Real Time Large Railway Network Re-Scheduling

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

November 23, 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 approach in a formal way;

G1 motivate the scoping 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) 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].

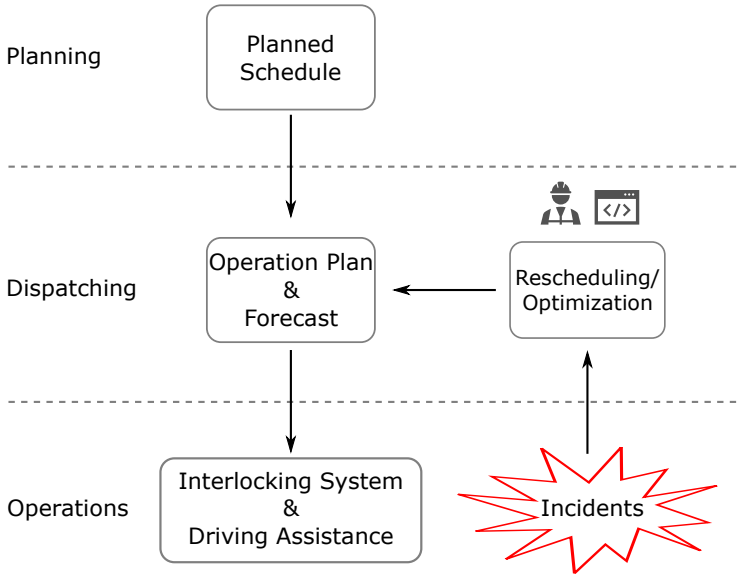


Figure 1: *Simplified schema of the rescheduling process during operations. There are three main areas involved:* planning, dispatching *and* operation *(interlocking and driving assistance). Dispatchers have access to the current situation and an automated forecast and make decisions at the level of 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.

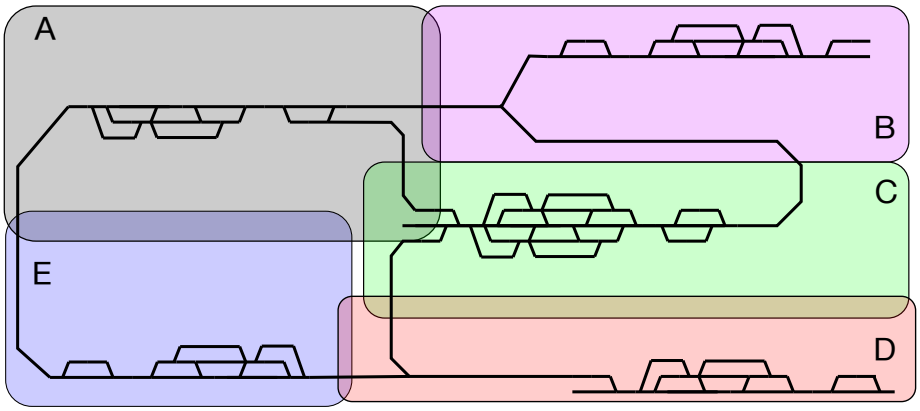


Figure 2: *Due to the vast complexity of a railway network, the re-scheduling task is decomposed into smaller geographical areas (A-E), withiwin which human dispatchers optimize traffic flow. Inter-region coordination is mostly done by informal means of communication such as telephone.*

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 sys-

tems. For larger areas, we cannot currently find feasible re-scheduling solutions in real-time.

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

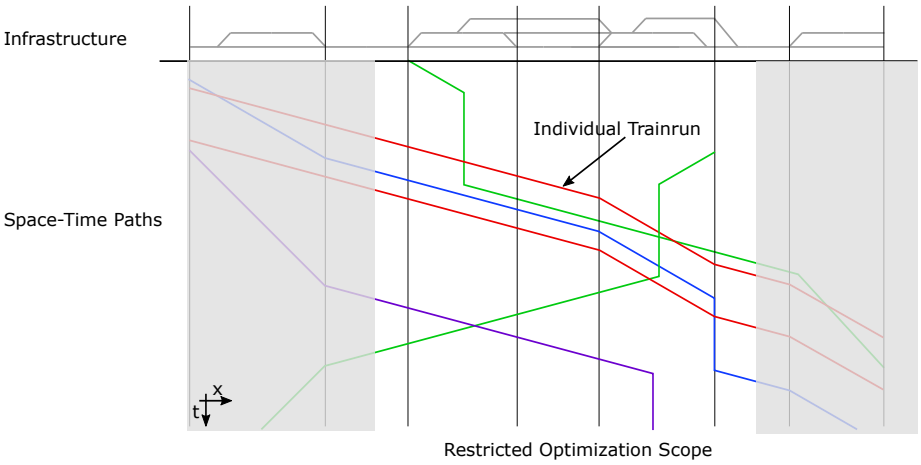


Figure 3: *Real-time rescheduling has so far been applied to restricted geographic areas only. This means that human intervention or buffer times are necessary outside of the optimization scope to guarantee conflict free traffic in the whole network. The limited geographical scope leads to a cost-function that can only locally be minimized and might have negative effects at 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, we refer to this as the **core problem**. In contrast to current models we do not follow a spatial decomposition according to network properties but rather investigate the possibility of predicting the relevant core problem given a schedule and a malfunction.

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

- Departure times of trains
- Routes of trains

and hence spans both space and time, in contrast to spatial decomposition.

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.

Our approach combines Operations Research and Domain-specific Learning to get the best of both worlds: a "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 constrains the search space for the solver or at least drives its search more efficiently (driving the branching process).

2.2 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 railway elements in 2D grid. Each railway element occupies

a square cell in this grid. The railway element defines the orientation depended possible transitions of a train to the four direct neighbouring cells (e.g. switches allow for different transitions depending on the orientation of a train on the switch). Infrastructure details can be found in section 3.2

Synthetic Timetable and Train Dynamics

Given a specific infrastructure we generate a simplified synthetic schedule with the following constraints:

- 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.
- There are no time reserves in the schedule, which means that trains cannot reduce delay during their journey.
- 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 scheduled connections between trains or vehicle tours (turnrounds).

Synthetic route alternatives

We use k-shortest paths from source to target as list of route alternatives; path cycles are not allowed. In reality, the choice and prioritization of route alternatives is intricate.

Simple Disturbance Model

We simulate one train a being stopped for d discrete time steps at some time t and call this a malfunction $M = (t, d, a)$; This setup is shown in Figure 4. In reality, the delay might not be known or only probabilities can be assumed and multiple disturbances can occur independently.

2.3 Reduction in Space and Time

The main idea of problem scope reduction is shown in Figure 5, where the scoper predicts what parameters in time and space are relevant to solve the core problem caused by the malfunction.

2.4 Online and Offline Scoping

We distinguish two types of scopes: We distinguish between two classes of scopes:

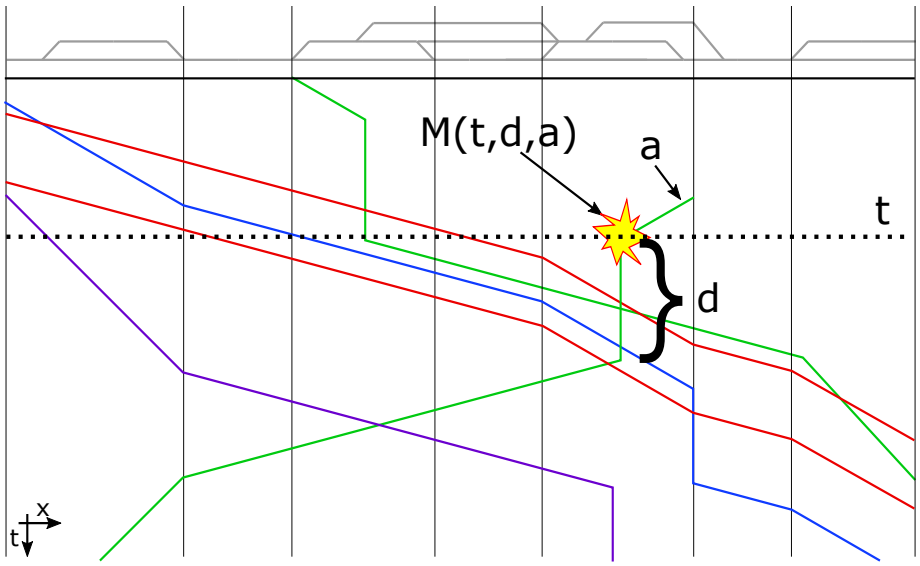


Figure 4: *This figure illustrates a simple operations plan with 6 trains on a small infrastructure. A single malfunction $M(t, d, a)$ occurs for agent a at time t with a total duration of d time steps. In reality of course there exists many different disturbances in the network which we will discuss further in Section 5.4*

online the scoper does not have access to a full re-schedule solution[1]

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

The rationale of offline scoping depending on a is shown in Figure 6.

We will consider the following problem scope reduction schemes (scopers):

online_unrestricted this scoper does not restrict the re-scheduling problem, it is the "full" re-scheduling problem with "empty" scope reduction, supposed to give a trivial lower bound on speed-up with respect to the solver; this excludes unreachable alternative paths after the malfunction

offline_fully_restricted this scoper takes the re-scheduling solution from the online_unrestricted scoper as its scope. This gives a trivial upper bound on speed-up with respect to the specific solver since it measures the specific solver's overhead.

offline_delta this scoper reduces the scope to the difference between the initial schedule and the online_unrestricted solution,

- only edges from either schedule or online_unrestricted are allowed;

[1]More precisely, online scopers know at most how many agents have changed but not which ones, see below offline delta weak vs. online random.

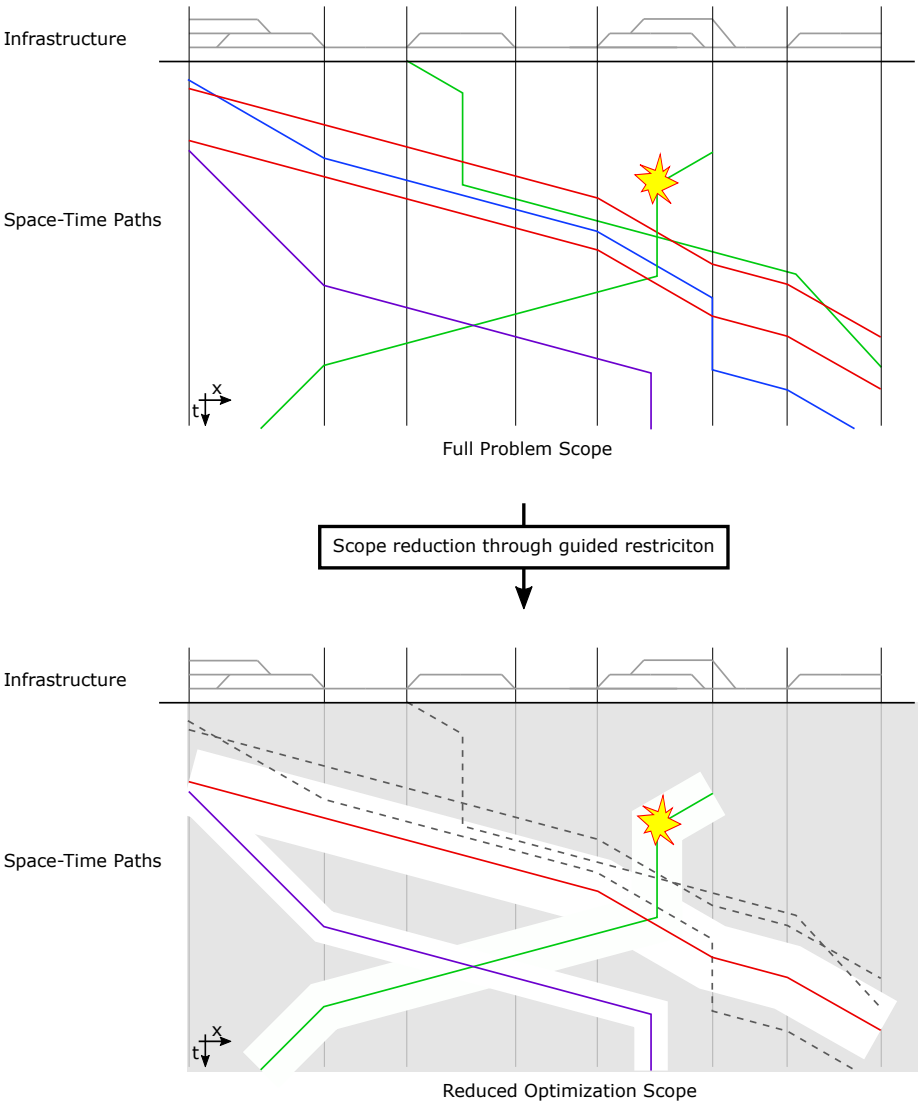


Figure 5: *The scope reducer condenses the initial full problem description to the core problem in the time-space parameters. This vastly reduces the search space of an optimizing algorithm.*

