The present work was submitted to the LuFG Theory of Hybrid Systems

MASTER OF SCIENCE THESIS

ANALYZING DELAY PROPAGATION IN RAILWAY NETWORKS

Valerie Li-Li Tan

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Nils Nießen

Additional Advisor: Rebecca Haehn

Aachen, 09. March 2022

Abstract

Symbolic Simulation is an exact, synchronous railway simulation that takes in macroscopic inputs. It models primary delays as discrete probability distributions, and in one single run, computes all possible executions under all possible scenarios while also accounting for all stochastic dependencies. The output of Symbolic Simulation consists of final arrival times, each associated with a train and an exact probability.

This thesis extends upon Symbolic Simulation by finding explanations and improvements for secondary delays. The first approach involves computing first-order secondary delays that a train causes on another; that is, it records the expected amount of time that one train occupies an infrastructure element which another train needs to move into. The second approach experiments with smaller and faster pruned versions of the base simulation of some dataset, and determines cases in which increasing primary delays of one train decreases the overall amount of delays of the pruned simulation. The latter approach has shown to be promising, and in addition to determining explanations for secondary delays, also provides possible solutions.

Acknowledgements

Thank you very much to my supervisors, Professor Ábrahám and Rebecca Haehn for your tremendous help and patience. I am very thankful for having the opportunity to work with you on this very interesting and applicable topic. Moreover, many thanks to Professor Nießen for being my second supervisor. I am also very thankful for my family and friends for their care and support.

Contents

Chapter 1

Introduction

In 2020, the European Commission approved of the European Green Deal with the goal of reaching a carbon neutral economy in the EU by 2050 [9]. The transport sector is currently the second highest emitter of greenhouse gases, particularly due to the growth of emissions from road transportation [9]. On the other hand, railways produce fewer emissions, have lower external costs, decrease congestion on roads, and contribute to better air quality [9]. In Germany alone, the usage of passenger trains has grown steadily since 2001, and despite a drop in 2020, the Federal Ministry for Digital and Transport forecasts 3.129 billion passengers in 2024, the highest value since 2001 [10, 4]. Similarly, for freight trains, the amount of goods transported has also had a general upward trend since 2001 [10, 4].

Due to increased demand and the necessity of greener transport, improving the performance of railway systems is important, especially when higher traffic leads to more delays [7]. With more traffic, a single train delay can cause further delays in other trains. Therefore, it is crucial to improve railway systems by reducing propagated delays, also known as "secondary delays," and to create better timetables that are more resilient to such delays. Simulation allows us to inexpensively model trains and their movement in many scenarios, and can provide opportunities to improve timetables without requiring expensive changes in infrastructure [7].

All railway simulation systems take in input that is simplified to a certain extent, and then make further assumptions on train movement during the execution. To the best of our knowledge, all other railway simulations do some form of estimation of secondary delays, such as through Monte Carlo simulation in which only a small fraction of all possible combination of initial delays are considered. The approach worked with in this thesis, called Symbolic Simulation, is a novel system described in the paper, "Symbolic Simulation of Railway Timetables under Consideration of Stochastic Dependencies" [7]. It computes *exact* probabilistic descriptions of trains at any time point, given the input primary delays with distributions [7]. This approach even accounts for all stochastic dependencies, and therefore accurately accounts for all secondary delays under all possible scenarios [7].

While the invariant of accounting for all stochastic dependencies is maintained, Symbolic Simulation does not yet output explanations for secondary delays, suggest trains that have a tendency to cause more secondary delays on other trains, or even find cases in which further delaying one train decreases the total amount of delays of other trains. While we cannot avoid the inevitable "primary delay" such as a signal fault or a large number of passengers, Symbolic Simulation's strength in accounting for stochastic dependencies exactly makes it a good system for analyzing and potentially reducing secondary delays.

In this thesis, we explore two approaches in providing explanations and suggestions for secondary delays. First, we describe a scoring approach that is updated in each time step in an attempt to determine how much direct negative influence a train has on another. Unlike with primary delays, computing the amount of secondary delay that one train causes on another is difficult to determine due to the large number of stochastically dependent events that occur during a simulation, akin to trying to find the butterfly (realistically, multiple butterflies) that cause a major event in some "Butterfly Effect" situation. Nonetheless, this first approach tries an approach to storing direct effects between an instance of a train and another train in general. The second approach is a pruning approach that selects one instance of a train with a high expected delay, and then "prunes" the simulation to only include simulation input relevant to the selected train instance. Under a pruned simulation, we determine cases in which further delaying a different train by a particular number of minutes decreases the total delay of the selected train, and even the total amount of delay over all trains.

We begin this thesis by discussing a few other railway systems in the beginning of Chapter 2, and furthermore describe Symbolic Simulation in detail, based on the work in [7]. Chapter 3 discusses two attempts at computing scores for train instances and suggestions for further work and improvement. Chapter 4 describes the second approach on pruning and analyzes preliminary results. Finally, Chapter 5 summarizes our work and outlines further work.

Chapter 2

Preliminaries

In this section, we will describe some basics of railway simulation and give an overview of a subset of railway simulations. We also describe with moderate detail how our Symbolic Simulation works and provide an example to help with understanding.

2.1 Overview of Railway Simulation Systems

2.1.1 Delays

Since *delays* play a central role in our work, it is important to classify them into two main groups. The first is *primary delays*, which are delays that are not caused by other delays [7]. These delays could occur as a result of direct real-world events such as a large number of passengers, or technical issues such as signal or brake faults [6]. We define *secondary delays* of a train as delays that occur when the train needs to wait to move to its next infrastructure element, which is currently at full capacity. Possible reasons for such secondary delays could include other late trains occupying its next infrastructure element, or the train itself is running late.

In other literature, secondary delays are also referred to as “knock-on delays” or “propagation of delays” [14, 13].

2.1.2 Classification of Existing Simulation Systems

Various existing railway simulation systems can be classified. One way is based on the level of detail of their inputs. *Microscropic* models represent the infrastructure in great detail, while *macroscopic* models describe the infrastructure on a higher level. Next, we can classify railway systems as either *synchronous* or *asynchronous.* Synchronous models simulate all trains at once, in a similar vein to real-world operations. On the other hand, asynchronous models simulate each train run, one at a time. The highest priority train is simulated and its schedule is “locked-in”, then the next highest priority train run is simulated around the first train, and so on. Therefore, in asynchronous simulations, a lower priority train run does not impact a higher priority train run [8].

Many systems employ a *Monte Carlo* simulation, where many runs are made, with each run containing randomly selected discrete primary delays. After many runs, the system runs statistical analysis on all of them as a whole. Nonetheless, even with many runs, such systems usually encompass only a fraction of all simulations that are possible under the input set of primary delays.

Many systems also employ some form of analytical estimation of secondary delays, with or without Monte Carlo simulation. A notable example of analytical estimation is the STRELE formula, created by Schwanhäußer in 1974, which has been used in many railway tools in Germany [14, 15]. This formula is based off of a queueing system [14].

2.1.3 A Subset of Existing Railway Simulations

Below, we discuss existing railway simulations that specifically work with timetable robustness and delays. Due to the propriety nature of them, information is limited.

Open Track is a system developed by the Swiss Federal Institute of Technology and was used by the Swiss National Railroad along with many other customers [1]. It takes in microscropic models as inputs and computes Monte Carlo–style synchronous simulations [1, 8].

RailSys is a system developed by the Institute of Transport, Railway Construction and Operation at the University of Hannover in the early 2000s. Like Open Track, it also uses microscopic inputs and computes Monte Carlo–style synchronous simulations [11].

MOSES/WiZug is a system for simulating freight trains for the Deutsche Bahn and was developed by a number of various institutes in Germany, including the Institute of Transport Science at RWTH Aachen University. Unlike the previously mentioned systems, it uses macroscropic models and asynchronous simulation [12]. Moreover, it also employs Monte Carlo simulation [7]. A "newer approach" described in the paper discusses using Queue Theory for more precise calculations of secondary delays [12].

LUKS is a system developed by VIA Consulting & Development in Germany. It uses microscropic models as input, and utilizes both a synchronous and asynchronous approach [2, 7]. In addition, it uses Monte Carlo simulation [5].

OnTime is a system developed specifically for assessing the quality of timetables and is in use in Switzerland and Belgium [5]. It can take in either microscopic or macroscopic input data. Moreover, OnTime models primary delays as cumulative distribution functions, and uses *analytical* simulation instead of Monte Carlo simulation [3, 5].

2.2 Symbolic Simulation

While other railway simulation systems typically use Monte Carlo simulation, Symbolic Simulation differs by being an *exact* symbolic simulation. It takes in macroscopic inputs and is also synchronous. Only one run is computed in the simulation, since a single run computes all possible executions at once, given the primary delays. At the end of the simulation, for each train, we can compute all of its possible arrival times at its target with their respective exact probabilities.

The definitions and algorithms in this section have been copied or adapted from [7].

2.2.1 Railway System Model

We begin by defining how our macroscropic model is represented in our simulation.

Definition 2.2.1. *An* infrastructure network *is a tuple $I = (V,E,c,B,b)$ with a finite set V of* vertices*, a set $E \subseteq V \times V$ of (directed)* edges*, a* capacity function $c : (V \cup E) \to \mathbb{N}_{>0}$*, a* bidirectional edge *set $B \subseteq E$ and a* coupling function $b : B \to B$ *such that for all $e \in B$ and $b(e) = e'$ we have $b(e') = e$ and $c(e) = c(e')$.*

Definition 2.2.2. *An* infrastructure element $x \in V \cup E$ *is either a* vertex *or an* edge*.*

Throughout this work, we will refer to a generalization of vertices and edges as infrastructure elements as defined in Definition 2.2.2.

We define a capacity function c, to determine the number of concurrent trains allowed on an infrastructure element at any time point. If the infrastructure element is a vertex, the capacity refers to the number of trains allowed at a station at the same time. If it is an edge, the capacity refers to the number of parallel tracks available.

Bidirectional tracks exist in our model, and we represent a single bidirectional edge as two unidirectional edges, with b as a coupling function to record such pairing. We also ensure in our implementation that at any given time point, only one direction is used. To simplify our examples in this thesis, we neglect discussing bidirectional edges, although they are accounted for in our implementation.

In our implementation, we also added a *source* node to V and connected every other $v \in V$ to the source node with an edge of unlimited capacity. This is to model the possibility that a train may not be able to start on time because its initial vertex is already fully occupied. In addition, we added a *target* node and connected each $v \in V$ with an edge with unlimited capacity, to model a train leaving the infrastructure or starting a new train ride. Doing this prevents finished trains from occupying their final nodes beyond what was intended in our simulation or in the real world.

Definition 2.2.3. *A* time window $\mathbb{T} = [t_{min}, t_{max}] \subseteq \mathbb{R}$ *with t_{min},$t_{max} \in \mathbb{N}$, $t_{min} < t_{max}$, represents the time interval considered in our simulation.*

Definition 2.2.4. *A* timed path *is a finite sequence $\pi = (v^1(a^1 \mapsto d^1), \ldots, v^k(a^k \mapsto d^k))$ such that for all i,$i' \in [1..k]$ and $j \in [1..k-1]$: (i) $v^i \in V$ and $(v^j, v^{j+1}) \in E$, (ii) $a^i \leq d^i$ and $d^j \leq a^{j+1}$ for* arrival *and* departure *times a^i,$d^i \in \mathbb{T}$, and (iii) $i \neq i'$ implies $v^i \neq v^{i'}$ (loop-free).*

Definition 2.2.4 from [7] ensures that all vertices in π indeed exist in V, all edges formed by two consecutively listed vertices within a sequence also exist in E, arrival and departure times are ordered properly, and there are no loops in the path.

