
eCO-friendly urban Multi-modal route PlAnning Services for mobile uSers

**FP7 - Information and Communication Technologies**

**Grant Agreement No: 288094**
**Collaborative Project**
**Project start: 1 November 2011, Duration: 36 months**

# D3.4.2 - New realistic approaches to multi-modal route planning using methods from stochasticity and machine learning and their empirical assessment

| | |
|---|---|
| **Workpackage:** | WP3 - Algorithms for Multimodal Human Mobility |
| **Due date of deliverable:** | 30 June 2013 |
| **Actual submission date:** | 14 July 2013 |
| **Responsible Partner:** | ETHZ |
| **Contributing Partners:** | ETHZ, KIT |

**Nature:** ☒ Report ☐ Prototype ☐ Demonstrator ☐ Other

**Dissemination Level:**
☒ PU: Public
☐ PP: Restricted to other programme participants (including the Commission Services)
☐ RE: Restricted to a group specified by the consortium (including the Commission Services)
☐ CO: Confidential, only for members of the consortium (including the Commission Services)

**Keyword List:** Multi-modal, Multi-criteria, Route planning, Algorithms, Optimization, Algorithm engineering, Public transportation, Timetable information systems, Shortest Path, Uncertainty, Robustness, Stochasticity, Machine Learning.


The eCOMPASS project (www.ecompass-project.eu) is funded by the European Commission, DG CONNECT (Communications Networks, Content and Technology Directorate General), Unit H5 - Smart Cities & Sustainability, under the FP7 Programme.

## The eCOMPASS Consortium


Computer Technology Institute & Press 'Diophantus' (CTI) (coordinator), Greece


Centre for Research and Technology Hellas (CERTH), Greece


Eidgenössische Technische Hochschule Zürich (ETHZ), Switzerland


Karlsruher Institut fuer Technologie (KIT), Germany


TOMTOM INTERNATIONAL BV (TOMTOM), Netherlands


PTV PLANUNG TRANSPORT VERKEHR AG. (PTV), Germany

| Document history | | | |
|---|---|---|---|
| Version | Date | Status | Modifications made by |
| 0.1 | 31.05.2013 | Table of Contents draft | Sandro Montanari, ETHZ |
| 0.2 | 14.06.2013 | First Draft | Sandro Montanari, ETHZ |
| 1.0 | 17.06.2013 | Sent to internal reviewers | Sandro Montanari, ETHZ<br>Julian Dibbelt, KIT |
| 1.1 | 20.06.2013 | Reviewers' comments received | Sandro Montanari, ETHZ<br>Julian Dibbelt, KIT |
| 1.1 | 24.06.2013 | Reviewers' comments incorporated (sent to PQB) | Sandro Montanari, ETHZ<br>Julian Dibbelt, KIT |
| 1.2 | 28.06.2013 | PQB's comments received | Sandro Montanari, ETHZ |
| 1.3 | 30.06.2013 | PQB's comments incorporated | Sandro Montanari, ETHZ<br>Christos Zaroliagis, CTI |
| 1.4 | 14.07.2013 | Final (approved by PQB and sent to the Project Officer) | Christos Zaroliagis, CTI |

**Deliverable manager**

- Sandro Montanari, ETHZ

**List of Contributors**

- Tobias Pröger, ETHZ
- Katerina Bohmova, ETHZ
- Rastislav Sramek, ETHZ
- Julian Dibbelt, KIT
- Ben Strasser, KIT

**List of Evaluators**

- Michael Marte, TomTom
- Damianos Gavalas, CTI

**Summary**
The purpose of this deliverable is to present the research results obtained with respect to Task 3.4 in the first 20 months of the project. The focus is on presenting algorithms and models developed for solving problems concerning robust multi-modal route planning in public transportation networks using methods from stochasticity and machine learning. We also present some experimental results aiming at assessing the quality of the proposed solutions. For each problem considered, we briefly present state-of-the-art techniques for dealing with it, and we illustrate the new solutions developed within the scope of the project.

# Contents

# 1 Introduction

This deliverable presents the research results obtained by the project partners in the first 20 months of the project with respect to multi-modal route planning in urban areas. It describes the models and algorithms developed so far for the problems related to WP3, while also showing experimental results aiming to assess the quality of the proposed solutions.

## 1.1 Objectives and scope of D3.4

The aim of WP3 is to provide novel methods for route planning in urban public transportation networks, considering the environmental impact as a main parameter of the optimization objective.

The present deliverable is the outcome of Task 3.4 "Algorithms for multi-modal route planning using methods from stochasticity and machine learning and their empirical assessment". This task aims at developing models and algorithms outperforming the state-of-the-art techniques in both precision and reliability when the underlying information is subject to unpredictable changes because of delays or modifications to the planned schedules.

## 1.2 Motivation

Public transportation networks, such as buses, trains, trams, or subways are a key ingredient to significantly reducing carbon dioxide emissions in passenger transportation. An effectively filled bus produces considerably less environmental pollution than a large fleet of cars carrying the same amount of people, but only a single person each. However, to assure the acceptance of such mass transport systems it is important to reach a similar level of convenience as cars have.

A car departs at the moment that the user wishes but a bus has a fixed schedule. A car brings the user from his starting position directly to his destination without the risk of missing an intermediate connection train as is the case with train journeys. Unfortunately delays are frequent in large public transportation networks which makes it even more difficult for the public transportation to compete with individual car transportation. In the present work we address different aspects of this problem and propose solutions to increase reliability and flexibility of traveling when using public transportation.

We believe that it is of large importance for every mass transit information system to provide a similar level of flexibility and reliability as cars do. As a result, this increases the motivation of the potencial users to switch to public transportation, and thus, decreases the level of produced emissions. In this document we make a step in this direction.

## 1.3 Structure of the Document

In Section 2, we focus on identifying journeys that are robust for typical delays and on supplying the user with information about the reliability of a selected journey. The reliability of a journey is an important criterion for a user who plans his journey ahead of time and wants to pick one that will ensure that he arrives at his destination on time. Such a user is willing to accept a journey that takes slightly longer than the quickest one, but has better chances of being feasible and on time even in the presence of delays.

In Section 3, we focus on flexibility of the presented solutions to promote public transportation as a competitive alternative to car traveling. If the user decides to change his destination midway, because for example he might have realized that he still needs to buy groceries, then a car will bring him directly to his new destination. Buses on the other hand follow a strict route fixed for months in advance. We aim to provide flexible passenger routing information that allows the user to miss a couple of trains and still get to his destination. If the user misses a bus then the system should tell him in advance what backup bus he may take. Using this knowledge the user can also interrupt his journey at a stop with a grocery store and resume his journey once he has finished shopping.

# 2 Mining-Based Robust Routing in Public Transportation Networks

In this section we focus on robust routing in public transportation networks. In order to propose robust solutions we assume that we have past observations of real traffic situation available. In particular, we assume that we have "snapshots" (i.e., traces) containing the observed travel times in the whole network for a few past days. We introduce a new model to express a solution, a so called *journey*, that is feasible in any snapshot of a given public transportation network. We adapt the method of Buhmann et al. [7] for optimization under uncertainty, and develop algorithms that allow its application for finding a robust journey from a given source to a given destination. Finally, we introduce a measure for reliability of a given journey, and develop algorithms for its computation. The worked presented in this section is published as part of the eCOMPASS technical report series [4].

## 2.1 Related Work

In this section we survey the most relevant literature on robust routing in public transportation networks. We first review the modeling question, and then we discuss robustness concepts.

A classical method is to model the public transportation network as a graph, which then allows to solve the problem using standard algorithms for routing in graphs. The most widely considered graph-based models are the *time-expanded* [6, 19, 21, 22] and *time-dependent* [18, 23, 26] models.

In the time-expanded model, each public transportation station is represented by many vertices, each of them corresponding to a time event associated with the station. Usually, each departure/arrival of a train (or a tram, bus, etc.) passing through the station is one such event. Moreover, for each station, other vertices are often added to simulate additional events and situations such as a possible transfer from a train to another. Each edge in the graph models the possibility and the time requirements of getting from a particular event to another one. For example, the event of departing of a train $t$ from a station $s$ is connected by an edge to the event of the train $t$ arriving to the next station $s'$. Time-expanded models are quite versatile and a lot of aspects of public transportation can be captured into them by refining and adding specific vertices and edges to the graph in a rather straightforward way. An advantage of time-expanded models is that the corresponding graph is usually directed and acyclic. The drawback is that the more properties the model captures, the more vertices and edges must have the corresponding graph, and thus the size of this graph grows rapidly. The size of the graph can be reduced to some extent, for example by remodeling less important stations [9].

In the time-dependent model, each station is modeled as one vertex. The time-dependent nature of the timetable is captured by the edges between the stations. In particular, for each edge there is a table (a function) containing several entries, each indicating when a particular train (or bus, tram, etc.) enters and leaves the edge. To make this model more realistic, it is possible to also simulate transfers by splitting the vertex that corresponds to a station into several vertices and adding transfer edges between them. For more detailed overview on the time-expanded and time-dependent model we refer to [16, 24].

Recently, several models are considered that adopt different approaches than to try to capture all the properties of the problem into a graph. The basic idea behind these approaches is that the problem contains a certain structure (e.g. public transportation lines) that can be taken into account to simplify the routing, but this structure is usually obfuscated when modeled into a graph.

As an example of a non graph-based approach we mention round-based public transit routing introduced in [10]. The authors describe an algorithm to find all Pareto-optimal journeys between two given stops $a$ and $b$ that minimize arrival time and the number of transfers. Their approach is centered around transportation lines (such as train, or bus lines), and does not explicitly construct a graph with vertices representing the stations and edges connecting these vertices. The routing

algorithm operates in rounds: after round $i$, all non-dominated paths from stop $a$ to each stop reachable with at most $i$ transfers are found. In each round the algorithm considers each line at most once and uses it to extend the current paths in a non-dominated fashion. As shown later see, we adopt a similar approach.

In practice, the efficiency of developed algorithms is an important measure of their applicability. Precomputing may decrease query times substantially. Bast et al. [2] observe that for given two stations $a$ and $b$, we can find and encode each sequence of intermediate transfer stations (i.e., stations where we change from one line to another) that can lead to an optimal route. The set of these sequences of transfers is called *transfer patterns*. The transfer patterns can be precomputed and stored (compressed into a prefix tree). Using transfer patterns it is possible to achieve very fast query times. For a particular query it is enough to extract the sequence of intermediate stations and then to find the corresponding direct lines that connect the stations and both of these steps can be done quickly.

For further references and details concerning models and algorithms for routing in fixed public transportation networks, we refer to Deliverable D3.1.

Various approaches for robust multi-modal route planning have been developed. Perhaps the earliest work was done on stochastic networks [12, 5, 20], where either the lengths of edges (in the case of car transportation), or the delays between successive edges (in the case of bus transportation) are random variables. In a situation when timetables are fixed, Disser *et al.* [11] used a generalization of Dijkstra's algorithm to compute pareto-optimal multi-criteria journeys. With this algorithm, the reliability of a journey can simply be considered as an additional criterion. The reliability is expressed as a function depending on the minimal time to change two subsequent trains on the journey. Müller-Hannemann and Schnee [17] and Schnee [25] introduced the concept of a *dependency graph* for a prediction of secondary delays caused by some current primary delays, which are given as input. Another approach to handle robustness in public transportation networks was presented by Goerik *et. al.* [14]. They consider a given set of delay *scenarios* and introduce two concepts of robustness: while the goal of *strict robustness* is the computation of a journey that arrives on time for every scenario, the goal of *light robustness* is the computation of a journey whose travel time lies only a certain factor over the optimum travel time. For a complete overview of multi-modal route planning under uncertainty, see Deliverable D3.1.