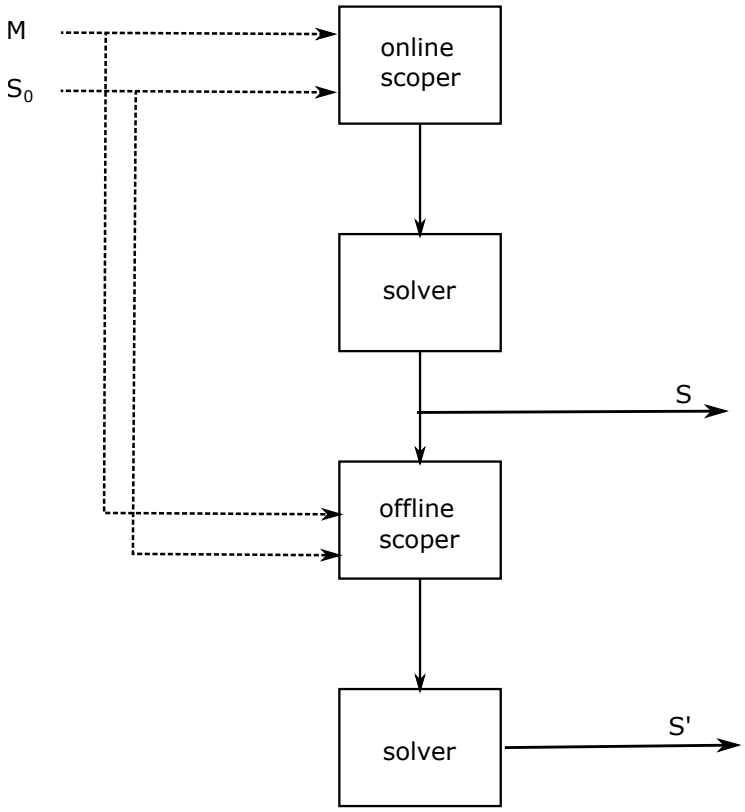


Figure 6: *Signal-flow diagram for online and offline scoping. It shows the online re-schedule S and the the offline re-schedule S'. It shows that offline scoper takes the online schedule as an additional in addition to the malfunction M and the initial schedule S_0.*

- if location and time is the same in schedule and full-reschedule, then we stay at them

supposed to give a non-trivial but unrealistic baseline on speed-up (upper-bound for non-trivial scopers); in this case, the solution from online_unrestricted is contained in the solution space, so we expect the same (or an equivalent solution modulo costs) to be found.

offline_delta_weak this scoper gives a lose delta by opening up the same as online_unrestricted for changed agents. This gives a first impression how much speed-up we gain by scoping on a train-by-train basis. We expect this to work reasonably well in sparse infrastructures / schedules, but less so in denser infrastructure/schedules, which do not separate well the effect of a malfunction. This might give us hints as to how well a scoper needs to work (with respect to a solver) in order to achieve desired speed-ups.

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 delay 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); we expect the solution quality to deterioriate if there are false negatives.

online_transmission_chains_route_restricted in contrast to the precedent scoper, unaffected agents are given time flexibility (they are constrained to use the same path as in the schedule, though)

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: 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; furthermore, we have time flexibility to trains not chosen since 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 on the chosen affected agents.

These scopers will be introduced formally with pseudo-code below in Section 3.7.

2.5 Research Programme: two Hypotheses

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

Hypothesis 1 (H1) : if we have access to an offline solution of the re-scheduling problem given unbounded resources, we can design an offline scope reduction that allows for a large speed-up in solution time (see below, Section 2.5.1); by this, we achieve goal **G2**.

Hypothesis 2 (H2) it is possible to design an online scope reduction, either from the problem structure or from historic data, which achieves similar speed-up factors as H1 (see below, Section 2.5.2); by this, we would over-achieve goal **G3**.

2.5.1 Hypothesis H1

Hypothesis 1 is a sanity hypothesis: if we cannot achieve a substantial speed-up with the offline_delta scoper, the whole approach must be re-evaluated and adjusted.

Consider Figure 7: The initial schedule S_0 and a malfunction M are passed

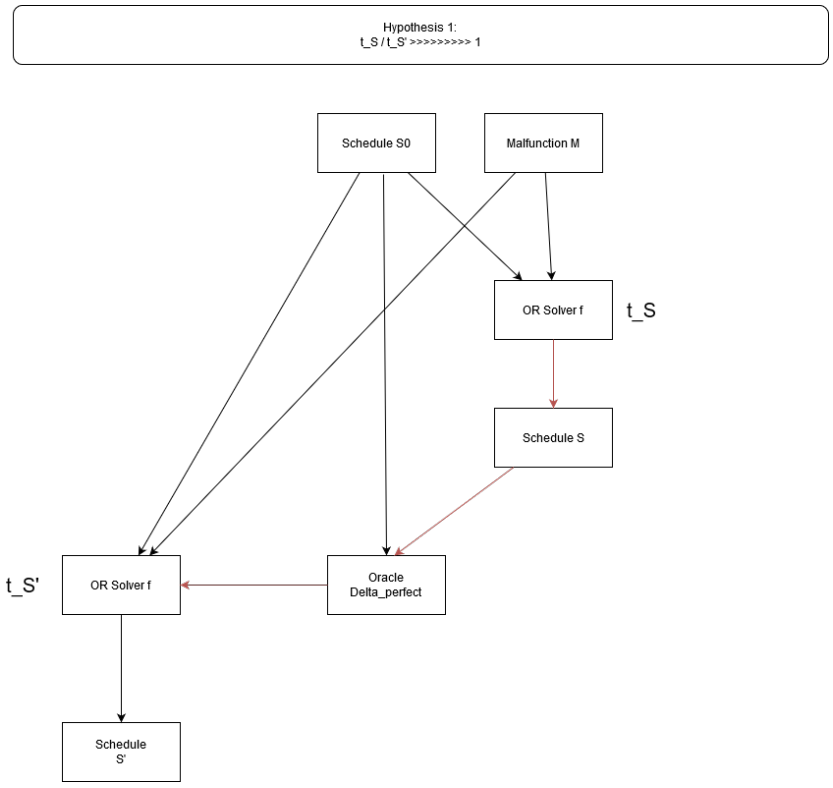


Figure 7: *Rationale of hypothesis H1.*

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 ($offline\_delta$) and define a new and reduced problem scope as follows:

- every temporal difference between $S_O\$$ and S at the same (partial) route opens up timing flexibility: the (partial) routes are kept fixed, but the time variable becomes part of the new problem scope.
- every (partial) route difference between S_0 and S opens up routing flexibility: routing and time variables become part of the new problem scope.

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

We investigate if an OR solver f, given the reduced problem scope, has a much shorter solution time t_S' compared to the time t_S for the full problem without the restriction.

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

- *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.

2.5.2 Hypothesis H2

Hypothesis 2 asks whether we can build an oracle that provides a considerable speed-up with access only to iformation at the current time.

Consider Figure 8: Our scoper *Omega_realistic* only has access to the current information and possibly its knowledge base, not to the perfect re-schedule for the current situation. Is such a scoper able to reduce the scope to a degree that allows for a speed-up $t\_S/t\_S' >> 1$?

The implementation of such a scoper *Omega_realistic* is outside the scope of this research, but we

- report on some first ideas (Section 3.7) which should give a baseline, showing that the scoping is not trivial (goal **G4**)
- illustrate the limitation of a geographic decomposition in our synthetic setting 5 (goal **G1**).

We hope this motivates and encourages research into the area of real-time problem reduction.

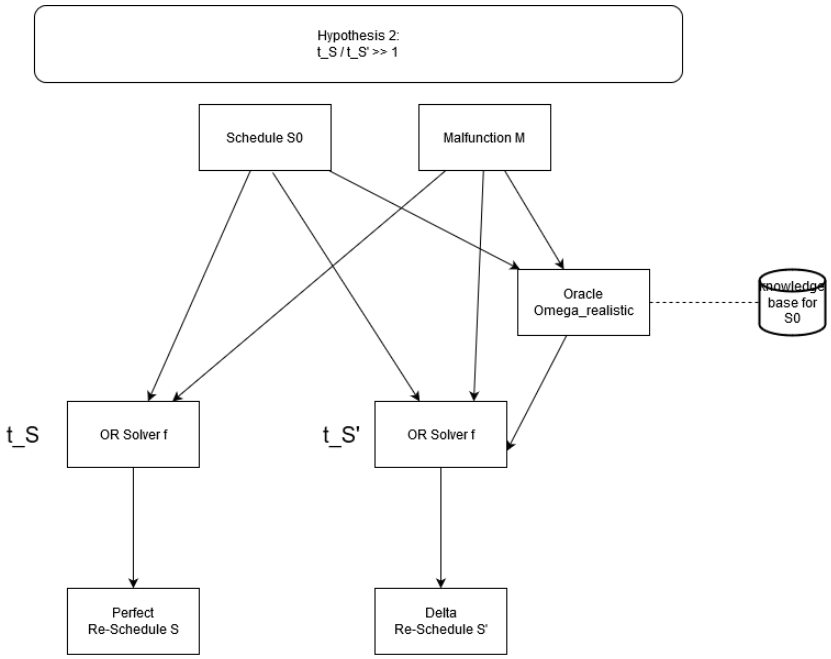


Figure 8: *Rationale of hypothesis H2.*

3 The Experiment Pipeline for H1 and H2

In this section, we formalize the ideas of our research approach as outlined in Section 2 (goal **G0**).

3.1 Overview

We now give a more detailed view of the pipeline for hypothesis H1 with respect to Section 2.5.1. We refer to Figure 9: the pipeline decomposes into five top-level stages:

1 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[2]

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 Section 3.4.

