Real Time Large Railway Network Re-Scheduling

Erik Nygren[*], Christian Eichenberger[†], Emma Frejinger[‡]

October 22, 2020

Contents

[*] erik.nygren@sbb.ch
[†] christian.markus.eichenberger@sbb.ch
[‡] emma.frejinger@unmontreal.ca

Abstract

In this paper, we describe our novel approach to tackle the Real-Time Re-Scheduling Problem for Large Railway Networks by a combination of techniques from operations research and machine learning.

The Industry State of the Art is limited to resolve re-scheduling conflicts fully automatically only at narrowly defined geographical regions due to the exponential growth of computational time for combinatorial problems. Our aim is to improve the scalability of the optimization to larger regions by a new two-step approach: we aim to use the generalization capabilities of machine learning algorithms to reformulate any new problem instance to a smaller equivalent instance, which can be solved in a much shorter time span. This allows us to keep the rigor of the OR formulation while relying on compressed knowledge in machine learning algorithms to vastly increasing the size of solvable problem instances. Early investigations support this assumption by showing that large problem instances can be reduced to much smaller core problems even with the use of domain specific heuristics. Instead of decomposing large problems and then iteratively solving the global problem through coupling of subproblems, we aim at re-scoping the problem.

We believe that our approach is even applicable to other transportation and logistic systems outside of railway networks.

The goal of this paper is four-fold:

G0 report the problem scope reduction formulation in a formal way;

G1 motivate the approach empirically;

G2 show the validity of the approach for one specific OR solver;

G3 report our first steps in tackling automated problem scope reduction;

G4 provide an extensible playground open-source implementation `https://github.com/SchweizerischeBundesbahnen/rsp` for further research into automated problem scope reduction.

We hope that this will draw the attention of both academic and industrial researchers to find other and better approaches and collaboration across Railway companies and from different research traditions.

1 Context and Goals

1.1 Real-world Context

Switzerland has one of the world's densest railway networks with both freight and passenger trains running on the same infrastructure. More than 1.2 million people use trains on a daily basis [1]. Besides the publicly available railway schedule, there is also the more detailed operational schedule, which maps specific trains to planned train runs and specifies which railway infrastructure will be utilised by which train at any time.

The operational schedule has to be continually re-computed due to many smaller and larger delays and disturbances that arise during operations. While many of the minor incidents (up to 1 million per day)

TODO Erik needs citation?

can be fully resolved by the rail control system, some larger disturbances require human or more advanced algorithmic interception. To limit the spread of disturbances and minimize the delay within the dynamic railway system, decisions on re-ordering or re-routing trains have to be taken to derive a new feasible operational plan.

The following Figure 1 shows the re-scheduling control loop adapted from [1, 2].

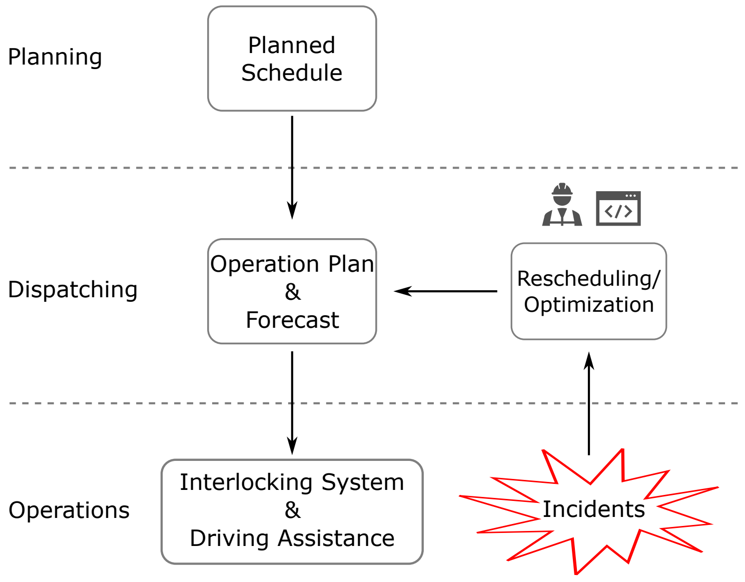
Planning
Planned Schedule
Dispatching
Operation Plan & Forecast
Rescheduling/ Optimization
Operations
Interlocking System & Driving Assistance
Incidents

Figure 1: *Simplified schema of the rescheduling process during operations. There are three main areas involved:* planning, dispatching *and* operation *(interlocking and driving assistance). Dispatchers have access to the current situation and an automated forecast and make decisions on the level of the operational plan, which are then translated either automatically or by dedicated dispatching team members to the interlocking system; some areas are under the control of an automatic optimization system. As time goes on, new elements from the planned schedule will be introduced to the operational schedule and the planned schedule gives also constraints to re-scheduling: passenger trains must not depart earlier than communicated to customers and the service intention may define further hard or soft constraints such as pecuniar penalties for delay or train dropping. The current situation will be reflected in the operational schedule and the forecast.*

The industry state of the art systems efficiently re-compute a traffic forecast for the next two hours, allowing dispatchers to detect potential conflicts ahead of time. The predicted conflicts, mostly have to be resolved by humans by explicitly deciding on re-ordering or re-routing based on their experience.

Today, the whole railway system is decomposed into disjoint geographic cells of responsibility as depicted in Figure 2. This vastly reduces the complexity within each region and allows human dispatchers to resolve conflicts locally with support of different IT systems. Using structured (e.g. IT system of incident messages) or informal ways (e.g. direct talks with fellow dispatchers) to communicate, different regions can coordinate their dispatching strategies to achieve system wide stability.

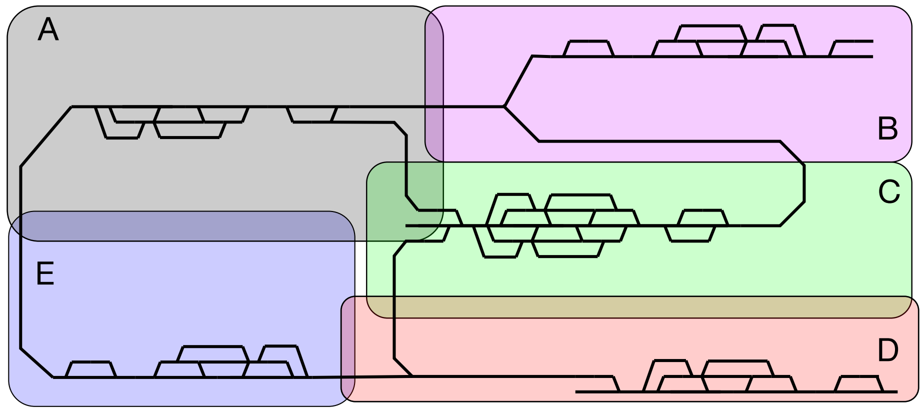

Figure 2: *Due to the vast complexity of a railway network the re-scheduling task is decomposed into smaller geographical areas (A-E), withiwin which human dispatchers optimize traffic flow. Inter-region coordination is mostly done by informal means of communication such as telephone.*

Each region is in itself decomposed according to network properties:

> [...], a network separation approach is applied to divide the railway network into zones of manageable size by taking account of the network properties, distinguishing condensation and compensation zones. Condensation zones are usually situated near main stations, where the track topology is complex and many different routes exist. As such an area is expected to have a high traffic density, it is also a capacity bottleneck and trains are required to travel through with maximum allowed speed and thus without time reserves. Collisions are avoided by exploiting the various routing possibilities in the station area. Conversely, a compensation zone connects two or more condensation zones and consists of a simpler topology and less traffic density. Here, time reserves should be introduced to improve timetable stability. The choice of an appropriate speed profile is the most important degree of freedom to exploit in these compensation

zones. [3]

The complexity of the combinatorial problem of re-scheduling grows exponentially with the number of involved trains. Hence, there are only a few small so-called condensation areas [3] that can be operated through automatic systems. For larger areas, we can't currently find feasible re-scheduling solutions in real-time.

This situation is depicted in a schematic way in Figure 3:

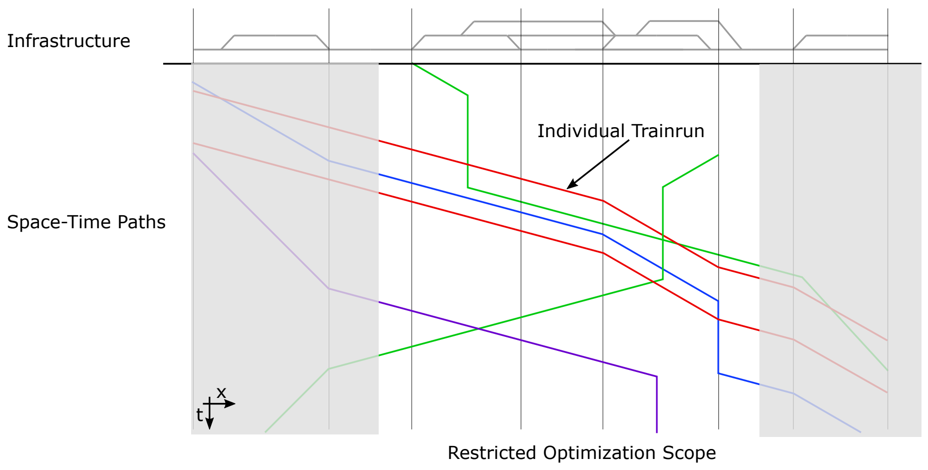

Figure 3: *Real-time rescheduling has so far been applied to restricted geographic areas only. This means that human intervention or buffer times are necessary outside of the optimization scope to guarantee conflict free traffic in the whole network. The limited geographical scope leads to a cost-function that can only locally be minimized and might have negative effects on the whole network scale.*

2 Research Approach

In this work we propose a novel solution approach to overcome the limited spatial restrictions of automated algorithmic re-scheduling by extracting the core problem variables from the original full problem formulation. We propose to utilize the strength of state-of-the-art machine learning algorithms to extract the core problem, which in turn is solved by any known optimizer such as Simplex or ASP. We test our novel research approach through experiments on a simplified railway simulation.

