SCOPE RESTRICTION FOR SCALABLE REAL-TIME RAILWAY RESCHEDULING: AN EXPLORATORY STUDY

Erik Nygren[*]
erik@nygren.ch,

Christian Eichenberger[*]
christianmarkus@oakmountain.ch,

Emma Frejinger
emma.frejinger@unmontreal.ca

ABSTRACT

With the aim to stimulate future research, we describe an exploratory study of a railway rescheduling problem. A widely used approach in practice and state of the art is to decompose these complex problems by geographical scope. Instead, we propose defining a *core problem* that restricts a rescheduling problem in response to a disturbance to only trains that need to be rescheduled, hence restricting the scope in both time and space. In this context, the difficulty resides in defining a *scoper* that can predict a subset of train services that will be affected by a given disturbance. We report preliminary results using the Flatland simulation environment that highlights the potential and challenges of this idea. We provide an extensible playground open-source implementation based on the Flatland railway environment and Answer-Set Programming.

Keywords: Railway traffic management, rescheduling, Flatland Simulation Environment

1 Introduction

Railways face a multitude of planning problems, the most important ones can be categorized as routing or scheduling problems (Cordeau et al., 1998). Train scheduling, also known as train timetabling, is a tactical planning problem that has been extensively studied in the literature (e.g., Cacchiani et al., 2015, Caprara et al., 2002). A schedule is in place over a certain period of time (e.g., a season) and often only minor changes are made over different periods. These schedules should be robust to minor perturbations. More important perturbations referred to as disturbances (e.g., Cacchiani and Toth, 2012, Liu and Dessouky, 2019) may induce primary delay on directly affected trains which, in turn, can propagate to other trains in the network (secondary delay). Disturbances can be resolved by, e.g., changing the impacted trains' schedule or routes. Severe perturbations – referred to as disruptions – caused by, for example, extreme weather, infrastructure or equipment failures may in addition require rescheduling of rolling stock or crew.

The focus of our work lies on real-time railway traffic management (aka traffic control, train dispatching or train rescheduling). The purpose is to identify mitigating strategies in short computing time so as to minimize the propagation caused by disturbances and recover the original schedule – henceforth real-time rescheduling (or simply rescheduling).

There is an extensive literature on this topic and an exhaustive literature is out of the scope of this paper. Instead, we refer the reader to Luan (2019) for a recent review as well as surveys (e.g., Ahuja et al., 2005, Lusby et al., 2011). Real-time railway traffic management is challenging because (i) the computing time budget is very restricted, and (ii) controlling trains requires a detailed representation of the system (so-called microscopic level) that considers block sections and signals. For these reasons, it is not possible to solve the full traffic management problem for real railway networks in real time at a microscopic level. Hence the problem is typically decomposed.

Luan et al. (2018) describe three decomposition methods of a time-space graph used in the literature: geography-based, train-based and time interval based. However, decomposition introduces a challenging coordination problem. Consider, for example, a decomposition according to geographical scope into different areas. In this case the decisions

[*]Work done while at Swiss Federal Railways (SBB).

between the different areas need to be coordinated, e.g., through conditions at the boarders. Corman et al. (2012) study this coordination problem. A difficult challenge resides in striking the right balance between the size of the areas and the difficulty of the coordination problem. In particular the areas of interest can change depending on the nature and location of the disturbance. Van Thielen et al. (2018) focus on conflict prevention and they propose a heuristic to dynamically define so-called impact zones.

In this work, we outline ideas for a solution approach that consists of a problem scope reduction step before solving a restricted traffic rescheduling problem. We conjecture that any rescheduling problem arising within the entire system can be reduced to a much smaller problem that specifies which subset of trains and which parts of the scheduled trainruns might be re-scheduled (adjusted). We refer to this as the *core problem*. The definition of the core problem depends on the perturbation and can vary in both geographical and time dimensions. The objective is to dynamically reduce the scope of the full rescheduling problem so that it can be solved within the limited computing time budget while reducing the need for coordination (or avoiding it all together). This idea is closely related dynamic impact zones in Van Thielen et al. (2018). However, the core problem is a more general concept and does not necessarily correspond to a given zone.