[2]Conceptually, the placement of agents should be separated from infrastructure generation. However, the way FLATland is designed, it is not possible without refactoring to separate the two as the information about cities/stations is only available during infrastructure generation and agent placement.

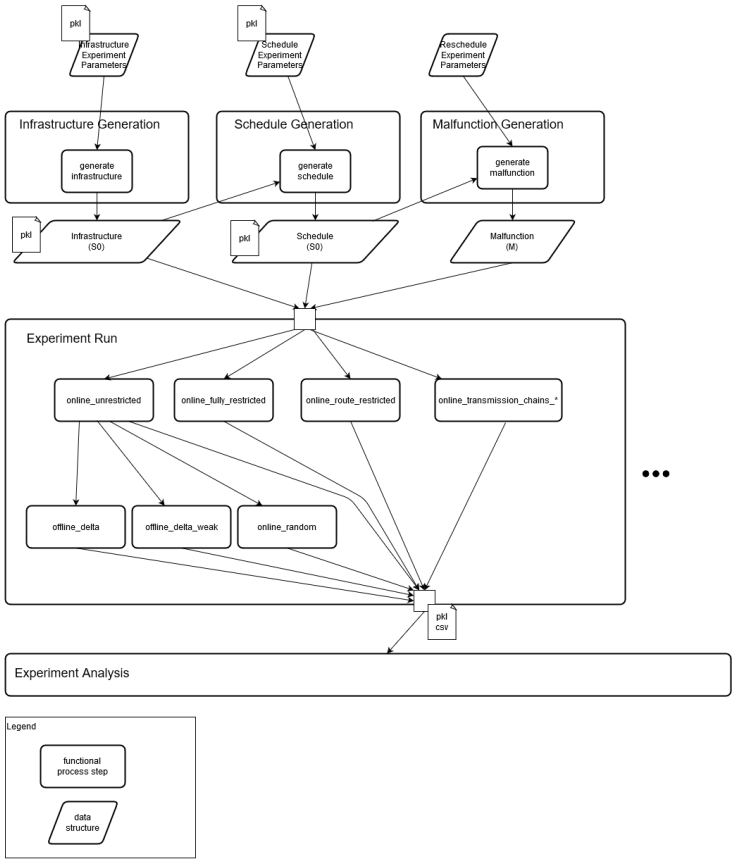


Figure 9: *Pipeline for hypothesis H1 as functional diagram with intermediate data structures at two levels. The top level consists of four stages.*

3 Malfunction Generation We define the type and time of the malfunction. Throughout this research project we use single train time delay as malfunction, i.e. Agents gets halted for x timesteps during simulation. See Section 3.5.

4 Experiment Run Here, the different scopers are run on the same problem instance. We will cover these data structures and functional process steps in Sections 3.3 and 3.7.

5 Experiment Analysis This stage aggregates the data and generates the plots that help to verify hypothesis H1 and H2 baseline. 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. We refer to `https://github.com/SchweizerischeBundesbahnen/rsp/blob/master/README.md` for practical getting started.

3.2 Infrastructure Generation

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. For each direction a train can pass through a cell, there are at most two options available, there are no 3-way forks. Figure 10 shows eight basic cell types of a FLATland grid (top) and their translation into a directed graph (bottom). The result of this

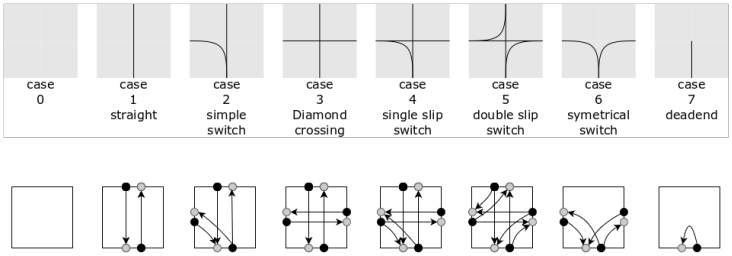


Figure 10: *Overview of the eight basic types in the top row. These basic types can be rotated in steps of* 45° *and mirrored along the North–South or East–West axis to give all possible cell types. The bottom row shows the corresponding translation into a graph structure: squares represent cells, black dots represent an entry pin into the depicted cell and a grey dot represents the pin in the opposite direction into the neighboring cell.*

translation is captured formally as *railway infrastructure*, which define to be a tuple $(\mathcal{C}, \mathcal{V}, c, \mathcal{E})$, 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),
- $c : \mathcal{V} \to \mathcal{C}$ 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;
- $(\mathcal{V}, \mathcal{E})$ is an acyclic graph.

Intuitively, we have a directed graph of routing possibilities where in addition each edge (identified by its entering node) is related to a cell (or more generally a resource) that it occupies.

To generate the infrastructure for our experiments we utilize the *sparse_rail_generator* which is part of the FLATland environment. It uses the following parameters to generate a railway infrastructure.

- **number_of_agents:** The number of trains in the schedule
- **width:** Number of cells in the width of the environment
- **height:** Number of cells in the height of the environment
- **flatland_seed_value:** Random seed to generate different configurations
- **max_num_cities:** Maximum number of cities to be places in the environment. Cities are the only places where agents can start or end their journey. Cities consists of parallel track and entry/exit ports.
- **max_rail_in_city:** Maximum number of parallel tracks in the city
- **max_rail_between_cities:** Maximum number of parallel track at entry/exit ports of the cities

The generated cities are illustrated in figure 11, which in turn are connected together to form a closed infrastructure system as shown in figure 12.

3.3 General Train Scheduling Problem

We introduce the abstract model from [5], which we 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 detailed in Sections 3.4 and 3.6, respectively, as well as different search heuristics in the solver. Furthermore, the model introduced in this section is more general than we actually need for our experiments 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

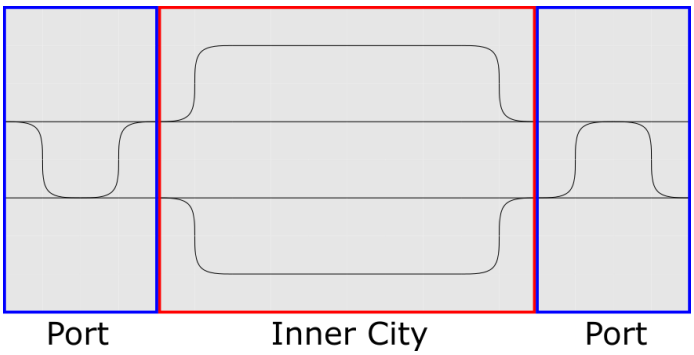


Figure 11: *Example of a generated city in flatland. The parallel tracks are used as starting or target points for the agents. Ports connect to neighbouring cities through long rail tracks.*

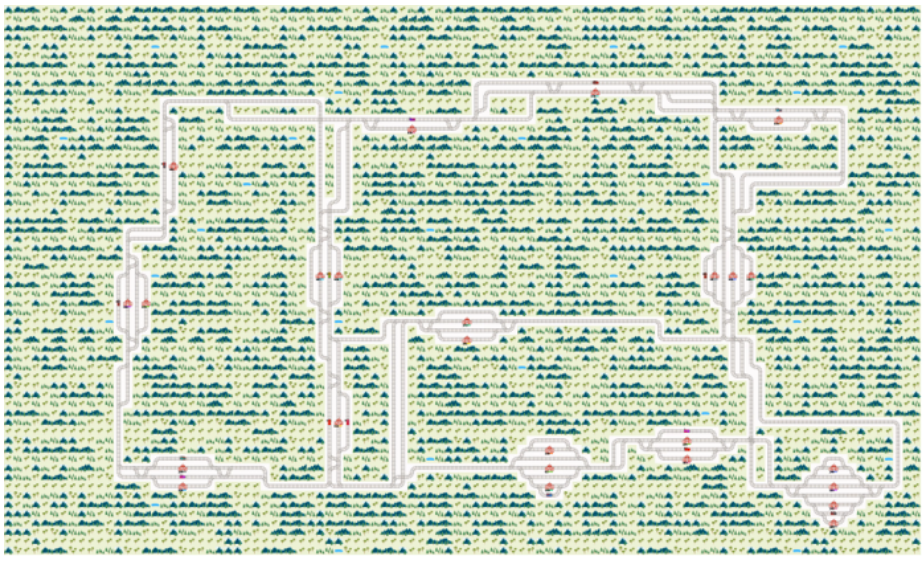


Figure 12: *Example infrastructure generated by sparse_rail_generator. The cities vary with number of parallel tracks as specified in the generator parameters. The complexity of the infrastructure can be enhances by reducing the number of available tracks within and between cities.*

- 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,
  - $m : E \to \mathbb{N}$ assigns the minimum travel time of an edge,
  - $a : R \to 2^E$ associates resources with edges in the railway network, and
  - $b : R \to \mathbb{N}$ gives the time a resource is blocked after it was accessed by a train line.
- $\mathcal{A}$ is an indexed set of train lines to be scheduled on network N. Each train $\mathcal{A}(a)$ for agent id $a \in \mathrm{dom}(\mathcal{A})$ is represented as a tuple $\mathcal{A}(a) = (S, L, e, l, w)$, where
  - (S, L) is an acyclic subgraph of (V, E),
  - $e : S \to \mathbb{N}$ gives the earliest time a train may arrive at a node,
  - $l : S \to \mathbb{N} \cup \{\infty\}$ gives the latest time a train may arrive at a node, and
  - $w : L \to \mathbb{N}$ 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.

We refer to [5] for more details. In this setting, edges can be interpreted as time slices of a train run referring to a certain 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 are abstract, there is no notion of location in general – resources might represent track sections that need to be reserved, but they may also be gates or sideway tracks or signals that need to be reserved while travelling the edge. Notice that different trains may have the same edges in their route DAG (directed acyclic

graph), which means that they can drive with the same speed profile at the same place.

We now define solutions: 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

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

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

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

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

We start schedule generation from 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 start schedule generation from the following elements:

- railway infrastructure $(\mathcal{C}, \mathcal{V}, \mathfrak{c}, \mathcal{E}$
- $\mathcal{A}$ of trains or agents;
  - each train $a \in \operatorname{dom} \mathcal{A}$ has a source $\sigma(a) \in \mathcal{V}$ and some target in $\tau(a) \subseteq \mathcal{V}^3$
  - a speed $v(a) \in [0, 1]$
  - a path restriction represented by acyclic subgraph $(\mathcal{V}_a, \mathcal{E}_a)$
- a release time r[4], 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.

In summary, we have $\mathcal{C}, \mathcal{V}, \mathfrak{c}, \mathcal{E}, \mathcal{A}, \sigma, \tau, v, r, \{ (\mathcal{V}_a, \mathcal{E}_a) \}_{a \in \mathcal{A}}$. We now show how a general train scheduling problem $(N, \tilde{\mathcal{A}}, C, F)$ can be derived:

- $\tilde{\mathcal{A}}$ consists of a tuple (S, L, e, l, w) for each $a \in \mathcal{A}$ where
  - $S = \mathcal{V}_a$

[3]The asymmetry comes from FLATland where we always start in a cell with a direction; however the target cell may be reached by any direction.

[4]Technically, we need a release time $r > 0$ if we want to be able to replay solutions in FLATland, which allows the next agent (in the order of agent indices) to enter a grid cell; with $r > 0$ we can force agents to stop and control by actions which agent will be the next to enter the cell.