Definition 2.2.5. *A* train (ride) $z = (type,\pi)$ *specifies a train type (e.g. freight train) from a finite ordered domain and a timed path π.*

Definition 2.2.6. *A railway* timetable $T = \{z_1, \ldots, z_n\}$ *is a finite ordered set of trains. We call i the* identity *of train z_i,and recall that $z_i = (type_i,\pi_i)$ and $\pi_i = (v_i^1(a_i^1 \mapsto d_i^1), \ldots, v_i^{k_i}(a_i^{k_i} \mapsto d_i^{k_i}))$ for $i \in [1..n]$.*

Definition 2.2.7. *The* safety distance $\delta \in \mathbb{N}$ *is the number of time units each infrastructure element is blocked for safety reasons after a train has left it.*

Definition 2.2.8. *A timetable $T = \{z_1, \ldots, z_n\}$ with δ in consideration, is* executable *over $\mathbb{T}$ if it does not exceed the available capacities in the absence of delay.*

Additional formal statements for verifying whether a timetable is even executable without delays are detailed further in [7].

In the following, we define primary delays for each train and infrastructure pair exactly as in [7].

Definition 2.2.9. *For each $i \in [1..n]$, $j \in [1..k_i]$ and $j' \in [1..k_i{-}1]$, we define stochastically independent discrete* random variables $p_i^{v_j}$ *and* $p_i^{(v_i^{j'}, v_i^{j'+1})}$ *with sample space $\mathbb{N}$. Let P be the set of all these random variables. For each $p_i^x \in P$ we denote the* probability *that p_i^x has the value $\Delta \in \mathbb{N}$ as $\mathbb{P}(p_i^x = \Delta) \in [0{,}1]$ and its finite* support *as $\mathcal{D}(p_i^x) = \{\Delta \in \mathbb{N} | \mathbb{P}(p_i^x = \Delta) > 0\}$. It holds that $\sum_{\Delta \in \mathcal{D}(p_i^x)} \mathbb{P}(p_i^x = \Delta) = 1$ for each $p_i^x \in P$.*

For some infrastructure element x and train z_i, p_i^x outputs an integer delay value representing the *additional* delay z_i has in its *departure* from x. These random variables are also assumed to be stochastically independent from each other as explained in [7]. $\mathcal{D}(p_i^x)$ is the set of all possible delay values for z_i at x that have nonzero probability.

The implementation also uses random variables representing a special type of primary delay known as an *initial delay*, p_i^{entry} to represent the delay in a train's *arrival* at its first vertex off its route.

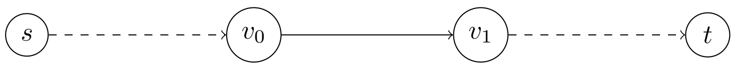
Definition 2.2.10. *A* random inclusion *s for $p_i^x \in P$ has the form $p_i^x \lhd D$ for some $D \subseteq \mathcal{D}(p_i^x)$, $D \neq \emptyset$; we define $\mathbb{P}(s) = \sum_{\Delta \in D} \mathbb{P}(p_i^x = \Delta)$. As shorthand, we also denote a random inclusion $p_i^x \lhd \{\Delta\}$ with cardinality of one as $p_i^x = \Delta$.*

Definition 2.2.11. *A* scenario *S is a set that contains exactly one random inclusion for each random variable. Let $\mathbb{S}$ be the set of all scenarios. For $S \in \mathbb{S}$ and $(p_i^x \lhd D) \in S$ we define $S(p_i^x) = D$, and set $\mathbb{P}(S) = \Pi_{s \in S} \mathbb{P}(s)$.*

To make our written scenarios appear more concise, we typically leave out trivial random inclusions $p_i^x \lhd \mathcal{D}(p_i^x)$. We also denote the maximal scenario, where each random variable's random inclusion contains all probable delay values, as $\emptyset$.

Example 2.2.1. *Let us consider a simple infrastructure with two nodes, v_0 and v_1 with a unidirectional edge in between $e = (v_0, v_1)$.*

Let us also add a source node s and target node t. In the implementation, we would normally connect s to both v_0 and v_1 but for simplicity, we only depict an edge between s and v_0, as our example will not contain a path starting at v_1. Likewise, we only depict an edge between v_0 and t for visual simplicity, as no path in our example will end on v_0.



Suppose there are two trains that traverse these nodes with the names z_0 and z_1. We define the set of (relevant) random variables in our example as

$$P = \{p_0^{entry}, p_0^{v_1}, p_0^{v_2}, p_0^{e}, p_1^{entry}, p_1^{v_1}, p_1^{v_2}, p_1^{e}\}.$$

Let us define the respective finite support for each random variable as follows:

$$\begin{aligned}
\mathcal{D}(p_0^{entry}) &= \{0,1,2\}\\
\mathcal{D}(p_0^{v_1}) &= \{0\}\\
\mathcal{D}(p_0^{v_2}) &= \{0\}\\
\mathcal{D}(p_0^{e}) &= \{0\}\\
\mathcal{D}(p_1^{entry}) &= \{0,1,2\}\\
\mathcal{D}(p_1^{v_1}) &= \{0\}\\
\mathcal{D}(p_1^{v_2}) &= \{0\}\\
\mathcal{D}(p_1^{e}) &= \{0\}
\end{aligned}$$

The only random inclusions here that contain nonzero delay values are of those that represent initial delays for trains at the very start of their respective rides. This is done to simplify our example. In the implementation of Symbolic Simulation, we also work with just initial delays for now to minimize runtimes [7].

The scenario $\{p_0^{entry} = 1\}$ is equivalent to $\{p_0^{entry} \lhd \{1\}\}$, which is also equivalent to the verbose representation including the trivial random inclusions which we typically exclude in our notation:

$$\{p_0^{entry} = 1, p_0^{v_1} = 0, p_0^{v_2} = 0, p_0^{e} = 0, p_1^{entry} \lhd \{0,1,2\}, p_1^{v_1} = 0, p_1^{v_2} = 0, p_1^{e} = 0\}.$$

A nonexample of a scenario is $\{p_1^{v_1} \lhd \emptyset\}$, since a train must have a positive or zero delay at v_1.

Definition 2.2.12. *We call S* complete *iff $|S(p_i^x)| = 1$ for each $p_i^x \in P$.*

Example 2.2.2. *Using the infrastructure and finite support from the previous example, an example of a valid complete scenario is:*

$$\{p_0^{entry} = 0, p_1^{entry} = 1\}.$$

Examples of incomplete scenarios are:

$$\{p_0^{entry} = 0, p_1^{entry} \lhd \{0,1\}\}$$

$$\{p_0^{entry} = 1\} = \{p_0^{entry} = 1, p_1^{entry} \lhd \{0,1,2\}\}$$

Definition 2.2.13. *We say that $S \in \mathbb{S}$* refines *$S' \in \mathbb{S}$ (written $S \preceq S'$) iff $S(p_i^x) \subseteq S'(p_i^x)$ for all $p_i^x \in P$; we also say that S'* contains *S.*

Example 2.2.3. *Using the setup from the previous examples, scenario $\{p_0^{entry} = 0\}$ contains the scenarios $\{p_0^{entry} = 0, p_1^{entry} = 0\}$, $\{p_0^{entry} = 0, p_1^{entry} = 1\}$, and $\{p_0^{entry} = 0, p_1^{entry} = 2\}$. Equivalently we can say, for example, that $\{p_0^{entry} = 0, p_1^{entry} = 0\}$* refines *$\{p_0^{entry} = 0\}$.*

Definition 2.2.14. *We call S and S'* compatible *iff $S(p_i^x) \cap S'(p_i^x) \neq \emptyset$ for all $p_i^x \in P$. For two compatible scenarios S and S' we define $S \bigtriangleup S'$ as the scenario $\{p_i^x \lhd (S(p_i^x) \cap S'(p_i^x)) \mid p_i^x \in P\}$.*

Example 2.2.4. *Consider two scenarios related to our example:*

$$S = \{p_0^{entry} \lhd \{0, 1\}, p_1^{entry} \lhd \{1,2\}\}$$
$$S' = \{p_0^{entry} \lhd \{0, 2\}, p_1^{entry} \lhd \{0,1\}\}$$

S and S' are compatible and $S \bigtriangleup S' = \{p_0^{entry} = 0, p_1^{entry} = 1\}$.
An example of two incompatible scenarios would be:

$$S = \{p_0^{entry} = 1, p_1^{entry} \lhd \{1,2\}\}$$
$$S' = \{p_0^{entry} \lhd \{0, 2\}, p_1^{entry} \lhd \{0,1\}\}$$

A complete scenario fixes exactly one delay value for each p_i^x. If we were to imagine a real-life-like example where we observed each train ride within some $\mathbb{T}$, and recorded each delay for each train ride at each infrastructure element, we would end up with data synonymous to a complete scenario. However, executing our simulation with only complete scenarios is very computationally expensive, so we therefore start our simulation working with incomplete and therefore more generalized scenarios in which each would cover more possible delay values per ${p_i}^x$.

2.2.2 Symbolic Simulation Algorithm

Our Symbolic Simulation uses such global-like scenarios defined in Section 2.2.1 that characterize all of the considered random inclusions for each random variable representing a train and infrastructure element pair. Now we need to run the simulation by moving *train instances* through our actual infrastructure itself. One train could be in different places depending on whether how much it gets delayed at each stop, as well as how much other trains get delayed and whether the desired infrastructure element is fully occupied for whatever set of reasons. Thus, we have a motivation to consider an instance of a train ride with a specific scenario. This allows us to do define different "parallel universes" for one specific train itself using scenarios.

Definition 2.2.15. *A* train instance *is a tuple (i, s, t), where i is the train identity, s is the scenario, and t is the time point of the train z_i's next planned movement.*

A train instance containing an incomplete scenario and residing at some infrastructure element x represents the notion that this train can exist at x under a set of different parallel universes.

Initial Inputs

We initialize our algorithm with the following input as defined in [7]:

- $I = (V,E,c)$: infrastructure network;
- $\delta \in \mathbb{N}$: safety distance;
- $T = \{z_1, \dots, z_n\}$: timetable for the time window $\mathbb{T} = [t_{min}, t_{max}]$;
- $P \subseteq \{p_i^x \mid i \in [1..n], x \in V \cup E\}$: set of random variables according to Def. 2.2.9 which also implicitly carry their probability distributions.

As global variables, we use the following sets:

- *times*: an ordered sequence of the time points $t \in \mathbb{T}$ in ascending order at which some trains want to change infrastructure element. For each time point, we maintain the invariant that no train can change its infrastructure element before the earliest point in *times*. *times* is also modified and updated during the simulation to keep this invariant true.
- for each infrastructure element $x \in V \cup E$:
  - *occupy*$[x]$: set of train instances occupying x at the current time point;
  - *block*$[x]$: set of train instances that left x but are still blocking it at the current time point due to the safety distance. The the time point t encodes the end time of blocking;
  - *req*$[x]$: set of train instances that want to move to x at the current time point;
  - *cap*$[x]$: set of scenario–int pairs, where the integer describes the number of trains occupying or blocking x in its respective scenario at the current time point.