Our playground implementation is an abstraction of real railway traffic (see Section 2.2 below), shown in Figure 1: we only consider an operational plan and a single incident (malfunction) and try to re-schedule such that we have a conflict-free plan again that stays close to the (published) planned schedule. We hint at a fully automatic re-scheduling loop as follows:

1. operational plan is updated according to real time data
2. core problem scope is extracted from operational plan
3. optimizer solves core problem and generate new operational plan
4. repeat.

2.1 Core idea

We assume that any re-scheduling problem that arises within the global railway system can be reduced to a much smaller re-scheduling problem consisting only of a subset of active trains at any given time. In contrast to current models we don't follow a spatial decomposition according to network properties but rather investigate the possibility of predicting the relevant problem scope given a schedule and a malfunction.

In other words, we assume that it is possible to predict the range of influence from a given disturbance and that a feasible solution can be found within that range while keeping the operational schedule fixed outside.

Thus the fundamental assumptions that we want to prove with this research project are:

1. it is possible to predict the affected time-space area, either from the problem structure or from historic data
2. solutions within the restricted problem scope can be found much faster than the full railway system
3. feasible solutions within the restricted problem scope show a similar quality to full network solutions.

To tackle this problem, our approach is to combine Operations Research and Domain-specific Learning to get the best of both worlds: an "Scope predictor" is able to predict the "impact" of a disturbance, with or without a knowledge base learnt by training; we hope the scope-predictor could predict which trains and which departures are or could be affected by the disturbance based on past decisions. This piece of information from the scope-predictor then helps the solver to constrain the search space or at least drive its search more efficiently (driving the branching process).

TODO Erik Add information about what we mean by automated problem scope reduction: Heuristic, historical, structures, ...

2.2 Research Approach: a Synthetic Playground

We now give a short introduction to our playground implementation (G4) and its limitation with respect to real-world features.

Synthetic Infrastructure and Simplified Resource Model We use the FLATland toolbox [4] to generate a grid world infrastructure consisting of 2D square cells; the infrastructure defines the possible movements to the 4 neighbor cells mirroring railway infrastructure (switches etc.)

Synthetic Timetable and Train Dynamics

- Every train has one single source and target, no intermediate stops, as provided by the FLATland toolbox [4].
- Every train has a constant speed (which may be different from train to train), as provided by the FLATland toolbox [4].
- The schedule (departure and passing times at all cells) is generated such that the sum of travel times of all trains is minimized within a chosen upper bound; this the generated timetable might not have a realistic structure.
- There are no time reserves in the schedule, trains cannot catch up.
- There is no distinction between published timetable and operational schedule, we only have an operational timetable; in reality, trains must not depart earlier than published.
- There are no connections or vehicle tours (turnrounds).

Synthetic route alternatives We use shortest paths (in reality, in particular in the case of disturbances affecting a whole area, we might need a different scheme knowing the parts that cannot be taken); path cycles are not allowed.

Simple Disturbance Model We simulate one train a being stopped for d discrete time steps at some time t and call this a malfunction $M = (t, d, a)$; in reality, the delay might not be known or only probabilities can be assumed. In reality, update information also comes in batches and we would need to consider multiple delays in the same update interval. This setup is shown in Figure 4.

2.3 Research Approach: Decomposition in Space and Time

TODO Erik naming (Oracle -¿ scoper), figures

We will now detail the general line of argument of Section 2.1 by introducing two hypotheses

H1 offline scoping: if we have access to an offline solution of the re-scheduling problem given unbounded resources, we can design a scope reduction that allows for a large speed-up (see below, Section 2.3.1); by this, we achieve goal **G2**.

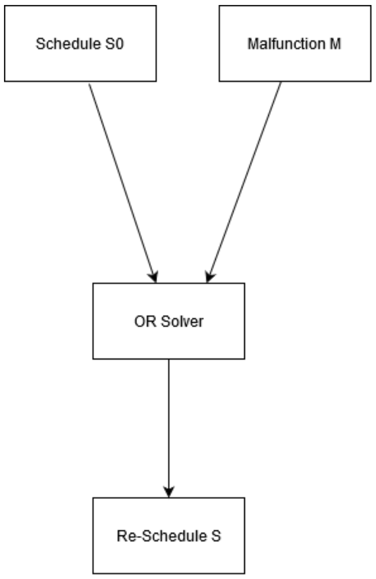

Figure 4: Simple non-iterative re-scheduling with single delay.

H2 online scoping: it is possible to design a scope reduction, either from the problem structure or from historic data (see below, Section 2.3.2); by this, we would over-achieve goal **G3**.

This approach is shown in Figure 5: the Oracle predicts restrictions in time and space, which are passed to the solver.

TODO Christian finalize naming? do we need delta naive?

We will consider the following scopings:

full after malfunction this is the full offline solution where we consider the full situation after the malfunction; this excludes paths not reachable any more after the malfunction.

delta perfect this is the almost perfect scoper which opens up only differences between the initial schedule and the full re-scheduling solution

- only paths from either schedule or full after malfunction are allowed;
- if location and time is the same in schedule and full-reschedule, then we stay at them

In this case, the solution from full after malfunction is contained in the solution space, so we expect the same (or an equivalent solution modulo costs) to be found.

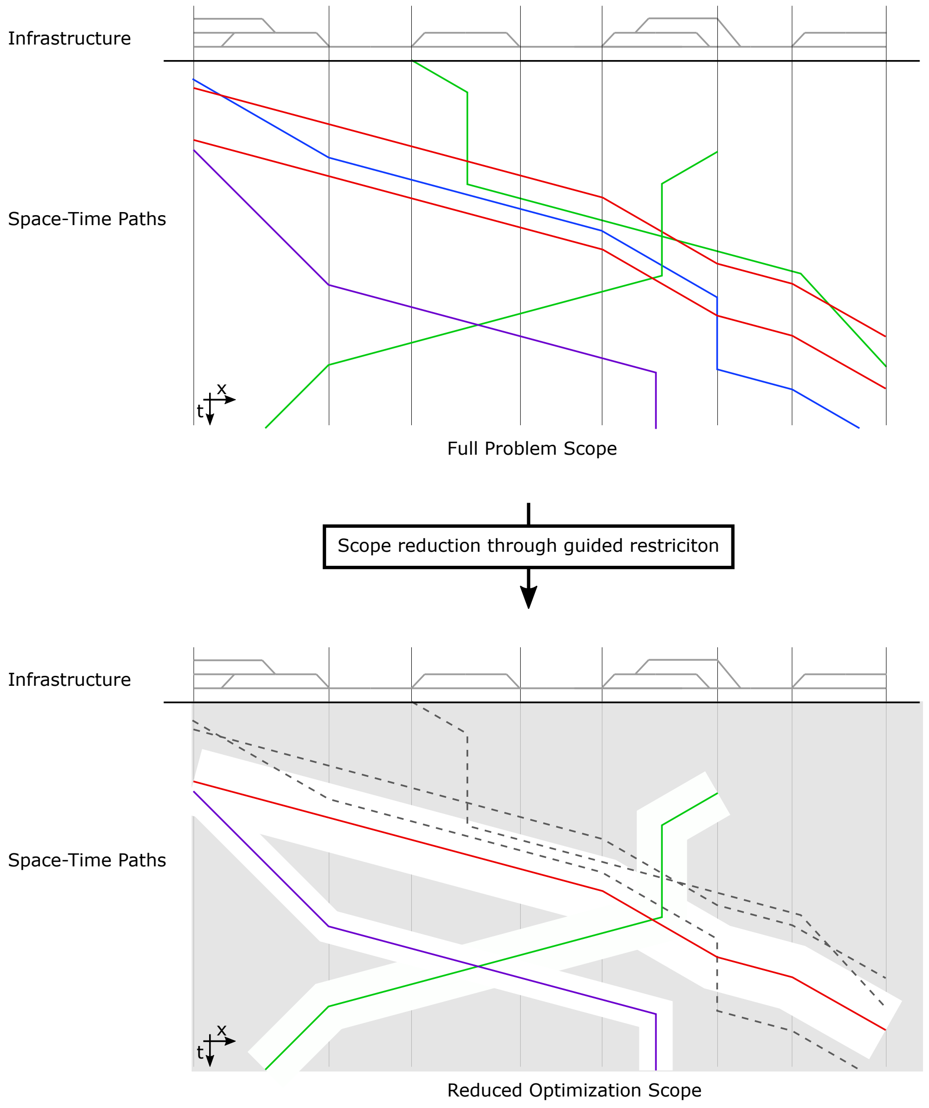

Figure 5: The Oracle gives degrees of freedom in time and space.

delta naive offline scoping, a bit less perfect: for each agent,

- if change between schedule and full after malfunction, no restriction after malfunction;
- if no change between schedule and full after malfunction, we do not open up any freedom for these agents

Again, the solution from full after malfunction is contained in the solution space, so we expect the same (or an equivalent solution modulo costs) to be found.

delta online online scoping: we propagate the delay from one train to the next along paths and times in schedule and open up agents that are predicted affected by the heuristic; we expect the solution quality to deterioriate if there are false negatives.

delta random this is a sanity check online scoping: if we predict affected trains randomly, we expect solutions to be worse than the ones found by the other scopes or that no solution can be found at all; for example if a train is scheduled to pass through the malfunction train during its malfunction and is not opened, there is no solution even if we enlarge time windows.