## 2.2 Mining-Based Model

**Stops and lines.** Let $\mathcal{S}$ be a set of stops, and $\mathcal{L} \subset \bigcup_{i=2}^{|\mathcal{S}|} \mathcal{S}^i$ be a set of lines, or services, (e.g., bus lines, tram lines or lines of other means of transportation). Every line $l \in \mathcal{L}$ is a sequence of $S(l)$ stops $\langle s_1^{(l)}, \ldots, s_{S(l)}^{(l)} \rangle$, where, for every $i \in \{1, \ldots, S(l)-1\}$, the stop $s_i^{(l)}$ is served before $s_{i+1}^{(l)}$ by the line $l$. Notice that we explicitly distinguish two lines that serve the same stops but have oppositional directions (these may be operated under the same identifier in reality). For a stop $s \in \mathcal{S}$ and a line $l \in \mathcal{L}$, we write $s \triangleleft l$ if there exists an index $i \in \{1, \ldots, S(l)\}$ such that $s = s_i^{(l)}$, i.e. $s$ is a stop on the line $l$. Furthermore, for two stops $s_1, s_2 \in \mathcal{S}$ and a line $l \in \mathcal{L}$ we write $s_1 \triangleleft s_2 \triangleleft l$ if there exist indices $i, j \in \mathbb{N}$, $1 \leq i \leq S(l)-1$, $i+1 \leq j \leq S(l)$ such that $s_1 = s_i^{(l)}$ and $s_2 = s_j^{(l)}$, i.e. if both $s_1$ and $s_2$ are stops on $l$ and $s_1$ is served before $s_2$. For two lines $l_1, l_2 \in \mathcal{L}$, we define $l_1 \cap l_2$ to be the set of all stops $s \in \mathcal{S}$ that are served both by $l_1$ and $l_2$.

**Trips and timetables.** While the only information associated with a line itself are its consecutive stops, it usually is operated multiple times per day. Each of these concrete realizations at a given time of the day is called a *trip*. With every trip $\tau$ we associate a line $L(\tau) = \langle s_1^{(l)}, \ldots, s_{S(l)}^{(l)} \rangle \in \mathcal{L}$. On the other hand, $L^{-1}(l)$ denotes the set of all trips associated with a line $l \in \mathcal{L}$. For a trip $\tau$ and

a stop $s \in \mathcal{S}$, we define

$$A(\tau, s) := \begin{cases} \text{Arrival time of } \tau \text{ at stop } s_i & \text{if } s = s_i^{(l)},\ i = 2, \ldots, S(l) \\ -\infty & \text{otherwise} \end{cases} \tag{1}$$

and

$$D(\tau, s) := \begin{cases} \text{Departure time of } \tau \text{ from stop } s_i & \text{if } s = s_i^{(l)},\ i = 1, \ldots, S(l) - 1 \\ +\infty & \text{otherwise} \end{cases} \tag{2}$$

In the following, we assume time to be modelled by integers. We have two more natural requirements on the arrival and the departure times for a given trip $\tau$. First we require $A(\tau, s) \leq D(\tau, s)$ for every stop $s \in \mathcal{S}$. Note that for all stops $s$ not served by $L(\tau)$, this requirement is trivially satisfied by definition. Furthermore we require $D(\tau, s_1) \leq A(\tau, s_2)$ for every two stops $s_1, s_2 \in \mathcal{S}$ with $s_1 \triangleleft s_2 \triangleleft L(\tau)$. A set of trips is called a *timetable*. We note that we are considering various timetables.

1) On one hand, we have the scheduled timetables for every day of the week. For the sake of simplicity we assume that these seven timetables are the same timetable $T$. The assumption of having scheduled timetables especially implies, that every line realized by some trip $\tau$ will be realized by a later trip $\tau'$ again (in the worst case, not on the same day).

2) On the other hand, we also consider daily "snapshots" $T_i$ that describe how various lines were operated on a concrete day $i$, i.e. timetables that contain the actual travel times of the lines.

**Goal.** In the following, let $a, b \in \mathcal{S}$ be two stops, $m \in \mathbb{N}_0$ be the maximal allowed number of line changes, $\varepsilon \in \mathbb{N}_0$ be the minimal time required to switch lines, and $t_A \in \mathbb{N}$ be the latest arrival time. A *journey* consists of a departure time $t_D$, a sequence of lines $\langle l_1, \ldots, l_k \rangle$, $k \leq m + 1$ and a sequence of intermediate stops $\langle s_{\text{CH}}^{(1)}, \ldots, s_{\text{CH}}^{(k-1)} \rangle$. The intuitive interpretation of such a journey is to start at stop $a$ at time $t_D$, take the first line $l_1$, and for every $i \in \{1, \ldots, k-1\}$, leave $l_i$ at stop $s_{\text{CH}}^{(i)}$ and take the next arriving line $l_{i+1}$ immediately. Our goal is to compute a recommendation to the user in form of one or more (robust) journeys from $a$ to $b$ that will likely arrive on time (i.e., before time $t_A$) on a day, for which the concrete travel times are not known yet. We formalize the notion of robustness later in this report.

**Routes.** Let $k \in \{1, \ldots, m+1\}$ be an integer. A sequence of lines $r = \langle l_1, \ldots, l_k \rangle \in \mathcal{L}^k$ is called a *feasible route from $a$ to $b$* if there exist $k+1$ stops $s_0 := a, s_1, \ldots, s_{k-1}, s_k := b$ such that $s_{i-1} \triangleleft s_i \triangleleft l_i$ for every $i \in \{1, \ldots, k\}$, i.e. if both $s_{i-1}$ and $s_i$ are stops on line $l_i$, and $s_{i-1}$ is served before $s_i$ on line $l_i$. Notice that on a feasible route $r \in \mathcal{L}^k$ we need to change the line at $k-1$ intermediate stops. Let

$$\mathcal{R}_{ab}^m = \{r \in \mathcal{L} \cup \mathcal{L}^2 \cup \cdots \cup \mathcal{L}^{m+1} \mid r \text{ is a feasible route from } a \text{ to } b\} \tag{3}$$

be the set of all feasible routes from $a$ to $b$ using at most $m$ intermediate stops. In the next section we describe an algorithm that computes the set $\mathcal{R}_{ab}^m$ for two stops $a, b \in \mathcal{S}$ and a number $m \in \mathbb{N}_0$. If $a$, $b$ and $m$ are clear from the context, for simplicity we just write $\mathcal{R}$ instead of $\mathcal{R}_{ab}^m$. Notice that by definition, a line $l$ may occur multiple times in a route. This is reasonable because there might be two intermediate stops $s, s'$ on $l$ and one or more intermediate lines that travel faster from $s$ to $s'$ than $l$ does. Additionally, notice that a route does not contain *any* time information.

## 2.3 Computation of Feasible Routes

In this section we describe an algorithm that, given a set of stops $\mathcal{S}$ and a set of lines $\mathcal{L}$, finds all feasible routes that allow to travel from a given source stop $a$ to a given target stop $b$ using at

most $m$ intermediate stops (also called transfers). Recall that a route may contain a line multiple times.

**Input data.** The input to the algorithm consists of a set of stops $\mathcal{S}$ and a set of lines $\mathcal{L}$, where each line is a particular sequence of stops. Note that to compute the set of feasible routes $\mathcal{R}$ we only need the network structure. The information about arrival/departure times (i.e., a particular timetable) is not necessary.

**Preprocessing of the input data.** We preprocess the input data and construct data structures to allow efficient queries of several types that are explained below.

**Supported queries.**

- $Q(l_i, s_j) \to$ Position of $s_j$ on $l_i$.

  Given a line $l_i$, and a stop $s_j$, query $Q(l_i, s_j)$ tells whether the stop $s_j$ lies on the line $l_i$ and if this is the case, also the position of $s_j$ on $l_i$. In particular, if $s_j$ does not lie on $l_i$, the query $Q(l_i, s_j)$ returns 0. Otherwise, to indicate that $s_j$ is $k$-th stop on $l_i$, $Q(l_i, s_j)$ returns a positive integer $k$.

- $Q(l_i, s_j, s'_j) \to s_j \triangleleft s'_j \triangleleft l_i$? (i.e., "Is $s_j$ before $s'_j$ on $l_i$?")

  Given a line $l_i$ and two stops $s_j, s'_j$, query $Q(l_i, s_j, s'_j)$ returns whether the stop $s_j$ is before $s'_j$ on the line $l_i$. In case $s_j$ or $s'_j$ is not on the line $l_i$, the query $Q(l_i, s_j, s'_j)$ returns `FALSE`.

- $Q(l_i, l_j) \to l_i \cap l_j$ (i.e., stops shared by $l_i$ and $l_j$) in a compact, ordered format.

  Given two lines $l_i$, and $l_j$, query $Q(l_i, l_j)$ returns $l_i \cap l_j$, i.e., the set of stops that are shared by these lines. We encode the stops shared by $l_i$ and $l_j$ into an ordered set $I_{ij}$ of pairs of stops with respect to the line $l_i$ in such a way that $(s_q, s_r) \in I_{ij}$ indicates that $l_i$ and $l_j$ share the stops $s_q$, $s_r$, and all the stops in between on the line $l_i$. Thus, the query $Q(l_i, l_j)$ outputs the described sorted set $I_{ij}$ of pairs of stops that compress the information on $l_i \cap l_j$. The motivation to compress $l_i \cap l_j$ into $I_{ij}$ is that, in practice, there may be many stops shared by $l_i$ and $l_j$, but only a small number of contiguous intervals of such stops.

  Note that $Q(l_i, l_j)$ doesn't need to be equal to $Q(l_j, l_i)$, nor its reverse; a tricky example is given in Figure 1.

**Graph of line incidencies.**

The function $Q(l_i, l_j)$ induces the following directed graph $G$. The set $V$ of vertices of $G$ corresponds to the set of lines $\mathcal{L}$. There is an edge from a vertex (line) $l_i$ to a vertex $l_j$ if and only if $Q(l_i, l_j) \neq \emptyset$. Then, $Q(l_i, l_j)$ represents a tag on the edge of the edge $(l_i, l_j)$. We construct and represent the graph $G$ as adjacency lists, i.e., for each vertex $l_i$, we store the outgoing edges along with their tags.

**Notes on implementation of the query structures.**

- The query $Q(l_i, s_j)$ can be implement using a suitable associative array/hashing table.
- The query $Q(l_i, s_j, s'_j)$ can be easily implemented using two queries of the type $Q(l_i, s_j)$ to determine the position of $s_j$ and $s'_j$ on $l_i$. In other words, a query $Q(l_i, s_j, s'_j)$ outputs `TRUE` if and only if both $Q(l_i, s_j)$ and $Q(l_i, s'_j)$ are nonzero and $Q(l_i, s_j) < Q(l_i, s'_j)$.
- If the need is to query $Q(l_i, l_j)$ for an arbitrary pair of lines, hashing table or two dimensional array may be a good choice of a data structure. However, for our purposes, it may be sufficient to use the graph of line incidencies $G$ to store this data. There, every edge $(l_i, l_j) \in G$ stores $Q(l_i, l_j)$ and for any pair of lines $l_i$, $l_j$ that does not form an edge in $G$, the query $Q(l_i, l_j)$ returns the empty set.