The following observations on the rescheduling problem constitute the rationale for our work. First, the infrastructure (railway network) typically does not change over time. Second, train schedules are slow changing (often only minor changes are made from one schedule to the next) and a given schedule is in place for an extended period of time. Third, in practice, the problem is solved by experienced humans. The rescheduling problem hence arises in highly structured situations where experience from the past is valuable to find good-quality solutions. Therefore, we hypothesize that machine learning (ML) can be used to efficiently restrict the rescheduling problem scope.

The methodology can be viewed as a pre-processing phase that is somewhat generic to the optimization model and solution approach of the rescheduling problem. For a given disturbance, the goal is to define the core problem, i.e., the sub-problem affected by the perturbation. In our context the scope consists of trains' passing times and routes. The rescheduling problem is then solved keeping the solution fixed for all variables outside the scope of the core problem. This hence results in a heuristic solution approach. Assuming access to historical data or a simulator, we propose to learn from data how to restrict the scope to the core problem for any given perturbation. Based on the aforementioned observations, the scope restriction problem is a prime candidate for ML. However, in this first exploratory work whose purpose is to assess the potential of the approach, we resort to simple heuristics.

The motivation for our work is supported by the findings in Fischetti and Monaci (2017) in the domain of operations research (OR). They investigate if general-purpose Mixed-Integer Linear Programming solvers can successfully solve real-time rescheduling problems in an effective way, as opposed to using tailored heuristics. They show that thanks to simple pre-processing heuristics they can solve most of the benchmark instances within the given time budget. There are only two pre-processing operations: heuristic bound tightening and heuristic variable fixing. The idea we put forward here can be viewed as heuristic variable fixing.

Further support comes from Li et al. (2021) in the domain of ML, who address scalability of combinatorial optimization problems by learning to identify smaller sub-problems which can be readily solved by existing methods in a black-box manner. In contrast to our setting, they start from a feasible solution and put emphasis on an iterative solution improvement, whereas we start from an "almost feasible" solution corrupted only by a disturbance and do not consider the iterative setting. In addition, they also present a Transformer-based architecture for learning sub-problem selection, whereas we only show that speed-up is possible in the context of railway rescheduling problems. In the terminology of Bengio et al. (2020), our approach puts forward *learning to configure algorithms*, i.e. augmenting an operation research algorithm with valuable pieces of information (i.e. the scope restriction).

The contributions of this paper are:

- We introduce the scope restriction problem in railway re-scheduling, whose goal is to identify a core sub-problem in space and time, which can be be passed to an existing (general-purpose) solver.
- We assess the potential of scope restriction in an exploratory numerical study using a simulator (Flatland toolbox, Mohanty et al., 2020) and one specific optimization model and solver. Our exploratory results show that a significant computation time reduction for railway re-scheduling can be achieved when the problem scope can be reduced reliably.
- The main objective with sharing this preliminary work is to encourage future research on the topic. We contribute to such work by providing and describing an extensible playground implementation in our public GitHub repo (see *Data and Code Availability* below).

The remainder of the paper is organized as follows: In Section 2, we provide the real-world context by describing the train rescheduling process at the Swiss Federal Railways (SBB). We describe scope reduction in detail in Section 3.

Sections 4 and 5 are respectively dedicated to describing the synthetic environment and the computational results. Finally, Section 6 concludes.

2 Real-world Context: Train Rescheduling at the Swiss Federal Railways

The goal of this section is to describe traffic management in practice and use Swiss Federal Railways (SBB) as an illustration. 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 (Swiss Federal Railways, 2020a). SBB hence faces challenging rescheduling problems and we use this case to motivate the relevance of our work.

2.1 Planning Horizon and Control Loop

Planning and rescheduling is done at multiple time horizons with their different organizational responsibilities. We depict a simplified view of a rescheduling control loop in Figure 1 (adapted from Swiss Federal Railways, 2020a,b). As we further detail in the following when describing the figure, there are three main areas involved: planning, dispatching and operations.