These scopers will be introduced formally with pseudo-code below in Section 2.3.

2.3.1 Hypothesis H1: Rationale of Offline Scoping

Hypothesis 1 is a sanity hypothesis: if we do not have a big speed-up with a perfect oracle, the whole approach must be dismissed. If no speed-up can be found we want to identify the reason and document this to gain further insight and adjust our research aim.

Consider Figure 6: The initial schedule S_0 and a malfunction M are passed to an OR solver which is given unbounded time to solve the model to optimality; the resulting re-schedule is S.

If we compare the initial schedule S_0 and the re-schedule S, we can take the differences ($Delta\_perfect$):

- every time difference at the same node opens up timing flexibility: the nodes are kept fixed, but time is flexible
- every node difference opens up routing flexibility: both nodes and times are kept flexible

We will describe this in more detail below in Section 3.7.1.

Now we want to see whether the OR solver f, given the perfect information coming through the red arrows, allows for a speed-up, i.e. whether the time t_S' needed for the solver with the restriction is much smaller than the time t_S for the full problem without the restriction

The rationale of this hypothesis is non-general and asymmetric:

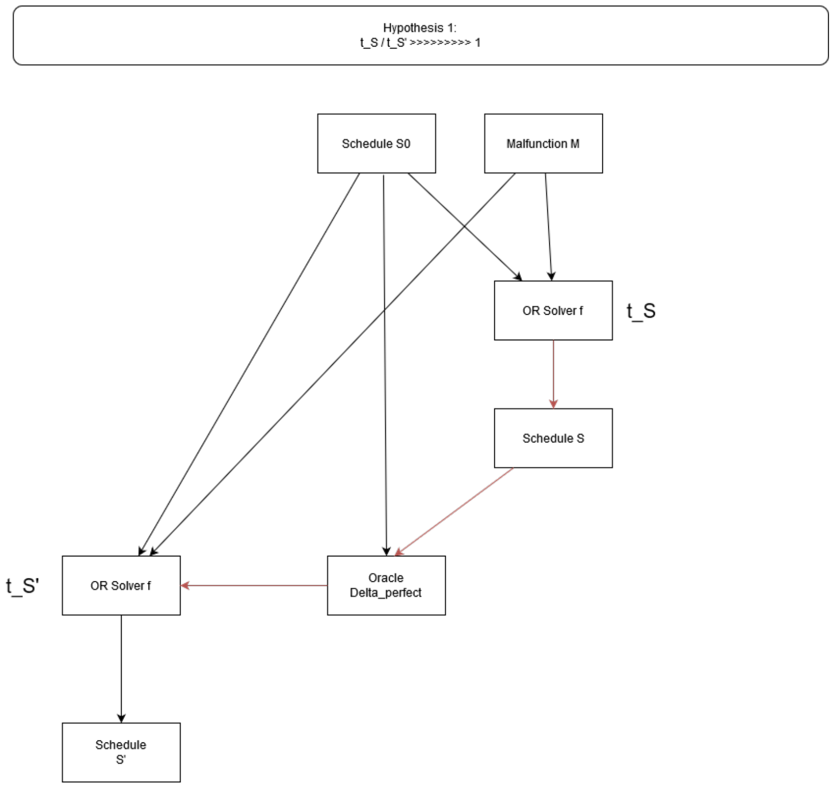

Figure 6: Rationale of hypothesis H1.

- *non-general*: if we show the speed-up for one particular OR solver, this does not mean that the information can be exploited by every other OR solver for speed-up. However, we conjecture that general setup may be applicable to other OR solvers and models, where the exact content of the Oracle's "hint" passed to the solver might differ. We might strengthen this conjecture by comparing with
- *asymmetric*: If we cannot show the speed-up for one particular solver, this does not exclude that the approach might work with a different OR solver and its particular shape of "perfect information". We might need to consider a different OR solver for our exposition.

2.3.2 Hypothesis H2: Rationale of Online Scoping

Hypothesis 2 asks whether we can build an oracle that provides a considerable speed-up without access to perfect information, only from the current situation.

Consider Figure 7: Our Oracle *Omega_realistic* now has only access to

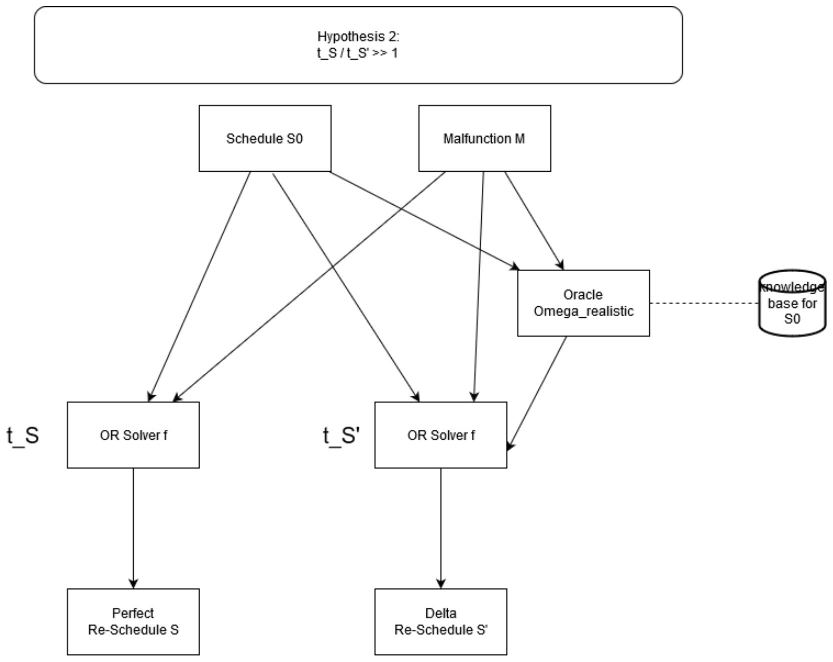

Figure 7: Rationale of hypothesis H2.

the current information and possibly its knowledge base, not to the perfect re-schedule for the current situation. Can it provide hints to the OR solver that allows for a speed-up $t\_S/t\_S' >> 1$?

We will not build such an *Omega_realistic*, but report on some first ideas which illustrate the limitation of a geographic decomposition in our synthetic setting 5. We hope this motivates researchers to invest in this task.

3 The Pipeline for H1 and H2

In this section, we will formalize the ideas of our research approach as outlined in Section 2.

3.1 Overview

We now give a more detailed view of the pipeline for hypothesis H1 with respect to Section 2.3.1. We refer to Figure 8:

TODO Christian update diagram

the pipeline decomposes into five top-level stages:

Infrastructure Generation This generates railway topologies and places agent start and targets in the infrastructure. We will cover this stage in detail in Section 3.2

Schedule Generation This generates the exact conflict-free paths and times through the infrastructure for all agents. We will cover this stage in detail in Sections 3.4.

Malfunction Generation Which train is delayed when during its run and for how long?

Experiment Run Here, the different scopers are run. We will cover these data structures and functional process steps in Sections 3.3 and 2.3.

Experiment Analysis This stage produces the plots that help to verify hypothesis H1. We will give more details in Sections 4 and 5.

Since the schedule generation step takes too long to repeat for every experiment, we will use the same infrastructure and schedule for many malfunctions. In fact, we can pre-generate infrastructures and schedules and then work on variations of the re-scheduling part of the pipeline more efficiently.

3.2 Infrastructure Generation

We now describe our railway infrastructure and how it is generated, mimicking a natural setup.

The infrastructure consists of a grid of cells. Each cell consists of a distinct tile type which defines the movement possibilities of the agent through the cell. There are 8 basic tile types, which describe a rail network. As a general fact in railway network when on navigation choice must be taken at maximum two options are available. Figure 9 (top) gives an overview of the eight basic types.