Algorithm

Algorithm 1 Initialization

```
1: procedure INITIALIZE( )
2:     $V \leftarrow V \cup \{source, target\}$; $c(source) \leftarrow \infty$; $c(target) \leftarrow \infty$; $times \leftarrow \emptyset$;
3:     for each $v \in V$ do
4:         $E \leftarrow E \cup \{(source, v), (v, target)\}$; $c((source, v)) \leftarrow \infty$; $c((v, target)) \leftarrow \infty$;
5:     for each $x \in (V \cup E)$ do $occupy[x] \leftarrow \emptyset$; $block[x] \leftarrow \emptyset$; $req[x] \leftarrow \emptyset$; $cap[x] \leftarrow \emptyset$;

6:     for each $i \in \{1, \ldots, n\}$ do
7:         $times \leftarrow times \cup \{a_i^1\}$;
8:         for each $t \in \mathcal{D}(p_i^{entry})$ do
9:             $times \leftarrow times \cup \{a_i^1 + t\}$;
10:            $occupy[(source, v_i^1)] \leftarrow occupy[(source, v_i^1)] \cup \{(i, \{p_i^{entry} = t\}, a_i^1 + t)\}$;
```

The main method SIMULATE is presented in Algorithm 2. First, it calls INITIALIZE as described in Algorithm 1. INITIALIZE adds source and target nodes with unlimited capacity in line 2 and in line 4 connects each of them to all other vertices in V. Each of the aforementioned edges are also assigned unlimited capacities. All sets in line 5 are initialized as empty. In line 7, we add all arrival times of all of the trains in ascending order. In line 10, for each z_i and entry delay value, we create one train instance, with its scenario and adjusted next movement time, and place it on the edge between the source and its first vertex. We also update *times* in line 9.

After initializing everything, SIMULATE iterates over t in *times* in ascending order in line 4. We only need to consider $t \in times$ because those are the only points in time in which any status change could occur in our simulation. Within each time point, we iterate over all vertices and then edges in line 5. It is important to process vertices before edges because some trains may plan to pass a vertex such that the arrival and

Algorithm 2 Symbolic simulation

```
1: procedure SIMULATE( )
2:     INITIALIZE(); let time point $t \leftarrow t_{min}$;
3:     while $t \leq t_{max} \wedge times \neq \emptyset$ do
4:         $t \leftarrow times$.GETSMALLEST(); $times \leftarrow times \setminus \{t\}$;
5:         for each $x \in V \cup E$ do                // first vertices then edges
6:             REQUESTS($t$, $x$);                      // update $req[x]$
7:             OCCUPATION($t$, $x$);                    // update $cap[x]$
8:             while $req[x] \neq \emptyset$ do     // requests have to be sorted (highest priority first)
9:                 $r \leftarrow req[x]$.POP(); UPDATE($t$, $x$, $r$);   // update $occupy$ and $block$
```

Algorithm 3 Collecting requests

```
1: procedure REQUESTS($t \in \mathbb{T}$, $x \in V \cup E$)
2:     $req[x] \leftarrow \emptyset$;
3:     for each $y \in pre(x)$ do        // either incoming edges or source vertex of $x$
4:         for each $(i,S,t') \in occupy[y]$ do
5:             if $t' \leq t$ then $req[x] \leftarrow req[x] \cup \{(i,S,t')\}$;
```

departure times are the same. Processing vertices first will allow the train to move from edge into the vertex (in which the train has the same arrival and departure time), and move on to the following edge in the same time point.

Within each point in time and for each infrastructure element x, we gather all train instances that want to move to x in line 6 using Algorithm 3. Note that it is often the case to have two separate train instances wanting to enter x that refer to the same train (say z_2) but have different and incompatible scenarios which denote different "universes." In line 7, we use Algorithm 4 to update the capacity of x under different scenarios. Line 8 has us looping over the list of requests $req[x]$ sorted by priority; for example, the ICE trains have priority over freight trains. Further detailing can be found in [7]. Finally within the while loop in line 9, we pop train instances in order of priority from our requests and use Algorithm 5 to move trains while updating $occupy[x]$ and $block[x]$. When all of the infrastructure elements and time points have been looped through, our algorithm terminates.

Algorithm 3, REQUESTS receives a single time point t and infrastructure element x, and initially sets global $req[x]$ to be empty. Then in line 3, it looks at each predecessor, either an incoming edge or source vertex. For each predecessor in line 4, it looks at each train instance currently at it, and if the time of its next planned movement, t', is already less than or equal to our current time, then we add this train instance to $req[x]$.

Algorithm 4, OCCUPATION does two main things. First, in line 2, it removes all blocking train instances whose blocking times are now in the past. Secondly, it computes $cap[x]$, which itself is a set of pairs. Line 3 in OCCUPATION allows us to consider all train instances that either are blocking or occupying x and are thus relevant to computing $cap[x]$. In line 4, we use a recursive helper function called SPLIT to store the number of trains at x for each scenario. SPLIT starts out with the maximal scenario $\emptyset$ and recursively splits it into many incompatible scenarios, each

Algorithm 4 Computing the occupation of an infrastructure element

```
procedure Occupation($t \in \mathbb{T}$, $x \in V \cup E$)
    $block[x] \leftarrow \{(\cdot, \cdot, t') \in block[x] \mid t' \geq t\}$;
    $trains \leftarrow occupy[x] \cup block[x]$;
    $cap[x] \leftarrow$ Split($\emptyset, 0, trains, x$);
```

Algorithm 5 Updating a train instance's position

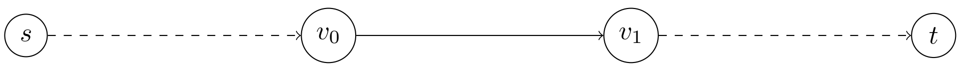
```
1:  procedure Update($t \in \mathbb{T}$, $x \in V \cup E$, $r = (i, S, t^*) \in [1..n] \times \mathbb{S} \times \mathbb{T}$)
2:      let set $\mathcal{S} \leftarrow \emptyset$;
3:      if $c(x) = \infty \vee |\{j|(j,\cdot,\cdot) \in occupy[x] \cup block[x] \cup req[x]\}| < c(x)$ then $\mathcal{S} \leftarrow \{S\}$
4:      else $\mathcal{S} \leftarrow$ Available($x$,$r$);
5:      for each $S' \in \mathcal{S}$ do
6:          for each $t' \in \mathcal{D}(p_i^x)$ do
7:              $occupy[x] \leftarrow occupy[x] \cup \{(i, S' \triangle \{p_i^x = t'\}, t + t' + t'')\}$;
8:              $times \leftarrow times \cup \{t + t' + t''\}$;            // $t''$ is waiting/driving time
9:          $block[pre(i,x)] \leftarrow block[pre(i,x)] \cup \{(i, S', t + \delta)\}$; $times \leftarrow times \cup \{t + \delta\}$;
10:     $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \setminus \{(i, S, t^*)\}$;
11:     for each $S' \in$ ScenarioDiff($S, \mathcal{S}$) do
12:         $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \cup \{(i, S', t^*)\}$;
```

with an integer capacity value to count how many train instances are at x for some scenario. After Split terminates, it holds that for each scenario S in $cap[x]$, each of its cases (e.g. a complete sub-scenario contained by S) has the same exact set of train instances occupying x. Further details on Split are elaborated upon in [7].

Algorithm 5, Update takes in a time point t, infrastructure element x, and a specific train instance r representing a train instance that wants to move to x. $\mathcal{S}$ is the set of all scenarios in which train instance r (paired with its own scenario) can move to x. Line 3 considers two simple cases, either the capacity of x is infinite, or when the sum of train instances blocking, occupying, and requesting for x is below the maximum capacity of x, even when we include train instances with scenarios that are not even compatible with our train instance's S. Essentially, there is just so much room that all of the requests, including r can move to x.

Otherwise, we call the helper function Available in line 4 to return the set of scenarios in which r can move to x, which is detailed in [7]. Lines 5-9 detail a great amount of bookkeeping. For each scenario and for each additional delay value associated with p_i^x, we update $occupy[x]$ with a modified train instance, as well as update *times* accordingly. In line 9, we add our current train instance to the blocking times of the previous infrastructure element. Similarly, line 9 allows us to remove our train instance from the the occupy list of the previous infrastructure element. It is important to note that r is a train instance likely associated with an incomplete scenario. That means that there are subscenarios of S where it is not possible for z_i to move to x. Lines 11-14 addresses this by placing train instances of z_i back to the previous infrastructure element in subscenarios where moving z_i to x is impossible. ScenarioDiff used in line 11 is a helper function also described in [7] and works similarly to a set difference function.

Example 2.2.5. *Recall example 2.2.1, with two trains z_1 and z_2 with their respective finite support and the following infrastructure:*



As in example 2.2.1, we defined the finite support of each random variable as $\mathcal{D}(p_0^{entry}) = \{0,1,2\}$, $\mathcal{D}(p_1^{entry}) = \{0,1,2\}$, and $\mathcal{D}(p_i^x) = 0$ for all other random variables. Let us define the capacities of $v_1, e = (v_1, v_2), v_2$ as 1, and infinite for the other infrastructure elements.

Consider a timetable $T = \{z_0, z_1\}$ with $z_0 = (Default, \pi_0)$ and $z_1 = (Default, \pi_1)$. Let us define the paths with associated arrival and departure times in minutes as such:

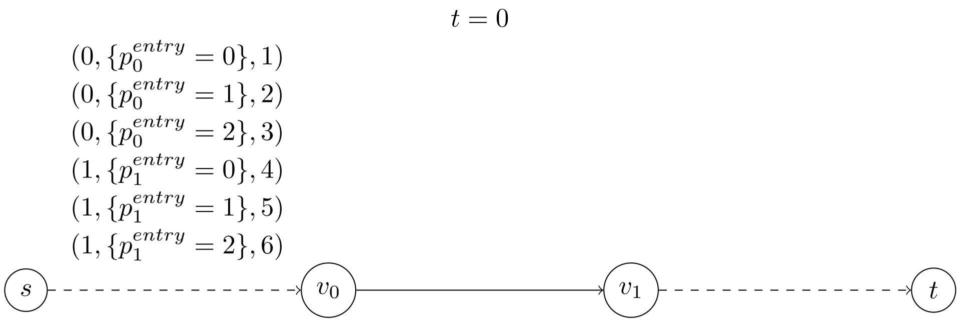
$$\pi_0 = (v^0(1 \mapsto 2), v^1(3 \mapsto 4))$$

$$\pi_1 = (v^0(4 \mapsto 5), v^1(6 \mapsto 7))$$

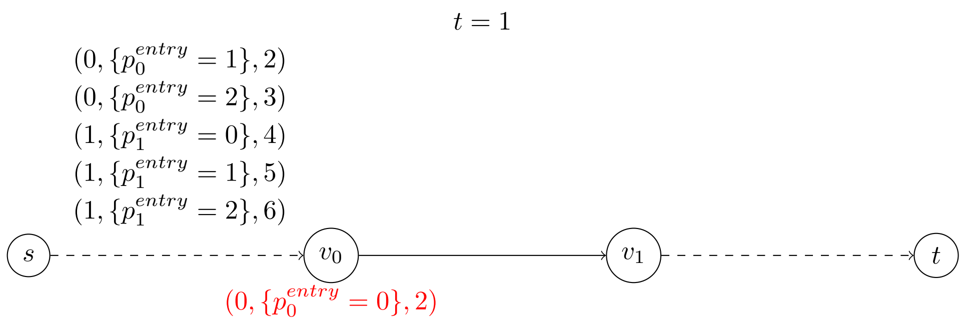
*For this example, we assume all blocking times of relevant infrastructure elements to be **one minute**. We also assume in this example that the train does not go faster on edge or spend less time at a vertex when it is behind schedule. The implementation however implements these considerations.*

To illustrate what a simulation looks like in essence, we will step through the simulation from $t = 0$ to $t = 5$ and illustrate the movement and splitting of train instances.