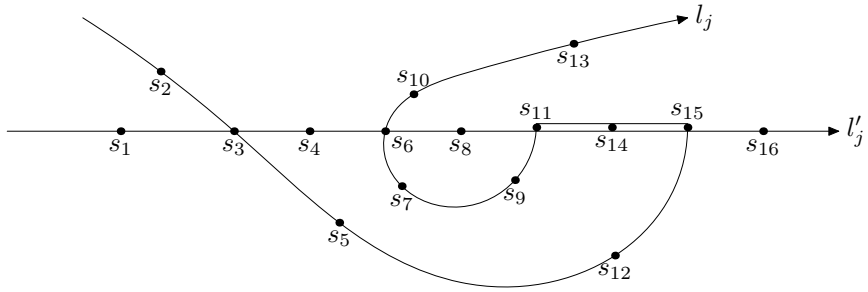



Figure 1: Lines $l_j$ and $l'_j$ have common stops $s_3$, $s_6$, $s_{11}$, $s_{14}$, and $s_{15}$. The ordered set $I_{jj'} = Q(l_j, l'_j)$ consists of pairs $\{(s_3, s_3), (s_{15}, s_{11}), (s_6, s_6)\}$. Thus, the last stop in the last interval of $I_{jj'}$ is the stop $s_6$. On the other hand, the ordered set $I_{j'j} = Q(l'_j, l_j)$ consists of pairs $\{(s_3, s_3), (s_6, s_6), (s_{11}, s_{15})\}$. Now, imagine that the current transfer stop $s_q$ for a partial path $P = l_1, \ldots, l_j$ is $s_{14}$, then the stop $s_{14}$ is the current transfer stop $s'_q$ for a partial path $P' = l_1, \ldots, l_j, l'_j$, as well. However, observe that if $s_q$ is $s_{12}$, then $s'_q$ needs to be $s_{11}$.

**Algorithm for computing all feasible routes.**

Given two stops $a$ and $b$, and a number $m$, we want to find all routes $\mathcal{R}$ that allow to travel from $a$ to $b$ using at most $m$ transfers in the given public transportation network described by a set of stops $\mathcal{S}$ and a set of lines $\mathcal{L}$.

Note that each such route $r = \langle l_1, \ldots, l_k \rangle \in \mathcal{L}^k$ with $0 < k \leq m + 1$ has the following properties.

- Both $Q(l_1, a)$ and $Q(l_k, b)$ are nonzero (i.e., $a \triangleleft l_1$, and $b \triangleleft l_k$).
- The vertices $l_1, \ldots, l_k$ form a path in $G$ (i.e., $l_i \cap l_{i+1} \neq \emptyset$ for all $i = 1, \ldots, k-1$).
- There exists a sequence of stops $a = s_0, s_1, \ldots, s_{k-1}, s_k = b$ such that $Q(l_i, s_{i-1}, s_i)$ is `TRUE` (i.e., $s_{i-1} \triangleleft s_i \triangleleft l_i$) for all $i = 1, \ldots, k$. Note that such a sequence can be found iteratively as follows.
  - set $s_0 = a$,
  - for all $i = 1, \ldots, k-1$: set $s_i$ to a stop that minimizes (but being nonzero) the value of $Q(l_{i+1}, s_i)$, and still satisfies $Q(l_i, s_{i-1}, s_i)$.

These observations lead to the following algorithm to find the set of routes $\mathcal{R}$.

- For the stop $a$, determine the set $\mathcal{L}_a$ of all lines passing through $a$.
- Explore the graph $G$ from the set $\mathcal{L}_a$ of vertices in the following fashion. For each vertex $l_1 \in \mathcal{L}_a$, perform a kind of depth-first search in $G$ up to the depth $m$. Any particular vertex may be processed several times, when reached from different paths. In each step, try to extend a partial path $l_1, \ldots, l_j$ to a neighbor $l'_j$ of $l_j$ in $G$. We keep track of the *current transfer stop* $s_q$. This is a stop on the currently considered line $l_j$ such that $s_q$ is the stop with the smallest order on $l_j$ at which it is possible to transfer from $l_{j-1}$ to $l_j$, considering the partial path from $l_1$ to $l_{j-1}$. Each step of the algorithm is characterized by a *search state*: a partial path $P = l_1, \ldots, l_j$, and a current transfer stop $s_q$ that allowed the transfer to line $l_j$. The initial search state consists of the partial path $P = l_1$ and the current transfer stop $a$.

  More specifically, to process a search state with the partial path $P = l_1, \ldots, l_j$, and the current transfer stop $s_q$, we perform the following tasks:

– Check whether the line corresponding to the vertex $l_j$ contains the stop $b$. If this is the case (i.e., $Q(l_j, b) > 0$), check that also $s_q \triangleleft b \triangleleft l_j$ holds (i.e., the query $Q(l_j, s_q, b)$ returns `TRUE`), then the partial path $P$ corresponds to a feasible route and is output as one of the solutions in $\mathcal{R}$.

– If the partial path $P$ contains at most $m-1$ edges (thus the corresponding route has at most $m-1$ transfers, and it can be further extended) then for each $l'_j$ neighbor of $l_j$ do:

Check whether extending $P$ by $l'_j$ is possible (and if so, update the current transfer stop) as follows.
Let $I_{jj'} = Q(l_j, l'_j)$ be the set of pairs of stops sorted as described in the previous section. Recall that each pair $(s_u, s_v) \in I_{jj'}$ encodes an interval of one or several consecutive stops on $l_j$ that are also stops on the line $l'_j$. Let $s_z$ be the last stop in the last interval of $I_{jj'}$. Similarly, let $I_{j'j} = Q(l'_j, l_j)$.
If $Q(l_j, s_q, s_z)$ is `TRUE`, then $s_q \triangleleft s_z \triangleleft l_j$, and the path $P$ can be extended to $l'_j$.

◇ We determine the current transfer stop $s'_q$ for $l'_j$ by considering the pairs/intervals of $I_{j'j}$ in ascending order and deciding whether the position of $s_q$ on the line $l_j$ is before one of the endpoints of the currently considered interval.
We refer to Figure 1 for a nontrivial case of computing of the current transfer stop.

◇ Perform the depth search with the search state consisting of the partial path $P' = l_1, \ldots, l_j, l'_j$ and the current transfer stop $s'_q$.

Otherwise, if $Q(l_j, s_q, s_z)$ is `FALSE`, it is not possible to extend $P$ to $l'_j$.

• output the set of feasible routes $\mathcal{R}$.

We can express the just described algorithm in a pseudocode as follows. We call this algorithm *All-Routes algorithm*. The algorithm uses a subroutine *All-Routes-Recursion*.

---

ALL-ROUTES$(a, b, \mathcal{S}, \mathcal{L}, G)$

---

1 $\mathcal{R} \leftarrow \emptyset$
2 $\mathcal{L}_a \leftarrow \{l \in \mathcal{L} \mid Q(l, a) > 0\}$
3 **for each** $l \in \mathcal{L}_a$ **do**
4 $\quad P \leftarrow \langle l \rangle$; $s_q \leftarrow a$
5 $\quad \mathcal{R} \leftarrow \mathcal{R} \cup$ ALL-ROUTES-RECURSION$(a, b, \mathcal{S}, \mathcal{L}, G, P, l, s_q)$
6 **return** $\mathcal{R}$

---

ALL-ROUTES-RECURSION$(a, b, \mathcal{S}, \mathcal{L}, G, P, l, s_q)$

```
1  if Q(l, b) > 0 and Q(l, s_q, b) then
2          R ← {P}
3  else
4          R ← ∅
5  if length(P) > m then
6          return R
7  for each l' such that (l, l') ∈ G do
8          I ← Q(l, l')
9          s_z ← last element of the last interval of I
10         if Q(l, s_q, s_z) then
11                 P' ← append(P, l'); I' ← Q(l', l)
12                 for each (s_u, s_v) ∈ I' do
13                         if Q(l, s_q, s_u) or Q(l, s_q, s_v) then
14                                 if Q(l, s_q, s_u) and Q(l, s_q, s_v) then
15                                         s'_q ← s_u
16                                 else
17                                         s'_q ← s_q
18                                 R ← R ∪ ALL-ROUTES-RECURSION(a, b, S, L, G, P', l', s'_q)
19                                 break
20 return R
```

## 2.4 Computing the earliest arrival of a journey

**Recursive computation.** As previously stated, let $a \in \mathcal{S}$ be the initial stop, $b \in \mathcal{S}$ be the destination stop, $\varepsilon$ be the minimum time to change lines and $t_A \in \mathbb{N}$ be the latest arrival time. In the previous section we showed how the set $\mathcal{R}$ of feasible routes from $a$ to $b$ can be computed. However, instead of presenting just a route $r \in \mathcal{R}$ to the user, we want to compute a departure time $t_0$ and a *journey* that arrives at $b$ before time $t_A$. For the following considerations, we assume the underlying timetable (either the scheduled timetable or a daily snapshot) to be fixed. We describe an algorithm that computes the earliest arrival of a journey when the stops $a, b \in \mathcal{S}$, an initial departure time $t_0 \in \mathbb{N}$, and a route $r = \langle l_1, \ldots, l_k \rangle \in \mathcal{R}_{ab}^{k-1}$ are given. The idea is simple: we start the journey at the stop $a$ at time $t_0$ and take the first line $l_1$ that arrives. We compute an appropriate intermediate stop $s \in l_1 \cap l_2$ (that is served both by $l_1$ as well as by $l_2$) and the arrival time $t_1$ at $s$, leave $l_1$ there and compute recursively the earliest arrival time when departing from $s$ at time at least $t_1 + \varepsilon$, following the route $\langle l_2, \ldots, l_k \rangle$. Notice that the selection of an appropriate intermediate stop $s$ is the only non-trivial part due to mainly two reasons:

1) The lines $l_1$ and $l_2$ may operate with different speeds (e.g., because $l_1$ is a fast tram while $l_2$ is a slow bus line), or $l_1$ and $l_2$ separate at a stop $s$ and join later again at a stop $s'$ but the overall travel times of $l_1$ and $l_2$ differ between $s$ and $s'$. Depending on the situation, it may be better to leave $l_1$ as soon or as late as possible.

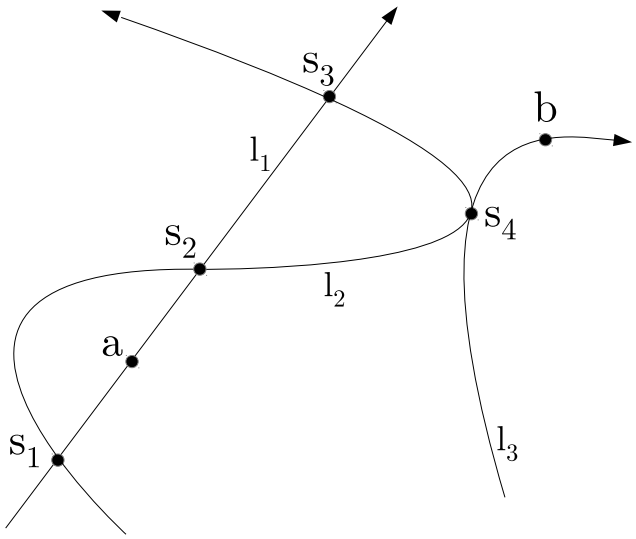



Figure 2: A network where not every stop in $l_1 \cap l_2 = \{s_1, s_2, s_3\}$ is suitable for changing from $l_1$ to $l_2$. We cannot choose $s_1$ as intermediate stop since it is served before $a$. If $s_3$ was chosen, then $l_3$ can never be reached without travelling back. Thus, the only valid stop to change the line is $s_2$.

2) The lines $l_1$ and $l_2$ may separate at a stop $s$ and join later again at a stop $s'$. If all intermediate stops in $l_2 \cap l_3$ are served by $l_2$ before $s'$, then leaving $l_1$ at $s'$ is not an option since $l_3$ is not reachable anymore. See Figure 2 for a visualization.

It is not hard to find an optimum intermediate stop recursively by the following algorithm. The idea is to find the earliest trip of line $l_1$ that departs from $a$ at time $t_0$ or later, iterate over all stops $s \in l_1 \cap l_2$, and compute recursively the earliest arrival time when continuing the journey from $s$ with a changing time of at least $\varepsilon$. We return the smallest arrival time that was found in one of the recursive calls. These considerations lead to the following algorithm. Notice that if some inappropriate intermediate stop was chosen at some point, the corresponding recursive call simply returns $\infty$, thus, it is ignored due to the minimum computation in line 7.

