly be simplified. We make use of the facts that $\mathrm{Ne}(c) = 0$, $C_{s,2} = \emptyset$ and $C_{s,1} = C_s$ and get:

$$
\begin{aligned}
\alpha(t,s) &= \begin{cases} t & \text{if } s = s_{\text{targetstop}} \\ \beta\left(t, s, \{c \mapsto 0\}\right) & \text{otherwise} \end{cases} \\
\beta(s,t,\{c \mapsto 0\}) &= \min_{\substack{c \in C_s \\ t \le c_{\text{deptime}}}} \gamma(c,0) \\
\gamma(c,0) &= \min_e \delta(c,0,e) \\
\delta(c,0,e) &= \sum_{d_{\text{new}} \in D} \Pr(c,0,d_{\text{new}}) \epsilon(c,d_{\text{new}},e) \\
\epsilon(c,d,e) &= \zeta(c,d,e) \text{ if } c \neq e \\
\epsilon(c,d,e) &= \eta(c_{\text{arrtime}} + d, c_{\text{arrstop}}) \text{ if } c = e \\
\zeta(c,d,e) &= \delta(c_{\text{next}}, 0, e) \\
\eta(t,s) &= \min \left\{ \begin{array}{c} \alpha(t + s_{\text{minchange}}, s), \\ \min_{x \in F_s} \alpha(t + s_{\text{minchange}} + x_{\text{dur}} + (x_{\text{arrstop}})_{\text{minchange}}, x_{\text{arrstop}}) \end{array} \right\}
\end{aligned}
$$

By inlining β and ζ and eliminating constant zero parameters this can be simplified to:

$$
\begin{aligned}
\alpha(t,s) &= \begin{cases} t & \text{if } s = s_{\text{targetstop}} \\ \min\limits_{\substack{c \in C_s \\ t \le c_{\text{deptime}}}} \gamma(c) & \text{otherwise} \end{cases} \\
\gamma(c) &= \min_e \delta(c,e) \\
\delta(c,e) &= \sum_{d_{\text{new}} \in D} \Pr(c,0,d_{\text{new}}) \epsilon(c,d_{\text{new}},e) \\
\epsilon(c,d,e) &= \delta(c_{\text{next}}, e) \text{ if } c \neq e \\
\epsilon(c,d,e) &= \eta(c_{\text{arrtime}} + d, c_{\text{arrstop}}) \text{ if } c = e \\
\eta(t,s) &= \min \left\{ \begin{array}{c} \alpha(t + s_{\text{minchange}}, s), \\ \min_{x \in F_s} \alpha(t + s_{\text{minchange}} + x_{\text{dur}} + (x_{\text{arrstop}})_{\text{minchange}}, x_{\text{arrstop}}) \end{array} \right\}
\end{aligned}
$$

We show that δ does not depend on the exact value of c. More formally we show that:

Lemma 8. *For all connections x, y, z that are in the same trip where y is the connection directly after x and z is some connection after y it holds that $\delta(x,z) = \delta(y,z)$.*

Proof. The key idea is that the delays are reset after one connection and therefore do not propagate.

$$
\begin{aligned}
\delta(x,z) &= \sum_{d \in D} \Pr(x,0,d) \epsilon(x,d,z) \\
&= \sum_{d \in D} \Pr(x,0,d) \delta(y,z) \text{ because } x \neq z \\
&= \delta(y,z) \underbrace{\sum_{d \in D} \Pr(x,0,d)}_{=1} \\
&= \delta(y,z)
\end{aligned}
$$

□

Using this lemma we show the following lemma:

Lemma 9. *For all consecutive connections x and y it holds that $\gamma(x) = \min\{\gamma(y), \delta(x,x)\}$.*

Proof. Consider all the connections $c_1 c_2 \dots c_n$ that follow x in the same trip with $x = c_1$ and $y = c_2$. We can write $\gamma(x)$ as

$$\begin{aligned}
\gamma(x) &= \min_{i\in\{1\dots n\}} \delta(x, c_i) \\
&= \min\{\delta(c_1,c_1), \delta(c_1,c_2), \dots, \delta(c_1,c_n)\} \\
&= \min\{\delta(c_1,c_1), \delta(c_2,c_2), \dots, \delta(c_2,c_n)\} \text{ using Lemma 8} \\
&= \min\{\delta(c_1,c_1), \min\{\delta(c_2,c_2), \dots, \delta(c_2,c_n)\}\} \\
&= \min\{\delta(c_1,c_1), \gamma(c_2)\} \\
&= \min\{\delta(x,x), \gamma(y)\}
\end{aligned}$$

□

Using this lemma allows us to reformulate γ as

$$\begin{aligned}
\gamma(c) &= \min\{\gamma(c_{\text{next}}), \delta(c,c)\} \\
&= \min\left\{\gamma(c_{\text{next}}), \sum_{d\in D} \Pr(c,0,d)\epsilon(c,d,c)\right\} \\
&= \min\left\{\gamma(c_{\text{next}}), \sum_{d\in D} \Pr(c,0,d)\eta(c_{\text{arrtime}}+d, c_{\text{arrstop}})\right\}
\end{aligned}$$

and that allows us to reduce the system to

$$\begin{aligned}
\alpha(t,s) &= \begin{cases} t & \text{if } s = s_{\text{targetstop}} \\ \min\limits_{\substack{c\in C_s \\ t\le c_{\text{deptime}}}} \gamma(c) & \text{otherwise} \end{cases} \\
\gamma(c) &= \min\left\{\gamma(c_{\text{next}}), \sum_{d\in D} \Pr(c,0,d)\eta(c_{\text{arrtime}}+d, c_{\text{arrstop}})\right\} \\
&\quad \text{where } \gamma(c_{\text{next}}) = +\infty \text{ if } c_{\text{next}} = \perp \\
\eta(t,s) &= \min\left\{\begin{array}{c} \alpha(t + s_{\text{minchange}}, s), \\ \min_{x\in F_s} \alpha(t + s_{\text{minchange}} + x_{\text{dur}} + (x_{\text{arrstop}})_{\text{minchange}}, x_{\text{arrstop}}) \end{array}\right\}
\end{aligned}$$

At first these transformations seem to lead nowhere, however, the structure of the corresponding dynamic program (as described in Section 5.2.2) is now very similar to the profile algorithm described in Chapter 4. The expression $\eta(t,s)$ is a step function (for a fixed s) corresponds to $d(s)(t)$ and the function $\gamma(c)$ has the same role as $g(c)$. The jumps correspond to the states that are not dominated. The dynamic program visits the states using a topological ordering. Recall that ordering the connections by departure time yields such an ordering.