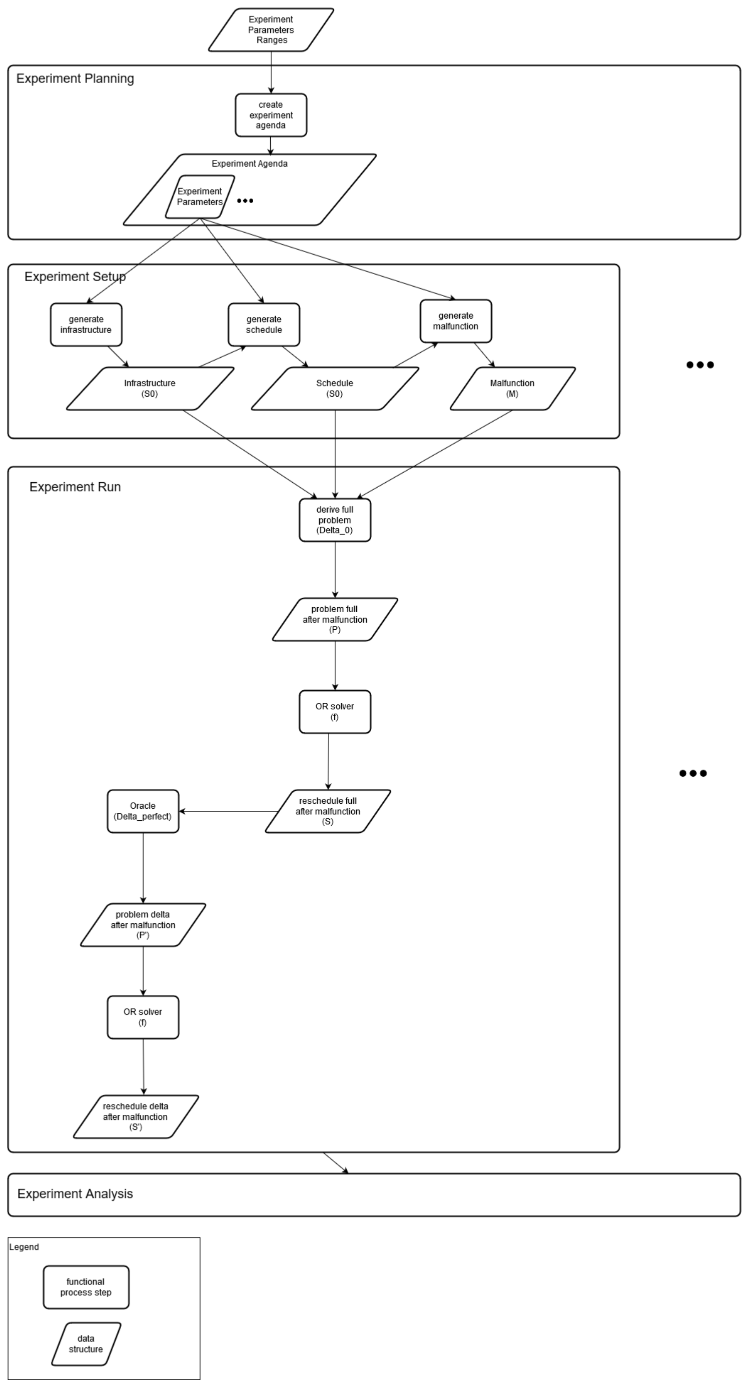

Figure 8: Pipeline for hypothesis H1 as functional diagram with intermediate data structures at two levels. The top level consists of four stages.

These can be rotated in steps of 45° and mirrored along the North–South of East–West axis. The bottom row shows the corresponding translation into a graph structure: squares represent cells, black dots represent an entry to pin to the depicted cell and a grey dot represents the pin in the opposite direction into the neighboring cell. Formally, the *railway infrastructure* is a tuple $(\mathcal{C}, \mathcal{V}, \mathfrak{c}, \mathcal{E})$,

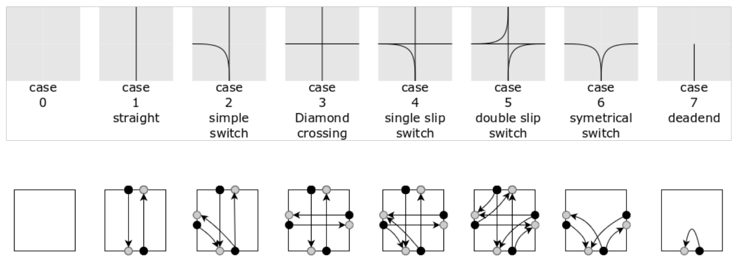

Figure 9: Eight basic cell types of a FLATland grid (top) and their translation into a directed graph (bottom).

where

- $\mathcal{C}$ is a set of cells,
- $\mathcal{V}$ is the set of pins by which the cell can be entered (they are positioned to the north, east, south or east of the cell),
- $\mathfrak{c} : \mathcal{V} \to \mathcal{C}$ which associates to each "pin" the cell it enters (there are at most 4 pins for every cell),
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, the possible directed transitions in the grid;

furthermore, we require $(\mathcal{V}, \mathcal{E})$ to be an acyclic graph.

TODO Erik describe infrastructure generation: placement of stations and lines (FLATland)

3.3 General Train Scheduling Problem

We will introduce the abstract model from [5], which we will use both for schedule generation and for re-scheduling; we could use any schedule as input, but we use the same solver model for practical reasons; the difference between these two applications is only in the optimization objective used as detailled in Sections 3.4 and 12, respectively. Furthermore, the model introduced in this section is more general than we actually need for our pipeline for H1; this will allow us to review the synthetic assumptions of Section 2.2 in a more formal setting.

According to [5], the *(general) train scheduling problem* is formalized as a tuple $(N, \mathcal{A}, C, F)$ having the following components

- N stands for the railway network (V, E, R, m, a, b), where

  - (V, E) is a directed graph,
  - R is a set of resources,
  - $m : E \rightarrow \mathbb{N}$ assigns the minimum travel time of an edge,
  - $a : \mathbb{R} \rightarrow 2^E$ associates resources with edges in the railway network, and
  - $b : R \rightarrow \mathbb{N}$ gives the time a resource is blocked after it was accessed by a train line.

- $\mathcal{A}$ is a set of train lines to be scheduled on network N. Each train in $\mathcal{A}$ is represented as a tuple (S, L, e, l, w), where
  - (S, L) is an acyclic subgraph of (V, E),
  - $e : S \rightarrow \mathbb{N}$ gives the earliest time a train may arrive at a node,
  - $l : S \rightarrow \mathbb{N} \cup \{\infty\}$ gives the latest time a train may arrive at a node, and
  - $w : L \rightarrow \mathbb{N}$ is the time a train has to wait on an edge.

  Note that all functions are total unless specified otherwise.

- C contains connections requiring that a certain train line a' must not arrive at node n' before another train line a has arrived at node n for at least α and at most ω discrete time steps. More precisely, each connection in C is of form $(t, (v, v'), t', (u, u'), \alpha, \omega, n, n')$ such that $a = (S, L, e, l, w) \in \mathcal{A}$ and $a' = (S', L', e', l', w') \in \mathcal{A}$, $a \neq a'$,$(v, v') \in L$,$(u, u') \in L'$,$\{\alpha, \omega\} \subseteq \mathbb{Z} \cup \{\infty, -\infty\}$, and either $n = v$ or $n = v'$, as well as, either $n' = u$ or $n' = u'$.

- Finally, F contains collision-free resource points for each connection in C. We represent it as a family $(F_c)_{c \in C}$. Connections removing collision detection are used to model splitting (or merging) of trains, as well as reusing the whole physical train between two train lines. More importantly, this allows us to alleviate the restriction that subgraphs for train lines are acyclic, as we can use two train lines forming a cycle that are connected via such connections. Refer to [5] for more details.

In this setting, edges can be interpreted as time slices of a train run referring to a certain a speed profile since the resources to be reserved ahead may depend on the speed profile; however, the model has no reference to the underlying geography (coordinates etc.); also, resources have no location – they can be interpreted as track sections that need to be reserved, but they may also be gates or sideway tracks that need to be reserved while travelling the edge. Notice that different trains may have the same edges in their route DAG, which means that they can drive with the same speed profile at the same place.

We now define solutions: As above, the *solution to a train scheduling problem* $(N, \mathcal{A}, C, F)$ is a pair (P, A) consisting of

1. a function P assigning to each train line the path it takes through the network, and
2. an assignment A of arrival times to each train line at each node on their path.

A path is a sequence of nodes, pair-wise connected by edges. We write $v \in p$ and $(v, v') \in p$ to denote that node v or edge (v, v') are contained in path $p = (v_1, ..., v_n)$, that is, whenever $v = v_i$ for some $1 \leq i \leq n$, or this and additionally $v' = v_{i+1}$, respectively. A path $P(a) = (v_1, ..., v_n)$ for $a = (S, L, e, l, w) \in \mathcal{A}$ has to satisfy

$$v_i \in S \text{ for } 1 \leq i \leq n, \tag{1}$$

$$(v_j, v_{j+1}) \in L \text{ for } 1 \leq j \leq n-1 \tag{2}$$

$$\mathit{in}(v_1) = 0 \text{ and } \mathit{out}(v_n) = 0, \tag{3}$$

where *in* and *out* give the in- and out-degree of a node in graph (S, L), respectively. We will write $\tau(a, P) = v_n$ for the target node used by agent a in the schedule (P, A), and, similarly $\sigma(a, P) = v_1$ for the source node of agent a in the schedule (P, A). Intuitively, conditions (1) and (2) enforce paths to be connected and feasible for the train line in question and Condition (3) ensures that each path is between a possible start and end node. An assignment A is a partial function $\mathcal{A} \times V \to \mathbf{N}$, where $A(a, v)$ is undefined whenever$v \notin P(a)$. In addition, given path function P, an assignment A has to satisfy the conditions in (4) to (8):

$$A(a, v_i) \geq e(v_i) \tag{4}$$

$$A(a, v_i) \leq l(v_i) \tag{5}$$

$$A(a, v_j) + m((v_j, v_{j+1})) + w((v_j, v_{j+1})) \leq A(a, v_{j+1}) \tag{6}$$

for all $a = (S, L, e, l, w) \in \mathcal{A}$ and $P(a) = (v_1, ..., v_n)$ such that $1 \leq i \leq n$,$1 \leq j \leq n-1$, either

$$A(a, v') + b(r) \leq A(a', u) \text{ or } A(a', u') + b(r) \leq A(a, v) \tag{7}$$

for all $r \in R$, $a, a' \subseteq \mathcal{A}$, $a \neq a'$,$(v, v') \in P(a)$,$(u, u') \in P(a')$ with $\{(v, v'), (u, u')\} \subseteq a(r)$ whenever for all $(a, (x, x'), a', (y, y'), \alpha, \omega, n, n') \in C$ such that $(x, x') \in P(a)$,$(y, y') \in P(a')$, we have $(a, (v, v'), a', (u, u'), r) \notin F_c$, and finally

$$\alpha \leq A(t', n') - A(t, n) \leq \omega \tag{8}$$

for all $(a, (v, v'), a', (u, u'), \alpha, \omega, n, n') \in C$ if $(v, v') \in P(a)$ and $(u, u') \in P(a')$. Intuitively, conditions (4), (5) and (6) ensure that a train line arrives at nodes neither too early nor too late and that waiting and traveling times are accounted for. Furthermore, Condition (7) resolves conflicts between two train lines that travel edges sharing a resource, so that one train line can only enter after another has left for a specified time span. This condition does not have to hold if the two trains use a connection that defines a collision-free resource point for the given edges and resource. Finally, Condition (8) ensures that train line a connects to a'at node n and n', respectively, within a time interval from α to ω. Note that this is only required if both train lines use the specific edges specified in the connections. Furthermore, note that it is feasible that n and n' are visited but no connection is required since one or both train lines took alternative routes.