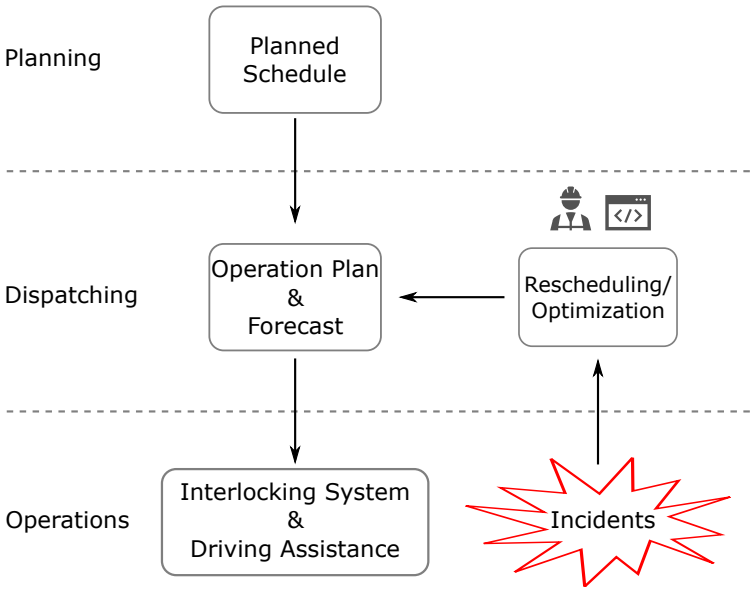


Figure 1: *Simplified schema of the rescheduling process during operations.*

The *planned schedule* is the customer interface of the schedule, i.e. the published schedule giving a table of train departures and arrivals that can be used or booked by passengers or train paths that can be reserved by railway undertakings for transportation of goods.

Dispatchers have access to information about the current state, and an automated forecast and they make decisions at the level of the operational plan:

Operational plan is the railway infrastructure managers internal view of mapping railway services to infrastructure at the microscopic level in the sense of interlocking systems. Over a certain time-horizon ahead of production, the operation plan needs to be conflict-free, ensure mutual exclusion of infrastructure usage and additional safety time buffers. To ensure communicability and to ensure service level agreements, decisions should stay closely in agreement with the planned schedule. At SBB, most of the interlockings can be controlled remotely in four operations centers. However, as the planning process is still largely manual and software tool support is limited to graphical editing and feasibility checking is not fully implemented in the IT systems, the planned schedule has no fully conflict-free associated operational plan. Furthermore, there is no global time horizon for the operational plan, conflicts are therefore resolved iteratively. Nevertheless, in the experiments in this paper, we will assume a conflict-free operational plan.

Forecasts. Given status updates based on incident reports, there is a forecast of differences and conflicts with respect to the operation plan. Note that not every difference needs to be conflicting, as there may be enough buffer times between trains.

The dispatchers' decisions – the operational plan – are translated either automatically or by dedicated dispatching team members to the interlocking system and driving assistance systems. The instructions for the interlocking systems can be time-dependent or event-dependent (for instance, give way to train X after train Y has left). At SBB, most of the operation plan is automatically compiled into interlocking instructions. Whereas the interlocking system must be fail-safe and comply with highest safefty standards, the the driving system assistance has only limited safety integrity level, and it only issues recommendations to drivers who remain fully responsible for the actions they take (Swiss Federal Railways, 2020a).

Incidents (disturbances) are reported as status updates that express small deviations from the operational plan stemming from driving decisions (braking, accelerating), from passenger boarding and alighting, signal box failures, etc. In this work, we do not consider disruptions such as temporary removal of infrastructure, cancellation of trains or connections.

Over time, new elements from the planned schedule are introduced to the operation plan (rolling time horizon) and the planned schedule also imposes constraints on re-scheduling. For instance, passenger trains must not depart earlier than communicated to customers and the service intention may define further hard or soft constraints such as specific penalties for delay or train dropping.

Rescheduling/optimization is the adaptation of the operational plan. It is done both after the update of the forecast based on incidents in the infrastructure, and the unrolling of the time horizon by including future planned schedule into the operational plan. As long as no other trains are affected, the forecast can be directly integrated into the operational plan. The time available to produce a conflict-free solution is typically restricted to a few seconds.

There exist advanced traffic management systems for efficiently re-computing a traffic forecast for a given time horizon, allowing dispatchers to detect potential conflicts ahead of time. However, the predicted conflicts mostly have to be resolved by humans by explicitly deciding on re-ordering or re-routing of trains based on their experience (Van Thielen et al., 2018).