There are two differences to the backward profile algorithm. The first is how the user is allowed to end his journey. The backward profile algorithm does not account for the minimum change time at the target stop and it supposes that the final connection arrives on time. The equation system accounts for the minimum change time at the target stop

Algorithm 6.1 evaluation operation for M.E.A.T. algorithm

Recall that for every step function an array of jumps $(d_1, a_1) \dots (d_n, a_n)$ is stored and that enough memory has been allocated to extend it at the beginning.

```
i ← 1;
while d_i < x do
    i ← i + 1;
p ← P_c(d_i − x);
t ← p · a_i;
while p ≠ 1 do
    i ← i + 1;
    p′ ← P_c(d_i − x);
    t ← t + (p′ − p) · a_i;
    p ← p′;
return t;
```

Procedure `evaluate(`x`)`

and supposes that the arrival time can be delayed. Both of these differences are minor and can be fixed by initializing the arrival time for connections that arrive at the target stop with

$$c_{\text{arrtime}} + \left(\sum_{\substack{d \in D \\ d \leq d_{\max}}} \Pr(c, 0, d) \right) + (s_{\text{targetstop}})$$

instead of only c_{arrtime}.

The second difference is that the evaluation of the step functions is replaced by the computation of an expected value. Fortunately this is also an easy operation on step functions. We start by computing the distribution function of Pr, i.e.,

$$P_c(d_{\max}) = \sum_{\substack{d \in D \\ d \leq d_{\max}}} \Pr(c, 0, d)\,.$$

This allows us to sum up the probabilities in a given interval in constant time using

$$\sum_{\substack{d \in D \\ d \leq d_{\max} \\ d > d_{\min}}} \Pr(c, 0, d) = P_c(d_{\max}) - P_c(d_{\min})\,.$$

Consider some stop s, some connection c and some moment in time x. The goal is to evaluate $\sum_{d \in D} \Pr(c, 0, d)\eta(x + d, s)$. We denote by $(d_1, a_1) \dots (d_n, a_n)$ the list of jumps that corresponds to $\eta(\cdot, s)$. Recall that $d_i < d_{i+1}$, $a_i < a_{i+1}$ and $(d_n, a_n) = (+\infty, +\infty)$ must hold. Determine the first jump after x, i.e., the smallest i such that $x \leq d_i$. This allows use to compute the needed expression as

$$\sum_{d \in D} \Pr(c, 0, d)\eta(x + d, s) = P_c(d_i - x) \cdot a_i + \sum_{j \in \{i+1 \dots n\}} (P_c(d_j - x) - P_c(d_{j-1} - x)) \cdot a_j\,.$$

Algorithm 6.1 illustrates how this formula can evaluated in pseudo-code. The pseudo-code of the remaining parts of the algorithm are identical to the backward profile algorithm.

6.3 Delay Distribution Functions

In Section 2.4 we described how we determine c_{maxdelay}. The probability distributions we use are solely parametrized in this value. In this section we present a number of ways

to define the distribution function P_c. Recall that $P_c(x) = 0$ for $x < 0$ and $P_c(x) = 1$ for $x \geq c_{\text{maxdelay}}$ is required. Further every distribution function must further be strictly ascending. We only indicate the values for delays x with $0 \leq x < c_{\text{maxdelay}}$. We evaluate the following options:

1. *The distribution function is constant.* Strictly speaking this choice is not even valid but it certainly is the function that can be evaluated the fastest.

$$P_c(x) = 0.5$$

2. *The distribution function grows linearly*, i.e.,

$$P_c(x) = 0.5 + \frac{x}{2c_{\text{maxdelay}}}$$

3. *The distribution function follows an exponential law.* We use

$$P_c(x) = 1 - 0.4 \cdot \exp\left(-\frac{15x}{4c_{\text{maxdelay}}}\right).$$

   This choice is largely inspired by Disser et al. [7]. They use

$$s - \exp\left(\ln(1-a) - \frac{x}{b}\right)$$

   with $a = 0.6$, $b = 8$ and $s = 0.99$. We round s to 1 and observe that for $x = 30$ the function is nearly 1. After a few simplifications our function is obtained.

4. *A discretized exponential distribution function.* This is nearly the same function as number 3 but it is discretized with 30 interpolation points. The only advantage of this approach over number 3 is that it can be evaluated faster.

7. Evaluation

We compiled our programs using GCC v4.6.2 with full optimizations (i.e. `-O3`). We ran our experiments on an Intel Xeon E5-2670 processor with 8 cores clocked at 2.6 GHz with 64 GiB of RAM, 20 MiB of level 3 cache and 256 KiB of level 2 cache per core. If not stated otherwise the experiments used only one core.

We ran our tests on three different types of public transit networks.

1. *The German railway network.* The data was kindly provided by HaCon [26]. It is based upon the winter schedule of 2001/2002 and contains all trains operated by Deutsche Bahn. The data includes minimum change times but no footpaths.
2. *The European long distance railway network.* Again, the data was kindly provided by HaCon. It is based on the winter schedule of 1996/1997 and contains mostly long-distance trains. The data includes minimum change times but no footpaths.
3. *The public transit network of London.* It includes tube (subway), buses, tram, Dockland Light Rail (DLR), and ferries. We extracted a Tuesday from the periodic summer schedule of 2011, which is publicly available from the London Data Store [27]. The data includes footpaths but no minimum change times. The footpaths only model paths within the same station as described in Section 2.2. The footpaths are transitively closed.

Table 7.1 contains the sizes of these networks. Note that we removed a number of strange connections (for example those that occur twice or where $c_{\text{deptime}} = c_{\text{arrtime}}$) from the data and therefore the precise sizes of the network differ from those in other papers based on the same data.