3.4 Schedule Generation

A *railway service intention* defines the commercial contract between railway infrastructure manager (IM) and railway undertaking companies (RU). In our simplified setting, we take it as a set $\mathcal{A}$ of trains or agents; each train a has a source $\sigma(a) \in \mathcal{V}$ and a some target in $\tau(a) \subseteq \mathcal{V}$ and a speed $v(a) \in [0,1]$; furthermore, a release time r, which specifies how long a cell remains blocked after a train has left it and which we assume the same for all resources in our setting. This is exactly the output of infrastructure generation as described in Section 3.2; correctly, we should have introduced an additional layer into our hierarchical approach, splitting our infrastructure generation into infrastructure and service intention; for purely practical reasons only did we refrain from doing this (FLATland does not store the station areas in the final layout, they are only temporary data structures during grid generation, and we would have had to extend FLATland for this purpose).

We now show how a general train scheduling problem can be derived from such a railway service intention. Let $(\mathcal{A}, \sigma, \tau, v, r)$ be a service intention in a railway infrastructure $(\mathcal{C}, \mathcal{V}, \mathfrak{c}, \mathcal{E})$. Then, $(N, \tilde{\mathcal{A}}, C, F)$ with

- $\tilde{\mathcal{A}}$ consists of a tuple (S, L, e, l, w) for each $a \in \mathcal{A}$ where
  - $S = \{v : v \in P_a\}$
  - $L = \{(v_1, v_2) : (v_1, v_2) \in P_a\}$
  - $e(v) = \min_{p:\sigma(a)\text{–}v \text{ path}} |p| \cdot v(a)^{-1}$
  - $l(v) = \max_{p:v\text{–}\tau(a) \text{ path}} U - |p| \cdot v(a)^{-1}$
  - $w(e) = v(a)^{-1}$ [1].

  for a set P_a of $\sigma(a)$–$\tau(a)$ paths in $\mathcal{E}$.
- $N = (V, E, R, m, a, b)$ where
  - $V = \bigcup_{a \in \mathcal{A}} V_a$
  - $E = \bigcup_{a \in \mathcal{A}} E_a$

[1]This does not respect the intended semantics of the general model. Therefore, it would be better to use $S = \{(v, a) : v \in P_a\}$, $L = \{((v_1, a), (v_2, a)) : (v_1, v_2) \in P_a\}$, $w(e) = 0$ and $m(((v_1, a), (v_2, a))) = v(a)^{-1}$.

  - $R = \mathcal{C}$
  - $m((v_1, v_2, a)) = 0$
  - $a(c) = \{e = (v_1, v_2, a) : \mathfrak{c}(v_1) = c\}$
  - $b(c) = r$

- $C = \emptyset$
- $F = \emptyset$

is a train scheduling problem. Some remarks on this transformation:

- We only have one resource per edge, i.e. we only reserve the train's own track, per cell.
- The earliest and latest windows are designed such that they represent the earliest possible time the train can reach the vertex, respectively, the latest possible time the train must pass in order to be able to reach the target within the time limit.

The time windows given by e and l can be computed by Algorithm 1 and 2, respectively. They propagate the minimum running forward from an initial set of earliest and backwards from an initial set of latest constraints. An illustration can be found in Figures 10 and 11, respectively. The time windows are thus given by

$$e \leftarrow \mathit{propagate\_earliest}(\{\, e(\sigma(a)) = 0 \,\}, (S, L), \{\, \sigma(a) \,\}, v(a)^{-1})$$

and

$$l \leftarrow \mathit{propagate\_latest}(\{\, e(\tau) = U : \tau \in S, \mathit{out}(\tau) = 0 \,\}, (S, L), \{\, \tau \in S : \mathit{out}(\tau) = 0 \,\}, v(a)^{-1}).$$

Finally, we can remove nodes v with empty time window $e(v) > l(v)$.

Algorithm 1 *propagate_earliest*

```
Input: $e$, $(S, L)$, $F$, $mrt$ s.t. $F \subseteq \mathrm{dom}(e)$
Output: $e$
 1: $\mathit{Open} \leftarrow F$
 2: for $v \in \mathit{Open}$ do
 3:     for $v' \in S - F : (v, v') \in L$ do
 4:         if $v' \notin \mathrm{dom}(e)$ then
 5:             $e(v') \leftarrow \infty$
 6:         end if
 7:         $e(v) \leftarrow \min\{\, e(v'), e(v) + mrt \,\}$
 8:         $\mathit{Open} \leftarrow \mathit{Open} \cup \{\, v' \,\}$
 9:     end for
10:     $\mathit{Open} \leftarrow \mathit{Open} - \{\, v \,\}$
11: end for
```

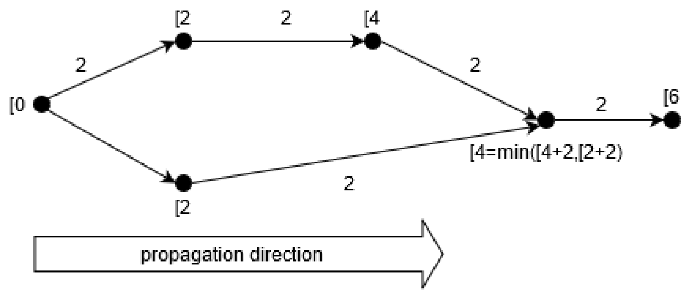

Figure 10: Illustration of *propagate_earliest* for an agent a with speed $v(a) = \frac{1}{2}$ (i.e. $mrt = 2$).

Algorithm 2 *propagate_latest*

Input: l, (S, L), F, mrt s.t. $F \subseteq \mathrm{dom}(l)$

Output: l

```
1:  $Open \leftarrow F$
2:  for $v \in Open$ do
3:      for $v' \in S - F : (v', v) \in L$ do
4:          if $v' \notin \mathrm{dom}(l)$ then
5:              $l(v') \leftarrow -\infty$
6:          end if
7:          $l(v) \leftarrow \max\{\, l(v'), l(v) - mrt \,\}$
8:          $Open \leftarrow Open \cup \{\, v' \,\}$
9:      end for
10:     $Open \leftarrow Open - \{\, v \,\}$
11: end for
```

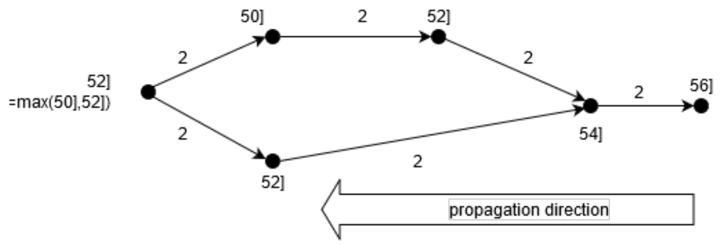

Figure 11: Illustration of *propagate_latest* for an agent a with speed $v(a) = \frac{1}{2}$ (i.e. $mrt = 2$) and for upper bound $U = 56$.

For schedule generation, we use a different objective function than for rescheduling, namely minimizing

$$\sum_{a\in\mathcal{A},v,v'\in P(a),in(v)=0,out(v')=0} A(a,v') - A(a,v) \tag{9}$$

and restricting

$$\max_{a\in\mathcal{A},v\in P(a)} A(a,v) \leq U, \tag{10}$$

where we use the following heuristic for the upper bound,

$$U = \delta \cdot \left(w + h + \frac{|\mathcal{A}|}{|S|} \right) \tag{11}$$

and $\delta = 8$, to find a schedule S for a given service intention. This objective function is intended to mimick a green wave behaviour of real-world train scheduling (schedules are constructed such that there are only planned stops for passenger boarding and alighting); however, this also keeps us from introducing time reserves in the schedule and trains will not be able to catch up in our synthetic world.

Furthermore, we only use the shortest path for each agent for scheduling in order to speed-up schedule generation; if the upper bound U is chosen liberal enough, then a solution is found where no train stops during its run.

Notice that trains have no length in the model: an edge is exited when the next edge is entered, but remains blocked for r time steps after exit. Train extension can be introduced into this model by requiring consecutive edges to require the same resource; this again highlights that edges in the general train scheduling problem do not represent physical track sections; physical track sections have to be represented by resources.

3.5 Malfunction Generation

Our experiment parameters contain the following parameters for malfunction generation:

- $m_{earliest} \in \mathbb{N}$: the earliest time step the malfunction can happen after the scheduled departure;
- $m_{duration} \in \mathbb{N}$: how much the train will be delayed;
- $m_{agent} \in \mathcal{A}$: the train that is concerned.

Given the schedule, we derive our malfunction $M = (m_{time\_step}, m_{duration}, m_{agent})$ where

$$m_{time\_step} = \min \left\{ A(m_{agent}, \sigma(a,P)) + m_{earliest}, A(m_{agent}, \tau(a,P) \right\},$$

which ensures that the agent malfunction happens while the train is running.