- $L = \mathcal{E}_a$
- $e(v) = \min_{p:\sigma(a)\text{–}v \text{ path in } (\mathcal{V}_a,\mathcal{E}_a)} |p| \cdot v(a)^{-1}$
- $l(v) = \max_{p:v\text{–}\tau(a) \text{ path in } (\mathcal{V}_a,\mathcal{E}_a)} U - |p| \cdot v(a)^{-1}$
- $w(e) = v(a)^{-1}$ [5];

- $N = (V, E, R, m, a, b)$ where
  - $V = \bigcup_{a\in\mathcal{A}} V_a$
  - $E = \bigcup_{a\in\mathcal{A}} E_a$
  - $R = \mathcal{C}$
  - $m(e) = 0$, $e \in E$
  - $a(c) = \{e = (v_1, v_2) : \mathfrak{c}(v_1) = c\}$, $c \in R$
  - $b(c) = r$, $c \in R$;
- $C = \emptyset$;
- $F = \emptyset$.

Some remarks on this transformation:

- In our setting, we only have one resource per edge, i.e. we only reserve the train's own track, per cell. There is a $n : 1$ correspondence between edges and resources. In general, the correspondence can be $n : m$ (a train may reserve path ahead or signals along its position depending on its speed)
- 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.
- In our setting, b is the same for all resources.
- In our setting, the minimum running per cell (by abuse, $w(e) = v(a)^{-1}$ for all $e \in L$ for all agents $a \in \mathrm{dom}(\mathcal{A})$)
- In our setting, we do not have intermediate stops (by abuse, $m(e) = 0$ for all $e \in E$)
- In our setting, there are no connections nor collision-free resources.

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

The time windows given by e and l can be computed by Algorithm 1 and 2, respectively. They propagate the minimum running time forward from an initial set of earliest and backwards from an initial set of latest constraints. An illustration can be found in Figures 13 and 14, respectively. The time windows are thus given by

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

and

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

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:  Open ← F
2:  for v ∈ Open do
3:      for v' ∈ S − F : (v, v') ∈ L do
4:          if v' ∉ dom(e) then
5:              e(v') ← ∞
6:          end if
7:          e(v) ← min{ e(v'), e(v) + mrt }
8:          Open ← Open ∪ { v' }
9:      end for
10:     Open ← Open − { v }
11: end for
```

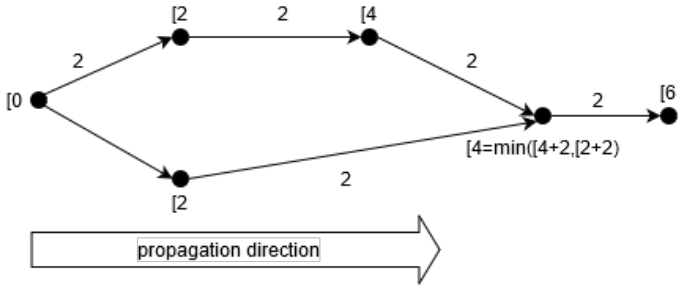


Figure 13: *Illustration of propagate_earliest for an agent a with speed $v(a) = \frac{1}{2}$ (i.e. $mrt = 2$).*

For schedule generation, we use a different objective function than for re-scheduling, namely minimizing

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

Algorithm 2 *propagate_latest*

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