All of these networks contain data for one day. However, journeys exist that run over midnight and therefore we also consider the periodic expansion of these networks as described in Section 2.7. Note that in the European long distance network journeys with

Network	#stops	#connections	#trips	#footpaths	avg. c_{maxdelay} [min]
Germany	6 822	487 649	47 660	—	27.5
Europe	30 517	1 621 269	162 508	—	27.5
London	20 843	4 941 823	127 881	45 652	20.0

Table 7.1: Sizes of various networks.

Network	expand time	#stops	#connections	#trips	#footpaths
Ger. Per.	6 min	6 822	856 791	95 094	—
Eur. Per.	130 min	30 517	3 944 507	477 911	—
Lon. Per.	300 min	20 843	6 421 670	272 758	45 652

Table 7.2: Sizes of various periodic networks.

Network	periods unrolled	#stops	#connections	#trips	#footpaths
Ger. Unroll	3	6 822	1 462 947	142 980	—
Eur. Unroll	5	30 517	8 106 345	812 540	—
Lon. Unroll	2	20 843	9 883 646	255 762	45 652

Table 7.3: Sizes of various networks.

a length of several days are so common that we do not consider the one period variant. Note that unrolling periods has nearly no effect on the average c_{maxdelay}. Table 7.2 contains the expanded sizes of these networks and the time needed to compute the expansion. The number of extra connections generated by the expansion depends on the network. It ranges from 33% to 59% extra connections. The average footpath duration is 2:30 min. The average minimum waiting time on the German network is about 4:30 min and 5:00 on the European network.

In Section 5.3 we demonstrate that periodic network expansions do not necessarily contain all the connections needed to correctly answer periodic M.E.A.T. queries. For this reason we also consider unrolled versions (i.e. versions with several shifted copies of the first period) of the networks when evaluating these algorithms. Table 7.3 contains the detailed network sizes. Note that we only unrolled enough periods to cover most journeys and therefore the periodic expansion of London has a few more trips than the unrolled version. This, however, is not a problem as the user is most likely not interested in journeys that last longer than a day through London.

The German network is the smallest one and the journeys usually last less than a day. The European network has more connections and stops and journeys last several days most of the time. The London network has more connections but less stops but the journeys rarely last longer that a day. The London network also includes footpaths.

We compare our algorithms to time-dependent approaches. It is important to note that we do not use the "Realistic Time-Dependent Model" as described by Pyrga et al. [14] but the "Coloring Model" as introduced by Delling et al. [5]. (The model is only described in the journal version of the paper.) This alternative model produces smaller graphs by exploiting information about routes and yields a measurable speedup.

We evaluated our algorithms with minimum change times enabled and disabled. Note that disabling them differs from setting them to zero. If it is disabled then the algorithms do not need the was-relaxed-flag g nor do they need the information about the previous connections c_{prev}. Removing them instead of setting them to trivial values reduces cache misses and the memory footprint and makes a measurable difference. Disabling minimum change times also disables trips but this does not affect the earliest arrival times. It does, however, make a difference for the minimum expected arrival times.

7.1 Earliest Arrival Problem

We evaluate the algorithms proposed in Chapter 3 and compare their running times to other algorithms that solve the same problem. Table 7.4 shows the running times of the

Network	SO	Time [ms]
Germany	○	2.18
Germany	•	1.13
Europe	○	9.97
Europe	•	5.31
London	○	3.90
London	•	2.21

Table 7.4: Running times of Time-Dependent Dijkstra using the "Coloring Model". The flag "SO" indicates if a stop criterion was used.

Time-Dependent Dijkstra algorithm [17, 16] using the "Coloring Model". Note that the implementation we have treats networks as periodic. The London instance contains more connections but also more routes and therefore the "Coloring Model" results in better graphs and therefore the running times are lower on the Europe network.

We ran 10 000 random queries with start and target stops selected uniformly at random. The starting time is selected uniformly at random within the first day. We evaluate the variant with and without minimum change times (MC) on the German and the European railway network. On the London network we only evaluate the variant with footpaths but without minimum change times. We ran tests with the start criterion enabled and disabled (SA). Unfortunately it nearly never possible to reach every stop and therefore the one-to-all stop criterion nearly never aborts the loop. We therefore only present running times for the one-to-one stop criterion (SO). The London network fulfills the requirements for the optimization described in 3.3.6 and therefore we can evaluate it (TR).

We measured the running times and the total number of connections processed in the main loop. Further, we measure the number of relaxations that were applied and improved the solution at the arrival stop of the connection and those that did not. If applicable we also measured the number of footpath relaxations. We also indicate the numbers relative to the total number of connections respectively footpaths. The results for networks without minimum change times are presented in Table 7.5. The result for the London network is given in Table 7.1.

We first discuss the results without footpaths. On the German network the running times are about the same as those of the time-dependent algorithms. Their algorithm needs about 83% of the running time of our algorithm on one-to-all queries and 138% on one-to-one queries. On the Europe network their running time is always slightly lower. On one-to-all queries their algorithm needs 66% of our algorithm's running time and on one-to-one queries 79%.

A first observation is that the many connections are processed but are not relaxable. This, however, is not expensive as every discarded connection only results in a single if-statement that is not taken. The connections that are relaxed but do not improve the solution are a bit more expensive but also a cheap constant time operation. Interestingly only a very small amount of relaxations lead to an improvement of the solution. The start criterion is an optimization that always pays out but the speed gain of 15%-20% is moderate. The gain of the one-to-one stop criterion of up to 50% is significantly higher. If one is only interested in a one-to-one query then one should always use this optimization. Considering minimum change times increases executions time by about 50%.