EARLIEST-ARRIVAL$(a, b, t_0, \langle l_1, \ldots, l_k \rangle)$

1 $\tau_1 \leftarrow \arg\min_{\tau \in L^{-1}(l_1)} \{D(\tau, a) \mid D(\tau, a) \geq t_0\}$
2 **if** $k = 1$ **and** $a \triangleleft b \triangleleft l_1$ **then return** $A(\tau, b)$
3 **else if** $k = 1$ **then return** $\infty$
4 **else** A-min $\leftarrow \infty$
5 **for each** $s \in l_1 \cap l_2$ **do**
6 **if** $a \triangleleft s \triangleleft l_i$ **then**
7 A-min $\leftarrow \min\{\text{A-min}, \text{EARLIEST-ARRIVAL}(s, A(\tau_1, s) + \varepsilon, \langle l_2, \ldots, l_k \rangle)\}$
8 **return** A-min

**Issues and improvement of the recursive algorithm.** An issue with this naïve implementation is the running time, which might be exponential in $k$ in the worst-case (if $|l_i \cap l_{i+1}| > 1$ for $\Omega(k)$ many $i \in \{1, \ldots, k-1\}$). To improve the running time, we make two simple observations:

1) Let $\tau$ and $\tau'$ be two trips with $L(\tau) = L(\tau')$. If $\tau$ leaves before $\tau'$ at some stop $s$, then it will never arrive later than $\tau'$ at any subsequent stop $s'$, $s \triangleleft s' \triangleleft L(\tau)$, i.e. consecutive trips of the same line do not overtake (i.e., the FIFO property applies).

2) Let $l_1, l_2 \in \mathcal{L}$ be two lines and $T_{l_2} \subseteq L^{-1}(l_2)$ be the set of all trips that can be reached from one of the stops in $l_1 \cap l_2$ with a changing time of at least $\varepsilon$. It follows from the previous observation that taking the earliest trip in $T_{l_2}$ never results in a later arrival at $b$ than taking any other trip from $T_{l_2}$. Due to the first observation, a trip $\tau \in T_{l_2}$ is operated earlier than a trip $\tau' \in T_{l_2}$ iff $A(\tau, s) < A(\tau', s)$ for *any* stop $s \in l_1 \cap l_2$.

Thus, we can iterate over some appropriate stops in $l_1 \cap l_2$ to find the earliest reachable trip associated with $l_2$. We just need to ignore those stops where changing to $l_3$ is no longer possible (see Figure 2 for an example).

**Computing appropriate intermediate stops.** The problem to find these appropriate stops can be solved by first sorting $l_1 \cap l_2 = \{s_1, \ldots, s_n\}$ such that $s_j \triangleleft s_{j+1} \triangleleft l_1$ for every $j \in \{1, \ldots, n-1\}$. If the first $f \geq 0$ stops $s_1, \ldots, s_f$ are served before $a$, they cannot be used for changing to $l_2$. This problem can easily be solved by considering only those stops $s_j$ where $a \triangleleft s_j \triangleleft l_1$. Unfortunately, the last $g \geq 0$ stops $s_{n-g+1}, \ldots, s_n$ might also not be suitable for changing to $l_2$ because they may prevent us later to change to some line $l_j$ (e.g., if *all* stops of $l_2 \cap l_3$ are served before $s_{n-g+1}, \ldots, s_n$ on $l_2$, then changing to $l_3$ is no longer possible). We solve this problem by precomputing (the index of) the last stop $s_j$ where all later lines are still reachable. This can be done backwards: we start at $b$, order the elements of $l_k \cap l_{k-1}$ as they appear on line $l_k$, and find the last stop that is served before $b$ on $l_k$. We recursively continue with $l_1, \ldots, l_{k-1}$ and use the stop previously computed as the stop that still needs to be reachable.

**Iterative algorithm.** The improved algorithm works as follows. First, for every $i \in \{1, \ldots, k-1\}$, we use the algorithm described in the previous paragraph to precompute the index last[$i$] of the last stop where changing from $l_i$ to $l_{i+1}$ is still possible (with respect to the route $\langle l_1, \ldots, l_k \rangle$). After that, for every $i \in \{1, \ldots, k-1\}$, we iterate over the appropriate intermediate stops $s \in l_i \cap l_{i+1}$ where changing to $l_{i+1}$ is possible, and find among those the stop $s^{(i)}_{\text{CH}}$ where the earliest trip $\tau_{i+1}$ associated with line $l_{i+1}$ departs.

This computes a sequence of trips $\tau_1, \ldots, \tau_k$ along with intermediate stops $s^{(0)}_{\text{CH}} := a, s^{(1)}_{\text{CH}}, \ldots, s^{(k)}_{\text{CH}}$ to change lines. Since we gradually compute the earliest trips $\tau_i$ for each of the lines $l_i$, the earliest time to arrive at $b$ is simply $A(\tau_k, b)$.

EARLIEST-ARRIVAL$(a, b, t_0, \langle l_1, \ldots, l_k \rangle)$

```
1  last[k] ← b
2  for i ← k, ..., 2 do
3      Order the elements of l_i ∩ l_{i-1} = {s_1, ..., s_n} such that s_j ◁ s_{j+1} ◁ l_i ∀j ∈ {1, ..., n-1}.
4      last[i-1] ← max{j ∈ {1, ..., n} | s_j ◁ last[i] ◁ l_i}
5  τ_1 ← arg min_{τ ∈ L^{-1}(l_1)} {D(τ, a) | D(τ, a) ≥ t_0};   s^{(0)}_CH ← a
6  for i ← 1, ..., k-1 do
7      Order the elements of l_i ∩ l_{i+1} = {s_1, ..., s_n} such that s_j ◁ s_{j+1} ◁ l_i ∀j ∈ {1, ..., n-1}.
8      τ_{i+1} ← null;   s^{(i)}_CH ← null;   A^{(i+1)}_{s_n} ← ∞
9      for j ← 1, ..., last[i] do
10         if s^{(i-1)}_CH ◁ s_j ◁ l_i then
```

11 $\tau' \leftarrow \arg\min_{\tau \in L^{-1}(l_{i+1})}\{D(\tau, s_j) \mid D(\tau, s_j) \geq A(\tau_i, s_j) + \varepsilon\}$

12 **if** $A(\tau', s_n) < A_{s_n}^{(i+1)}$ **then** $\tau_{i+1} \leftarrow \tau'$; $s_{\text{CH}}^{(i)} \leftarrow s_j$; $A_{s_n}^{(i+1)} \leftarrow A(\tau', s_n)$

13 **return** $A(\tau_k, b)$

## 2.5 Maximizing the Unexpected Similarity

**Computing the optimum journey for a fixed timetable.** Given two stops $a, b \in \mathcal{S}$ and a departure time $t_0 \in \mathbb{N}$, we can already compute the earliest arrival of a journey from $a$ to $b$ starting at time $t_0$. This section is concerned with a slightly different problem: we want to compute the latest departure time at $a$ when a latest *arrival* time $t_A$ at $b$ is given. For this purpose we present a sweepline algorithm that uses the previous algorithm Earliest-Arrival. This sweepline algorithm will later be extended to count journeys (instead of computing a single one) and can be used for finding robust journeys, i.e. journeys that are likely to arrive on time.

The sweepline algorithm works as follows. We consider the trips departing at stop $a$ before time $t_A$, sorted in reserve chronological order. Everytime we find a trip $\tau$ of any line departing at some time $t_0$, we check whether there exists a route $r = \langle L(\tau), l_2, \ldots, l_k\rangle \in \mathcal{R}$ that starts with the line $L(\tau)$. If yes, then we use the previous algorithm to compute the earliest arrival time at $b$ when we depart at $a$ at time $t_0$ and follow the route $r$. If the time computed is not later than $t_A$, we found the optimal solution and stop the algorithm. Otherwise we continue with the previous trip departing from $a$.

**Computing the Unexpected Similarity.** We will now describe how we can compute robust journeys using the approach of Buhmann *et. al.* [7] (see Deliverable D2.2 for a detailed information). Let $a, b \in \mathcal{S}$ be the departure and the target stop of the journey, $t_A$ be the latest arrival time at $b$, and $\mathcal{T}$ be a set of traffic snapshots, i.e. timetables for some concrete times. When applying the aforementioned approach to the model delevoped in this section, we can use $\mathcal{R}$ as the set of feasible solutions. For a snapshot $T \in \mathcal{T}$ and a value $\gamma$, the *approximation set* $A_\gamma(T)$ is a multiset of routes. For every journey along a route $r \in \mathcal{R}$, we add $r$ to $A_\gamma(T)$ if the starting time of the journey is $t_A - \gamma$ or later and the arrival time is at most $t_A$. We represent the approximation set by a function $\mu_\gamma^T : \mathcal{R} \to \mathbb{N}_0$, where for a route $r \in \mathcal{R}$, $\mu_\gamma^T(r)$ is the number of journeys starting at time $t_A - \gamma$ or later, arriving at time $t_A$ or earlier and following the route $r$ (see Figure 3 for an example). Thus, we have $|A_\gamma(T)| = \sum_{r\in\mathcal{R}} \mu_\gamma^T(r)$, and for two snapshots $T_1, T_2$, we need to compute $\gamma$ that maximizes the *similarity*

$$S_\gamma = \frac{|A_\gamma(T_1) \cap A_\gamma(T_2)|}{|A_\gamma(T_1)||A_\gamma(T_2)|} = \frac{\sum_{r\in\mathcal{R}} \min(\mu_\gamma^{T_1}(r), \mu_\gamma^{T_2}(r))}{\left(\sum_{r\in\mathcal{R}} \mu_\gamma^{T_1}(r)\right) \cdot \left(\sum_{r\in\mathcal{R}} \mu_\gamma^{T_2}(r)\right)}. \tag{4}$$

Let $\gamma_{\text{OPT}}$ be the value of $\gamma$ maximizing the ratio (4). After computing this value, we pick a route $r$ from $A_{\gamma_{\text{OPT}}}(T_1) \cap A_{\gamma_{\text{OPT}}}(T_2)$ at random according to the probability distribution defined by

$$p_r := \frac{\min(\mu_{\gamma_{\text{OPT}}}^{T_1}(r), \mu_{\gamma_{\text{OPT}}}^{T_2}(r))}{\sum_{r\in\mathcal{R}} \min(\mu_{\gamma_{\text{OPT}}}^{T_1}(r), \mu_{\gamma_{\text{OPT}}}^{T_2}(r))}, \tag{5}$$

and search in the scheduled timetable $T$ for a journey from $a$ to $b$ that departs at time $t_A - \gamma_{\text{OPT}}$ or earlier, and that arrives at time $t_A$ or earlier.