```
1: Open ← F
2: for v ∈ Open do
3:     for v' ∈ S − F : (v', v) ∈ L do
4:         if v' ∉ dom(l) then
5:             l(v') ← −∞
6:         end if
7:         l(v) ← max{ l(v'), l(v) − mrt }
8:         Open ← Open ∪ { v' }
9:     end for
10:    Open ← Open − { v }
11: end for
```

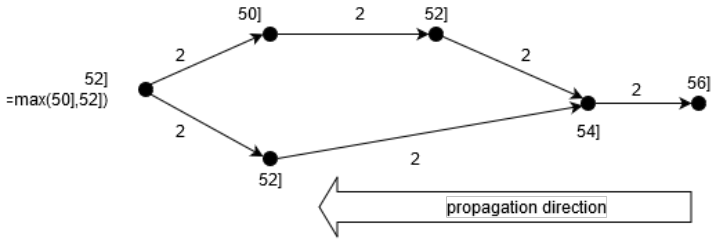


Figure 14: *Illustration of propagate_latest for an agent a with speed $v(a) = \frac{1}{2}$ (i.e. $mrt = 2$) and for upper bound $U = 56$.*

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,

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

and $\alpha = 2$ and $\delta = 4$, 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 once a delay has occured.

The choice of U represents a worst case: $w + h$ is an approximation of the longest possible path, $\frac{|\mathcal{A}|}{|S|}$ is the expected number of agents per city, hence the sum represents the time required for the last agent at the station if they all start one after the other and have constant speed; δ represents the minimum time required per cell for an agent running at minimum speed level $\frac{1}{4}$ (we use 4 levels $\frac{1}{i}$, $i = 1, \ldots, 4$) and α is a tolerance term that should allow for feasibility (there is no guarantee though).

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 abstract 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 many time steps the train will be delayed;
- $m_{agent} \in \text{dom}(\mathcal{A})$: the id of 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 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 $\delta_{step}, \delta_{cutoff}, \delta_{penalty}$ 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

We now look at the different scopes of Section 2.4 in more detail. All these scopers have the same interface, i.e. take the same parameters:

- an indexed set $\mathcal{A}$ of agents $\mathcal{A}(a) = (S, L, e, l, w)$ that needs to be restricted; in our setup, S, L will contain more routing alternatives than in the scheduling problem $\bar{\mathcal{A}}$, but e, v, w will be defined just the same as in $\bar{\mathcal{A}}$
- initial schedule (P_{S_0}, A_{S_0})
- malfunction $M = (m_{time\_step}, m_{duration}, m_{agent})$;
- agent speeds $v(a)$, $a \in \mathrm{dom}(\mathcal{A})$
- 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 will ensure feasibility of the re-scheduling problem, as we will discuss below.
- offline scopers will also receive the full re-schedule solution (P_S, A_S)

The result of the scoper will be a modification of $\mathcal{A}$; in the pseudo-code for the different scopers, we will not show w, which is constant $w(e) = v(a)^{-1}$ for all

$e \in L$. The railway network $N = (V, E, R, m, a, b)$ of the re-scheduling problem will be composed of the union of all these agents,

$$V = \bigcup_{(S,L,e,l,w)=\mathcal{A}(a),a\in\mathrm{dom}(\mathcal{A}} S, E = \bigcup_{(S,L,e,l,w)=\mathcal{A}(a),a\in\mathrm{dom}(\mathcal{A}} L, \tag{14}$$

and m, a, b the restrictions to E and used resources by E.

In our setting, each edge has exactly one resource; therefore, each vertex points at exactly one resource. Hence, the following definition is sound:

$$R(v) = r \text{ s.t. } (v, v') \in a(r) \tag{15}$$

Christian is there something missing here?

3.7.1 Scoping Re-scheduling (online_unrestricted)

We first describe the online_unrestricted scope, which does not restrict the problem scope, as a general train scheduling problem. Let $N = (V, E, R, m, a, b)$ be the network of the train scheduling problem, $S_0 = (P, A)$ be a solution to it and let $M = (m_{time\_step}, m_{duration}, m_{agent})$ be a malfunction.

For each train the *scoper_online_unrestricted* reduces the problem scope through 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 direct impact on the train and we do not allow the train to start earlier than scheduled.

Before describing the third case of the malfunction happening while the train is running (lines 12–13), 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 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). In *propagate*, we first remove the 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 (line 13). Notice that different vertices at the same resource may have different time windows e, l. Since in our simplified setting, we do not have intermediate stations, we do not require propagate to respect the schedule at intermediate stations, where we would not want trains to depart earlier than published to customers.

Algorithm 3 *scoper_online_unrestricted* for train $a \in \mathrm{dom}(\mathcal{A})$

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

if $\max_{v \in S} A(v) \leq m_{time\_step}$ **then**
  $S \leftarrow \emptyset$, $L \leftarrow \emptyset$
else if $\min_{v \in S} A(v) > m_{time\_step}$ **then**
  $v_1 \leftarrow \arg\min_{v \in S} A(v)$
  $e(v_1) \leftarrow A(v_1)$
  for $\tau \in S : out(\tau) = 0$ **do**
    $l(\tau) = U$
  end for
  $F_v \leftarrow F_e \leftarrow \{ v_1 \}$
  $F_l \leftarrow \{ \tau \in S : out(\tau) = 0 \}$
  $e, l, (S, L) \leftarrow propagate(e, l, (S, L), F_e, F_l, F_v, mrt, U, c)$
else
  $e, l, (S, L) \leftarrow scoper\_online\_unrestricted\_running((S, L), (P, A), M, mrt, U, c)$
end if

Algorithm 4 *propagate*

Input: $e, l, (S, L), F_e, F_l, F_v, mrt, U, c$
Output: $e, l, (S, L)$

for $v \in F_v$ **do**
  $S \leftarrow \{ v' \in S :$ there is a v–v' path or a v'–v path in $L \}$
  $L \leftarrow \{ (v, v') \in L : v, v' \in S \}$
end for
$e \leftarrow propagate\_earliest(e, (S, L), F_e, mrt)$
$l \leftarrow propagate\_latest(l, (S, L), F_l, mrt)$
if $c < \infty$ **then**
  for $v \in S - F_e$ **do**
    $l(v) \leftarrow \min\{ l(v), e(v) + c \}$
  end for
  $l \leftarrow propagate\_latest(l, (S, L), F_l, mrt)$
end if
$S \leftarrow \{ v \in S : e(v) \leq l(v) \}$, $L \leftarrow \{ (v, v') \in L : v, v' \in S \}$

We now describe the remaining case of Algorithm 3, namely when the malfunction happens while the train is running. We refer to Algorithm 5 for *scoper_online_unrestricted_running*: 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).

Algorithm 5 *scoper_online_unrestricted_running* for running train $a \in \text{dom}(\mathcal{A})$

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

$(v_1, v_2) \leftarrow (v_1, v_2) \in L$ s.t. $A(v_1) \leq m_{time\_step}$ and $A(v_2) > m_{time\_step}$
$e(v_1) \leftarrow A(v_1)$, $l(v_1) \leftarrow A(v_1)$
if $a = m_{agent}$ **then**
    $e_1(v_2) \leftarrow A(v_1) + mrt + m_{duration}$
else
    $e_1(v_2) \leftarrow A(v_1) + mrt$
end if
$F_e \leftarrow \{ v_1, v_2 \}$, $F_l \leftarrow \{ v_1 \} \cup \{ \tau \in S : out(\tau) = 0 \}$
for $\tau \in S - \{ v_1 \} : out(\tau) = 0$ **do**
    $l(\tau) \leftarrow U$
end for
$e, l, (S, L) \leftarrow propagate(e, l, (S, L), F_e, F_l, \{ v_1, v_2 \}, mrt, U, c)$

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)

The scoper offline_fully_restricted of Algorithm 6 takes a schedule as input and yields a scheduling problem which has exactly the same schedule as only solution. This allows to determine the solver overhead as a lower bound of re-scheduling computation time.