Next we discuss the results with footpaths. If a connection arrives at a stop it can improve the solution at its arrival stop or at a stop reachable by foot or at both. We only measured how many connections improve the solution at their arrival stop. The effects of the

Network	MC	SO	SA	Time [ms]	#conn. processed		#conn. relax. no improve		#conn. relax. improve	
Germany	○	○	○	0.90	487 649	(100%)	136 034	(27%)	4 567	(0%)
Germany	○	○	●	0.75	285 612	(58%)	136 034	(27%)	4 567	(0%)
Germany	○	●	○	0.62	361 893	(74%)	29 014	(5%)	2 587	(0%)
Germany	○	●	●	0.44	159 858	(32%)	29 014	(5%)	2 587	(0%)
Germany	●	○	○	1.50	487 649	(100%)	132 689	(27%)	4 521	(0%)
Germany	●	○	●	1.06	285 612	(58%)	132 689	(27%)	4 521	(0%)
Germany	●	●	○	1.04	365 682	(74%)	29 076	(5%)	2 607	(0%)
Germany	●	●	●	0.62	163 646	(33%)	29 076	(5%)	2 607	(0%)
Ger. Per.	○	○	○	1.86	856 796	(100%)	478 005	(55%)	6 707	(0%)
Ger. Per.	○	○	●	1.46	654 759	(76%)	478 005	(55%)	6 707	(0%)
Ger. Per.	○	●	○	0.73	400 746	(46%)	45 568	(5%)	3 440	(0%)
Ger. Per.	○	●	●	0.56	198 710	(23%)	45 568	(5%)	3 440	(0%)
Ger. Per.	●	○	○	3.24	856 796	(100%)	473 919	(55%)	6 698	(0%)
Ger. Per.	●	○	●	2.63	654 758	(76%)	473 919	(55%)	6 698	(0%)
Ger. Per.	●	●	○	1.26	405 664	(47%)	46 055	(5%)	3 498	(0%)
Ger. Per.	●	●	●	0.82	203 629	(23%)	46 055	(5%)	3 498	(0%)
Eur. Per.	○	○	○	10.82	3 944 507	(100%)	1 964 366	(49%)	28 894	(0%)
Eur. Per.	○	○	●	10.02	3 303 523	(83%)	1 964 366	(49%)	28 894	(0%)
Eur. Per.	○	●	○	5.74	2 113 145	(53%)	317 215	(8%)	15 258	(0%)
Eur. Per.	○	●	●	4.82	1 472 162	(37%)	317 215	(8%)	15 258	(0%)
Eur. Per.	●	○	○	16.85	3 944 507	(100%)	1 944 964	(49%)	28 873	(0%)
Eur. Per.	●	○	●	14.75	3 303 522	(83%)	1 944 964	(49%)	28 873	(0%)
Eur. Per.	●	●	○	8.54	2 133 278	(54%)	316 883	(8%)	15 323	(0%)
Eur. Per.	●	●	●	6.76	1 492 295	(37%)	316 883	(8%)	15 323	(0%)

Table 7.5: Results of the Connection Scan algorithm for the earliest arrival problem on networks without footpaths. The flag “MC” indicates if minimum change times were used, “SO” indicates if the one-to-one stop criterion was used and “SA” whether the start criterion was used.

Network	TR	SO	SA	Time [ms]	#conn. processed		#conn. relax. no improve at arr.		#conn. relax. improve		#foot. relax. no improve		#foot. relax. improve	
London	○	○	○	43.66	4 941 823	(100%)	2 637 364	(53%)	10 850	(0%)	9 569 918	(209x)	9 662	(21%)
London	○	○	●	40.36	2 961 017	(59%)	2 637 364	(53%)	10 850	(0%)	9 569 918	(209x)	9 662	(21%)
London	○	●	○	6.36	2 379 576	(48%)	118 323	(2%)	5 364	(0%)	528 267	(11x)	5 496	(12%)
London	○	●	●	2.75	398 771	(8%)	118 323	(2%)	5 364	(0%)	528 267	(11x)	5 496	(12%)
London	●	○	○	10.31	4 941 823	(100%)	2 637 361	(53%)	10 850	(0%)	0	(0%)	9 662	(21%)
London	●	○	●	6.93	2 961 017	(59%)	2 637 361	(53%)	10 850	(0%)	0	(0%)	9 662	(21%)
London	●	●	○	4.85	2 379 581	(48%)	118 324	(2%)	5 364	(0%)	0	(0%)	5 496	(12%)
London	●	●	●	1.27	398 776	(8%)	118 324	(2%)	5 364	(0%)	0	(0%)	5 496	(12%)
Lon. Per.	○	○	○	61.44	6 421 670	(100%)	4 095 752	(63%)	11 337	(0%)	12 585 855	(275x)	9 944	(21%)
Lon. Per.	○	○	●	58.31	4 440 864	(69%)	4 095 752	(63%)	11 337	(0%)	12 585 855	(275x)	9 944	(21%)
Lon. Per.	○	●	○	6.70	2 421 028	(37%)	139 347	(2%)	5 463	(0%)	577 394	(12x)	5 580	(12%)
Lon. Per.	○	●	●	3.07	440 224	(6%)	139 347	(2%)	5 463	(0%)	577 394	(12x)	5 580	(12%)
Lon. Per.	●	○	○	13.38	6 421 670	(100%)	4 095 749	(63%)	11 337	(0%)	0	(0%)	9 944	(21%)
Lon. Per.	●	○	●	9.92	4 440 864	(69%)	4 095 749	(63%)	11 337	(0%)	0	(0%)	9 944	(21%)
Lon. Per.	●	●	○	4.89	2 421 033	(37%)	139 347	(2%)	5 463	(0%)	0	(0%)	5 579	(12%)
Lon. Per.	●	●	●	1.36	440 228	(6%)	139 347	(2%)	5 463	(0%)	0	(0%)	5 579	(12%)

Figure 7.1: Results of the Connection Scan algorithm for the earliest arrival problem on networks without minimum change times. The flag "TR" indicates if the footpath transitivity was exploited, "SO" indicates if the one-to-one stop criterion was used and "SA" whether the start criterion was used. The "x" in "209x" means "times" (i.e. 20900%).

Network	SO	cores	Time [ms]
German	○	1	46.25
German	●	1	29.34
German	●	8	6.21
Europe	○	1	100.80
Europe	●	1	64.15
Europe	●	8	17.12
London	○	1	307.00
London	●	1	176.36
London	●	8	33.32

Table 7.6: Timing results of the PSPCS algorithm. SO indicates if the stop criterion was used. The queries are periodic. The results are averaged over 1000 queries.

footpaths were measured separately. The first observation is the huge number of footpath relaxations if the transitive footpath optimization is not used. Every footpath is touched on average about 209 to 275 times without improving the solution. When using this optimization then no footpath is relaxed without it having an effect. The reason for this effect is that at least two connections are needed that arrive at the same clique of stops at nearly the same time and both improve the time at their direct arrival stop. This can happen (it just has not been trigged by the random queries) but is extremly unlikely to happen and therefore explains the numbers. The average journey on this network is relatively short and therefore the one-to-one stop criterion gives a significant speed boost (75%). Our one-to-one running times are below the Time-Dependent Dijkstra running times and the one-to-all times are above.