2.2 Hierarchical Organization into Areas of Responsibility

Due to the complexity of a railway network, the re-scheduling task is decomposed into smaller geographical areas within which human dispatchers optimize traffic flow. This leads to a multi-level hierarchy of operation centers. In turn, it vastly reduces the complexity within each area and allow human dispatchers to resolve conflicts locally. We show an example Figure 2 with five areas (A–E). We intentionally make the areas overlap to illustrate the fact that there is coordination between the areas. Such inter-area coordination is typically done by informal means of communication such as telephone and supported by automated traffic forecasting software. Moreover, dispatchers have access to complete information and can take actions in other regions in agreement with the respective responsible dispatchers.

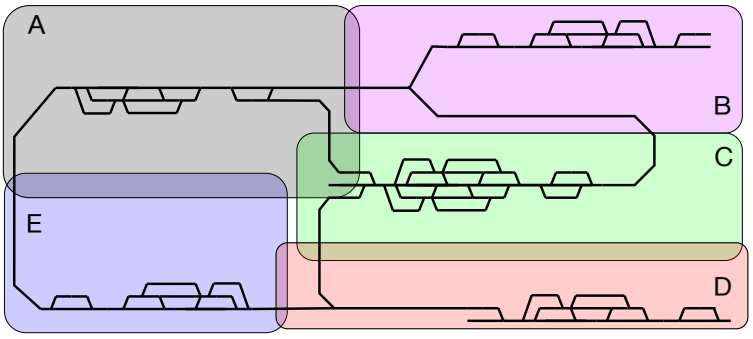


Figure 2: *Illustration of geographical decomposition*

2.3 Deployment of Algorithmic Online Re-Scheduling

At SBB, only very limited areas are fully controlled algorithmically (Caimi et al., 2012, Fuchsberger, 2012, Swiss Federal Railways, 2020a). These are small zones around hubs and in tunnels (so-called condensation zones in Caimi,

2009) and current implementations can handle up to one million train re-ordering conflicts per day which are fully resolved by the rail control system (Swiss Federal Railways, 2020a). The goal of such condensation zones is to exploit the hub infrastructure such as approaching areas ahead of major stations or tunnels at capacity. This means, for instance, that trains should enter and pass through the critical sections at full speed.

Railway networks run on expensive infrastructure and thus an optimal utilization of the given resources is crucial to keep transportation costs low. Therefore, automatic control is expected to be deployed to more areas and those areas are expected to grow; thereby, time reserves for catching up and time budget for taking compensation actions will go down, and coordination between different condensation areas will be required. Humans are not expected to be able to cope with the increased load of decisions to take, and for larger areas, it is currently not possible to find feasible re-scheduling solutions in real time. The solution space of the train rescheduling problem grows rapidly with important dimensions of the problem, e.g., the number of train routes and the length of the time horizon. Simply deploying the current optimization techniques to a higher number, and larger areas, is not feasible without a scalable strategy for the compensation/coordination work. Van Thielen et al. (2018) highlight two related limitations in the state of the art. First, exact methods based on decomposition constitute a promising direction but existing approaches are limited in their applicability to large networks. Second, relying on macroscopic approaches may be insufficient since such solutions do not necessarily lead to a feasible microscopic solution.

Based on the above description of the real-world context and the limitations of the state of the art, we believe it is essential that the coordination effort that human train dispatchers make so greatly through communication and experience with an eye on the global system state, can be transported into algorithmic means. Experienced dispatchers have an intuition about which conflicts to pay attention to, while keeping the overall negative impacts under control. We therefore believe that ML could encode the knowledge of defining a restricted optimization scope in both space and time that results in a reduced problem that is computationally tractable with existing solution approaches.

3 Scope Restriction: Identifying the Core Problem

In this work, we propose ideas that may lead to a solution approach to overcome the limited spatial restrictions of automated algorithmic re-scheduling by extracting the core problem from the original full problem formulation. We share this preliminary study with the hope that it motivates and encourages further research into the area of real-time problem reduction. In this section, we describe the idea in more detail, and then we outline several different ways to restrict the problem scope. These are used in Section 5 to assess the potential of scope restriction.

3.1 Central Idea

We describe the idea in a simplified setting. Referring to Figure 1, we consider an operational plan and a single incident (malfunction) which can cause several conflicts, and we aim 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. Incident is found by updating forecast according to real time data.
2. Core problem is extracted from operational plan and forecast.
3. Optimizer solves core problem and generates new operational plan.
4. Repeat (go to 1).