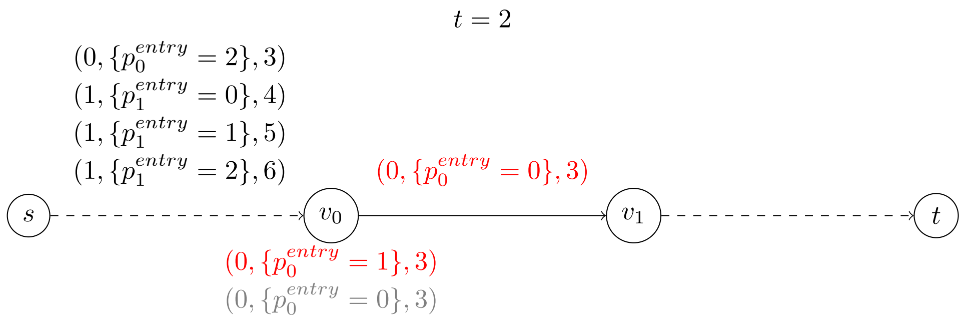
At time $t = 0$, we are just starting and therefore have six train instances, and not two because of the different initial primary delays. We begin with all train instances on the virtual edge (s, v_1).


$t = 0$
$(0, \{p_0^{entry} = 0\}, 1)$
$(0, \{p_0^{entry} = 1\}, 2)$
$(0, \{p_0^{entry} = 2\}, 3)$
$(1, \{p_1^{entry} = 0\}, 4)$
$(1, \{p_1^{entry} = 1\}, 5)$
$(1, \{p_1^{entry} = 2\}, 6)$
s
v_0
v_1
t

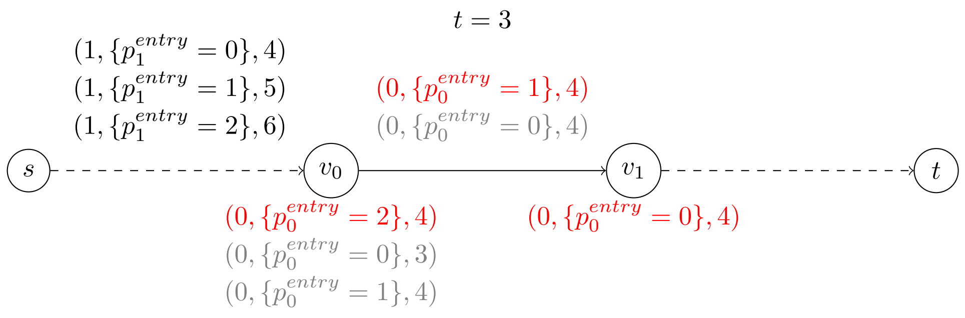
At $t = 1$, only one train instance has an earliest possible departure time earlier or at $t = 1$ and its completed movement with an updated earliest departure time is depicted in red, as shown below. We do not include blocking train instances on virtual edge (s, v_0) as it has an infinite capacity.


$t = 1$
$(0, \{p_0^{entry} = 1\}, 2)$
$(0, \{p_0^{entry} = 2\}, 3)$
$(1, \{p_1^{entry} = 0\}, 4)$
$(1, \{p_1^{entry} = 1\}, 5)$
$(1, \{p_1^{entry} = 2\}, 6)$
s
v_0
v_1
t
$(0, \{p_0^{entry} = 0\}, 2)$

At $t = 2$, we first look for train instances that need to move into vertices as described earlier. $(0, \{p_0^{entry} = 1\}, 2)$ is able to move to v_0 because the train instance $(0, \{p_0^{entry} = 0\}, 2)$ (which has not yet been moved to an edge) has a scenario which is incompatible with $\{p_0^{entry} = 1\}$. Therefore, there are no capacity issues at v_0 for this moving train instance. Afterwards, we move possible train instances into edges and include blocking train instances, depicted in gray. In this example, we move $(0, \{p_0^{entry} = 0\}, 2)$ onto the edge and update its next planned movement, and additionally the corresponding train instance $(0, \{p_0^{entry} = 0\}, 3)$ blocks v_0 up to and including $t = 3$.


$t = 2$
$(0, \{p_0^{entry} = 2\}, 3)$
$(1, \{p_1^{entry} = 0\}, 4)$
$(1, \{p_1^{entry} = 1\}, 5)$
$(1, \{p_1^{entry} = 2\}, 6)$
$(0, \{p_0^{entry} = 0\}, 3)$
s
v_0
v_1
t
$(0, \{p_0^{entry} = 1\}, 3)$
$(0, \{p_0^{entry} = 0\}, 3)$

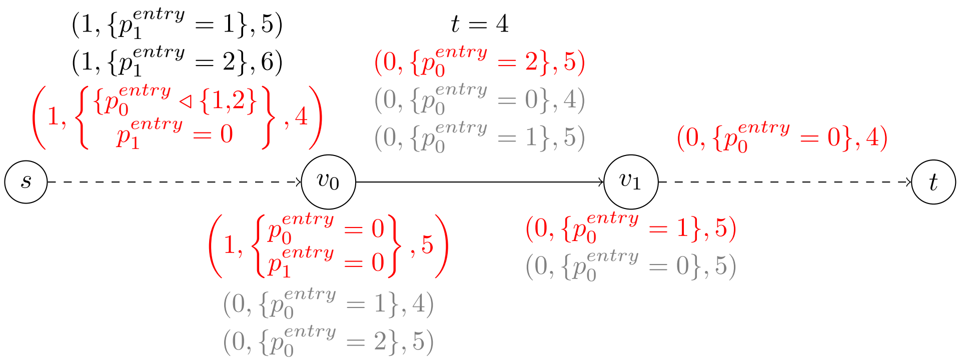
For $t = 3$, we continue to move train instances as shown. First, we consider all train instances that want to move to vertices. They are $(0, \{p_0^{entry} = 2\}, 3)$ and $(0, \{p_0^{entry} = 0\}, 3)$. These train instances successfully move to their respective vertices without capacity issues and their instances are updated appropriately as shown below. Next, we consider all train instances that want to move to edges. $(0, \{p_0^{entry} = 1\}, 3)$ successfully moves to the edge. All completed train instance updates and blockers are shown below:


$t = 3$
$(1, \{p_1^{entry} = 0\}, 4)$
$(1, \{p_1^{entry} = 1\}, 5)$
$(1, \{p_1^{entry} = 2\}, 6)$
$(0, \{p_0^{entry} = 1\}, 4)$
$(0, \{p_0^{entry} = 0\}, 4)$
s
v_0
v_1
t
$(0, \{p_0^{entry} = 2\}, 4)$
$(0, \{p_0^{entry} = 0\}, 3)$
$(0, \{p_0^{entry} = 1\}, 4)$
$(0, \{p_0^{entry} = 0\}, 4)$

In our simulation, it is important to note that we only clear blocking train instances when the block until time is strictly *less than the current time t. Therefore, $(0, \{p_0^{entry} = 0\}, 3)$ is still blocking v_0 until the next time point, $t = 4$, which makes the effective blocking time moreso two minutes instead of the originally prescribed one. This happens because this example works with integers for simplicity. In more realistic simulations with edge speedups, the actual blocking times are close to the user–defined one, as the simulation works with smaller time steps.*

At $t = 4$, we erase all blockers with times that are (strictly) earlier than the current time. We move all possible train instances to vertices, then edges, splitting train instances when needed, and additionally add blockers.

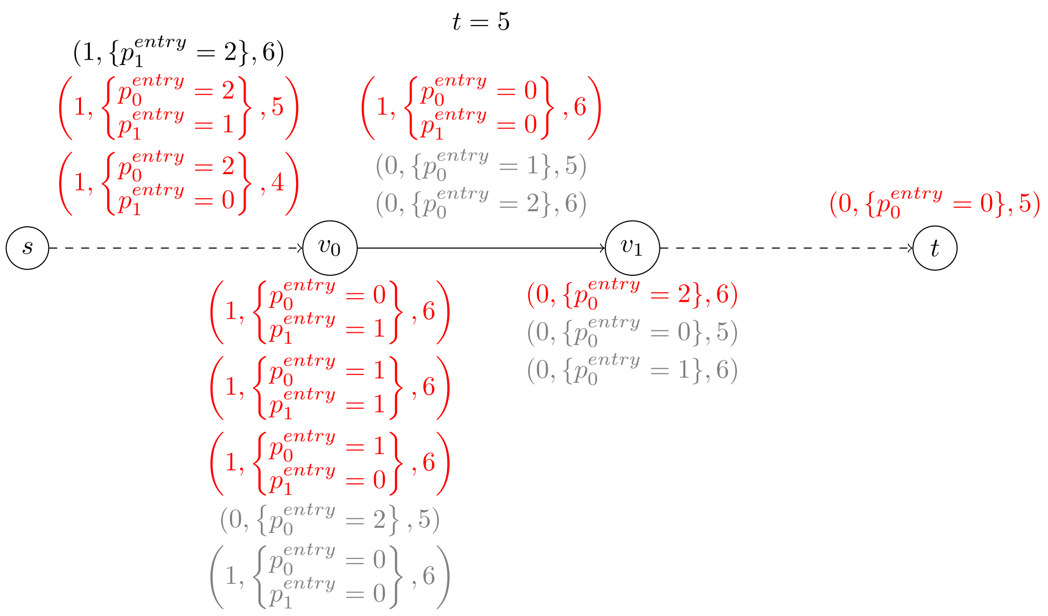
For this time point, let us focus on the train instance $(1, \{p_1^{entry} = 0\}, 4)$, which wants to move into v_0 during the vertices phase. This train instance is compatible with the train instance $(0, \{p_0^{entry} = 2\}, 4)$ and blocking train instance $(0, \{p_0^{entry} = 1\}, 4)$, both at v_0. Only certain scenarios within $\{p_1^{entry} = 0\}$ can move into v_0. Therefore, we split our train instance into two train instances such that one instance that successfully moves into v_0 only contains scenarios that are incompatible with the respective scenarios of either of the occupying or blocking train instances at v_0. We also make sure to leave the other split train instance "behind" at (s, v_0) with a complement scenario as shown below.



$(1, \{p_1^{entry} = 1\}, 5)$
$(1, \{p_1^{entry} = 2\}, 6)$
$\left(1, \left\{\begin{matrix}\{p_0^{entry} \lhd \{1,2\}\} \\ p_1^{entry} = 0\end{matrix}\right\}, 4\right)$
$t = 4$
$(0, \{p_0^{entry} = 2\}, 5)$
$(0, \{p_0^{entry} = 0\}, 4)$
$(0, \{p_0^{entry} = 1\}, 5)$
$(0, \{p_0^{entry} = 0\}, 4)$
s
v_0
v_1
t
$\left(1, \left\{\begin{matrix}p_0^{entry} = 0 \\ p_1^{entry} = 0\end{matrix}\right\}, 5\right)$
$(0, \{p_0^{entry} = 1\}, 4)$
$(0, \{p_0^{entry} = 2\}, 5)$
$(0, \{p_0^{entry} = 1\}, 5)$
$(0, \{p_0^{entry} = 0\}, 5)$

At $t = 5$, we continue moving train instances and splitting them when needed. In particular, one train instance which was previously "held back" at (s, v_0) is split again into two instances. One of them becomes $(1, \{p_0^{entry} = 1, p_1^{entry} = 0\}, 6)$, which has a complete scenario that allows it to move to v_0, while the other becomes $(1, \{p_0^{entry} = 2, p_1^{entry} = 0\}, 4)$, which stays behind due to the blocker $(0, \{p_0^{entry} = 2\}, 5)$ at v_0.