7.2 Profile Search

We evaluated the profile search algorithms proposed in Chapter 4 and compare them to the PSPCS algorithm [5]. Table 7.6 shows its running times on the "Color Model". We evaluate PSPCS using a single core and on all of the 8 cores. Our algorithms always just use one core. Note that this algorithm solves one-to-all problem variant, whereas our algorithm solves the all-to-one variant.

We ran 10$\dot{0}$00 random queries. We chose the start and the target stop uniformly at random. We consider the variants with and without minimum change times (MC). Further, we evaluate the variant with worst case guarantees proposed in Section 4.4 (CE). Finally, we also evaluate the algorithm given an earliest departure time and a latest arrival time as described in Section 4.3.1, i.e., the algorithm computes the result for a given time window (WI). We choose a departure time and an arrival time uniformly at random within the first day. If the arrival time is before the departure time then we swap them. We measure the running time and the number of iterations in the step function evaluation method (eval. scan). If the loop body is only executed once then this value is one. Note that this evaluation loop does not exist if the variant with worst case guarantees is used. We further measure the number of final jumps per stop (this value includes the jump at infinity) and the number of pruned jumps (i.e. the calls to the jump insert method that do not result in a new jump being stored). If footpaths are considered we also measure the number of jumps per stop before the postprocessing phase to account for the initial footpaths. We further measure the running time of this postprocessing phase. The results for networks without minimum change times are presented in Table 7.7. The result for the London network is given in Table 7.8.

We first discuss the results for the algorithms without footpaths. A first observation is that

Network	MC	WI	CE	Time [ms]	eval. scan	#jumps p. stop	#jumps pruned p. stop
Germany	○	○	○	2.86	1.02	10.2	62.3
Germany	○	●	○	2.24	1.01	1.6	9.2
Germany	○	○	●	9.59	—	10.2	62.3
Germany	○	●	●	3.25	—	1.6	9.2
Germany	●	○	○	4.65	1.10	10.1	62.4
Germany	●	●	○	3.22	1.07	1.5	8.8
Germany	●	○	●	11.20	—	10.1	62.4
Germany	●	●	●	4.47	—	1.5	8.8
Ger. Per.	○	○	○	6.08	1.02	19.0	107.6
Ger. Per.	○	●	○	3.44	1.01	1.6	9.2
Ger. Per.	○	○	●	17.37	—	19.0	107.6
Ger. Per.	○	●	●	5.91	—	1.6	9.2
Ger. Per.	●	○	○	8.57	1.11	18.9	107.6
Ger. Per.	●	●	○	5.51	1.07	1.5	8.8
Ger. Per.	●	○	●	20.21	—	18.9	107.6
Ger. Per.	●	●	●	8.51	—	1.5	8.8
Eur. Per.	○	○	○	33.66	1.01	13.0	117.3
Eur. Per.	○	●	○	16.74	1.00	1.0	1.6
Eur. Per.	○	○	●	115.70	—	13.0	117.3
Eur. Per.	○	●	●	18.47	—	1.0	1.6
Eur. Per.	●	○	○	46.34	1.10	12.9	117.3
Eur. Per.	●	●	○	26.03	1.02	1.0	1.5
Eur. Per.	●	○	●	139.67	—	12.9	117.3
Eur. Per.	●	●	●	29.51	—	1.0	1.5

Table 7.7: Results of the Connection Scan algorithm for the profile problem on networks without footpaths. the flag "MC" indicates whether minimum change times were used, "WI" whether a time window is given and "CE" whether the constant time evaluation is used.

Network	WI	Time [ms]	Time in postpro. [ms]	eval. scan	#jumps p. stop	#jumps p. stop before postpro.	#jumps pruned p. stop
London	○	235.49	1.08	1.38	84.8	87.8	150.3
London	●	135.90	4.15	1.39	24.1	24.4	49.3
Lon. Per.	○	279.77	1.01	1.34	103.9	105.9	203.2
Lon. Per.	●	157.40	5.34	1.39	24.1	24.4	49.3

Table 7.8: Results of the Connection Scan algorithm for the profile problem on networks without minimum change times. The "WI" flag indicates whether the query is restricted to a given random time window.

a one-to-all profile search is only about 3-4 times slower than the one-to-all earliest arrival algorithm on the same network. It is not surprising that the constant time evaluation is slower as it is a lot less cache-efficient. However, it is interesting to see that the evaluation loop nearly always exits after one iteration and therefore is about constant on our data. On this data the running times of our algorithm are about three times faster than SPCS (i.e. PSPCS with one core). The PSPCS with 8 cores is about 30% faster than our sequential algorithm.

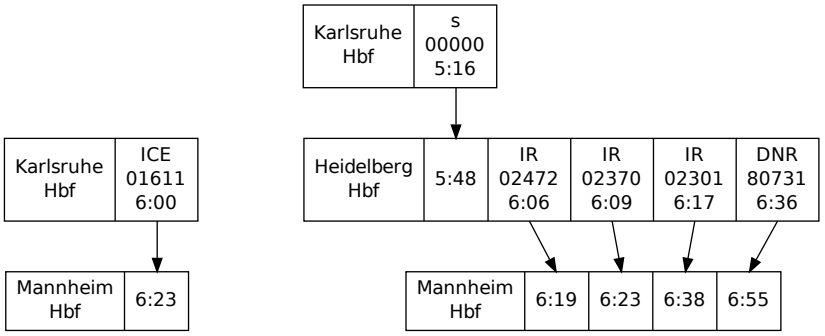
Next we discuss the results that consider footpaths. Note that we do not exploit the footpath transitivity and therefore the number of touched footpaths is significantly higher than in the earliest arrival setting. As a consequence the running times are higher. The running time is however still slightly below the one-to-all SPCS. We compute a windowed one-to-all query and SPCS a one-to-one query and therefore these optimizations can not be compared directly. The evaluation loops need more iterations on this network than on the previous networks. However, the number of iterations still is nearly constant. It is interesting that the postprocessing phase is cheaper than expected and it actually reduces the number of jumps instead of increasing it. If a time window is given then the postprocessing phase takes longer. This is due to the fact that this phase does not make use of these optimizations and therefore propagates arrival times into regions that are pruned by the main algorithm. These regions do not contain any jumps and therefore a lot less jumps are pruned there.