We assume that any re-scheduling problem that arises within the full railway system can be reduced to a much smaller re-scheduling problem defining which trains or even only parts of trainruns might need to be re-scheduled, and we refer to this as the *core problem*. In contrast to the standard approach, 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: (i) passing times of trains and (ii) routes of trains. It hence spans both space and time. Compared to spatial decomposition, it is closer to a train-based decomposition introduced by Luan et al. (2018).

The main idea of problem scope restriction is shown in Figure 3, where a *scoper* predicts which parameters in time and space are relevant to solve the core problem caused by the malfunction. If the scoper can significantly reduce the problem size while allowing for a high-quality feasible solution, this should speed up the solution process. The scoper should hence identify all trains and departures that could be affected by a given disturbance, i.e., that might

need adjustment to include the disturbance in the updated operational plan. We note that this problem is related to delay propagation which is a problem that has been extensively studied in the literature (see, e.g., Goverde, 2010).

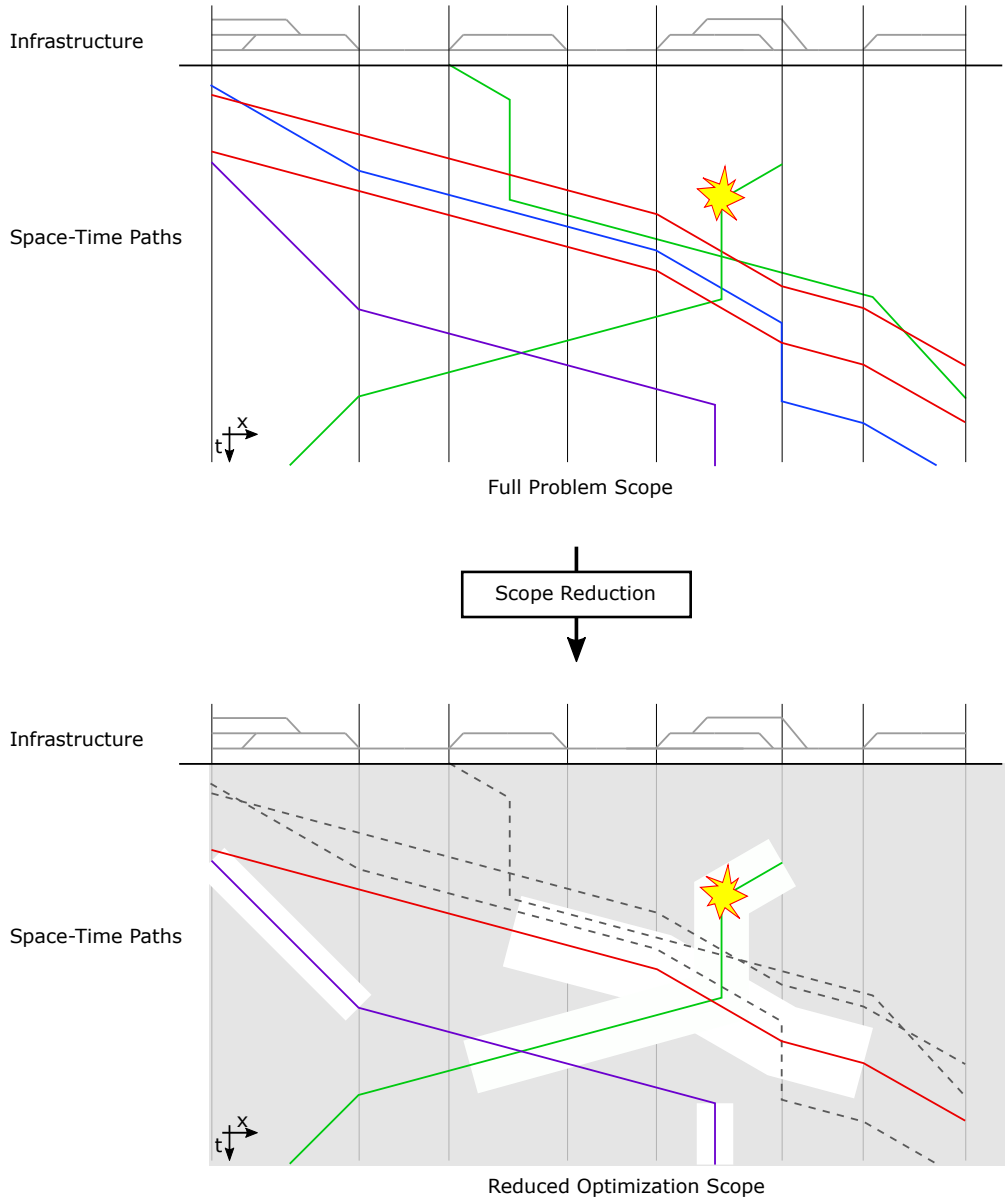


Figure 3: *Illustration of scope restriction. The aim is to vastly restrict the search space of an optimizing algorithm, condensing the initial full problem description to the core problem in the time-space parameters.*