For $i \in \{1, 2\}$, we represent the function $\mu_\gamma^{T_i}$ by an $|\mathcal{R}|$-dimensional vector $\mu_i$ such that $\mu_i[r] = \mu_\gamma^{T_i}(r)$ for every $r \in \mathcal{R}$. We can compute the value $\gamma_{\text{OPT}}$ by a simple extension of the sweepline algorithm presented in the previous paragraph. The modified algorithm again starts at time $t_A$, and considers all trips in $T_1$ and $T_2$ in reserve chronological order. The sweepline stops at every

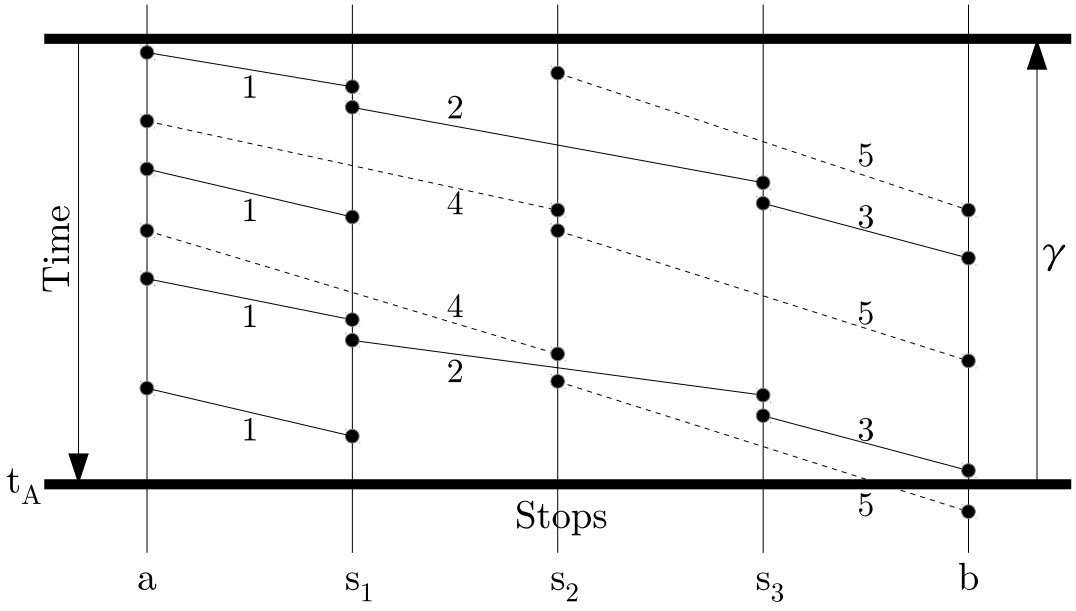



Figure 3: An example with five lines $\{1, \ldots, 5\}$ and two routes $r_1 = \langle 1, 2, 3\rangle$ (solid) and $r_2 = \langle 4, 5\rangle$ (dotted). The $x$-axis illustrates the stops $\{a, s_1, s_2, s_3, b\}$, whereas the $y$-axis the time. If a trip leaves a stop $s_d$ at time $t_d$ and arrives at a stop $s_a$ at time $t_a$, it is indicated by a line from $(s_d, t_d)$ to $(s_a, t_a)$. We have $\mu^T_\gamma(r_1) = 3$, because we have three possible journeys from $a$ to $b$ when departing at time $t_A - \gamma$ or later. Notice that we always have to take the first occurence of a line that arrives. Thus, taking the first 1 and waiting for the second 2 is not counted because it is not a valid journey in our setting although it would still arrive on time. Furthermore we have $\mu^T_\gamma(r_2) = 1$, because taking the second 4 will not result in an arrival on time.

time when one or more trips in $T_1$ or in $T_2$ depart. Assume that the sweepline stops at time $t_A - \gamma$, and assume that it stopped at time $t_A - \gamma' > t_A - \gamma$ in the previous step. Of course, we have $\mu^{T_i}_\gamma(r) \geq \mu^{T_i}_{\gamma'}(r)$ for every $r \in \mathcal{R}$ and $i \in \{1, 2\}$. Let $\tau_1, \ldots, \tau_k$ be the trips that depart in $T_1$ or $T_2$ at time $t_A - \gamma$. The idea is to compute the values of $\mu_i$ (representing $\mu^{T_i}_\gamma$) from the values computed in the previous step (representing $\mu^{T_i}_{\gamma'}$). This can be done as follows: for every trip $\tau_j$ occuring in $T_i$ and departing at time $t_A - \gamma$, we check whether there exists a route $r \in \mathcal{R}$ starting with $L(\tau_j)$. If yes, we distinguish two cases:

1) If $\mu_i[r] = 0$, then $\mu^{T_i}_{\gamma'}(r) = 0$, thus $r \notin A_{\gamma'}(T_i)$. If there exists a journey from $a$ to $b$ along $r$ departing at time $t_A - \gamma$ or later, and arriving at time $t_A$ or earlier, then $A_\gamma(T_i)$ contains $r$ exactly once. Thus, if EARLIEST-ARRIVAL$(a, b, t_A - \gamma, r) \leq t_A$, we set $\mu_i[r] \leftarrow 1$.

2) If $\mu_i[r] > 0$, then $\mu^{T_i}_{\gamma'}(r) > 0$, thus $A_{\gamma'}(T_i)$ contains $r$ at least once. Thus, there exists a journey from $a$ to $b$ along $r$ departing at time $t_A - \gamma'$ or later, and arriving at time $t_A$ or earlier. Since $\tau_i$ is the only possibility to depart at $a$ between time $t_A - \gamma$ and $t_A - \gamma'$, $\tau_i$ is the first trip on a journey we never found before. Therefore it is sufficient to simply increase $\mu_i[r]$ by 1.

Up to now, we did not define when the algorithm terminates. In fact we stop if $\gamma$ exceeds a value $\gamma_{\text{MAX}}$. Let $t_A - \gamma_i$ be the starting time of an optimal journey in $T_i$. Of course, $\gamma_{\text{MAX}}$ has to be larger than $\max\{\gamma_1, \gamma_2\}$. We believe that from a practical point of view it is sufficient to set $\gamma_{\text{MAX}} = f(\max\{\gamma_1, \gamma_2\})$ for some function $f : \mathbb{N} \to \mathbb{N}$; good choices for $f$ will be investigated in a future experimental evalutation.

The above considerations lead to the following algorithm for the computation of $\gamma_{\text{OPT}}$ and the corresponding functions $\mu^{T_i}_{\gamma_{\text{OPT}}}$. In the following pseudocode, the parameter list of EARLIEST-ARRIVAL now contains the additional parameter $T_i$ to determine which instance is considered.

COMPUTE-MAXIMUM-SIMILARITY$(a, b, t_A)$

```
1  S^OPT ← 0;  μ_1 ← μ_2 ← μ_1^OPT ← μ_2^OPT ← 0^|R|;  γ^OPT ← 0
2  for each stopping time of the sweepline do
3      γ ← Time difference of the sweepline to t_A
4      τ_1, ..., τ_n ← Trips departing at a at time t_A − γ in T_1 or T_2
5      for i ← 1, ..., n do
6          for each r = ⟨L(τ_i), l_2, ..., l_k⟩ ∈ R do
7              if τ_i occurs in T_1 then
8                  if μ_1[r] > 0 then μ_1[r] ← μ_1[r] + 1
9                  else if EARLIEST-ARRIVAL(T_1, a, b, t_A − γ, r) ≤ t_A then μ_1[r] ← 1
10             else
11                 if μ_2[r] > 0 then μ_2[r] ← μ_2[r] + 1
12                 else if EARLIEST-ARRIVAL(T_2, a, b, t_A − γ, r) ≤ t_A then μ_2[r] ← 1
13     S_γ ← (Σ_{r∈R} min(μ_1[r], μ_2[r])) / ((Σ_{r∈R} μ_1[r]) · (Σ_{r∈R} μ_2[r]))
14     if S_γ ≥ S^OPT then γ^OPT ← γ;  S^OPT ← S_γ;  μ_1^OPT ← μ_1;  μ_2^OPT ← μ_2
```

## 2.6 Journey Reliability

**Goal.** In the previous sections, we have considered the problem of *finding* a robust journey in public transporation network. In this section we will look at a related, but conceptually much simpler problem: Estimating the reliability of a given journey with respect to a given latest arrival time $t_A$. We will take the most straightforward approach and express the reliability as the "probability" of the given journey to finish before $t_A$, and present a scan-line algorithm that computes the "probability".

**Input.** We are given a public transportation network with its stops $\mathcal{S}$ and lines $\mathcal{L}$. Furthermore, we are given a journey $J$ from stop $a$ to stop $b$, and a (small) set of past daily snapshots $T_1, \ldots, T_m$ of the timetable, upon which we want to assess the reliability of the journey. We note that such data are commonly collected by transportation companies, and thus they are, in practice, principially available at virtually no cost. Recall also that a journey is specified by a departure time $t_D$ (i.e., the time when we start the journey), by a sequence of lines $\langle l_1, \ldots, l_k \rangle$, and by a sequence of transfer/changing stops $\langle s^{(1)}_{CH}, \ldots, s^{(k-1)}_{CH} \rangle$.

**Definitions of reliability.** In the most straightforward way, we express the reliability of a journey as the fraction of days (timetables) in which the journey arrived on time (i.e., before the latest arrival time $t_A$). We call this the *coupled reliability*.

**Definition 1 (Coupled reliability)** *The* coupled reliability of a journey $J$ with respect to a given latest arrival time $t_A$ and timetables $T_1, \ldots, T_m$ *is the ratio of the number of timetables $T_i$, in which*

*the journey $J$ (departing at stop $a$ no sooner than $t_D$) arrives at stop $b$ no later than $t_A$, to the total number of timetables, i.e.,*

$$\frac{|\{j \mid \textit{journey } J \textit{ arrives before } t_A \textit{ in } T_j\}|}{m}.$$

If the number of available timetables is small, and when "delays" of lines in the snapshot-timetables are independent (uncoupled), then we can heuristically evaluate the travel-time of each of the lines $< l_1, \ldots, l_k >$ of journey $J$ in a different snapshot $T_i$, and check whether the journey $J$ would arrive before $t_A$ in such a "virtual snapshot". This motivates the following definitions.

**Definition 2** *Let $T'_1, \ldots, T'_k$ be timetables, and $J$ a journey with departure time $t_D$, sequence of lines $\langle l_1, \ldots, l_k \rangle$, and a sequence of transfer stops $\langle s^{(1)}_{CH}, \ldots, s^{(k-1)}_{CH} \rangle$. Journey $j$ is* realizable in $T'_1 \times T'_2 \times \ldots \times T'_k$ *with respect to a given latest arrival time $t_A$ if for every line $l_i$ there exists a trip $t_i$ (of the line $l_i$) in $T'_i$ such that:*

1. *The departure time of trip $t_1$ from station $a$ is after $t_D$,*
2. *the arrival time of trip $t_k$ at station $b$ is before $t_A$, and*
3. *for every $i = 1, \ldots, k-1$, the arrival time of trip $t_i$ in station $s^{(i)}_{CH}$ is before the departure time of trip $t_{i+1}$.*

**Definition 3 (Decoupled reliability)** *The* decoupled reliability of a journey $J$ with respect to the latest departure time $t_A$ and timetables $T_1, \ldots, T_m$ *is the ratio of the number of all possible $k$-tuples of timetables $(T'_1, \ldots, T'_k)$, $T'_i \in \{T_1, \ldots, T_k\}$, such that $J$ is realizable in $T'_1 \times \ldots \times T'_k$ with respect to $t_A$, to the total number of $k$-tuples.*

Clearly, if the delays occuring on one line are dependent on delays in other parts of the transportation network, we conservatively prefer to use one timetable $T_i$ for the assessing the reliability of a journey, i.e., the coupled reliability. This gives a realistic feedback on the quality of the journey with respect to the an actually travelled past snapshot $T_i$. However, if the number of available snapshots $T_1, \ldots, T_m$ is small, we obtain only several such feedbacks. On the other hand, if delays on one line in the transportation network are independent from delays in the other part of the network, we can increase the feedback we get by simulating "artificial" timetables: for a decoupled reliability, we evaluate the travel times of a line between two transfer stops in different timetables.

**Computing reliability.** We will address the computational aspects of the coupled and decoupled reliability. Computing the coupled reliability is very easy: For every timetable $T_i \in \{T_1, \ldots, T_m\}$ we need to check whether the journey in question finished before time $t_A$ or not. This can be done by a simple linear time algorithm that simply "simulates" the journey in the timetable $T_i$, and checks whether the arrival time of the journey lies before or after $t_A$. The computation of decoupled reliability is not so trivial anymore, as the straightforward approach would require to enumerate all the $m^k$ $k$-tuples of the set $\{T_1, \ldots, T_m\}$, and thus an exponential time. In the next part, we will present an algorithm that avoids such an exponential enumeration.

### 2.6.1 Computing decoupled reliability

In this section we present an algorithm for computing the decoupled reliability of a journey $J$ with respect to a given latest arrival time $t_A$ and timetables $T_1, \ldots, T_k$. The algorithm is based on dynamic programming.