In addition, the train instance $(1, \{p_1^{entry} = 1\}, 5)$ gets split into multiple train instances also due to the blocker $(0, \{p_0^{entry} = 2\}, 5)$.



$t = 5$
$(1, \{p_1^{entry} = 2\}, 6)$
$\left(1, \begin{Bmatrix} p_0^{entry} = 2 \\ p_1^{entry} = 1 \end{Bmatrix}, 5\right)$
$\left(1, \begin{Bmatrix} p_0^{entry} = 2 \\ p_1^{entry} = 0 \end{Bmatrix}, 4\right)$
$\left(1, \begin{Bmatrix} p_0^{entry} = 0 \\ p_1^{entry} = 0 \end{Bmatrix}, 6\right)$
$(0, \{p_0^{entry} = 1\}, 5)$
$(0, \{p_0^{entry} = 2\}, 6)$
$(0, \{p_0^{entry} = 0\}, 5)$
s
v_0
v_1
t
$\left(1, \begin{Bmatrix} p_0^{entry} = 0 \\ p_1^{entry} = 1 \end{Bmatrix}, 6\right)$
$\left(1, \begin{Bmatrix} p_0^{entry} = 1 \\ p_1^{entry} = 1 \end{Bmatrix}, 6\right)$
$\left(1, \begin{Bmatrix} p_0^{entry} = 1 \\ p_1^{entry} = 0 \end{Bmatrix}, 6\right)$
$(0, \{p_0^{entry} = 2\}, 5)$
$\left(1, \begin{Bmatrix} p_0^{entry} = 0 \\ p_1^{entry} = 0 \end{Bmatrix}, 6\right)$
$(0, \{p_0^{entry} = 2\}, 6)$
$(0, \{p_0^{entry} = 0\}, 5)$
$(0, \{p_0^{entry} = 1\}, 6)$

Chapter 3

Scorekeeping

3.1 Problem Definition

Our simulation computes when each train will arrive at its target with some probability, given the primary delays. This occurs because each train instance comes with a scenario from which we can compute such probability. However, it does not provide a formal explanation for any of the propagated secondary delays. Furthermore, our simulation assumes that the given timetable is executable in absence of delay, so if z_i cannot move to some infrastructure element, then some train, itself or otherwise, was delayed at some point and its delay is causing capacity issues.

One way to determine why a train is late is to come up with some systematic way of keeping track of which train instances are causing secondary delays for other train instances *while the simulation is running.* When a train instance is unable to proceed to its next infrastructure element at its planned time, we consider placing "blame" on this train itself or on other trains occupying or blocking the next infrastructure element. Such blame would be stored within a train instance as an integer or real value.

3.2 First Attempt: Blaming Late Trains

Suppose for some fixed scenario, a train instance r_i wants to move to some infrastructure element x, but x is currently full and occupied by other trains. If r_i itself is already late, we increment its "score" to signify the idea that it was unable to proceed further because it already is at "fault" for being late in the first place. The reasoning is, if the train was not late in the first place, perhaps it would have had the chance to move to its next infrastructure element.

The initial attempt expands our definition of a train instance as follows.

Definition 3.2.1. *A* scored train instance *is a tuple $r = (i, s, t, \alpha)$, where i is the train identity, s is the scenario, t is the time point of the train z_i's next planned movement, and $\alpha \in \mathbb{N}$ is a* score *representing its level of "blame" for being unable to proceed while also being behind schedule.*

Algorithm 6 Updating a train instance's position and blaming late trains

```
1: procedure LATE TRAIN UPDATE($t \in \mathbb{T}$, $x \in V \cup E$, $r = (i, S, t^*, \alpha) \in [1..n] \times \mathbb{S} \times \mathbb{T} \times \mathbb{N}$)
2:     let set $\mathcal{S} \leftarrow \emptyset$;
3:     if $c(x) = \infty \vee |\{j|(j,\cdot,\cdot) \in occupy[x] \cup block[x] \cup req[x]\}| < c(x)$ then $\mathcal{S} \leftarrow \{S\}$
4:     else $\mathcal{S} \leftarrow$ AVAILABLE($x$,$r$);
5:     for each $S' \in \mathcal{S}$ do
6:         for each $t' \in \mathcal{D}(p_i^x)$ do
7:             $occupy[x] \leftarrow occupy[x] \cup \{(i, S' \Delta \{p_i^x = t'\}, t + t' + t'')\}$;
8:             $times \leftarrow times \cup \{t + t' + t''\}$;          // $t''$ is waiting/driving time
9:         $block[pre(i,x)] \leftarrow block[pre(i,x)] \cup \{(i, S', t + \delta)\}$; $times \leftarrow times \cup \{t + \delta\}$;
10:    $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \setminus \{(i, S, t^*)\}$;
11:    for each $S' \in$ SCENARIODIFF($S, \mathcal{S}$) do
12:        $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \cup \{(i, S', t^*, \alpha + 1)\}$;
```

Algorithm Modifications:

In Algorithms 2, 3, and 4 described in Section 2.2.2 as well as other subroutines mentioned in [7], we modify all train instances of the original format $r = (i, S, t')$ to $r = (i, S, t', \alpha)$ where α is its current score. The modification of the train instance in Algorithm 1 is also quite simple, where we initialize the first train instances in Line 10 as $(i, \{p_i^{entry} = t\}, a_i^1 + t, 0)$ to start off with a zero score. Where most of the score changes happen in this approach is shown in Algorithm 6. The scoring happens whenever "our" train r under certain scenarios is unable to move to x and its planned arrival into x is at or before the current time. Algorithm 6 is only called when $t^* \leq t$, and it also holds that z_i's planned arrival time at x, a_i^k for some k must be at or earlier than t^*, because we never move trains ahead of schedule. Therefore, we can conclude that all of the train instances held back in line 12 of Algorithm 6 have become late train instances.

Problems with this approach:

This approach comes with multiple issues which could be fixed or adapted in the future. First, this scoring is inaccurate when values in *times* are not just integers, which is almost always the case. *times* does not contain time points in equal intervals, making a train instance's score not proportional to how late our train actually is. This issue can be solved by updating α by the next time interval and increasing α by a value equivalent to the time interval passed. We later used this idea in the following approach. Another main issue of this approach is that the amount of "blame" a train picks up appears to be dependent on how much longer the train still needs to travel. For example, a train that is late in the beginning of a very long ride may have been delayed by another train at the very beginning. Then, assuming that speedups on edges and shortened wait times on vertices are negligible, our train tends to pick up more blame simply for having a lot of traveling left and the other train does not get any blame for causing the problem. Furthermore, a third issue with this approach is that we do not really store interactions on trains being in the way of other trains, as we only blame the late train every time. The added value of this approach is also lacking as there should not be a difference between this scoring scheme's expected score over all train instances of some z_i versus simply calculating the expected delay of z_i at the end of the simulation. Nonetheless, this approach provided ideas which will be described next.

3.3 Expected First-Order Secondary Delay

After a first attempt, we decided to modify the scored train instance such that it stored all direct "interactions" it had with other trains. Our goal here is to examine the first level of causes of secondary delays, that is, when one train instance is occupying or blocking the infrastructure element that another train wants to move to.

3.3.1 Definitions

Definition 3.3.1. *For some time interval $[t_k, t_{k+1}] \in \mathbb{T}$ ($t_k, t_{k+1} \in times$), we define the* expected first-order delay r_j has on all train instances of z_i *as*

$$\beta^x_{r_j \to i, t_k} := \sum_{(i,S',\cdot,\cdot) \in remain(x,i)} \mathbb{P}(S \bigwedge S')(t_{k+1} - t_k). \tag{3.1}$$

We define remain(x,i) as the set of all train instances of z_i that were unable to move to x at time t_k (see Algorithm 5, line 14) until some later determined t_{k+1} and r_j has occupied or blocked x at time t_k.

Note that *times* gets updated regularly as changes in other train instances occur in Algorithm 5, and therefore, at t_k for some k, we compute $\beta^x_{r_j \to i, t_k}$ before finding out what the following t_{k+1} is. It is also important to make note that r_j is one *train instance* of z_j, not z_j itself. Considering only the intersection, $S \bigwedge S'$ allows us to accurately only blame the relevant subscenarios of r_j that do indeed prevent any train instances of z_i from proceeding.

Definition 3.3.2. *For the entire duration of the simulation in, $\mathbb{T}$, we define the* expected first order delay r_j has on all train instances of z_i *as*

$$\beta^x_{r_j \to i} := \sum_{k=0}^{|times|-1} \beta^x_{r_j \to i, t_k}. \tag{3.2}$$

At the end of the simulation, $\beta^x_{r_j \to i}$ indicates the expected amount of time (typically in minutes) in which r_j directly prevents all train instances of z_i from moving to x under a specific scenario associated with r_j.

Using the aforementioned definitions, we update this expanded definition of a train instance as follows:

Definition 3.3.3. *A* scored train instance *is a tuple $r_j = (j, S, t', \mathbf{B})$, where j is the train identity, s is the scenario, t' is the time point of the train z_j's next planned movement, and $\mathbf{B}$ tracks a set of scores of when r_j occupies some infrastructure element x which is currently at full capacity, and prevents some train instance of z_i from moving to x, for all $i, j \in [1 \dots n]$.*

We define $\mathbf{B}$ as

$$\mathbf{B} = \{(\beta^x_{r_j \to i,t}, \beta^x_{r_j \to i}) \in (\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}) \mid i \in [1..n], x \in (V \cup E), t \in \mathit{times}\}$$

where $t \in \mathit{times}$ is the current time.
We define the set of all possible set of score combinations as $\mathcal{B}$.

In the implementation, we store **B** inside the scenario datastructure of the responsible train.

3.3.2 Algorithm Modifications

As in Section 3.2, we made simple modifications to all train instances but from $r = (j, S, t')$ to $r = (j, S, t', \mathbf{B})$ instead. For each $j \in [1 \ldots n]$, we also initialize all pairs $(\beta^x_{r_j \to i,t_k}, \beta^x_{r_j \to i})$ in **B** as (0,0), for all $i \in [1 \ldots n]$.

The major change is with Algorithm 2 and Algorithm 5, and we also created an additional helper algorithm to update our scorekeeping.

For each time point, t_k and for each train instance, we update the first-order expected delay value of all responsible train instances that were occupying or blocking some x in line 12. This leads us to Algorithm 9. Lines 2-4 update parts of the momentarily incomplete scores of occupiers that are in the way of movement, in accordance to Definition 3.3.1. In lines 6-9, we do something similar with the blockers with an extra step. Because the implementation stores blockers and occupiers differently (although they are both train instances symbolically), it means that a blocker at x could correspond to more than one occupier at other infrastructure elements. This occurs with one occupier with scenario S corresponding to a blocker with the same S ends up splitting into more refined scenarios. Therefore, we collect all of such occupiers in line 7 and update scores accordingly.