7.3 Minimum Expected Arrival Time Problem

In this section we evaluate the M.E.A.T. algorithm proposed in Chapter 6. We ran 1000 random all-to-one queries with a target stop picked uniformly at random. For each of these queries we measure the running time, the number of iterations in the evaluation loop, the final number of jumps and the number of pruned jumps. Further, we also evaluated 1000 random one-to-one queries with the optimizations described in Section 4.3.1 activated. Finally, for every all-to-one query we evaluate 100 random decision graphs with a random start stop and a random start time within the first day. We measured the number of nodes and arcs in diagrams such as 7.2. We evaluate all of the distributions listed in Section 6.3.

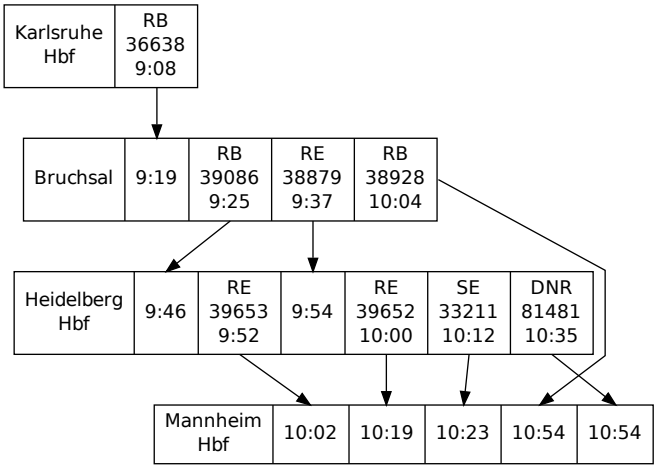
Note that disabling the minimum change time also disables trips and therefore forces the user to exit and reenter a train after each connection, if he wishes to continue his journey in the same train. If delays are not considered then this is no problem. However, if they are considered, then this leads to the possibility that the user misses the train that he is already sitting in (i.e., drops out of the train). This is not what we want and therefore we only consider the algorithm variant with minimum change times. The results are presented in Table 7.9. Figures 7.2, 7.3 and 7.4 present a few example decision graphs that all use the exponential distribution. (These were generated using graphviz [28].)

We first discuss the example decision graphs. In Figure 7.2 three decision graphs are presented that guide the user from Karlsruhe to Mannheim. The complexity of the decision graph largely depends on the departure time. Ideally the journey consists of only taking the ICE (i.e. a fast long distance train). However, it departs only at certain times a day. If a query is processed at another time then the decision graph is more complex. Note that we have not explicitly optimized the number of train changes but the produced results do not contain superfluous ones. (This is a problem with non-delayed train routing and is solved using multi-criteria routing.) Figure 7.3 illustrates a common problem. The user arrives with a train that has a high maximum delay and the possible connecting trains depart with a very high frequency and do not dominate each other. Unfortunately this scenario is very common when considering local public transport. This can cause the size of the decision graph to explode. On the German and the European networks this is no



(a)

(b)



(c)

Figure 7.2: Several decision graphs from Karlsruhe to Mannheim at various times of the day. (See Figure 1.2 for indications on how to read the diagram.)

Network	prob. dist.	all-to-one Time [ms]	one-to-one Time [ms]	eval. scan	#jumps p. stop	#jumps pruned p. stop	#stops in d.g.	#arcs in d.g.
Germany	const.	10.74	3.71	1.67	14.0	58.4	6.9	12.7
Germany	lin.	11.19	3.80	1.47	14.1	58.2	8.1	15.1
Germany	exp.	20.19	4.16	1.86	16.0	56.3	8.1	19.3
Germany	d. exp.	12.29	3.86	1.85	15.9	56.4	8.1	19.2
Ger. Per.	const.	18.86	6.17	1.79	30.9	95.4	7.7	13.2
Ger. Per.	lin.	20.63	6.26	1.52	28.6	97.7	9.0	16.0
Ger. Per.	exp.	39.17	6.66	1.98	34.9	91.3	9.2	20.3
Ger. Per.	d. exp.	23.66	6.37	1.97	34.7	91.5	9.2	20.1
Ger. Unroll	const.	33.26	10.08	1.92	56.3	158.7	7.7	13.3
Ger. Unroll	lin.	37.43	10.20	1.60	52.1	162.7	9.0	16.2
Ger. Unroll	exp.	76.88	10.65	2.17	64.5	150.1	9.2	20.7
Ger. Unroll	d. exp.	43.73	10.33	2.16	64.1	150.4	9.2	20.5
Eur. Per.	const.	80.78	27.70	1.45	22.8	107.3	14.8	24.3
Eur. Per.	lin.	82.96	27.63	1.31	20.5	109.6	18.3	33.1
Eur. Per.	exp.	131.02	28.05	1.56	24.9	105.2	20.0	44.4
Eur. Per.	d. exp.	92.95	28.12	1.55	24.8	105.3	19.7	43.3
Eur. Unroll	const.	170.14	57.05	1.54	45.8	220.5	14.5	24.5
Eur. Unroll	lin.	177.32	57.05	1.36	41.1	225.2	18.1	35.7
Eur. Unroll	exp.	308.55	57.93	1.70	51.4	214.7	19.5	46.2
Eur. Unroll	d. exp.	204.01	58.01	1.69	51.1	215.0	19.3	45.0
London	const.	615.39	235.19	3.34	123.5	108.8	1542.6	12056.4
London	lin.	770.44	275.89	2.60	122.1	109.6	4636.5	25899.5
London	exp.	2636.01	766.83	4.27	150.5	77.9	2724.6	30243.0
London	d. exp.	1050.37	349.71	4.23	149.4	79.1	3070.0	32366.2
Lon. Per.	const.	733.92	260.41	3.16	154.5	150.0	3292.8	14972.7
Lon. Per.	lin.	909.34	301.12	2.47	151.9	151.6	6175.9	30600.4
Lon. Per.	exp.	3006.43	795.22	4.02	187.4	112.7	5388.9	39428.8
Lon. Per.	d. exp.	1228.73	375.53	3.98	186.2	114.1	5596.3	41370.1
Lon. Unroll	const.	1264.89	338.15	3.39	251.1	211.0	3339.5	53066.7
Lon. Unroll	lin.	1604.94	382.72	2.62	245.7	214.9	8859.6	98953.1
Lon. Unroll	exp.	5426.19	883.55	4.34	306.4	147.4	4929.3	114428.0
Lon. Unroll	d. exp.	2175.95	457.72	4.30	304.2	149.8	5476.7	120848.0