3.6 Objective for Re-Scheduling

We chose to to minimize a linear combination of delay with respect to the initial schedule S_0 and penalizing diverging route segments (only the first edge of each such segment). This reflects the idea of not re-routing trains without the need of avoiding delay; if re-scheduling is applied in an iterative loop, this should help to avoid "flickering" of decisions. Formally, the objective is to minimize over solutions $S = (P_S, A_S)$ the sum

$$\sum_{a \in \mathcal{A}} \delta\left(S(a, \tau(a, S)) - S_0(a, \tau(a, S_0))\right) + \rho \cdot |\{v \in \mathcal{V}(S, a) - \mathcal{V}(S_0, a) : (v', v) \in P_{S_0}(a)\}| \tag{12}$$

where the *delay model* is step-wise linear,

$$\delta(t) = \begin{cases} \infty & \text{if } t \geq \delta_{cutoff}, \\ \lfloor t/\delta_{step} \rfloor \cdot \delta_{penalty} & \text{else,} \end{cases} \tag{13}$$

for hyper-parameters δ_{step}, δ_{cutoff}, $\delta_{penalty}$ and the second term of (12) penalizes re-routing with respect to the initial schedule S_0 by weight ρ for every first edge deviating from the original schedule S_0 .

3.7 Re-Scheduling Scopes

Informally, the *re-scheduling problem* is the train scheduling problem such that

- all decisions up to and including m_{time_step} are fixed from the solution. If the train is on an edge at m_{time_step}, then it has to use the same edge, reflecting that a train on a straight segment has to continue and the switch position cannot be changed once the train is on the switch;
- the train corresponding to m_{agent} is delayed by $m_{duration}$.

We here have two options:

1. we can constrain the original scheduling up to the malfunction
2. we can remove everything up to the malfunction and constrain only the last decision before the malfunction

In order to keep the exposition simple and since this is also in the spirit of a re-scheduling loop with moving time horizon, we adopt the second option in the following exposition.[2]

We distinguish between two classes of scopes:

online the scoper does not have access to a full re-schedule solution

offline the scoper has access to a full re-schedule solution

[2]The implementation follows option 1. Should we adapt the implementation? We suspect that the overhead is negligible.

We now define different scopings that further restrict the problem to different degrees. All these scopers have the same interface, i.e. take the same parameters:

- initial schedule $S_0 = (P, A)$
- agents $\mathcal{A}$ whose elements have the form (S, L, v) where (S, L) are the possible routes for the agent and v is the agent's speed, resp. v^{-1} is the minimum running time per edge;
- malfunction $M = (m_{time\_step}, m_{duration}, m_{agent})$;
- maximum time window size c: this is the maximum time window in the re-scheduling problem
- upper bound U: we will increase the upper bound from scheduling by $m_{duration}$, which in most cases will ensure feasibility of the re-scheduling problem, as we will discuss below.

We consider the following scopings:

online_unrestricted this scoper does not restrict the re-scheduling problem, it is the "full" re-scheduling problem with empty scoping, supposed to give a trivial lower bound on speed-up with respect to the solver

offline_fully_restricted this scoper takes the re-scheduling solution from online_unrestricted, supposed to give a trivial upper bound on speed-up with respect to the solver

offline_delta this scoper takes the "difference" between the schedule and online_unrestricted, opening up route sections and times where there is a difference, supposed to give a non-trivial but unrealistic baseline on speed-up (upper-bound for non-trivial scopers)

online_route_restricted this scoper restricts the re-scheduling problem to the routes chosen in the schedule, but giving time flexibility, suppposed to show the influence of re-routing on computation times and solution quality

online_transmission_chains_fully_restricted this scoper uses a simple propagation algorithm along the scheduled paths to predict which trains will be affected by the malfunction, keeping unaffected trains exactly at their path and times, supposed to show a baseline speed-up (lower bound for non-trivial scopers)

online_transmission_chains_route_restricted this scoper uses a simple propagation algorithm along the scheduled paths to predict which trains will be affected by the malfunction, keeping unaffected trains exactly at their path and times, supposed to show a baseline speed-up (lower bound for non-trivial scopers)

online_random this scoper randomly chooses affected agents, giving no re-routing flexibility to the unaffected trains, supposed to show that the problem of predicting which agents will be affected is not trivial

3.7.1 (Empty) Scoping Re-scheduling (online_unrestricted)

We first describe the online_unrestricted scope, which does not restrict the scope, as a general train scheduling problem. Let $N = (V, E, R, m, a, b)$ be the network of train scheduling problem, (P, A) be a solution to it and let $M = (m_{time\_step}, m_{duration}, m_{agent})$ be a malfunction.

We now describe the transformations *scoper_online_unrestricted* for a single train in Algorithm 3: If the train is already done at the malfunction time step (line 1), it can be removed from the problem (line 2); if the train has not started yet at the malfunction time step (line 3), we keep its start node fixed from S_0 and do not start earlier than in S_0 and use *propagate* (described below) to derive the constraints (lines 4–11); in this case, the malfunction has no impact on the train, reflecting the idea that the train is not "in the game" yet and cannot be hit by the malfunction.

Algorithm 3 *scoper_online_unrestricted* for train a

```
Input: $(S, L)$, $(P, A)$, $M = (m_{time\_step}, m_{duration}, m_{agent})$, mrt, U, c
Output: $(S, L), e, l$
 1: if $\max_{v \in S} A(v) \leq m_{time\_step}$ then
 2:     $S \leftarrow \emptyset$, $L \leftarrow \emptyset$
 3: else if $\min_{v \in S} A(v) > m_{time\_step}$ then
 4:     $v_1 \leftarrow \arg\min_{v \in S} A(v)$
 5:     $e(v_1) \leftarrow A(v_1)$
 6:     for $\tau \in S : out(\tau) = 0$ do
 7:         $l(\tau) = U$
 8:     end for
 9:     $F_v \leftarrow F_e \leftarrow \{ v_1 \}$
10:     $F_l \leftarrow \{ \tau \in S : out(\tau) = 0 \}$
11:     $e, l, (S, L) \leftarrow propagate(e, l, (S, L), F_e, F_l, F_v, mrt, U, c)$
12: else
13:     $e, l, (S, L) \leftarrow Delta\_0\_running((S, L), (P, A), M, mrt, U, c)$
14: end if
```

Before describing the third case of the malfunction happening while the train is running, we describe *propagate* (Algorithm 4): we pass in an initial, incomplete set of earliest and latest constraints which we want to propagate in the graph (S, L) with the given minimum running time *mrt* and such that e in the output reflects the earliest possible time an agent can reach the vertex and that the l reflects the last possible time the agent must reach the vertex to be able to still find a path to the target. Furthermore, we want some initial values not to be modifiable (F_e and F_l) and we want to "force" some vertices that need be visited (F_v);

We first remove that nodes that cannot be reached given F_v (lines 1–4); then, we propagate earliest and latest (lines 5–6) and truncate time windows to c (lines 7–10); we need to propagate latest again since truncation might spoil the semantics of $l(v)$ being the latest possible passing time to reach the target in time (line 11). Finally, we remove nodes that are not reachable in time (because of an empty time window) or cannot be reached from one of the nodes F_v that must be visited (line 13).

Algorithm 4 *propagate*

```
Input: $e, l, (S, L), F_e, F_l, F_v, mrt, U, c$
Output: $e, l, (S, L)$
 1: for $v \in F_v$ do
 2:     $S \leftarrow \{\, v' \in S : \text{there is a } v\text{–}v' \text{ path or a } v'\text{–}v \text{ path in } L \,\}$
 3:     $L \leftarrow \{\, (v, v') \in L : v, v' \in S \,\}$
 4: end for
 5: $e \leftarrow \mathit{propagate\_earliest}(e, (S, L), F_e, mrt)$
 6: $l \leftarrow \mathit{propagate\_latest}(l, (S, L), F_l, mrt)$
 7: if $c < \infty$ then
 8:     for $v \in S - F_e$ do
 9:         $l(v) \leftarrow \min\{\, l(v), e(v) + c \,\}$
10:     end for
11:     $l \leftarrow \mathit{propagate\_latest}(l, (S, L), F_l, mrt)$
12: end if
13: $S \leftarrow \{\, v \in S : e(v) \leq l(v) \,\}$, $L \leftarrow \{\, (v, v') \in L : v, v' \in S \,\}$
```

We now describe the remaining case of Algorithm 3, namely when the malfunction happens while the train is running. We refer to Algorithm 5: we first determine the edge (v_1, v_2) the train is on when the malfunction happens; we then fix the time for v_1 as in S_0 and set the earliest for v_2, possibly delayed. Then, we use *propagate* to tighten the search space. Notice that *propagate* will here remove the nodes on the scheduled path before v_1 (since the forward propagation will not reach these nodes, they will be removed as the time window will be empty in the spirit of option 1 described above).

Notice that this problem always has a trivial feasible solution where all agents stop and restart in synchronicity with the malfunction agent (as said above, we extend the upper bound from scheduling by the malfunction duration).