Consider the situation in Figure 4. Different colors represent three different timetables (snapshots). Since we have "decoupled" the delays on different lines, there are $m^k = 3^2 = 9$ different $k$-tuples of timetables $\langle T'_1, T'_2 \rangle$. Each of the two lines can thus operate with any of the three timetables. We

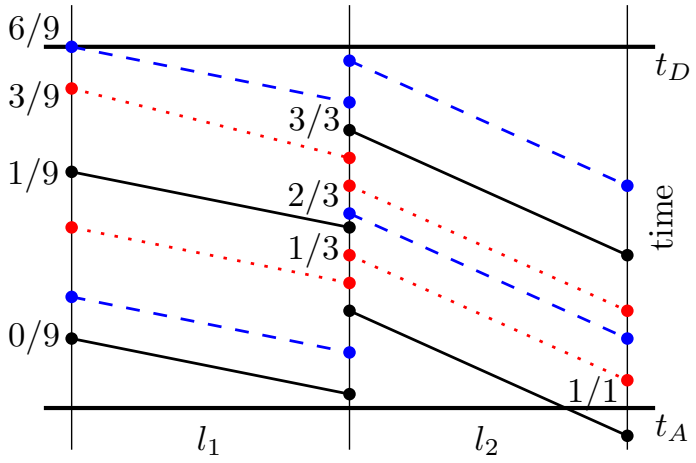



Figure 4: A journey with two lines $l_1$ and $l_2$ and three timetables (solid black, dotted red, dashed blue). Partial probabilities of finishing on time are shown.

now proceed from the last to first stop and for each relevant time (arrival or departure time of any connection) we compute how large fraction of all possible timetable combinations will make the journey to "make it" if making the next connection at the considered time. For the last stop, every time before before $t_A$ will be assigned the fraction 1/1, because once we are at the last stop before $t_A$, every combination of timetables will make us to get there. We process each stop by decreasing time, starting at $t_A$, going through all time points when one of the lines in any one of the timetables departs or arrives. If the time-point $t$ being processed corresponds to an arriving line, we simply copy the previously computed value (i.e., the later in time). If it corresponds to a departing line, we follow this trip and check the arriving time $t'$ at the next stop. We check at this stop and time $t'$ the fraction of "paths" that reach the final destination and multiply this fraction by $1/m$. Furthermore, for each timetable we store the maximum encountered fraction up to that point. If for the considered timetable $T_j$ the just computed fraction is larger than the stored one, we update this stored value. The total fraction for the processed time-point $t$ is then equal to the sum of the stored fractions of all timetables. The Figure 4 shows these compound fractions. The resulting fraction is 6/9, which can be easily checked by enumeration of the 9 possible options. The formal description is presented as Algorithm *CalculateSuccessRate*.

CalculateSuccessRate$(J, t_D, t_A, T_1, \ldots, T_m)$

1 SuccessRate$[b, t_A] \leftarrow 1$
2 **for** $i \leftarrow 1, \ldots, k-2$ **do** SuccessRate$[s_{\text{CH}}^{(i)}, t_A] \leftarrow 0$
3 **for** $i \leftarrow k-1, \ldots, 1$ **do**
4 **for** $j \leftarrow 1, \ldots, m$ **do** max$[j] \leftarrow 0$
5 **for** $j \leftarrow t_A, \ldots, t_D$ **do**
6 **if** connection at time $t_j$ is a departing one **then**
7 connection arrived at $s_{\text{CH}}^{(i+1)}$ at time $t'$ and is from timetable $T_k$
8 max$[k] \leftarrow \frac{1}{m}$SuccessRate$[s_{\text{CH}}^{(i+1)}, t']$
9 SuccessRate$[s_{\text{CH}}^{(i)}, t_j] \leftarrow \sum_{l=1}^{m}$ max$[l]$

10 **else** SuccessRate$[s_{\mathrm{CH}}^{(i)}, t_j] \leftarrow$ SuccessRate$[s_{\mathrm{CH}}^{(i)}, t_{j-1}]$

11 **return** SuccessRate$[s_{\mathrm{CH}}^{(1)}, t_D]$

## 2.7 Conclusions

We developed a novel model for public transportation networks. The main advantage is that it allows to distinguish between routes and concrete journeys, while most common models just allow to specify journeys by a sequence of lines and transfer stops. We also developed efficient algorithms for the computation of robust journeys. Furthermore, inspired by our considerations, we have introduced the natural concept of evaluating the robustness of a given journey (as defined in our model).

From a theoretical point of view, one of the next steps is to examine how the current methods have to be extended to support a fully multi-modal scenario, i.e. how walking, biking or park-and-ride can be integrated. We believe that the modelling itself is easy while the performance of the algorithms will descrease significantly without developing special techniques. For example, if we allow unrestricted walking, then all sequences of lines form a valid route since it is possible to walk from every stop to every other stop. However, in this situation it is not even clear how to compute the optimal overall travel time of a journey. It may be also worthwhile to study or develop different approaches to robustness than the one we used. Additionally, if the input data grows, preprocessing techniques may become necessary to allow an efficient computation of all feasible routes.

From a practical point of view, we will implement the presented algorithms. We are especially interested to evaluate their running time and the quality of the computed journeys. For an experimental evalutation, we need traffic snapshots which we do not have yet. If no traffic snapshots will be available, we will develop techniques for generating artificial delays.

# 3 Delay-Robust Stochastic Routing in Public Transportation Networks

In this section we first formalize a large number of different sources of delays in a mathematical framework. We then prove that it can be solved using a dynamic program. Unfortunately for many delay models this program has a superpolynomial running time and is therefore infeasible. We therefore restict ourselfs to one of the simpler models and show that it can solved efficently by doing experiments on real world data of realistic size and measuring the running times. This simpler model enables computation of decision graphs such as presented in Figure 5.

## 3.1 Problem Statement and Preliminaries

Our public transit networks are defined in terms of their aperiodic *timetable*, consisting of a set of *stops*, a set of *connections*, and a set of *footpaths*. A *stop* $p$ corresponds to a location in the network where a passenger can enter or exit a vehicle (such as a bus stop or train station). Stops may have associated minimum change times, denoted minchange, which represent the minimum time required to change vehicles at $p$. A *connection* $c$ models a vehicle departing from a stop $c_{\text{depstop}}$ at time $c_{\text{deptime}}$ and arriving to stop $c_{\text{arrstop}}$ at time $c_{\text{arrtime}}$ without intermediate halt. Connections that are subsequently operated by the same vehicle are grouped into *trips*. We denote by $c_{\text{next}}$ the next connection (after $c$) of the same trip, if available. Trips can be further grouped into *routes*. A route is a set of trips serving the exact same sequence of stops. For correctness, we require trips of the same route to not overtake each other. *Footpaths* enable walking transfers between nearby stops. Each footpath consists of two stops with an associated walking duration. Note that our footpaths are transitively closed. A *journey* is a sequence of connections and footpaths. If two subsequent connections are not part of the same trip, their arrival-departure time-difference must be at least the minimum change time of the stop. Because our footpaths are transitively closed, a journey never contains two subsequent footpaths.

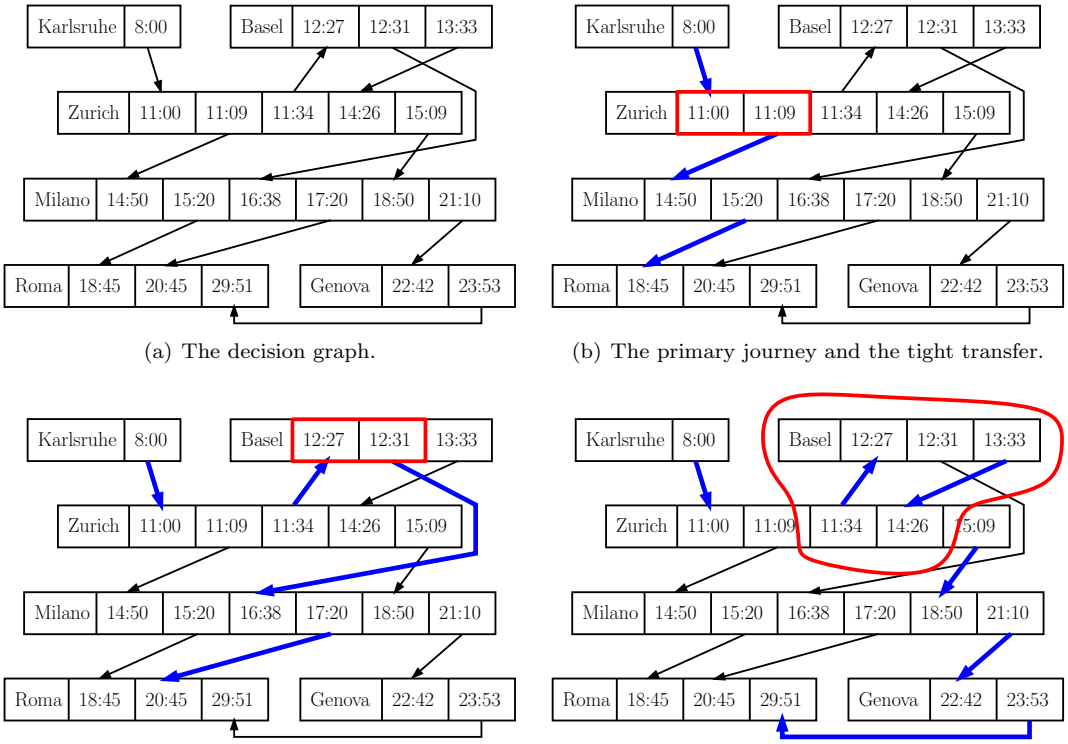
## 3.2 Related Work

Usually, these journey routing problems have been solved by (variants of) Dijkstra's algorithm on an appropriate graph (representing the timetable), see Deliverable D3.1 for an overview. Most relevant to our work is the realistic *time-expanded model* [24]. It expands time in the sense that it creates a vertex for each *event* in the timetable (such as a vehicle departing or arriving at a stop). Then, for every connection it inserts an arc between its respective departure/arrival events, and also arcs that link subsequent connections. Arcs are always weighted by the time difference of their linked events. Special vertices may be added to respect minimum change times at stops. See [16, 24] for details.

We are not the first to consider stochastic methods to formulate variants of the delay robust routing problems [3, 14]. However the delay model we propose and evaluate is unique in the sense that routing algorithms are known for it that scale to large networks as they occur in reality. We achieve this by considering a model that is more simplistic than the existing alternatives and show that this simplicity can be leveraged into simpler and more efficient algorithms.

In [14] the authors propose a stochastic delay model that is used to identify transfers that are unlikely to work. They show that the identification problem is strongly NP-hard. Further they introduce a query algorithm that computes Pareto-optimal journeys with respect to arrival time, number of safe transfers and number of unsafe transfers. As the identification problem is a crucial component of every routing system it is most likely not possible to construct an efficient solution based on the proposed model. We conclude that a simpler mode is needed.

In [14] another stochastic delay model is proposed that is used to predict delays. The authors evaluate the quality of their algorithm's predictions by comparing it to real world delay data. They show that in some cases their model is of by a large margin which makes their model impractical.




(a) The decision graph.

(b) The primary journey and the tight transfer.

(c) The first backup journey and the tight transfer.

(d) The backup's backup journey and an example of speculative traveling.

Figure 5: The horizontal boxes represent stops. For every box the name of the stop is given and a number of points in time. These are ordered increasingly from left to right. The arrows correspond to trains and connect their departure time with their arrival time. These time are the ones given in the timetable and are not adjusted for delays. In every situation the user should try to get the first train that he can catch. If a train is delayed then the user can lookup the next best train by picking the next time box to the right of his arriving train that has a departing train. The example decision graph travels from Karlsruhe in the top left to Roma in the bottom left. The user is given the graph as depicted in 5(a). If there are no delays then he will take the journey as highlighted in 5(b). The transfer in Zurich with only 9 minutes is very tight and therefore a backup is needed. This backup is highlighted in 5(c). Unfortunately the backup has another tight transfer in Basel with only 4 minutes. The backup journey needs therefore its own backup which is highlighted in 5(d). Figure 5(d) contains also an example of speculative traveling. If the user misses his first train in Zurich, then for an extended period of time no good backup exists and therefore it is beneficial to travel to Basel and to speculate on getting the connecting train there. If this fails then the user should take the train back to Zurich. If the speculation fails then the user has lost nothing because no train departed in Zurich within this time frame that would have been beneficial.