Finally, we update the total first-order delay of each train instance collected so far in Algorithm 7. First, we complete the calculation of $\beta^x_{r_j \to i,t_k}$ for interval $[t_k, t_{k+1}]$ now that we know what t_{k+1} is in lines 8 and 9. Then, we update $\beta^x_{r_j \to i}$ in accordance with Definition 3.3.2 and then reset $\beta^x_{r_j \to i,t}$ to zero in line 10. In some following time interval $[t_{k+1}, t_{k+2}]$ for a particular train instance, we will again calculate $\beta^x_{r_j \to i,t}$ for $t = t_{k+1}$ and so on.

3.3.3 Verification

Because Symbolic Simulation is a novel approach to railway simulation, verification is considerably difficult. For this approach, even a very small dataset with two vertices, two trains, a short blocking time, and a small number of initial delays already yields a large output detailing all of the train instances, not to mention the many train instances during each time step. At the moment, the only way to verify approaches is to print out the intermediate and final train instances and manually check them. On a very small self-created dataset, we found our scoring to be accurate. However, we overlooked one thing because Symbolic Simulation applied on this particular dataset did not split a train instance after its relevant scores were updated to be nonzero. More concretely, if some train instance r_j is expected to have a first-order delay on some z_i with some nonzero $\beta^x_{r_j \to i}$, then if it splits in Algorithm 4 due to a primary delay or being held back at another train, the scenarios split, but the scores are duplicated. Furthermore, the expected first-order delay value does not store which

Algorithm 7 Modified Symbolic Simulation

```
1: procedure ScoredSimulate( )
2:     Initialize(); let time point $t \leftarrow t_{min}$;
3:     while $t \leq t_{max} \wedge times \neq \emptyset$ do
4:         $t \leftarrow times$.getSmallest(); $times \leftarrow times \setminus \{t\}$;
5:         for each $x \in V \cup E$ do
6:             for each $(\cdot, \cdot, \cdot, \mathbf{B}) \in occupy[x]$ do
7:                 for each $(\beta^x_{r_j \to i,t}, \beta^x_{r_j \to i}) \in \mathbf{B}$ do
8:                     $t_{k+1} = t$;
9:                     $\beta^x_{r_j \to i} \leftarrow \beta^x_{r_j \to i} + (t_{k+1} - t_k)\beta^x_{r_j \to i,t}$;
10:                    $\beta^x_{r_j \to i,t} \leftarrow 0$;
11:        for each $x \in V \cup E$ do                          // first vertices then edges
12:            Requests($t$, $x$);                                      // update $req[x]$
13:            Occupation($t$, $x$);                                    // update $cap[x]$
14:            while $req[x] \neq \emptyset$ do        // requests have to be sorted (highest priority first)
15:                $r \leftarrow req[x]$.pop(); ScoredUpdate($t$, $x$, $r$);   // update occupy and block
```

specific delays in the scenario of r_j are responsible so we cannot proportion the scores between the scenarios as well.

A solution to this is to globally store scores pertaining to expected first-order delay of z_j on z_i instead. Expanding the train instance is no longer necessary and storage is much more efficient. The original reason for calculating r_j on z_i was in hopes of a more granular analysis that may aid in computing delay propagation chains. Another possible solution that was tried previously was to split every blamed train instance in Algorithm 9. However, this approach creates *many* additional train instances and slows down the simulation even on small datasets.

3.3.4 Discussion

An interesting question that should also be discussed is, “why score all occupying and blocking train instances that are present at x when we should only score those that are running late?” This was considered, but also placing blame on trains that are on-time would allow us to consider problems with the set timetable in which a train that itself is on time happens to get in the way of other trains when some other delays propagate. Perhaps delaying an on time train will be better if it could overall reduce secondary delays of more trains overall (weighted probabilistically of course).

If the aforementioned issues are fixed, the calculated first-order secondary delays are still stochastically dependent on each other. If we implemented proposed modified approach of computing expected first-order delay between two trains in general, we can also miss genuine root causes of secondary delays. This situation easily occurs if z_k's path on the infrastructure only intersects the paths of z_j and z_i at infrastructure element x, and z_i follows z_j on similar paths. If z_k holds back z_j for a relatively brief amount of time at x, then z_j will later impede z_i frequently and pick up higher scores, even if z_k is the actual root cause of such secondary delays. Because of these issues, we proceeded to try a different approach instead of working on this approach further.

Algorithm 8 Updating a train instance's position with scoring

```
1: procedure ScoredUpdate($t \in \mathbb{T}$, $x \in V \cup E$, $r = (i, S, t^*, \mathbf{B}) \in [1..n] \times \mathbb{S} \times \mathbb{T} \times \mathcal{B}$)
2:     let set $\mathcal{S} \leftarrow \emptyset$;
3:     if $c(x) = \infty \vee |\{k' | (k', \cdot, \cdot) \in occupy[x] \cup block[x] \cup req[x]\}| < c(x)$ then $\mathcal{S} \leftarrow \{S\}$
4:     else $\mathcal{S} \leftarrow$ Available($x$,$r$);
5:     for each $S' \in \mathcal{S}$ do
6:         for each $t' \in \mathcal{D}(p_i^x)$ do
7:             $occupy[x] \leftarrow occupy[x] \cup \{(i, S' \triangle \{p_i^x = t'\}, t + t' + t'')\}$;
8:             $times \leftarrow times \cup \{t + t' + t''\}$;                // $t''$ is waiting/driving time
9:         $block[pre(i,x)] \leftarrow block[pre(i,x)] \cup \{(i, S', t + \delta)\}$; $times \leftarrow times \cup \{t + \delta\}$;
10:    $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \setminus \{(i, S, t^*)\}$;
11:    for each $S' \in$ ScenarioDiff($S, \mathcal{S}$) do
12:        BlameOcc($t, S', i, x \in V \cup E$);
13:        $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \cup \{(i, S', t^*)\}$;
```

Algorithm 9 Updating the relevant scores in occupiers and blockers.

```
1: procedure BlameOcc($t_k \in \mathbb{T}, S' \in \mathcal{S}, i \in [1..n], x \in V \cup E$)
2:     for each $(j, S_j, t_k, \mathbf{B}_j) \in occupy[x]$ do
3:         if $S_j$ and $S'$ are compatible then
4:             $(t_{k+1} - t_k)\beta^x_{r_j \to i,t} \leftarrow (t_{k+1} - t_k)(\beta^x_{r_j \to i,t} + \mathbb{P}(S_j \triangle S')) \mid \beta^x_{r_j \to i,t} \in \mathbf{B}_j$;// $t_{k+1}$ is not determined until the next time step
5:     for each $(j, S', t_k, \mathbf{B}_j) \in block[x]$ do
6:         if $S_j$ and $S'$ are compatible then
7:             $occs \leftarrow \{(j, S'', \cdot, \mathbf{B}'_j) \in occupy[x'] \mid S'' \preceq S', \forall x' \in V \cup E\}$;
8:             for each $(j, S'', \cdot, \mathbf{B}'_j) \in occs$ do
9:                 $(t_{k+1} - t_k)\beta^x_{r_j \to i,t} \leftarrow (t_{k+1} - t_k)(\beta^x_{r_j \to i,t} + \mathbb{P}(S_j \triangle S')) \mid \beta^x_{r_j \to i,t} \in \mathbf{B}'_j$;
                   // $t_{k+1}$ is not determined until the next time step
```

Chapter 4

Pruning

A main goal is to find explanations for why trains are late. Previously, we tried scoring techniques on the entire input of the simulation. Because of the discussed issues with the stochastically dependent nature of the scoring, along with the runtine, our next approach focuses on selecting one late train instance, rerunning faster pruned simulations using only delay values relevant to the selected late train instance, and then experimenting with delaying other relevant trains under modified pruned simulations.

4.1 Examining Late Train Instances

When Symbolic Simulation is completed, we end up with a list of train instances at the target vertex. Now we want a ranked list of train instances, with the topmost train instance being the latest. For each train instance, we have information of the train's actual departure from its final vertex and can compare this value to its expected departure from its final vertex. The difference of these is intuitively the lateness of a train. Each train instance also contains a scenario and therefore a probability. Multiplying the lateness of a train by its probability is therefore the *expected lateness* of a train. That is, a very late train instance but with very low probability should probably be ranked lower and therefore examined with less priority than a reasonably late train instance with significantly higher probability.

More concretely, we define a *expected lateness* of a train instance $r_i = (i, S, t)$ as

$$del_{r_i} := \mathbb{P}(S)(t - d^{end})$$

where t is the actual time r_i arrived at its target, and d^{end} is the time z_i is planned to depart from its last vertex.

Similarly, we can define a general expected lateness for z_i over all of its train instances as

$$del_i := \sum_{r=(i,S,t)\in occupy[target]} \mathbb{P}(S)(t - d^{end})$$

4.2 Pruning to One Train Instance

After selecting a train instance, say, r_i with a high level of expected lateness as described in Section 4.1, we examine this train instance and all relevant train instances that directly or indirectly interact with it, "pruning" away all train instances with unnecessary random inclusions. This allows us to save computation time on delays for certain trains if we already know those trains do not interact with our train z_i (and therefore r_i as well) *directly* or *indirectly.* To figure out which trains do this, we have to first modify the first simulation that runs all scenarios with all their random inclusions.

4.2.1 Bitstrings

To obtain high-level information of how trains interact with other trains, we use bitstrings. We globally store bitstrings b_i for each z_i and update them during our first run of our simulation.

Definition 4.2.1. *For each $i \in [1 \ldots n]$, we define a* bitstring, *$b_i \in \{0,1\}^n$. For $j \in [1 \ldots n]$, let $b_i[j]$ represent the j^{th} digit of b_i, counting from the right-hand side. $b_i[j] = 1$ either when $j = i$, or for $j \neq i$, z_j directly or indirectly blocks z_i from moving to its desired infrastructure element at any point in the simulation. Otherwise, $b_i[j] = 0$.*

b_i tracks all delaying interactions that z_i has with all other trains, over all scenarios. During the simulation, when some train instance of z_i is affected by a train instance of z_j, we update b_i by doing a bitwise OR with b_j. $b_i[i]$ needs to be initialized to 1 in order to allow bitwise OR operations to update.

Example 4.2.1. *Consider a timetable $T = \{z_0, z_1, z_2\}$ with three trains. Before simulation, the respective bitstrings are respectively initialized to $b_0 = 001$, $b_1 = 010$, and $b_2 = 100$.*

Suppose one train instance of z_0 is currently occupying infrastructure element x, which is at full capaicty, at the same time a train instance of z_1 wants to move to x, in some mutual scenario S. Therefore we update just b_1 as shown:

$$b_1 = b_1 | b_0 = 010|001 = 011$$

As we can see, initializing $b_0[0] = 1$ allows us to update b_1.

Later, in some mutual scenario S, a train instance of z_1 occupies some infrastructure element y, y is at full capacity, and a train instance z_2 wants to move to y. We update the bitstrings again as follows:

$$b_2 = b_2 | b_1 = 100|011 = 111$$

$b_2 = 111$ has meaning to us. $b_2[2] = 1$ is simply there to update other bitstrings when needed. In this particular example, $b_2[1] = 1$ means that z_1 directly blocked z_2 and $b_2[0] = 1$ is an example of z_0 indirectly blocking z_2.