3.7.2 Trivial upper-bound scoping (offline_fully_restricted)

TODO Christian

3.7.3 "Perfect" delta scoping (offline_delta)

The idea of the "perfect" scoping is to have a baseline where we extract as much information as possible from a pre-computed offline solution without giving the solution right away. Formally, let (P_{S_0}, A_{S_0}) be the solution to the original train scheduling problem and let $N_S = (V_S, E_S, R_S, m_S, a_S, b_S)$ be the network of the re-scheduling problem for an agent and let (P_S, A_S) be a solution to the re-scheduling problem. Algorithm 6 defines the perfect oracle for a running agent:

Algorithm 5 *Delta_0_running* for running train a

Input: (S, L), (P, A), $M = (m_{time\_step}, m_{duration}, m_{agent})$, mrt, U, c
Output: $e, l, (S, L)$

```
1: $(v_1, v_2) \leftarrow (v_1, v_2) \in L$ s.t. $A(v_1) \leq m_{time\_step}$ and $A(v_2) > m_{time\_step}$
2: $e(v_1) \leftarrow A(v_1)$, $l(v_1) \leftarrow A(v_1)$
3: if $a$ corresponds to $m_{agent}$ then
4:     $e_1(v_2) \leftarrow A(v_1) + mrt + m_{duration}$
5: else
6:     $e_1(v_2) \leftarrow A(v_1) + mrt$
7: end if
8: $F_e \leftarrow \{ v_1, v_2 \}$, $F_l \leftarrow \{ v_1 \} \cup \{ \tau \in S : out(\tau) = 0 \}$
9: for $\tau \in S - \{ v_1 \} : out(\tau) = 0$ do
10:     $l(\tau) \leftarrow U$
11: end for
12: $e, l, (S, L) \leftarrow propagate(e, l, (S, L), F_e, F_l, \{ v_1, v_2 \}, mrt, U, c)$
```

we keep fixed all times and nodes that are common to S_0 and S, respectively (lines 1–10). We delay the vertex after the malfunction by the malfunction duration (lines 11–18). Everything that is not reachable in path ($F = \Delta_P$) or time ($F_e = F_l = \Delta_A$)) will be removed by *propagate* (line 19). Notice that we have $v_1 \in\subseteq \Delta_A \subseteq \Delta_P$ and $v_2 \in \Delta_P$

Notice that the solution space of this scoping contains the full re-scheduling solution, so we will find the same or a cost-equivalent solution.

3.7.4 Route-restricted Re-scheduling (online_route_restricted)

TODO Christian

3.7.5 Re-Scheduling with Transmission Chains (online_transmission_chains_fully_restricted, online_transmission_chains_route_restricted)

If we do not consider routing alternatives, we can find an easy way to predict the trains affected by a malfunction by recursively propagating the delay along the scheduled paths of the reached agents. Let $S_0 = (P, A)$ be a schedule and let $P(a) = (v_1, \ldots, v_n)$ the scheduled path for an agent a, then the exit time of the edge after an entry vertex is

$$E(a, v_i) = A(a, v_{i+1}) + r \tag{14}$$

for $i = 1, \ldots, n-1$ and $E(a, v_n) = A(a, v_n) + r$.

Now we can have a look at Algorithm 7: we have a queue q where we store which agents are reached, at which vertex and how much of the delay is propagated (line 1). In order to prevent loops, we store the elements of the queue already dealt with (line 2). The output will be a set of agents reached, initialized with the malfunction agent which we know will be delayed since the

Algorithm 6 $Delta_{perfect\_after\_malfunction_running}$ for running train a

```
Input: $(S, L)$, $(P_{S_0}, A_{S_0})$, $(P_S, A_S)$, $M = (m_{time\_step}, m_{duration}, m_{agent})$, mrt, U, c
Output: $e, l, (S_1, L_1)$
 1: $\Delta_A \leftarrow \{ v : A_S(v) = A_{S_0}(v), v \in P_{S_0}, v \in P_S \}$
 2: $\Delta_P \leftarrow \{ v : v \in P_S, v \in P_{S_0}, v \in P_{S_0}, v \in P_S \}$
 3: $S_1 \leftarrow \{ v : v \in P_{S_0} \} \cup \{ v : v \in P_S \}$
 4: $L_1 \leftarrow \{ (v, v') \in P_S$ or $(v, v') \in P_{S_0} : v, v' \in S_1 \}$
 5: for $v \in \Delta_A$ do
 6:     $l(v) \leftarrow e(v) \leftarrow A_{S_0}(v)$
 7: end for
 8: for $v \in S_1 - \Delta_A : out(v) = 0$ do
 9:     $l(v) \leftarrow U$
10: end for
11: $(v_1, v_2) \leftarrow (v_1, v_2) \in L_1$ s.t. $A_{S_0}(v_1) \leq m_{time\_step}$ and $A_{S_0}(v_2) > m_{time\_step}$
12: if $v_2 \notin \Delta_A$ then
13:     if $a$ corresponds to $m_{agent}$ then
14:         $e_1(v_2) \leftarrow A_{S_0}(v_1) + mrt + m_{duration}$
15:     else
16:         $e_1(v_2) \leftarrow A_{S_0}(v_1) + mrt$
17:     end if
18: end if
19: $e, l, (S_1, L_1) \leftarrow propagate(e, l, (S_1, L_1), \Delta_A \cup \{ v_2 \}, \Delta_A, \Delta_P, mrt, U, c)$
```

schedule is constructed such that agents always run at maximum speed (line 3). We initialize the queue and the changed agents with the malfunction agent with pushing all vertices that are passed in the schedule after the malfunction; since the schedule is constructed such that the agents never stop, the delay is not reduced (lines 4–6). Then, we look at the next agent after the queued element (lines 8–13). If the delay cannot be absorbed by the time difference between the queued element's departure and the next agent entry (lines 14–15), then we mark the next agent as reached (line 16) and push all vertices of the next agent after the delay propagation (lines 17–20); here, we do not only consider the delay to be propagated to all resources after v, but also allow a' to wait for a on resources it is on in the range of the delay that could be transmitted through the common resource of the edge entered at v.

TODO Christian The pseudo-code does not reflect the implementation correctly, we should have the next agent occupying the resource and not not the next agent passing the same vertex!!!!
TODO Christian Proof of correctness that this algorithm guarantees feasibility?
TODO Christian Compare with algo in Handbook of Railway Optimization

The output of this prediction can then be passed to Algorithm 8: all agents predicted as changed will have full re-scheduling degrees of freedom (lines 2–3) and all others will be "freezed" to the original schedule (lines 4–9).

This scoping is clearly feasible since we can find a new schedule with no route-change. However, since false positives and false negatives with respect to the full offline solution are possible, it is not guaranteed that the minimum costs can be reached.

3.7.6 Random Online Re-scheduling (online_random)

As a sanity check, to show that our scoping is not trivial, we use Algorithm 8 with random prediction of Algorithm 9: we determine the fraction of changed agents among those running at or after the malfunction (lines 1–2) and choose randomly with the same fraction among those running after the malfunction, ensuring that the malfunction agent is contained (lines 3–7).

In this case, it is not guaranteed that the problem is feasible. A simple example is if another agent is scheduled to go through the malfunctioning agent as it stands still but the other agent is not opened up and is forced to keep its schedule; then the problem is always infeasible; there may be many more such situations by transitivity.

TODO Christian in the implementation, we do not consider which agents are running but choose p_{min} pretty high. Consolidate.

Algorithm 7 *transmissionchains*

```
Input: $\mathcal{A}$, $(P_{S_0}, A_{S_0})$, $M = (m_{time\_step}, m_{duration}, m_{agent})$
Output: $\mathcal{A}^* \subseteq \mathcal{A}$
 1: $q = queue()$
 2: $done = \emptyset$
 3: $\mathcal{A}^* \leftarrow \{ m_{agent} \}$
 4: for $v \in P_{S_0}(m_{agent}) : A_{S_0}(m_{agent}, v) \geq m_{time\_step}$ do
 5:     $q.push((m_{agent}, v, m_{duration}))$
 6: end for
 7: while $q \neq \emptyset$ do
 8:     $(a, v, d) \leftarrow q.pop()$
 9:     if $(a, v, d) \in done$ then
10:         continue
11:     end if
12:     $done \leftarrow done \cup \{ (a, v, d) \}$
13:     $a' \leftarrow \arg\min_{a' \in \mathcal{A}, v \in P_{S_0}(a')} A_{S_0}(a', v)$
14:     $\delta \leftarrow A_{S_0}(a', v) - D_{S_0}(a, v)$
15:     if $\delta < d$ then
16:         $\mathcal{A}^* \leftarrow \mathcal{A}^* \cup \{ a' \}$
17:         $d' \leftarrow d - \delta$
18:         for $v' \in P_{S_0}(a') : A_{S_0}(a', v') \geq D_{S_0}(a, v) + d'$ do
19:             $q.push((a', v', d'))$
20:         end for
21:     end if
22: end while
```

Algorithm 8 $Delta_{changed}$

```
Input: $\mathcal{A}$, $(P_S, A_S)$, $\mathcal{A}^* \subseteq \mathcal{A}$
Output: $\mathcal{A}^{**}$
 1: for $a = (S, L, v) \in \mathcal{A}$ do
 2:     if $a \in \mathcal{A}^*$ then
 3:         $e, l, S, L \leftarrow scoper\_online\_unrestricted((S, L), (P, A), M, v^{-1}, U, c)$
 4:     else
 5:         $S \leftarrow \{ v : v \in P_S \}$
 6:         $L \leftarrow \{ (v, v') \in P_S : v, v' \in S \}$
 7:         $e \leftarrow \{ (v, P_S(v) : v \in S \}$
 8:         $l \leftarrow \{ (v, P_S(v) : v \in S \}$
 9:     end if
10:     for $e \in L$ do
11:         $w(e) \leftarrow v^{-1}$
12:     end for
13:     $\mathcal{A}^{**} \leftarrow \mathcal{A}^{**} \cup \{ (S, L, e, l, w) \}$
14: end for
```

Algorithm 9 *$Delta_{random}$*

```
Input: $\mathcal{A}$, $(P_{S_0}, A_{S_0})$, $M = (m_{time\_step}, m_{duration}, m_{agent})$, $p_{min}$
Output: $\mathcal{A}^* \subseteq \mathcal{A}$
1: $\mathcal{A}_{running} = \{ a \in \mathcal{A} : \max_{v \in P_{S_0}} A_{S_0}(a, v) \geq m_{time\_step} \}$
2: $p_{changed} = \min\{ p_{min}, \frac{|\{ a \in \mathcal{A}_{running} : P_{S_0} \neq P_S \text{ or } A_{S_0}(a,\cdot) \neq A_S(a,\cdot) \}|}{|\mathcal{A}_{running}|} \}$
3: for $a \in \mathcal{A}_{running}$ do
4:     if $a$ corresponds to $m_{agent}$ or $random\_bool(p_{changed})$ then
5:         $\mathcal{A}^* \leftarrow \mathcal{A}^* \cup \{ a \}$
6:     end if
7: end for
```

4 Computational Results

4.1 Experiment Design

In light of the findings of the previous paragraphs, we design our experiments in a hierarchical fashion on multiple levels

1. infrastructure
2. schedule
3. malfunction
4. solver runs

where at each level we generate multiple instances for a fixed instance of the previous levels (i.e. we generate multiple infrastructures for the same parameters, we generate multiple schedules for the same infrastructure etc.). The lowest level is a sanity level to investigate the variance of the solver for the same problem.