Their model propagates delay information over time which makes the model more realistic but also has the drawback that errors in the information are also propagated and accumulate over time. We suppose that this effect is at the root of the large errors that the authors unfortunately observe. Every fully realistic model needs to propagate delay information. Every model that does this, however, needs to be highly detailed and needs very precise input delay information. Solving the routing problem on more detailed models, however, is likely to be hard in term of computational complexity, making the existence of efficient algorithms unlikely. A further problem is that precise input delay information must be gathered. To our knowledge current systems only gather very rough information which contains too much noise. To sum up, we conclude that a fully realistic model is not usable and settle with the most complex model that does not propagate delays.

## 3.3 Stochastic Models

In this section, we consider models for delay-robust stochastic routing. In Section 3.4, 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 3.6, 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 Section 3.7 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 all the 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 which next state is.

We consider the problem of guiding the user towards a target stop given a start stop and a 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 exists 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 each 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 6 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 $p_{\text{arrtime}}$ 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 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 3.6.2. 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

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

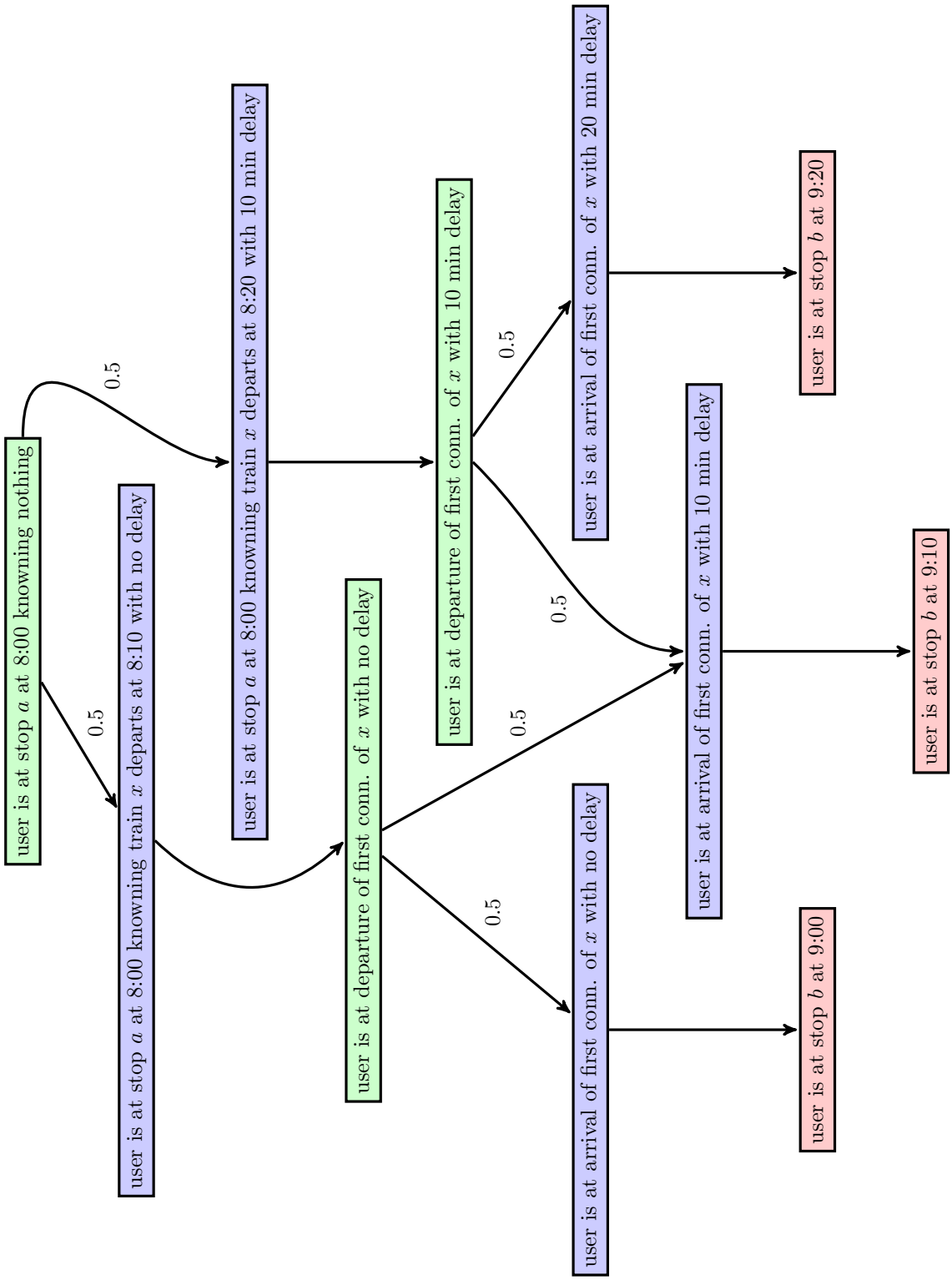



Figure 6: A simplified decision graph with a start stop $a$, a target stop $b$ and a trip $x$. The out-arcs of happen-states are labeled with their probability. The expected arrival time is $0.5^2 \cdot 9{:}00 + 2 \cdot 0.5^2 \cdot 9{:}10 + 0.5^2 \cdot 9{:}20 = 9{:}10$. Action states are in blue, happen states in green and target states in red.

if $q$ is a happen-state and as

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

if $x$ is an action-state.

## 3.4 Formal Definition

This section represents 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 3.7.1. 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 which 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 $(t, s, \text{Ne})$-triple. Note that this is the only state where the tuple components are not only simple numbers. This is an action-state.

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.

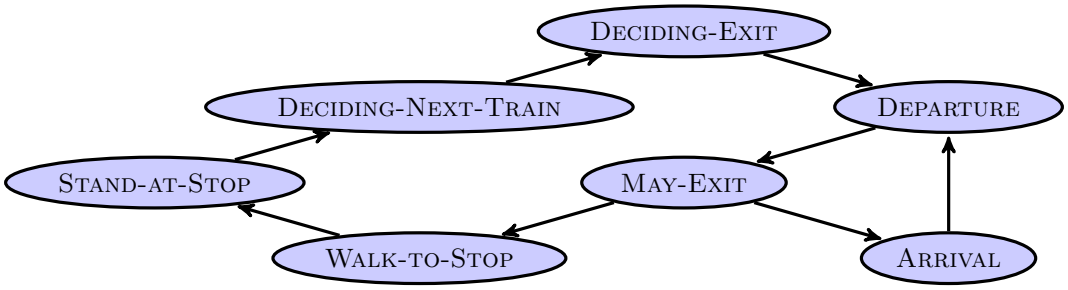



Figure 7: A graph containing the state classes as nodes. If a class $A$ contains a state which has a next state in class $B$ then the graph contains an arc $(A, B)$.

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 7 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.

## 3.5 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.

### 3.5.1 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 $(t, s, \mathrm{Ne})$ 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 a 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 $(t, s, \mathrm{Ne})$ 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 $(t, s, \mathrm{Ne})$ 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.

### 3.5.2 Next States of DECIDING-NEXT-TRAIN

The DECIDING-NEXT-TRAIN state represents a user that is at a stop and is aware of some exact delays. He must decide which train to take. This decision basically consists of choosing a departing connection. There are two groups of connections that depart from 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 $\min \emptyset = +\infty$.

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

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.

### 3.5.3 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

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

### 3.5.4 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.

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

### 3.5.5 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 $d_{\text{arr}}$ if it started with a delay of $d_{\text{dep}}$. 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 $d_{\text{arr}}$ such that $c_{\text{deptime}} + d_{\text{dep}} \geq c_{\text{arrtime}} + d_{\text{arr}}$). The functions relate as follows.

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

---

### 3.5.6 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 $(c_{\text{next}}, d, e)$ 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

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

## 3.6 The eCOMPASS Approach to Delay-Robust Stochastic Routing

In this section we describe algorithms that compute the minimum expected arrival time for the various models introduced. We start with a recursive program 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 Section 3.7 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.

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: 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 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 $(c_{\text{next}}, d, e)$ 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.

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

Table 1: Time potentials for problems without the Stop-Delay-Reset option. The departure time of a connection $c$ is $c_{\mathrm{dep}}$ and its arrival time is $c_{\mathrm{arr}}$.

### 3.6.1 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).

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 3.4.

We prove 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 $\epsilon$.* 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 $\epsilon$ exists. Consider two states of the same class. As shown in figure 7 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

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

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 $\epsilon$ is more complicated:

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

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 $d_{\mathrm{dep}} = 0$, $d_{\mathrm{arr}} = 0$ and $c$ to some element in $C$). Further it is finite because $C$ and $D$ are finite. Hence it has a minimum.

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 ;

Figure 8: Dynamic Program

The termination proof holds only for finite networks. See section 3.6.4 for an example where the recursive program would not terminate as the minimum decision graph is infinite.

### 3.6.2 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 Figure 8 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 an all-to-one algorithm.

### 3.6.3 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 in ascending order 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 7. 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.

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

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 3.6.1 for the exact value of $\epsilon$.
4. $t > t'$: This can not happen because it violated the time potential definition.

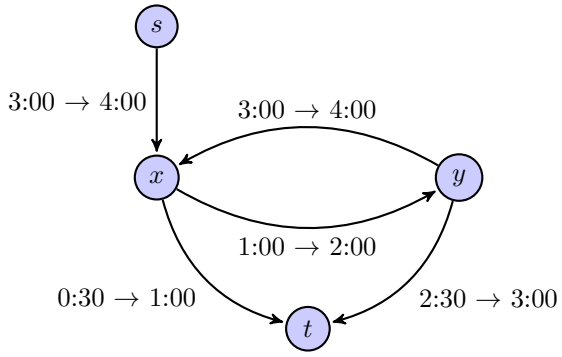



Figure 9: The user wants to get from $s$ to $t$. The start time is 3:00. The time period is 4 hours. A train has a maximum delay of 40 min. The chance that the delay is less than 30 min is $p$. The trains always depart on time.

### 3.6.4 Problems with Infinite Extensions

In this section we considers periodic networks. The proof of termination Theorem 3.6.1 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.

Consider the periodic network illustrated in figure 9. 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:

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.

## 3.7 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 section. All connections have an independent random

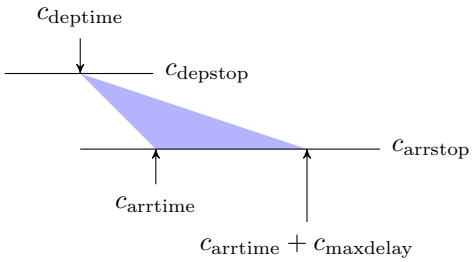



Figure 10: How a connection is represented.

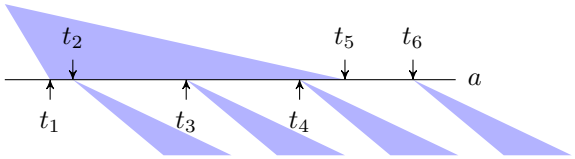



Figure 11: A stop with one incoming and four departing connections.

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 Section and construct an algorithm to solve this problem that is nearly identical to the backward profile algorithm introduced in Task 3.3. The only difference is the evaluation function of step functions. Finally, we describe how to generate the diagrams such as those shown in Figure 5.