Modifying the algorithms is also simple. We initialize each bitstring to be the binary equivalent of 2^i for some i in line 8 of Algorithm 10. This value is equivalent to setting all the bits to zero except for $b_i[i]$. We update the bitstrings in Algorithm 11, at lines 12 and 13.

Algorithm 10 Initialization

```
1: procedure INITIALIZE( )
2:     $V \leftarrow V \cup \{source, target\}$; $c(source) \leftarrow \infty$; $c(target) \leftarrow \infty$; $times \leftarrow \emptyset$;
3:     for each $v \in V$ do
4:         $E \leftarrow E \cup \{(source, v), (v, target)\}$; $c((source, v)) \leftarrow \infty$; $c((v, target)) \leftarrow \infty$;
5:     for each $x \in (V \cup E)$ do $occupy[x] \leftarrow \emptyset$; $block[x] \leftarrow \emptyset$; $req[x] \leftarrow \emptyset$; $cap[x] \leftarrow \emptyset$;

6:     for each $i \in \{1, \dots, n\}$ do
7:         $times \leftarrow times \cup \{a_i^1\}$;
8:         $b_i[i] \leftarrow 2^i_{10}$;                    // $b_i[i] = 1$, 0 for the rest of the bitstring
9:         for each $t \in \mathcal{D}(p_i^{entry})$ do
10:            $times \leftarrow times \cup \{a_i^1 + t\}$;
11:            $occupy[(source, v_i^1)] \leftarrow occupy[(source, v_i^1)] \cup \{(i, \{p_i^{entry} = t\}, a_i^1 + t)\}$;
```

Algorithm 11 Updating a train instance's position

```
1: procedure UPDATE($t \in \mathbb{T}$, $x \in V \cup E$, $r = (i, S, t^*) \in [1..n] \times \mathbb{S} \times \mathbb{T}$)
2:     let set $\mathcal{S} \leftarrow \emptyset$;
3:     if $c(x) = \infty \vee |\{j|(j, \cdot, \cdot) \in occupy[x] \cup block[x] \cup req[x]\}| < c(x)$ then $\mathcal{S} \leftarrow \{S\}$
4:     else $\mathcal{S} \leftarrow$ AVAILABLE($x$,$r$);
5:     for each $S' \in \mathcal{S}$ do
6:         for each $t' \in \mathcal{D}(p_i^x)$ do
7:             $occupy[x] \leftarrow occupy[x] \cup \{(i, S' \triangle \{p_i^x = t'\}, t + t' + t'')\}$;
8:             $times \leftarrow times \cup \{t + t' + t''\}$;          // $t''$ is waiting/driving time
9:         $block[pre(i,x)] \leftarrow block[pre(i,x)] \cup \{(i, S', t + \delta)\}$; $times \leftarrow times \cup \{t + \delta\}$;
10:    $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \setminus \{(i, S, t^*)\}$;
11:    for each $S' \in$ SCENARIODIFF($S, \mathcal{S}$) do
12:        for each $(j, S_j, t_k, \mathbf{B}_j) \in occupy[x] \cup block[x]$ do
13:            $b_i \leftarrow b_i | b_j$;
14:        $occupy[pre(i,x)] \leftarrow occupy[pre(i,x)] \cup \{(i, S', t^*)\}$;
```

4.2.2 Rerunning the simulation on a smaller set

We let the set of all random variables P be the original set of random variables with their support and probability distributions, and modify each $p \in P$ so that its support only includes values that are relevant to some selected late train instance r_i. For many random variables, we also need to modify their probability distributions such that they each add up to 1. Algorithm 12 demonstrates how we prune random variables after our first simulation. In lines 3 - 6, we reduce the finite support of all random variables associated with trains which do not directly or indirectly affect our late train instance r. Since they do not affect r, we can just choose one delay value and set the probability of that delay to 1.

For lines 8 - 11, we consider all p_j^x for z_j that affect r. In line 8, if the random inclusion of p_j^x in r's scenario is a strict subset of the original finite support $\mathcal{D}(p_j^x)$, then we update our finite support as so in line 11. In line 10, we update the probability of

Algorithm 12 Modifying Finite Support and Probabilities

```
procedure MODIFY SUPPORT($r = (i, S, t) \in [1..n] \times \mathbb{S} \times \mathbb{T}$)
    for each $p_j^x \in P$ do
        if $b_i[j] = 0$ then
            choose $\Delta \in \mathcal{D}(p_j^x)$      // Any value of $\Delta$ allowed, but typically $\Delta = 0$.
            $\mathcal{D}(p_j^x) \leftarrow \{\Delta\}$;
            $\mathbb{P}(p_j^x = \Delta) \leftarrow 1$;
        else
            if $S(p_j^x) \subset \mathcal{D}(p_j^x)$ then
                for each $\Delta \in S(p_j^x)$ do
                    $\mathbb{P}(p_j^x = \Delta) \leftarrow \mathbb{P}(p_j^x = \Delta)/\mathbb{P}(p_j^x \triangleleft S(p_j^x))$;      // Conditional probability
                $\mathcal{D}(p_j^x) \leftarrow S(p_j^x)$;      // Update support to have only relevant values
```

each support value, using the conditional probability of $\mathbb{P}(p_j^x = \Delta | p_j^x \triangleleft S(p_j^x))$. In some cases where z_j (indirectly) affects z_i for $i \neq j$, the scenario of r does not explicitly specify any refined random inclusion for p_j^x. Therefore, we must keep the original support of p_j^x with its respective probabilities. Algorithm 11 does this implicitly by not changing $\mathcal{D}(p_j^x)$.

After pruning our random variables, we can make one run of our pruned simulation by running Algorithm 2. This pruned simulation is unsurprisingly much faster than our original simulation since fewer delay values are used, and therefore we compute fewer train instances.

4.3 Evaluating Delay Changes

Now that we have run one pruned simulation and determined the relevant trains and delay values, we can try to answer the following question: if we increase the delays of another train, does it reduce the expected lateness of our late train? Furthermore, how does the sum of expected lateness over all trains change? Does it increase greatly or drop?

4.3.1 Statistics for Assessing Improvement

After completing one run of the pruned simulation, we must recompute del_{r_i} *under* the pruned simulation, to use as an updated statistical baseline of comparison when we experiment with different slight modifications of the pruned simulation. Additionally, we need to compute an overall baseline statistic over all trains. We define the total expected lateness of all trains in the (pruned) simulation as *total expected lateness*, as

$$del_{total} := \sum_{j \in [1 \ldots n]} del_j$$

4.3.2 Experiments

We have six anonymized datasets based off of real-world data from the DB Netz AG. Due to the currently limited scalability of Symbolic Simulation, running even the base simulation with only initial delays on a personal machine is only feasible with the two smallest datasets. One of these two datasets is over a one hour time period from 0:00 to 1:00 and is a subset of the other, which is from 0:00 to 2:00. For our simulation, we used the one hour dataset. The infrastructure of this dataset contains 2646 vertices and 5622 edges, and there are 78 trains in the timetable. We ran experiments on a computer with a 1.80 GHz $\times$ 8 Intel Core i7 CPU and 8 GB of RAM.

For primary delays, we only input entry delays with discrete distributions shown in Table 4.1. We also used a uniform block time of 2 minutes.

We experimented with primary delay increases of other trains by running many modified versions of our pruned simulation, with the only changes in each simulation being the selected random variable with increased primary delay, and the value in which this random variable's support is increased by. In our test, we chose the following set of increase values to experiment with: $\mathcal{I} = \{1, 2, 3, 4, 5, 10, 15, 20\}$. We selected many small values such as $\{1, 2, 3, 4, 5\}$ to have a more granular analysis of small changes, and only selected a smaller variety of larger delay values with the hypothesis that too large of an increased delay would increase the total expected lateness anyway.

More concretely, for each pruned simulation in which we wish to optimize del_{r_i}, we select one p_j^x where $b_i[j] = 1$, and increase all values of $\mathcal{D}(p_j^x)$ by a certain amount, and then run this one modified iteration of the pruned simulation. We then recompute del_{r_i} and del_{total} in this single iteration and record it. This brute force experimentation continues for all other values in $\mathcal{I}$, and furthermore with all other random variables that affect our chosen late train instance.

We repeated this process described above on other late train instances output in the base simulation.

Train Type	Support	Respective Probabilities
FRz	(0, 1, 4, 10)	(0.5, 0.17, 0.18, 0.15)
NRz	(0, 1, 4)	(0.5, 0.32, 0.18)
FGz	(0, 10, 30)	(0.6, 0.2, 0.2)
NGz	(0, 10, 30)	(0.5, 0.25, 0.25)
Gz	(0, 10, 30)	(0.5, 0.25, 0.25)
S	(0, 1, 4)	(0.8, 0.16, 0.04)
Lz	(0, 10, 30)	(0.5, 0.25, 0.25)
GzG	(0, 10, 30)	(0.5, 0.25, 0.25)

Table 4.1: Input entry delays

4.3.3 Results

Table 4.2 shows the results of twenty-five experiments. After the base simulation, we have a list of late train instances in descending order. For our results, we furthermore remove late train instances that are late only due to its own primary delay. We do this by removing all train instances of trains that do not interact with any other trains (based on its bitstring). After removing these train instances from the list, we then selected twenty-five train instances with the highest amount of expected lateness. For each train instance r_i, we ran a pruned simulation, and then proceeded to iteratively rerun the pruned simulation with different increases on different random variables that have impact on r_i. The classification of improvement in Table 4.2 is rather subjective and is detailed as follows.

Most of our experiments did not find any improvement ("None"), that is, all possible input support value changes reduced neither the expected lateness of the late train instance, nor the expected lateness of all train instances under the pruned simulations.

Two experiments found changes that create very "nominal" change. The experiments on z_5 and z_6 at best did not find any decrease in their respective expected lateness but respectively decreased the total expected lateness by about four and two minutes overall.

Four experiments found arguably "significant" change in which the expected lateness of the late train instance was decreased, and the overall lateness was either decreased (in the case of z_1) or increased by no more than two minutes. What constitutes as "significant" or "nominal" is arbitrary and also depends on how important the late train instance is. If it is important enough, then reducing its expected lateness at the cost of overall expected lateness by a little bit is acceptable. On larger datasets with more primary delays or larger time frames, we expect that we can find more significant reductions that greatly reduce the expected lateness of both the train instance and over all train instances.

Table 4.3 shows an example of how increasing the entry delay of z_1 by 2 minutes reduces the expected lateness of r_1 by about one minute and reduces the total expected lateness of the pruned simulation by about five minutes.

Table 4.4 shows an example of how increasing the entry delay of other trains by any value in $\mathcal{I}$ neither significantly reduces the expected lateness of r_{49} nor the total expected lateness of the pruned simulation.