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 7 defines the "perfect" problem scope for a

Algorithm 6 *scoper_offline_fully_restricted* for train $a \in \mathrm{dom}(\mathcal{A})$

Input: (S, L), (P, A), $M = (m_{time\_step}, m_{duration}, m_{agent})$, mrt, U, c
Output: $e, l, (S, L)$
1: $S \leftarrow \{\, v : v \in P \,\}$
2: $L \leftarrow \{\, (v, v') \in P : v, v' \in S \,\}$
3: $e \leftarrow \{\, (v, P(v) : v \in S \,\}$
4: $l \leftarrow \{\, (v, P(v) : v \in S \,\}$

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

Algorithm 7 *scoper_offline_delta* for running train $a \in \mathrm{dom}(\mathcal{A})$

Input: (S, L), (P_{S_0}, A_{S_0}), $M = (m_{time\_step}, m_{duration}, m_{agent})$, mrt, U, c, (P_S, A_S)
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: $L_1 \leftarrow \{\, (v, v') \in P_S \text{ or } (v, v') \in P_{S_0} : v, v' \in S_1 \,\}$
5: **for** $v \in \Delta_A$ **do**
6: $\quad l(v) \leftarrow e(v) \leftarrow A_{S_0}(v)$
7: **end for**
8: **for** $v \in S_1 - \Delta_A : out(v) = 0$ **do**
9: $\quad l(v) \leftarrow U$
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: $\quad$ **if** $a = m_{agent}$ **then**
14: $\quad\quad e_1(v_2) \leftarrow A_{S_0}(v_1) + mrt + m_{duration}$
15: $\quad$ **else**
16: $\quad\quad e_1(v_2) \leftarrow A_{S_0}(v_1) + mrt$
17: $\quad$ **end if**
18: **end if**
19: $e, l, (S_1, L_1) \leftarrow propagate(e, l, (S_1, L_1), \Delta_A \cup \{\, v_2 \,\}, \Delta_A, \Delta_P, mrt, U, c)$

The solution space of this scoping contains the full re-scheduling solution if $c >= m_{duration}$, so the optimizer will find the same solution up to equivalence modulo cost.

3.7.4 "Weak" delta scoping (offline_delta_weak)

To investigate the potential of simple online scopers we define a "weaker" offline scoper, which defines the reduced problem scope a all vertices and passing times for all agent which have any difference between S and $S0$. The procedure of this scoper is explained in Algorithm 8. The scoper frees up all the variables of changed trains, i.e., if a train has some change somewhere, it is given full routing and time flexibility. In Algorithm 8, the changed agents are determined (line 1) and then the scope is reduced by Algorithm 9: all agents predicted as changed will have full re-scheduling degrees of freedom (lines 2–3), all others will be "freezed" to the original schedule (lines 6–10) or only forced to re-use the same route (lines 4–5), depending on whether $time\_flexibility$ is enabled or not.

Algorithm 8 *scoper_offline_delta_weak*

Input: $\mathcal{A}$, (P_{S_0}, A_{S_0}),$M = (m_{time\_step}, m_{duration}, m_{agent})$,$v$,$U$,$U$,$c$, (P_S, A_S)
Output: $\mathcal{A}$
1: $\mathcal{A}^* \leftarrow \{\, a \in \mathrm{dom}(\mathcal{A}) : P_{S_0} \neq P_S \text{ or } A_{S_0}(a,\cdot) \neq A_S(a,\cdot) \,\}$
2: $\mathcal{A}^* \leftarrow scoper\_changed(\mathcal{A}, (P_{S_0}, A_{S_0}), \mathcal{A}^*, time\_flexibility = \bot, v, U, c)$

Algorithm 9 *scoper_changed*

Input: $\mathcal{A}$, (P, A), $\mathcal{A}^* \subseteq \mathrm{dom}(\mathcal{A})$, $time\_flexibility$, M, v, U, c
Output: $\mathcal{A}$
1: **for** $a \in \mathrm{dom}(\mathcal{A})$ **do**
2: **if** $a \in \mathcal{A}^*$ **then**
3: $e, l, S, L \leftarrow scoper\_online\_unrestricted((S, L), (P, A), M, v^{-1}(a), U, c)$
4: **else if** $time\_flexibility$ **then**
5: $e, l, S, L \leftarrow scoper\_online\_path\_restricted((S, L), (P, A), M, v^{-1}(a), U, c)$
6: **else**
7: $e, l, S, L \leftarrow scoper\_online\_fully\_restricted((S, L), (P, A), M, v^{-1}(a), U, c$
8: **end if**
9: **for** $e \in L$ **do**
10: $w(e) \leftarrow v^{-1}$
11: **end for**
12: $\mathcal{A}(a) \leftarrow (S, L, e, v, w)$
13: **end for**

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

3.7.5 Route-restricted Re-scheduling (online_route_restricted)

Algorithm 10 scopes the re-scheduling problem to the routes of the schedule S_0, but allows time flexibility: after discarding all vertices and edges not on

scheduled path (lines 1–2), *propagate* makes the constraints consistent. If $c \geq m_{duration}$, the resulting re-scheduling problem is clearly feasible; however, we expect solution quality to deterioriate.

Algorithm 10 *scoper_online_route_restricted* for running train $a \in \mathrm{dom}(\mathcal{A})$

Input: (S, L), (P, A), $M = (m_{time\_step}, m_{duration}, m_{agent})$, mrt, U, c
Output: $e, l, (S, L)$
1: $S \leftarrow \{\, v : v \in P \,\}$
2: $L \leftarrow \{\, (v, v') \in P : v, v' \in S \,\}$
3: $e, l, (S, L) \leftarrow propagate(e, l, (S, L), \emptyset, \emptyset, \emptyset, mrt, U, c)$

3.7.6 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

$$E(a, v_i) = A(a, v_{i+1}) + r \tag{16}$$

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

Now we can have a look at Algorithm 11: 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 (line 3). We initialize the queue and the changed agents with the malfunction agent by 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) [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), compute the transmitted delay (line 17) and enqueue all vertices of the next agent scheduled after it is "hit" by the propagated delay (lines 18–20); 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 even if those vertices are before the common resource of a and a' (lines 18–20)[7]. A simple example shows why we need this, at least in the case of no

[6] If there are time reserves in the schedule, the propagated delay could be compensated by consuming this time reserve. In our implementation, schedules have almost no

[7] Line 19: can d' be reduced when the wave moves backward? Is this the reason for the bad performance of the scoper?

time flexiblity[8]: 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 infeasible; there may be many more such situations by transitivity. The output of this prediction can then be passed to Algorithm 9 with or without time flexibility.

Algorithm 11 *changed_transmission_chains*

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

Again, we conjecture [9] that this algorithms produces a feasible scope if $c >= malfunctionduration$, in both cases with or without time flexibility. We expect false negatives and false positives in our prediction: the delay may open up the opportunity for a second train to depart earlier (false positive), causing a third train to be only slightly delayed (false negative).

We call our delay propagation transmission chains because in the full implementation we not only store the delayed effect but the full chains. This makes debugging easier and allows to easily compute various statistics such as the level the source delay has first reached another agent.

[8] In the case of time flexibility (route restricted only), it might suffice to consider only the vertices of a' after the common resource $R(v)$ (line 18)

[9] We do not prove it formally. The rationale is that every agent hit by the wave is opened up and this should allow for feasibility. The wave, can also move backwards in time in the range of the propagated delay reflecting a spill-back of congestion.