### 3.7.1 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 10. 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 its line. The left end of a stop's line represents an early moment at that stop whereas the right end represents a later one. Connections are represented using triangles. They depart at their departure stop precisely at $c_{\text{deptime}}$ and arrive some time between $c_{\text{arrtime}}$ and $c_{\text{arrtime}} + c_{\text{maxdelay}}$ at the target stop. The top edge of the triangle represents the user being at $c_{\text{deptime}}$ at the stop $c_{\text{depstop}}$ and the bottom side him being at $c_{\text{arrstop}}$ 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 section 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 11. The user arrives at a stop $a$ some time 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

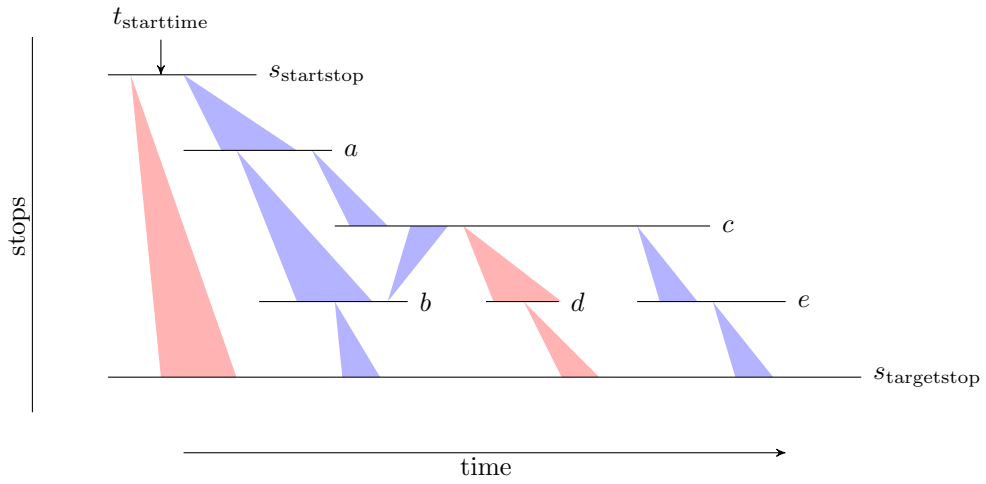



Figure 12: A timetable extract. The blue connections are in the decision graph. The red ones are not.

$t_3$. 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. 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 12. The user is at the start stop $s_{\text{startstop}}$ at the moment $t_{\text{starttime}}$ and wants to get to the target stop $s_{p_t}$. 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.

### 3.7.2 Formula Simplification and Algorithm

First let us recall the relevant formulas. We denote by $\alpha$ the M.E.A.T. of a Stand-at-Stop state, by $\beta$ the one of a Deciding-Next-Train state, by $\gamma$ the one of a Deciding-Exit state, by $\delta$ the one of a Departure state, by $\epsilon$ the one of a May-Exit state, by $\zeta$ the one of a Arrival state and by $\eta$ the one of a Walk-to-Stop state.

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

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:

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

By inlining $\beta$ and $\zeta$ and eliminating constant zero parameters this can be simplified to:

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

We show that $\delta$ does not depend on the exact value of $c$. More formally we show that: 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 $\delta(x,z) = \delta(y,z)$.

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

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

$$
\begin{aligned}
\gamma(x) &= \min_{i \in \{1 \ldots n\}} \delta(x, c_i) \\
&= \min\{\delta(c_1,c_1), \delta(c_1,c_2), \ldots, \delta(c_1,c_n)\} \\
&= \min\{\delta(c_1,c_1), \delta(c_2,c_2), \ldots, \delta(c_2,c_n)\} \text{ using Lemma 3.7.2} \\
&= \min\{\delta(c_1,c_1), \min\{\delta(c_2,c_2), \ldots, \delta(c_2,c_n)\}\} \\
&= \min\{\delta(c_1,c_1), \gamma(c_2)\} \\
&= \min\{\delta(x,x), \gamma(y)\}
\end{aligned}
$$

Using this lemma allows us to reformulate $\gamma$ 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

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

At first these transformations seem to lead nowhere, however, the structure of the corresponding dynamic program (as described in Section 3.6.2) is now very similar to the profile Connection Scan algorithm described in Deliverable D3.3. The expression $\eta(t,s)$ is a step function (for a fixed $s$) which can be interpreted as a profile mapping the departure time at $s$ onto the minimum expected arrival time at the target stop. Similarly the earliest trip arrival time is replaced by a minimum expected trip arrival time and corresponds to $\gamma(c)$. Note that the datastructures are exactly the same. The dynamic program introduced in section 3.6.2 visits the states using a topological ordering. Recall that ordering the connections by departure time yields such an ordering and therefore the adapted profile algorithm is an optimized variant of the dynamic program. The adapted profile algorithm is therefore correct.

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 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

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

instead of only $c_{\text{arrtime}}$.

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.,

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

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

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

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 $(d_1,a_1)\ldots(d_n,a_n)$ 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

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.

$i \leftarrow 1$;
**while** $d_i < x$ **do**
  $i \leftarrow i + 1$;
$p \leftarrow P_c(d_i - x)$;
$t \leftarrow p \cdot a_i$;
**while** $p \neq 1$ **do**
  $i \leftarrow i + 1$;
  $p' \leftarrow P_c(d_i - x)$;
  $t \leftarrow t + (p' - p) \cdot a_i$;
  $p \leftarrow p'$;
**return** $t$;

**Procedure** evaluate(x)

Figure 13: evaluation operation for M.E.A.T. algorithm

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 13 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.

### 3.7.3 Delay Distribution Functions

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. [11]. 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.

Table 2: Size figures for our timetables.

| Figures | London | Germany | Europe |
|---|---|---|---|
| Stops | 20 843 | 6 822 | 30 517 |
| Trips | 125 537 | 94 858 | 463 887 |
| Connections | 4 850 431 | 976 678 | 4 654 812 |
| Footpaths | 45 652 | 0 | 0 |

Table 3: Evaluating pCSA-P for the MEAT problem on all instances.

| Network | Max. Delay [min] | Decision Graph # Stops | Decision Graph # Arcs | All-To-One Time [ms] | One-To-One Time [ms] | One-To-One Dis. Time [ms] |
|---|---|---|---|---|---|---|
| Germany | 30 | 8 | 19 | 68.1 | 31.0 | 24.6 |
| Europe | 30 | 20 | 46 | 205.0 | 169.0 | 112.0 |
| London | 10 | 2 724 | 30 243 | 668.0 | 491.0 | 272.0 |

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.

## 3.8 Experiments

We ran experiments pinned to one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. We compiled our C++ code using g++ 4.7.1 with flags `-O3 -mavx`.

We consider three realistic inputs whose sizes are reported in Table 2. They are also used in [8, 13, 10], but we additionally filter them for (obvious) errors, such as duplicated trips and connections with non-positive travel time. Our main instance, London, is available at [27]. It includes tube (subway), bus, tram, Dockland Light Rail (DLR) and is our only instance that also includes footpaths. However, it has no minimum change times. The German and European networks were kindly provided by HaCon [15]. Both have minimum change times. The German network contains long-distance, regional, and commuter trains operated by Deutsche Bahn during the winter schedule of 2001/02. The European network contains long-distance trains, and is based on the winter schedule of 1996/97. To account for overnight trains and long journeys, our (aperiodic) timetables cover one (London), two (Germany), and three (Europe) consecutive days.

We ran for every experiment 10 000 queries with source and target stops chosen uniformly at random. Departure times are chosen at random between 0:00 and 24:00 (of the first day). We report the running time and the number of label comparisons, counting an SSE operation as a single comparison. Note that we disregard comparisons in the priority queue implementation.

### 3.8.1 Minimum Expected Arrival Time.

In Table 3 we present figures for the MEAT problem on all instances. Besides running time, we also report output complexity in terms of number of stops and arcs of the decision graph (see Figure 5 for an example). Real world delay data was not available to us. Hence, we follow Disser et al. [11] and assume that the probability of a train being delayed by $t$ minutes (or less) is $0.99 - 0.4 \cdot \exp(-t/8)$. After 30 min (10 min on London) we set this value to 1. Moreover, we also evaluate performance when discretizing the probability function at 60 equidistant points [11]. We run pCSA-P on 10 000 random queries and evaluate both the all-to-one and one-to-one (with earliest arrival pruning enabled) setting. Regarding output complexity, on the German and European networks the resulting decision

graphs are sufficiently small to be presented to the user. They consist of 8 stops and 19 arcs on average (Germany), roughly doubling on Europe. However, for London these figures are impractically large, increasing to 2 724 (stops) and 30 243 (arcs). Note that in a dense metropolitan network (such as London), trips operate much more frequently, therefore, many more alternate (and fall-back) journeys exist. These must all be captured by the output.

## 3.9 Conclusion and Outlook

We have successfully formalized the problem of computing robust journeys as the minimum expected arrival time (M.E.A.T.) problem. We have further shown that the problem can be solved efficiently even on large metropolitan areas such as London. The next step is to make it scale to even larger networks such as the complete transit network of Germany including not only the already evaluated long distance trains but also every single tram and bus. This network is significantly larger than the London network we considered up to now. We believe that this is not going to be possible without precomputing auxilary information in an offline phase. However, before we can make use of such techniques to solve the M.E.A.T. problem we will first have to incorporate them into the the algorithmic foundation on which we built our existing M.E.A.T. solver, i.e., the Connection Scan algorithm introduced in Task 3.3.

# References


[1] *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, 2011.

[2] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In Mark Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2010.

[3] Annabell Berger, Andreas Gebhardt, Matthias Müller–Hannemann, and Martin Ostrowski. Stochastic Delay Prediction in Large Train Networks. In ATMOS'11 [1], pages 100–111.

[4] Kateřina Böhmova, Matús Mihalák, Tobias Pröger, Rastislav Srámek, and Peter Widmayer. ecompass tr-023 - robust shortest path problems with uncertain costs. Technical report, The eCOMPASS Consortium, 2013.

[5] Justin Boyan and Michael Mitzenmacher. Improved results for route planning in stochastic transportation. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 895–902. Society for Industrial and Applied Mathematics, 2001.

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

[7] Joachim M. Buhmann, Matúš Mihalák, Rastislav Šrámek, and Peter Widmayer. Robust optimization in the presence of uncertainty. In *ITCS*, 2013.

[8] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. *ACM Journal of Experimental Algorithmics*, 17(1), July 2012.

[9] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering time-expanded graphs for faster timetable information. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 182–206. Springer Berlin Heidelberg, 2009.

[10] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140. SIAM, 2012.

[11] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.

[12] H Frank. Shortest paths in probabilistic graphs. *Operations Research*, 17(4):583–599, 1969.

[13] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer, May 2010.

[14] Marc Goerigk, Martin Knoth, Matthias Müller–Hannemann, Marie Schmidt, and Anita Schöbel. The Price of Robustness in Timetable Information. In ATMOS'11 [1], pages 76–87.

[15] HaCon. http://www.hacon.de/hafas/, 2013.

[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] Matthias Müller-Hannemann and Mathias Schnee. Efficient timetable information in the presence of delays. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2009.

[18] Matthias Müller-Hannemann and Karsten Weihe. Pareto shortest paths is often feasible in practice. In *Algorithm Engineering*, pages 185–197. Springer, 2001.

[19] Karl Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154–166, 1995.

[20] Evdokia Nikolova, Jonathan A Kelner, Matthew Brand, and Michael Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In *Algorithms–ESA 2006*, pages 552–563. Springer, 2006.

[21] Ariel Orda and Raphael Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM (JACM)*, 37(3):607–625, 1990.

[22] Ariel Orda and Raphael Rom. Minimum weight paths in time-dependent networks. *Networks*, 21(3):295–319, 1991.

[23] Stefano Pallottino and Maria Grazia Scutella. Shortest path algorithms in transportation models: classical and innovative aspects. *Equilibrium and advanced transportation modelling*, 245:281, 1998.

[24] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

[25] Mathias Schnee. *Fully realistic multi-criteria timetable information systems*. PhD thesis, 2009.

[26] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Algorithm Engineering and Experiments*, pages 43–59. Springer, 2002.

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