Table 4.5 shows an example of how at best (using only the set $\mathcal{I}$), increasing the entry delay of p_3^{entry} by two minutes does not noticeably decrease the expected lateness of r_5 but decreases the total expected lateness of the pruned simulation by about four minutes. In this type of situation, it is up to the user to decide whether to improve the timetable based on this result.

z_i	S	Original del_{r_i}	Improve-ment
z_{49}	$\{p_{49} = 30\}$	7.41	None
z_{37}	$\{p_{37} = 30\}$	7.29	None
z_{53}	$\{p_{53} = 30\}$	6.31	None
z_{70}	$\{p_{70} = 30\}$	6.10	None
z_5	$\{p_5 = 30\}$	5.95	Nominal
z_6	$\{p_6 = 30\}$	5.95	Nominal
z_{46}	$\{p_{46} = 30\}$	5.93	None
z_{73}	$\{p_{73} = 30\}$	5.93	None
z_4	$\{p_4 = 30\}$	5.90	Significant
z_{36}	$\{p_{36} = 30\}$	5.89	None
z_{40}	$\{p_{40} = 30\}$	5.88	None
z_2	$\{p_2 = 30, p_3 \triangleleft \{0, 30\}\}$	4.75	Significant
z_{74}	$\{p_{74} = 30, p_{75} \triangleleft \{0, 30\}\}$	4.75	None
z_1	$\{p_1 = 30, p_3 \triangleleft \{0, 10\}\}$	4.75	Significant
z_{54}	$\{p_{48} \triangleleft \{0, 30\}, p_{54} = 30\}$	4.65	None
z_{43}	$\{p_{32} = 0, p_{43} = 30\}$	4.02	None
z_{48}	$\{p_{36} = 0, p_{48} = 30\}$	3.54	None
z_{48}	$\{p_{36} \triangleleft \{10, 30\}, p_{48} = 30\}$	3.54	None
z_{44}	$\{p_{37} \triangleleft \{0, 10\}, p_{44} = 30, p_{46} \triangleleft \{0, 30\}\}$	3.49	Significant
z_{43}	$\{p_{32} = \triangleleft\{0, 30\}, p_{43} = 30\}$	2.28	None
z_{37}	$\{p_{37} = 10\}$	2.28	None
z_{48}	$\{p_{48} = 10\}$	2.10	None
z_2	$\{p_{10} = 30\}$	1.92	None
z_{76}	$\{p_{73} \triangleleft \{0, 10\}, p_{74} \triangleleft \{10, 30\}, p_{76} = 30\}$	1.90	None
z_{44}	$\{p_{44} = 10\}$	1.83	None
z_{70}	$\{p_{70} = 10\}$	1.71	None

Table 4.2: General results of experiments on twenty-five pruned simulations, each based off of a train instance in the original simulation with high expected lateness. All random variables listed above are assumed to be entry primary delays.

Selected p_j^{entry}	p_j^{entry} Increase	New del_{r_i}	New del_{total}
$p_3 \lhd \{0, 10\}$	20	1.13	82.82
$p_6 \lhd \{0, 10, 30\}$	2	3.78	60.34
$p_6 \lhd \{0, 10, 30\}$	3	3.79	63.27
$p_6 \lhd \{0, 10, 30\}$	4	3.81	64.41
$p_4 \lhd \{0, 10, 30\}$	10	3.81	70.47
$p_4 \lhd \{0, 10, 30\}$	15	4.57	80.04
$p_4 \lhd \{0, 10, 30\}$	5	4.67	71.19
$p_3 \lhd \{0, 10\}$	10	4.73	74.78
$p_2 \lhd \{0, 10, 30\}$	15	4.73	81.93
$p_2 \lhd \{0, 10, 30\}$	20	4.73	85.70
$p_3 \lhd \{0, 10\}$	3	4.73	68.77
$p_5 \lhd \{0, 10, 30\}$	20	4.73	84.61
$p_5 \lhd \{0, 10, 30\}$	15	4.73	79.55
$p_5 \lhd \{0, 10, 30\}$	10	4.73	76.35
$p_2 \lhd \{0, 10, 30\}$	2	4.73	67.84
$p_2 \lhd \{0, 10, 30\}$	5	4.73	70.77
$p_3 \lhd \{0, 10\}$	5	4.73	71.17
$p_3 \lhd \{0, 10\}$	1	4.73	66.59
$p_3 \lhd \{0, 10\}$	4	4.73	69.97
$p_6 \lhd \{0, 10, 30\}$	5	4.73	72.99
...	...	...	...

Table 4.3: Example of a "significant" result in which there was a reduction of delay found on r_1 with $S = \{p_1 = 30, p_3 \lhd \{0, 10\}\}$, and overall, when introducing an increase of primary delays for p_6^{entry} by 2 minutes. Under the pruned simulation, r_1 had an expected delay of 4.73 minutes and the total expected delay was 65.43 minutes. All random variables listed above are assumed to be entry primary delays.

Selected p_j^{entry}	p_j^{entry} Increase	New del_{r_i}	New del_{total}
$p_{48} \lhd \{0, 10, 30\}$	2	7.39	43.03
$p_{48} \lhd \{0, 10, 30\}$	4	7.40	44.17
$p_{48} \lhd \{0, 10, 30\}$	3	7.40	43.63
$p_{48} \lhd \{0, 10, 30\}$	5	7.40	44.93
$p_{48} \lhd \{0, 10, 30\}$	1	7.41	42.57
$p_{48} \lhd \{0, 10, 30\}$	15	7.43	53.77
$p_{48} \lhd \{0, 10, 30\}$	10	7.51	49.66
$p_{48} \lhd \{0, 10, 30\}$	20	7.63	62.57

Table 4.4: Example of no significant reduction of delay on r_{49} and overall when introducing any of the possible increases on the other primary delays. Under the pruned simulation, r_{49} had an expected delay of 7.41 minutes and the total expected delay was 40.47 minutes. All random variables listed above are assumed to be entry primary delays.

Selected p_j^{entry}	p_j^{entry} Increase	New del_{r_i}	New del_{total}
$p_6 \lhd \{0, 10, 30\}$	20	5.95	90.81
$p_6 \lhd \{0, 10, 30\}$	10	5.95	80.39
$p_1 \lhd \{0, 10, 30\}$	5	5.95	76.14
$p_3 \lhd \{0, 10, 30\}$	15	5.95	84.86
$p_1 \lhd \{0, 10, 30\}$	4	5.95	74.50
$p_3 \lhd \{0, 10, 30\}$	5	5.95	74.15
$p_2 \lhd \{0, 10, 30\}$	15	5.96	85.89
$p_1 \lhd \{0, 10, 30\}$	10	5.96	81.70
$p_3 \lhd \{0, 10, 30\}$	10	5.96	80.10
$p_1 \lhd \{0, 10, 30\}$	1	5.96	72.00
$p_3 \lhd \{0, 10, 30\}$	4	5.96	72.65
$p_3 \lhd \{0, 10, 30\}$	3	5.96	72.90
$p_3 \lhd \{0, 10, 30\}$	2	5.96	67.14
...	...	...	...

Table 4.5: Example of a "nominal" result in which there was no noticeable reduction of delay on r_5, but a small decrease in overall delay. Under the pruned simulation, r_5 originally had an expected delay of 5.96 minutes and the total expected delay was 71.27 minutes.

Chapter 5

Conclusion

In this thesis, we explained the relevant notation and algorithms of Symbolic Simulation based off the work of [7]. To aid with understanding, we provided examples, including a walkthrough of a portion of a simplified simulation.

The first approach with scoring consisted of two sub-approaches. The first sub-approach involved scoring or "blaming" late trains, and this approach had an intuitive motivation, but tended to anthropomorphize trains too much by only blaming the late ones, especially when an on-time train could frequently impede other trains at a particular infrastructure element, late or not. Due to the lack of added value of this sub-approach, we did not need to fix the other issues with scoring in relation to the differences between time points. The second sub-approach stored for each train instance, the expected first-order secondary delay it had on each of the other trains. Towards the end of this implementation, we found that the algorithm is not correct because it duplicates existing stored intermediate scores of a train instance when it splits further. We suggested a solution to this by storing the first-order delays globally between trains instead of between a train instance and a train. Because the first-order secondary delays were still stochastically dependent on each other, we proceeded to work on the pruning approach instead of further implementing this approach.

The second approach involved selecting a late train instance, pruning the simulation, and experimenting with increasing the delay values of one related random variable to see if the expected lateness of one train or all trains in a pruned simulation decreased. This approach appeared to correctly prune the necessary random variables and demonstrated opportunities to improve not only the expected lateness of the late train instance, but also the total expected lateness of all trains.

5.1 Future work

Our approach with pruning has the potential of being effective. Further work can experiment with a larger set of primary delay increases and determine a methodical way to minimize secondary delays. In addition, once we find a random variable in which an increase in its primary delays decreases the expected lateness of the late train instance, it is important to rerun this modification on the original simulation (with the only change being the increases in delays for that one random variable). This is necessary because adding a delay may improve the performance of a pruned simulation, but could potentially introduce more interactions between trains in the unpruned simulation. Integrating this approach with a GUI would also make selecting late train instances easier for the user and provide visuals. Another extension is to experiment with modifying the timetable itself as opposed to only the support of random variables.

Bibliography

[1] OpenTrack Railway Technology - Railway Simulation (2022, (accessed February 22, 2022)), http://www.opentrack.ch/opentrack/opentrack_e/opentrack_e.html

[2] LUKS (2022, (accessed February 23, 2022)), https://www.via-con.de/en/development/luks/

[3] OnTime (2022, (accessed February 23, 2022)), https://www.trafit.ch/en/ontime

[4] Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR) Institut für Verkehrsforschung, Deutsches Institut für Wirtschaftsforschung (DIW): Verkehr in Zahlen 2021/2022. Tech. rep., Bundesministerium für Verkehr und digitale Infrastruktur (2021)

[5] Franke, B., Seybold, B., Büker, T., Graffagnino, T., Labermeier, H.: Ontime–network-wide analysis of timetable stability. In: 5th International Seminar on Railway Operations Modelling and Analysis (2013)

[6] Haehn, R., Ábrahám, E., Nießen, N.: Probabilistic simulation of a railway timetable. In: OpenAccess Series in Informatics (OASIcs). vol. 85, pp. 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASICS.ATMOS.2020.16, https://publications.rwth-aachen.de/record/807139

[7] Haehn, R., Ábrahám, E., Nießen, N.: Symbolic simulation of railway timetables under consideration of stochastic dependencies. In: Abate, A., Marin, A. (eds.) Quantitative Evaluation of Systems. p. 257–275. Springer International Publishing (2021)

[8] Nash, A., Huerlimann, D.: Railroad simulation using OpenTrack. Computers in Railways IX p. 10 (2004)

[9] Pagand, I.: Fostering the railway sector through the European Green Deal. Tech. rep., European Union Agency for Railways (July 2020), https://www.era.europa.eu/sites/default/files/events-news/docs/fostering_railway_sector_through_european_green_deal_en.pdf

[10] Radke, S.: Verkehr in Zahlen 2019/2020. Tech. rep., Bundesministerium für Verkehr und digitale Infrastruktur (September 2019)

[11] Radtke, A., Bendfeldt, J.: Handling of railway operation problems with RailSys. In: Proceedings of the 5th World Congress on Rail Research (WCRR 2001), Cologne, Germany (2001)

[12] Schneider, W., Nießen, N., Oetting, A.: MOSES/WiZug: Strategic modelling and simulation tool for rail freight transportation. In: Proceedings of the European Transport Conference, Straßbourg (2003)

[13] Spanninger, T., Trivella, A., Corman, F.: Approaches for real-time train delay prediction. In: 20th Swiss Transport Research Conference (STRC 2020)(virtual). STRC (2020)

[14] Weik, N., Niebel, N., Nießen, N.: Capacity analysis of railway lines in germany–a rigorous discussion of the queueing based approach. Journal of Rail Transport Planning & Management **6**(2), 99–115 (2016)

[15] Zieger, S., Weik, N., Nießen, N.: The influence of buffer time distributions in delay propagation modelling of railway networks. Journal of rail transport planning & management **8**(3-4), 220–232 (2018)