Table 7.9: Results of the M.E.A.T. algorithm

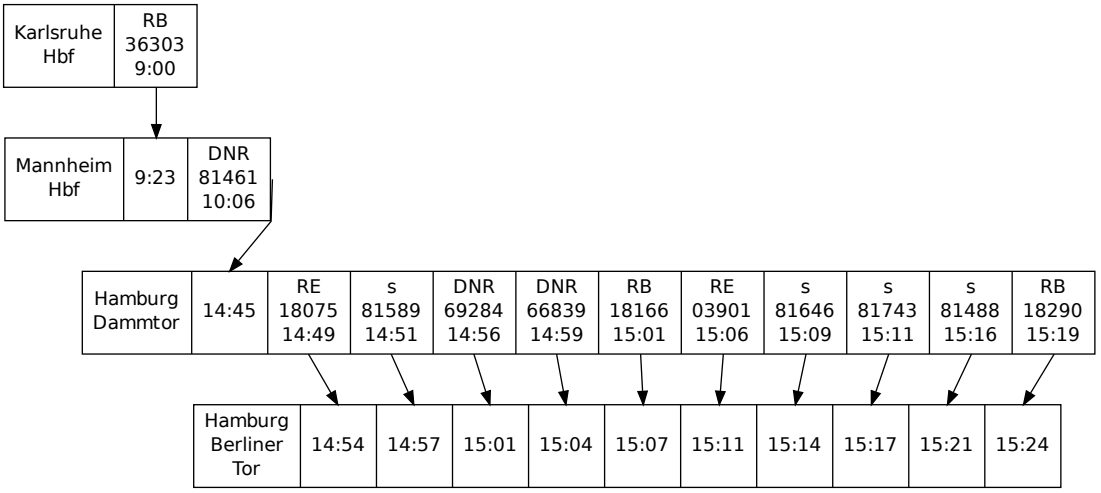


Figure 7.3: An example where a train arrives with a high maximum delay and the journey continues on a route with high frequency.

big problem as local public transport is largely ignored but on the London network this is a major problem. Figure 7.4 illustrates that the queries between two less important stops can become large.

We now discuss the running time results on the German and European data sets. A first observation is that the one-to-one running times are low enough to allow interactive applications. Further the decision graphs are small enough to be printed out and handed to the user. It is worth noting that the probability distribution functions matter and produce different results. The all-to-one running times are at most 7 times slower. The low number of iterations in the evaluation loop is surprising but is probably due to a high slack times within these networks. As expected the exponential distribution function is significantly slower than the other three options. It is no surprise that the discretized version is up to a factor 2 faster but it is surprising that it actually leads to measurably different results (although the differences are small on Germany and Europe). The exponential distribution leads to the largest decision graphs whereas the constant distribution produces the smallest decision graphs.

The running times on the London network are significantly higher and this is probably again due to the way footpaths are handled. (Note that every footpath relaxation requires a step function evaluation and therefore the number of evaluations is significantly higher.) Unfortunately, the decision graphs are too big to be handed to the user. In the London timetable there are a lot of routes that operate at high frequency (i.e. 10 min or even less). On average a train has a maximum delay of 12 min. This means that at every train change the decision graph contains on average at least two trains of the same route. As this effect occurs at every train change the number of possibilities quickly grows and produces large decision graphs. Many of the arcs in the decision graph belong to the same route. The decision graph sizes increase with the number of periods considered. This suggests that an effect similar to the one described in Section 5.3 (i.e. optimal but infinite decision graphs) actually occurs. Interestingly on this data set the exponential distribution produces smaller decision graphs than the linear distribution.

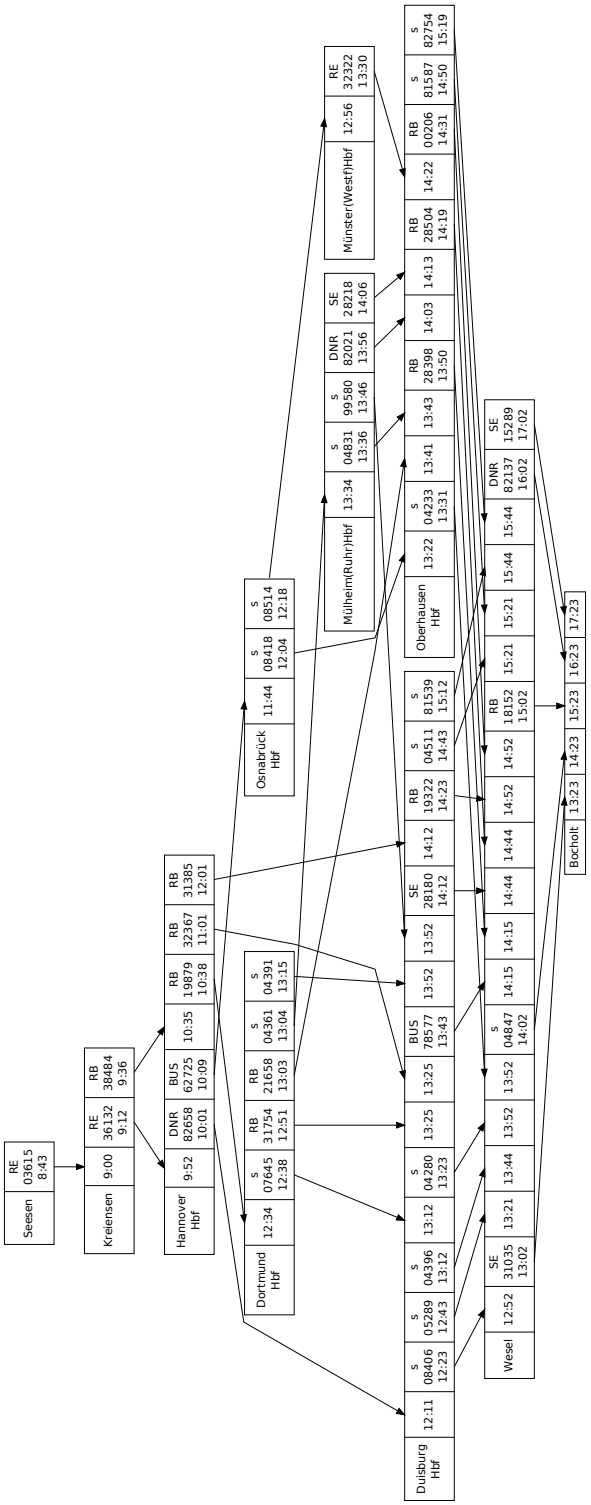


Figure 7.4: An example of a large decision graph.