Our approach is similar to delay propagation in [6] where delay propagation is illustrated in the setting of event activity graphs with given decisions.

We could optimize the pseudo-code by ensuring that only the largest delay of an agent at a resource is in the queue; we have refrained from this to keep the exposition simple.

3.7.7 Random Online Re-scheduling (online_random)

As a sanity check, to show that our scoping is not trivial, we use Algorithm 9 with random prediction of Algorithm 12: 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).

Algorithm 12 *scoper_offline_random*

Input: $\mathcal{A}$, (P_{S_0}, A_{S_0}),$M = (m_{time\_step}, m_{duration}, m_{agent})$,$v$,$U$,$U$,$c$, (P_S, A_S)
Output: $\mathcal{A}$

1: $\mathcal{A}_{running} = \{\, a \in \mathcal{A} : \max_{v \in P_{S_0}} A_{S_0}(a, v) \geq m_{time\_step} \,\}$
2: $n_{changed} = |\{\, a \in \mathcal{A}_{running} : P_{S_0} \neq P_S \text{ or } A_{S_0}(a, \cdot) \neq A_S(a, \cdot) \,\}|$
3: $\mathcal{A}^* \leftarrow$ choose $n_{changed}$ from $\mathcal{A}_{running}$
4: $\mathcal{A}^* \leftarrow scoper\_changed(\mathcal{A}, (P_{S_0}, A_{S_0}), \mathcal{A}^*, time\_flexibility = \top, v, U, c)$

The output of this prediction can then be passed to Algorithm 9 with time flexibility to determine the re-scheduling problem. Again, we conjecture that this algorithms produces a feasible scope if $c >= m_{duration}$ (without time flexibility, there is clearly no guarantee of feasibility).

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.

In order to show the speed-up, we want our problems to cover a good range of problems. These considerations lead us to the following agenda for this section, as shown in Figure 16. The full parameter ranges are shown in Appendix B.

We take our baseline (online unrestricted) as measure of the hardness of our problems; in Figure 15, we see that our experiments are biased to the left, i.e. it is hard to choose parameters such the problems get harder – we see no correlation between the experiment id (which reflects the infrastructure size) and online unrestricted.

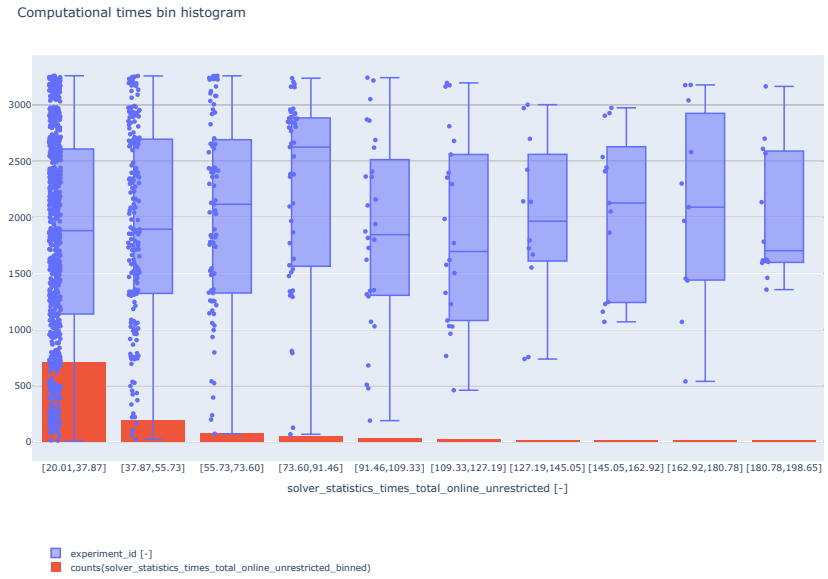


Figure 15: *Histogram of experiments in 10 equi-distant bins, filtered on data with* `solver_statistics_times_total_online_unrestricted` *between* $20s$ *and* $200s$.

We filter out experiments with small and large run times (if the solver took less than $20s$ or more than $200s$ for the online unrestricted scope; this leaves us with 125 data points.

4.2 Speed-Up and Solution Quality

In Figure 17, we do not recognize any block structure of running times as we could expect from Figure 16. In other words, grid size and number of agents are 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);

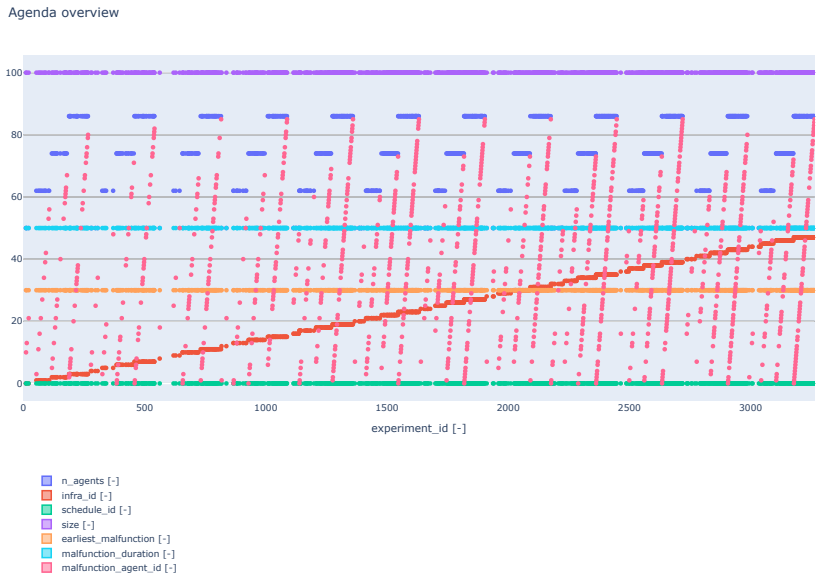


Figure 16: *Hierarchical Agenda Structure.*

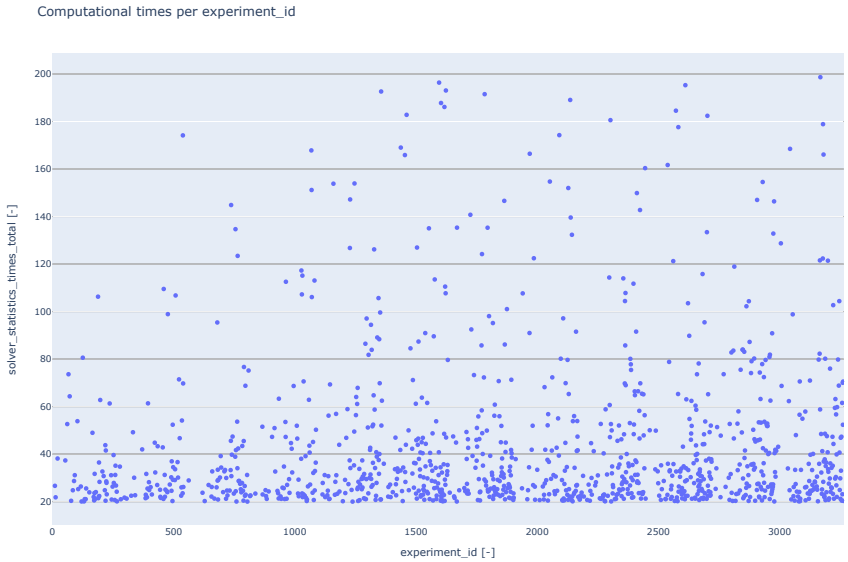


Figure 17: *Computation times per experiment.*

also other indicators as the number of agents running after the malfunction or the time horizon remaining after the malfunction did not give any hint on re-scheduling time (online unrestricted). We will come back to this in Section 5, where we give some illustrations.

Therefore, we for lack of a better take the full re-scheduling time (online unrestricted) as measure of problem complexity and plot the rescheduling times and speed-up for the different scopes against the full re-scheduling time in Figures 18, 19 and 20. We see a clear separation and increase of speed-up for larger online unrestricted in the following order:

1. online fully restricted
2. offline delta
3. online route restricted

We see a small speed-up for random and none for our transmission chains approach. We believe the small speed-up for random comes from randomly choosing agents whose paths are restricted.

4.3 Solution Quality

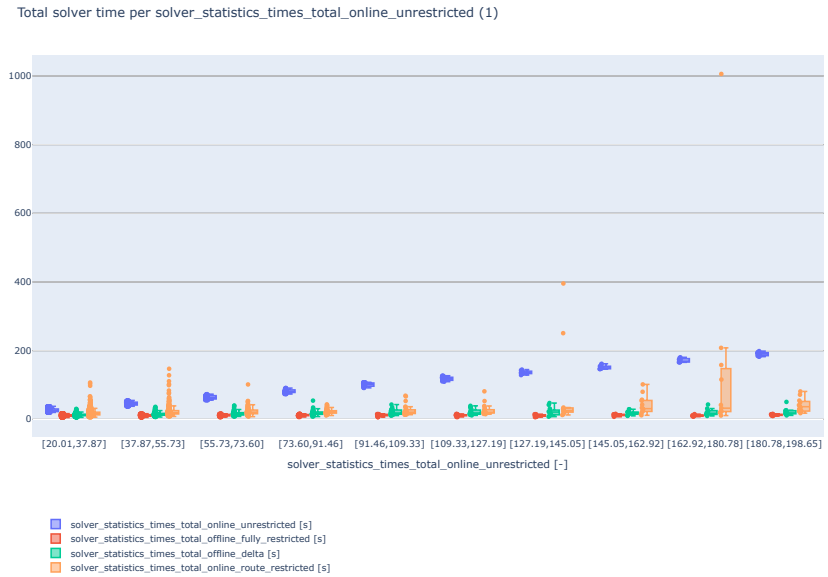
Figure 21 shows that, as expected, online fully restricted, offline delta yield a solution of the same costs; our transmission chains route restricted also yields the same quality solutions in our experiments[10], only when give the predicted agents no time flexibility (transmission chains fully restricted) does the solution quality slightly deterioriate. Online route restricted incurs additional costs, although it cannot have any routing penalties; the same applies to online random, which randomly chooses the agents having no routing flexibility; our transmission chains scopers seem to lead to good solutions (no additional costs for route restricted, almost no additional costs for fully restricted). There seems to be almost no tradeoff between routing penalties and lateness – we will come back to this in Section 5.

4.4 Prediction Quality

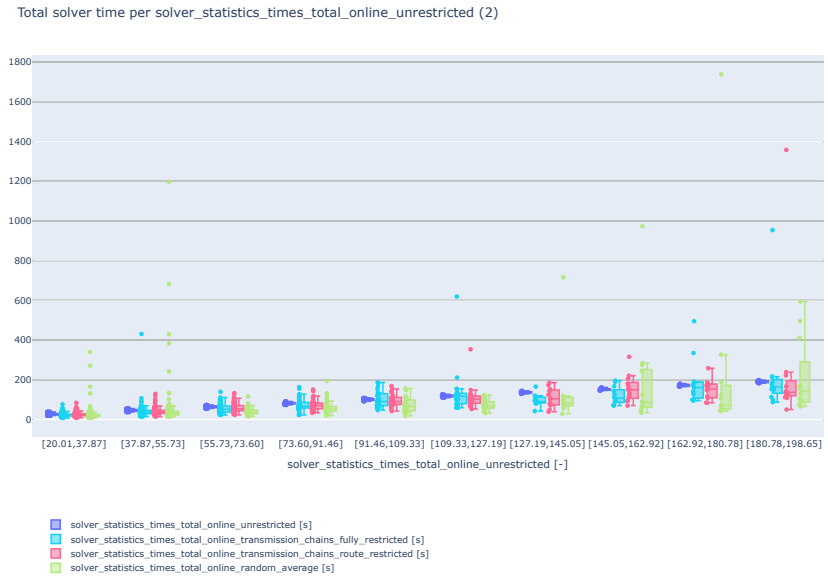
Figure 22 shows prediction quality for transmission chains and random: our transmission chains scoper has often many false positives, but almost no false negatives

Figure 23 shows that online route restricted compensates route restriction by changing more agents than all other scopes, however some experiments seem not to allow for such compensation and less agents are changed at still higher costs as we saw above. It is interesting to see that offline delta changes fewer agents, at the same cost as we saw above – we will come back to this below in Section 5.1.

[10] we do not know whether this is provably so

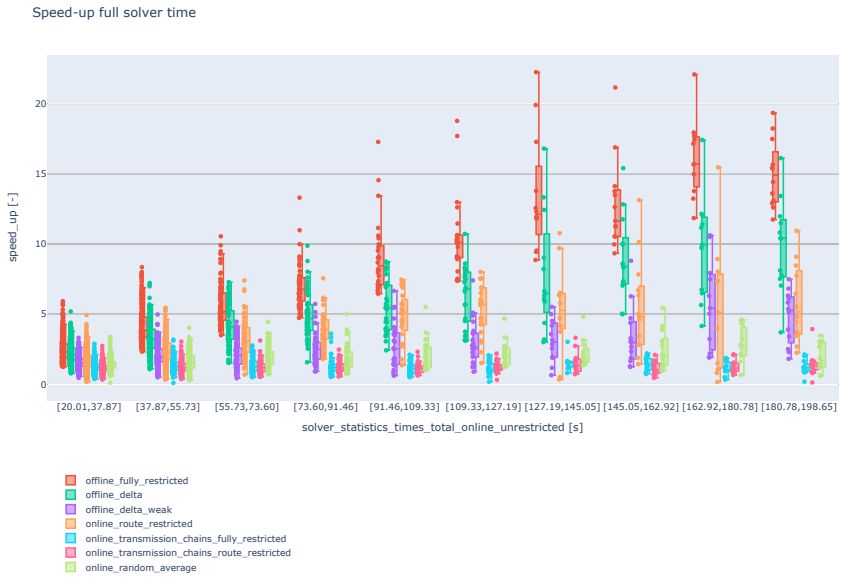


(a) *Total solver runtimes for online unrestricted, online full restricted, offline delta and online route restricted*

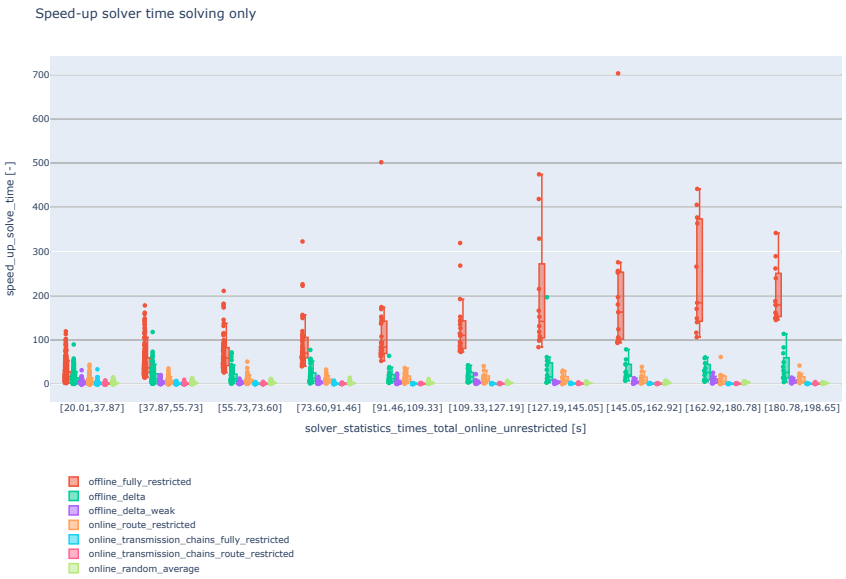


(b) *Total solver runtimes for online transmissions chains and online random against online unrestricted.*

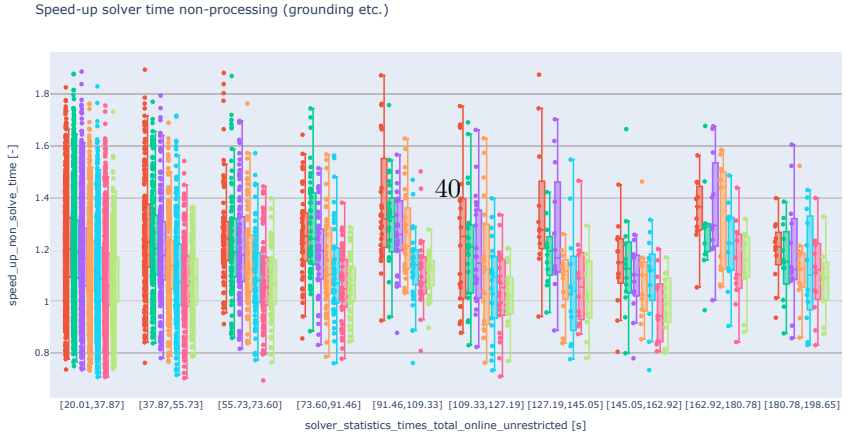
Figure 18: *Re-scheduling times against full re-scheduling time.*



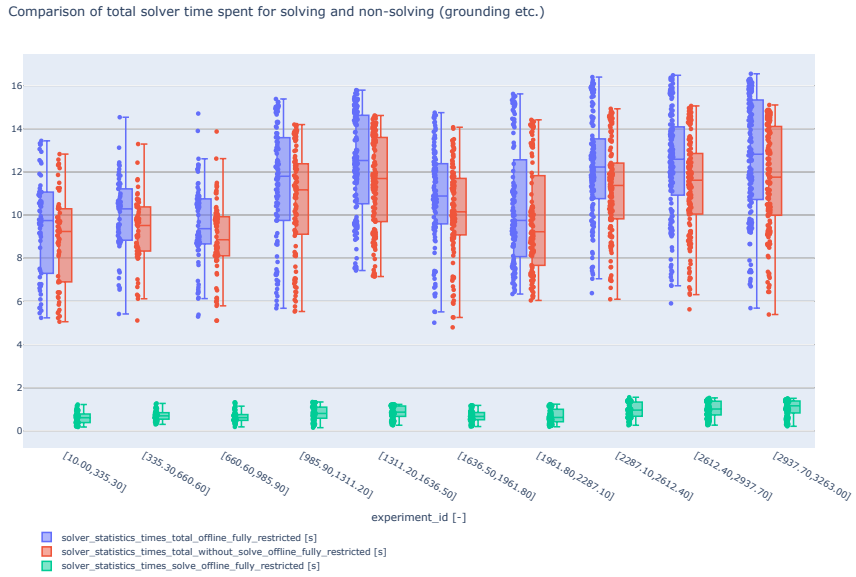
(a) *Speed-up for total solver time*



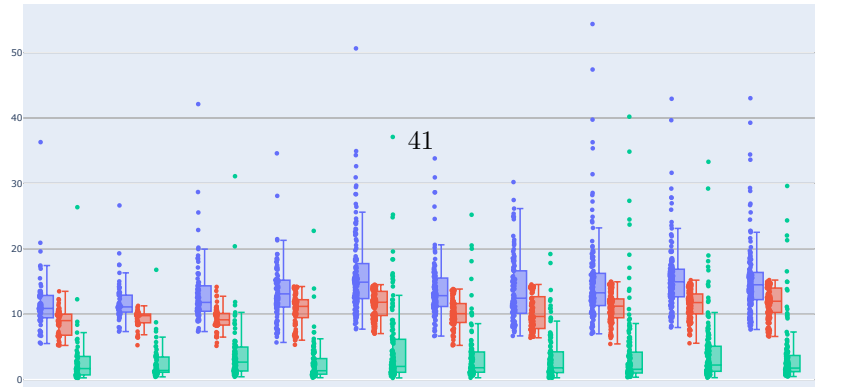
(b) *Speed-up for solver time spent solving*



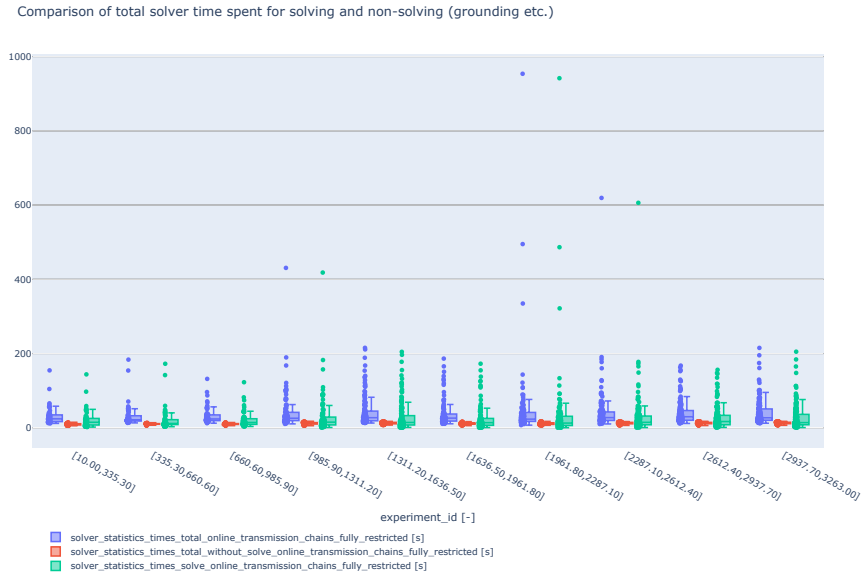
(a) *Total solver time, solve time and non-solve time for online unrestricted*

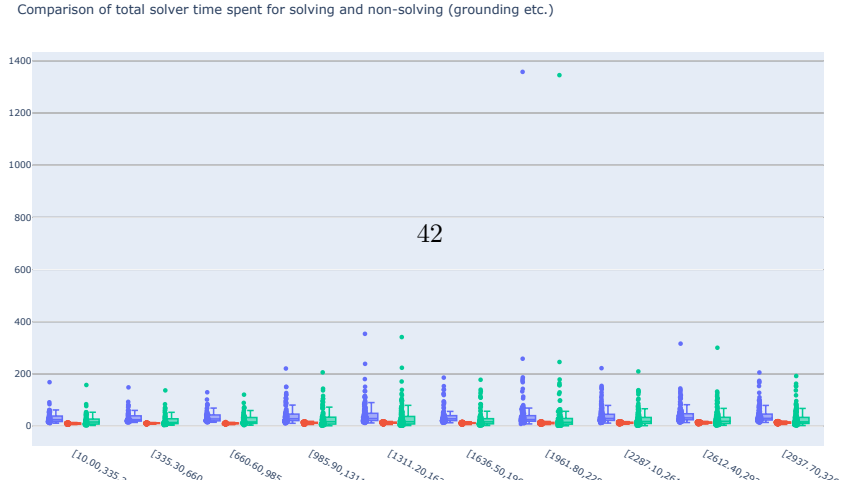


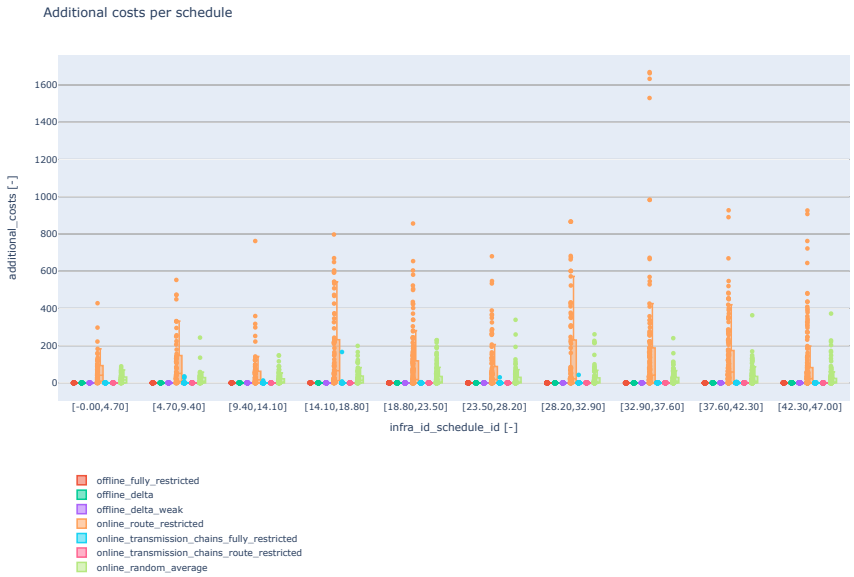
(b) *Total solver time, solve time and non-solve time for online fully restricted*



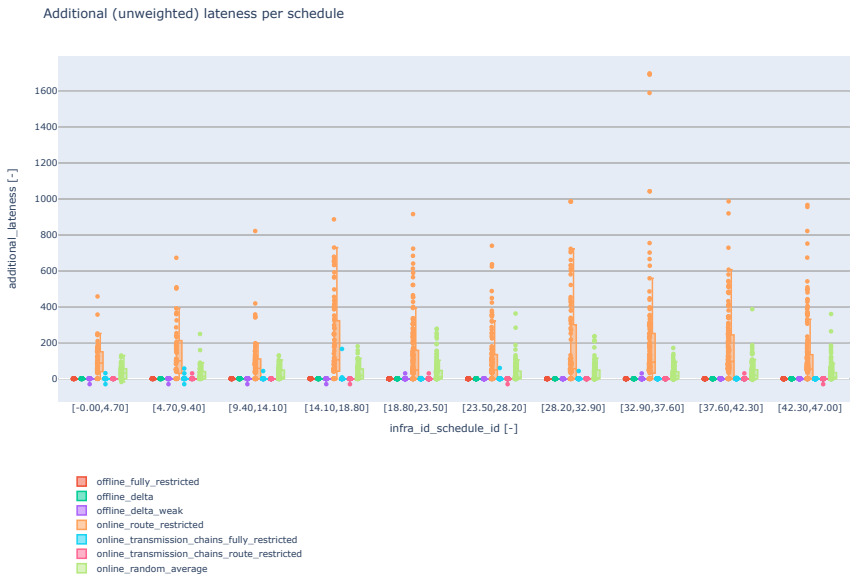
(e) Total solver time, solve time and non-solve time for online random



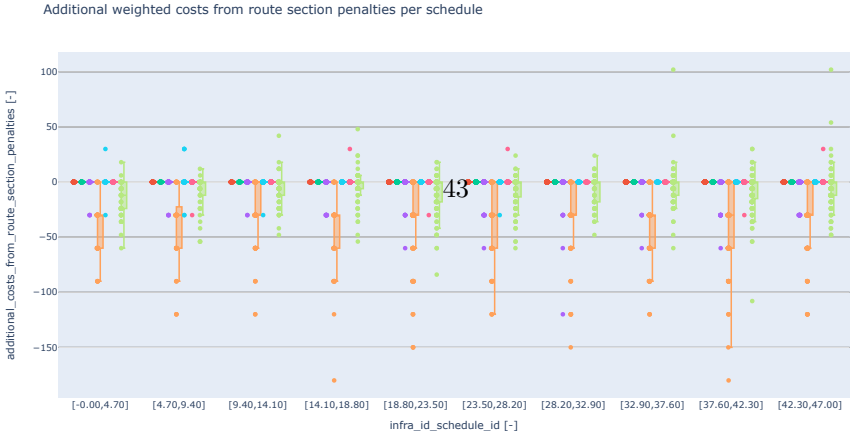




(a) *Additional costs*



(b) *Additional lateness*



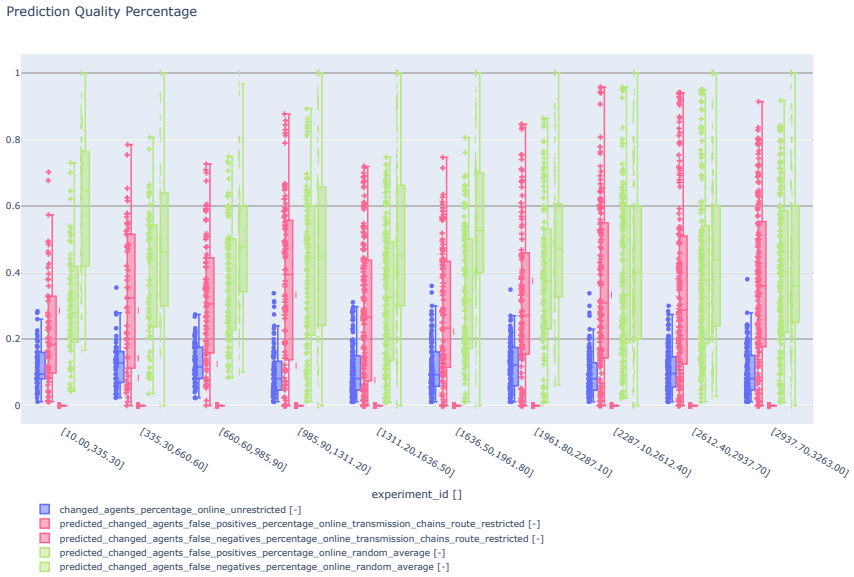


Figure 22: *Prediction quality: false positives and false negatives rate. Online transmission chains fully restricted is not shown as it is almost completely identical to online transmission chains route restricted.*

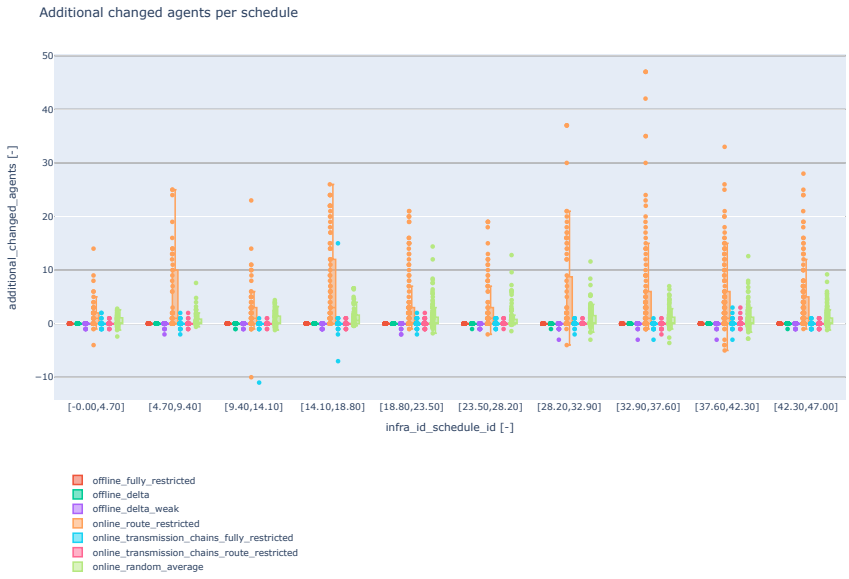


Figure 23: *Additional changed agents.*

4.5 Discussion

On the one hand, the results are good news for our approach since it supports our hypothesis that a huge speed-up is possible, especially if we ignore the non-solve part (we haven't invested in optimizing this part of the model and guess what it could be improved with the ASP solver and that it behaves completely differently with other solvers).