The agenda for the results of this section is shown in Figure 12: We have

- 8 infrastructures with 80 to 150 agents (i.e. the first infrastructure has 80 agents, the second has 88 agents)
- 10 schedules per infrastructure
- for every agent a malfunction 30 time steps after its scheduled start for 50 time steps
- 1 solver run

This gives an agenda of 8640 experiments. The data is available in a dedicated git repository `https://github.com/SchweizerischeBundesbahnen/rsp-data`. Infrastructure, schedule and the experiment results shown in this section are in h1_2020_08_24T21_04_42/h1_2020_08_24T21_04_42_baseline_2020_09_17T17_08_13; in these provisional results, only the first 803 experiments have been run.

TODO Christian Run and upload full agenda, finalize

We filter out experiments with small absolute and large relative run times:

- time full after malfunction < 10s
- time full after malfunction $>$ 97%-percentile of time full after malfunction

This filters out 25 experiments and leaves us with data from 778 experiments.

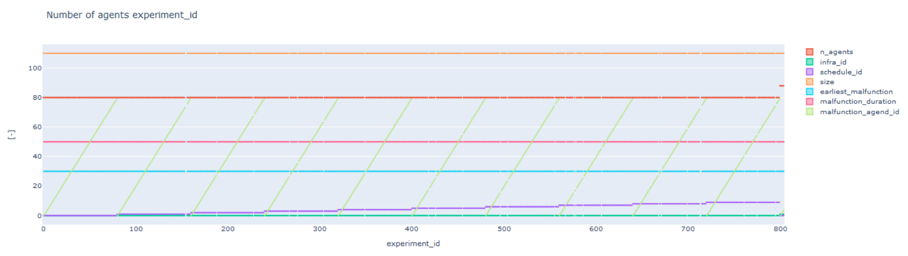

Figure 12: Hierarchical Agenda Structure.

4.2 Speed-Up and Solution Quality

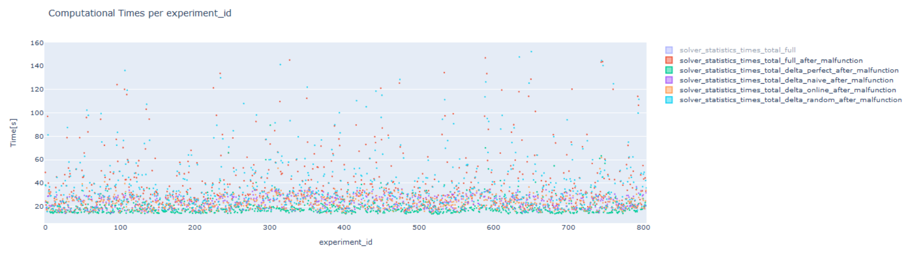

Figure 13: Computation times per experiment.

In Figure 13, we do no recognize any block structure of running times as we could expect from Figure 12. In other words, grid size and number of against is not a good measure of the difficulty of the re-scheduling problem (we conjecture that it may be even hard to determine beforehand whether a problem is hard); furthermore, the difficulty varies greatly for the same schedule when taking different agents (notice that this implies different time steps as well since the malfunction is performed 30 time steps after the malfunction agent's departure). We will come back to this in Section 5, where we give some illustrations.

Therefore, we trivially take the full re-scheduling time as measure of problem complexity and plot the rescheduling times and speed-up for the different scopes against the full re-scheduling time in Figure 14 We see a clear speed-up for almost all experiments except for the random scoper, which is reflected by the random scope has runtimes staying along the diagonal (full rescheduling time) and speed-up being around 1.0. Furthermore, the re-scheduling times for more difficult problems in the reduced scopes (delta perfect, delta naive, delta online) seem to stay in an almost constant range; this is reflected by an increasing speed-up with respect to the full re-scheduling time.

With these preliminary results, do not see a clear separation between the scopes, only a slight tendency for the speed-up being greatest in delta perfect.

TODO Christian results are wrong for random!

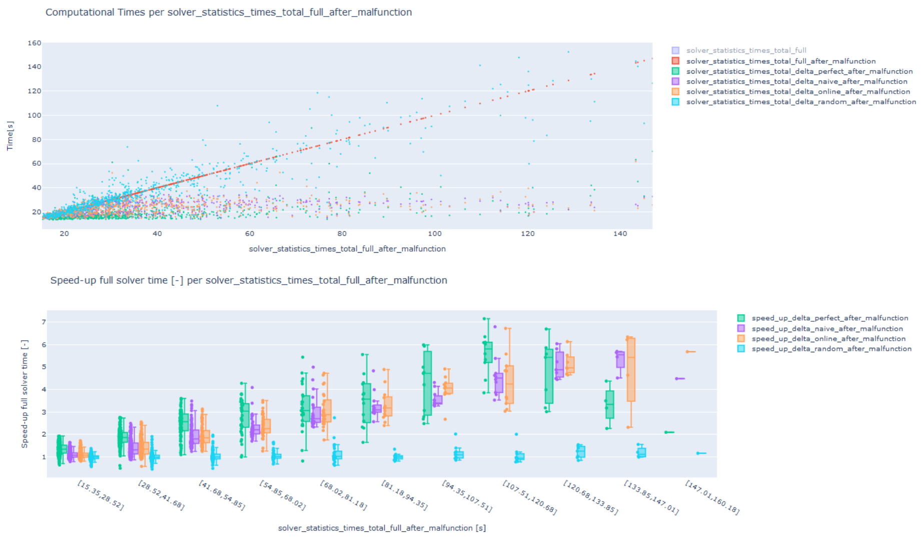

Figure 14: Re-scheduling times and speed-up against full re-scheduling time. The speed-ups are grouped in 10 bins of equal size on the horizontal axis between min and max value.

TODO Christian visualize effective costs and ratio: delta perfect and delta naive should reach same costs

4.3 Prediction Quality of Online Scopes

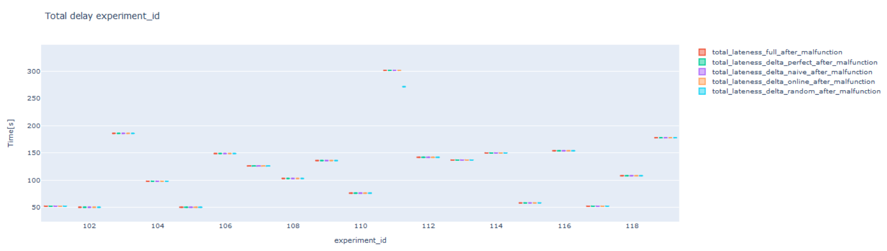

Figure 15: .

TODO Christian false positives, false negatives etc.

4.4 Model Calibration

TODO Christian ASP: preprocessing vs. solving; different parameters (delay model?): which parameters make the problem harder, what do we measure?

5 Case Studies

TODO Christian example of cost-equivalent solutions; example of false negative, false positive; malfunction variation

6 Extended Discussion/Related Work/Literature: show links to various Research Approaches

TODO Emma ML delay propagation / recourse / stability of iterative and batch processing simulation OR / heuristics (job insertion) decomposition approaches real-time rescheduling

References

[1] Rcs – traffic management for europe's densest rail network. controlling and monitoring with swiss precision. `https://company.sbb.ch/content/dam/sbb/de/pdf/sbb-konzern/sbb-als-geschaeftspartner/RCS/SBB_RCS_Broschuere_EN.pdf.sbbdownload.pdf`. Accessed: 2020-07-15.

[2] The swiss way to capacity optimization for traffic management. `https://www.globalrailwayreview.com/wp-content/uploads/SBB-RCS-Whitepaper-NEW.pdf`. Accessed: 2020-07-15.

[3] Gabrio C. Caimi. *Algorithmic decision suport for train scheduling in a large and highly utilised railway network*. PhD thesis, ETH Zurich, Aachen, 2009.

[4] Flatland challenge. multi agent reinforcement learning on trains. `https://www.aicrowd.com/challenges/flatland-challenge`. Accessed: 2020-07-15.

[5] Dirk Abels, Julian Jordi, Max Ostrowski, Torsten Schaub, Ambra Toletti, and Philipp Wanko. Train scheduling with hybrid answer set programming. *CoRR*, abs/2003.08598, 2020.