8. Conclusion

In this work we introduced the Connection Scan algorithm, a new approach to the earliest arrival problem and to the profile problem on time table networks that does not directly operate on graphs nor uses a priority queue as all Dijkstra based approaches do. We showed that it outperforms or is at least on par with existing algorithms. A big advantage of our algorithm is that it is easy to implement. We also introduce several stochastic models for train delays. We introduced the minimum expected arrival time (M.E.A.T.) algorithm to solve one of these models efficiently. It is a new approach to generate alternative journeys by computing one decision graph instead of a set of several pareto-optimal but unrelated journeys as is done by the multi-criteria approach. We have also shown how to compute a minimum number of routes efficiently and how to reduce periodic queries to aperiodic ones through a network transformation.

Future Work

We have presented a number of novel ideas and showed that they are useful. However, as often with new ideas, a lot of open questions remain. It is worth investigating whether the basic Connection Scan algorithm can be further improved. Some ideas include: Parallelization, data compression to improve cache efficiency, rearranging the stops to improve cache efficiency, exploiting routes, speedups through preprocessing, alternative connection orders and better footpath handling. It s also interesting to investigate how the Connection Scan algorithm can interact with existing algorithms such as Dijkstra's algorithm to answer queries in multi-modal settings that do not only involve train-like vehicles. An in-depth analysis of the adaptation of Dijkstra's algorithm as presented in Algorithm 2.1 could also yield interesting results. Another topic consists of extending the existing algorithm to answer multi-criteria queries. There are also a lot of open problems concerning the minimum expected arrival time problem. We do not yet know how to generate reasonably sized decision graphs for routes with trains that depart every few minutes. We do not know how to efficiently solve the more complex models introduced in this work. Especially handling delayed departures seems complicated. Other approaches to stochastic routing exist such as stochastic on time arrival (S.O.T.A.) problem [9, 13]. Investigating whether a Connection Scan based approach can solve this problem and outperforms existing approaches is interesting. A different approach to routing with delays is to use realtime data (i.e. an online approach whereas our approach is offline). It may be worth augmenting existing online approaches with offline components.

Acknowledgment

I want to thank my advisors Prof. Dr. Dorothea Wagner, Julian Dibbelt and Thomas Pajor for giving me the opportunity to work on this topic, for having time for questions and for providing me advises in our discussions.

Bibliography

[1] H. Bast. Car or public transport-two worlds. *Efficient Algorithms*, pages 355–367, 2009.

[2] I. Abraham, D. Delling, A.V. Goldberg, R.F. Werneck, A. Goldberg, and R. Werneck. Hierarchical hub labelings for shortest paths. Technical report, Tech. Rep. MSR-TR-2012-46, Microsoft Research, 2012.

[3] R. Bauer, D. Delling, and D. Wagner. Experimental study of speed up techniques for timetable information systems. *Networks*, 57(1):38–52, 2011.

[4] A. Berger, D. Delling, A. Gebhardt, and M. 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 2009), Dagstuhl Seminar Proceedings*, 2009.

[5] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. *ACM Journal of Experimental Algorithmics*, 2012. to appear.

[6] Round-based public transit routing.

[7] Y. Disser, M. Müller-Hannemann, and M. Schnee. Multi-criteria shortest paths in time-dependent train networks. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, pages 347–361, 2008.

[8] E. Miller-Hooks. Adaptive least-expected time paths in stochastic, time-varying transportation and data networks. *Networks*, 37(1):35–52, 2001.

[9] Y. Nie and Y. Fan. Arriving-on-time problem: discrete algorithm that ensures convergence. *Transportation Research Record: Journal of the Transportation Research Board*, 1964(-1):193–200, 2006.

[10] R.P. Loui. Optimal paths in graphs with stochastic or multidimensional weights. *Communications of the ACM*, 26(9):670–676, 1983.

[11] L. Fu and L.R. Rilett. Expected shortest paths in dynamic and stochastic traffic networks. *Transportation Research Part B: Methodological*, 32(7):499–516, 1998.

[12] E.D. Miller-Hooks and H.S. Mahmassani. Least expected time paths in stochastic, time-varying transportation networks. *Transportation Science*, 34(2):198–215, 2000.

[13] S. Samaranayake, S. Blandin, and A. Bayen. A tractable class of algorithms for reliable routing in stochastic networks. *Transportation Research Part C: Emerging Technologies*, 2011.

[14] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12:2–4, 2008.

[15] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[16] 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.

[17] Gerth Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03)*, volume 92 of *Electronic Notes in Theoretical Computer Science*, pages 3–15, 2004.

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

[19] R. Geisberger. Contraction of timetable networks with realistic transfers. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, pages 71–82, 2010.

[20] A. Berger, M. Grimmer, and M. Müller-Hannemann. Fully dynamic speed-up techniques for multi-criteria shortest path searches in time-dependent networks. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, pages 35–46, 2010.

[21] R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[22] `http://code.google.com/codejam/contest/dashboard?c=204113#s=a&a=2`.

[23] T.H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[24] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1), 1956.

[25] LR Ford Jr and DR Fulkerson.

[26] HaCon - Ingenieurgesellschaft mbH. `http://www.hacon.de`, 1984.

[27] London Data Store. `http://data.london.gov.uk/`, 2011.

[28] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph-static and dynamic graph drawing tools. *Graph Drawing Software*, pages 127–148, 2004.