On the other hand, the results must make as also cautious as we compare our transmission chains approach with online route restricted: our better prediction seems to make it harder for the solver to find solutions in some cases, resulting in much more variance in re-scheduling times than offline delta.

5 Case Studies

5.1 Cost Equivalence

TODO

5.2 False Positives and False Negatives

TODO take example of false positive and false negative from online transmission chains

5.3 Outlier Example

TODO

- very high online unrestricted
- very high random average with respect to its bin
- ...

5.4 Malfunction Variation

TODO take a schedule and agent and vary the time of malfunction: show monotone decreasing effect (number of agents affected, re-scheduling time)

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

TODO ML delay propagation / recourse / stability of iterative and batch

processing simulation OR / heuristics (job insertion) decomposition approaches real-time rescheduling

7 Conclusion

G0 We have a given a formal description of problem scope reduction.

G1 We have given multiple illustration of the malfunction spread and its non-monotonicity in time and space. We therefore think that it is more natural to consider the affected scope in the global scope than to decompose the global problem into geographic regions and iteratively solve the global problem by adapting boundary conditions which make the subproblems independent. However, we are not sure whether scope reduction is possible if

- the schedule becomes very dense everywhere (delays can nowhere be absorbed or compensated) and the problem therefore might not separate well
- the schedule loses regularity and we might therefore not be able to use historic data for scope reduction (we have not tackled

G2 We have shown promising speed-ups with the ASP solver model, however we suspect that behaviour could be significantly different with other solvers (for instance, solvers which do not have an initial grounding phase or solvers that exploit the graph-temporal problem structure better); however, we conjecture that scoping in time and space rather than decomposition could enhance solver times for other techniques as well.

G3 We have shown that the scoping problem is not trivial, by implementing a plausible scoper; however, we have not shown that it is possible to build an efficient online scoper.

G4 We provide an extensible playground open-source implementation. The code quality is acceptable, but lacks the quality to be used as a library in a black box way. We aim at turning our approach into a coding challenge; this could happen

- as part of the RSP pipeline by generating historic data (multiple malfunction and their re-scheduling) and trying to build an online scoper based on historic data, which not only memorizes the historic solution;
- as part of FLATland by adding an initial schedule in the sense of a conflict-free detailed operational plan and penalizing delay with respect to the initial schedule, still focussing on a trade-off between computation time and solution quality.

A Model Calibration

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

A.1 effect of SEQ heuristic (SIM-167)

A.2 effect of delay model resolution with 2, 5, 10 (SIM-542)

A.3 effect of –propagate (SIM-543)

B Experiment Parameters

Here we give the full parameter ranges for the results of Section 4. Values correspond to `https://github.com/SchweizerischeBundesbahnen/rsp-data/tree/master/PUBLICATION_DATA`, generated by `https://github.com/SchweizerischeBundesbahnen/rsp/blob/3c92f19f69abc1d8ea5bfc344526cb971e75a118/src/python/rsp/hypothesis_one_experiments_potassco.py`. The value ranges $[a, b, n]$ have the following meaning: if $n = 1$, we take only a; if $n > 1$, we take $n - 1$ points from the half-open interval $[a, b)$ of step size $step = \lfloor \frac{b-a}{a} \rfloor$; in the following tables, we abbreviate 100 instead for $[100, 100, 1]$[11].

[11] We should probably harmonize the implementation and take n points from $[a, b]$ with step size *step* instead.

B.1 Infrastructure Parameters

Parameter	Symbol	Description	Value
infra_id	–	infrastructure id	–
width	w	Number of cells in the width of the environment	100
height	h	Number of cells in the height of the environment	100
flatland_seed_value	–	Random seed to generate different configurations	190
max_num_cities	$\|S\|$	Maximum number of cities to be places in the environment. Cities are the only places where agents can start or end their journey. Cities consists of parallel track and entry/exit ports.	$[8, 15, 3]$
grid_mode	–		False
max_rail_between_cities	–	Maximum number of parallel track at entry/exit ports of the cities	$[1, 2, 2]$
max_rail_in_city	–	Maximum number of parallel tracks in the city	$[2, 3, 2]$
number_of_agents	$\|\mathcal{A}\|$		$[50, 100, 4]$
speed_data	$v : \mathcal{A} \rightarrow [0, 1]$	distrbution of speeds among agents	{ 1.0: 0.25, 1.0 / 2.0: 0.25, 1.0 / 3.0: 0.25, 1.0 / 4.0: 0.25, }
number_of_shortest_paths _per_agent	$\approx (\mathcal{V}_a, \mathcal{E}_a)$	We compute shortest paths only once; should be larger than the number in scheduling and re-scheduling.	10

B.2 Schedule Parameters

Parameter	Symbol	Description	Value
infra_id	–	reference to infrastructure	–
schedule_id	–	schedule id	–
asp_seed_value	–	Since we use 2 threads, the ASP solver behaves non-deterministically, so the seed value has no effect.	814
number_of_shortest_paths_per_agent_schedule	$\approx (\mathcal{V}_a, \mathcal{E}_a)$		1

B.3 Reschedule Parameters

Parameter	Symbol	Description	Value
earliest_malfunction	$m_{earliest}$	Used to determine m_{time_step}, the time step of the malfunction.	30
malfunction_duration	$m_{duration}$	Malfunction duration.	50
malfunction_agent_id	m_{agent}	Which agent is disturbed?	$[0, 200, 200]$
number_of_shortest_paths_per_agent	$\approx (S, L)$	Defines the route restrictions for the agents.	10
max_window_size_from_earliest	c	Truncate window sizes to this time window sizes to this maximum. Applies to windows on vertices and not on resources.	30
asp_seed_value	–	Since we use 2 threads, the ASP solver behaves non-deterministically, so the seed value has no effect.	99
weight_route_change	ρ	How many time steps delay is one route change equivalent to?	30
weight_lateness_seconds	$\delta_{penalty}$	Factor to scale delays. Should only be different than 1 if weight_route_change is equal to 1, and vice-versa.	1

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.

[6] Erwin Abbink, Andreas Bärmann, Nikola Bešinovic, Markus Bohlin, Valentina Cacchiani, Gabrio Caimi, Stefano de Fabris, Twan Dollevoet, Frank Fischer, Armin Fügenschuh, Laura Galli, Rob M.P. Goverde, Ronny Hansmann, Henning Homfeld, Dennis Huisman, Marc Johann, Torsten Klug, Johanna Törnquist Krasemann, Leo Kroon, Leonardo Lamorgese, Frauke Liers, Carlo Mannino, Giorgio Medeossi, Dario Pacciarelli, Markus Reuther, Thomas Schlechte, Marie Schmidt, Anita Schöbel, Hanno Schülldorf, Anke Stieber, Sebastian Stiller, Paolo Toth, and Uwe T. Zimmermann. *Handbook of Optimization in the Railway Industry*, volume 268. 2018.