3.2 Scopers

We now turn our attention to the scopers. As already stated, we believe that an adequately trained machine learning algorithm could constitute a strong scoper. This is left for future research. Instead, in this paper, we focus on assessing the potential of the idea. We therefore define several different scopers that can give useful performance benchmarks. In this context, we distinguish between two classes of scopers:

online The scopers have information about the current state of the trains / network (they do not have access to a full re-schedule solution). This corresponds to the setting that is of interest in practice.

offline The scopers have information about the current state of the trains / network as well as a full re-schedule solution. These scopers give us useful performance benchmarks.

We further illustrate the difference between online and offline scopers in Figure 4. It shows that both online and offline scopers use information about malfunctions (M) (state of the trains) and the original schedule (S_0). Based on this information, the online scoper defines a core problem that is given to a solver which produces a re-schedule S. The offline scoper can use the information of this re-schedule and produce another solution (S'). We note that an online scoper that does not restrict the scope (does nothing) is a special case of this scheme.

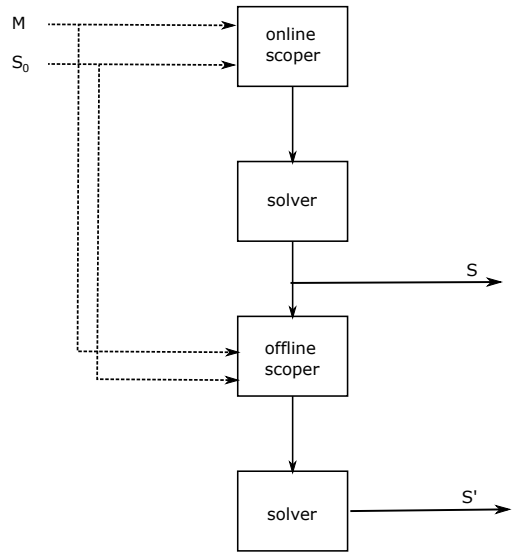


Figure 4: *Online versus offline scopers.*

If we have access to a full solution S of the re-scheduling problem, we can design non-trivial offline scopers that allow for a large speed-ups. Non-trivial means that the output of the offline scoper does not only trivially contain S, but a different S' is also a solution. Whereas this offline setting clearly does not correspond to a real application, it provides useful information to assess the potential of the idea. The main question of interest is to assess if it is possible to design an online scoper from data which achieves significant speed-ups with no, or limited impact on solution quality. In addition to the scoper, the answers to these questions depend on the problem instances and the optimization solver. For the experimental results in this preliminary study, we use one specific solver and we mostly focus on analyzing the potential for speed-up using offline scopers.

In the following we describe the different scopers we use for our numerical experiments (see Table 1 for a synopsis). More precisely, the online scopers are:

online_unrestricted this scoper does not restrict the re-scheduling problem, it is the "full" re-scheduling problem with "empty" scope restriction, supposed to give a reference of the hardness of the re-scheduling problem (see S going into *offline scoper* in Figure 4); it thus gives a trivial lower bound on speed-up with respect to the solver; this excludes unreachable alternative paths after the malfunction

heuristic 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.

random this scoper randomly chooses affected trains, giving no re-routing flexibility to the unaffected trains, supposed to show that the problem of predicting which trains 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 trains.

We now turn our attention to offline scopers used in our experiments:

upper_bound 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.

scoper	routing flexibility	time flexibility	role
online_unrestricted (online)	all trains	all trains	starting point for offline scopers, measure for hardness of the problem instance
upper_bound (offline)	–	–	trivial upper-bound on speed-up, measuring solver overhead when solution space has exactly one solution
max_speedup (offline)	routes occurring in either schedule or online_unrestricted solution	waypoints (nodes) with a difference between schedule and online_unrestricted solution	close to trivial upper-bound on speed-up
baseline (offline)	full routing flexibility for trains with difference online_unrestricted and schedule	trains with difference in online_unrestricted and schedule	a train-based baseline speed-up with access to full solution, giving a non-trivial, but offline upper-bound on speed-up
heuristic (online)	all possible paths for trains predicted by transmission chains propagation scheme	all nodes for trains predicted by transmission chains	lower bound for online scopers
random (online)	all paths for randomly selected trains (same number as predicted in by heuristic scoper)	all nodes from schedule	sanity check for baseline, solution quality expected to be worse than in baseline

Table 1: *Short-hand synopsis of scopers. Routing flexibility refers to degree of freedom of route choice. Time flexibility refers to where along those routes passing times are fixed and where the re-scheduling solution has time flexibility. The last column refers to the scopers' role in our line of argument.*

max_speedup this scoper restricts the scope to the difference between the initial schedule and the online_unrestricted solution,

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

this is 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.

baseline this scoper gives a lose core problem by opening up the same as online_unrestricted for changed trains. This gives a first impression of 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.

4 Synthetic Environment and Instance Generation

In Section 5, we report results for experiments based on a simplified railway simulation and simple heuristic problem scope restriction algorithms. Our playground implementation described in this section is an abstraction of real railway traffic. More precisely, we discuss a synthetic environment based on the Flatland toolbox (Mohanty et al., 2020) that we use to generate problem instances. This environment is available to other researchers and practitioners. We provide a high-level description here and refer to Appendix A for implementation details. Readers can reach out for more details. In brief, a given instance of the rescheduling problem corresponds to one train malfunction occurring for a given infrastructure and schedule. We therefore start by describing the infrastructure and the schedule generation followed by the malfunction simulation. Each of these aspects are parameterized to allow a generation of problem instances featuring different characteristics.

More details on the instance generation can be found in Appendix A and parameters in Appendix B.

4.1 Infrastructure

We use the Flatland toolbox (Mohanty et al., 2020) to generate a 2D grid world infrastructure. This world is built out of eight basic railway elements by assigning such an element to each position in the grid. We show the eight elements in the top row of Figure 5 each with a corresponding translation into a directed graph in the bottom row. The railway elements define the possible train movements from one position to the four neighbouring ones. From these basic railway elements, it is possible to generate cities which in turn are connected together to form a closed infrastructure system as shown in Figure 6. On each side of the city, there are parallel tracks that allow entries and exits, we refer to them as ports in the figure. This structure allows to generate track configurations automatically. More precisely, a fixed number of cities with certain internal track configuration are placed at random on the grid (uniformly distributed). Each city is connected by tracks (given a maximum number of parallel tracks between each city) according to a nearest neighbour algorithm.

In summary, different infrastructures can be generated by varying the dimension of the grid, the number of cities and the track configuration within and between each city. In turn, these settings can impact the solution space of the re-scheduling problem. For example, parallel tracks increase the re-routing options.

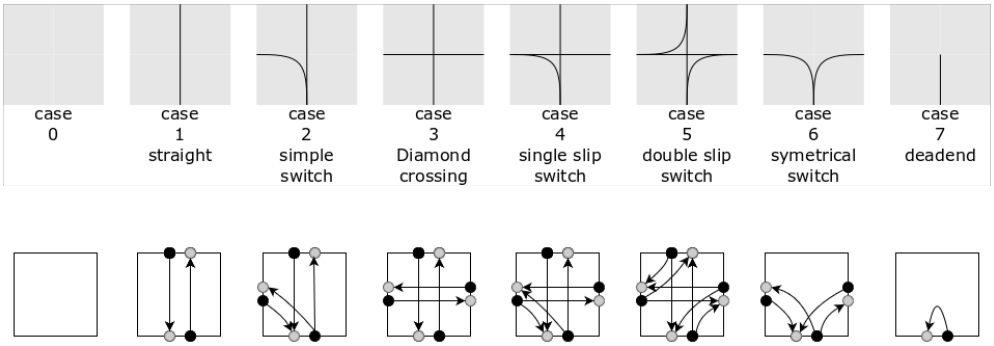


Figure 5: *Flatland basic railway elements. The bottom row shows the corresponding translation into a directed double-point graph structure.*

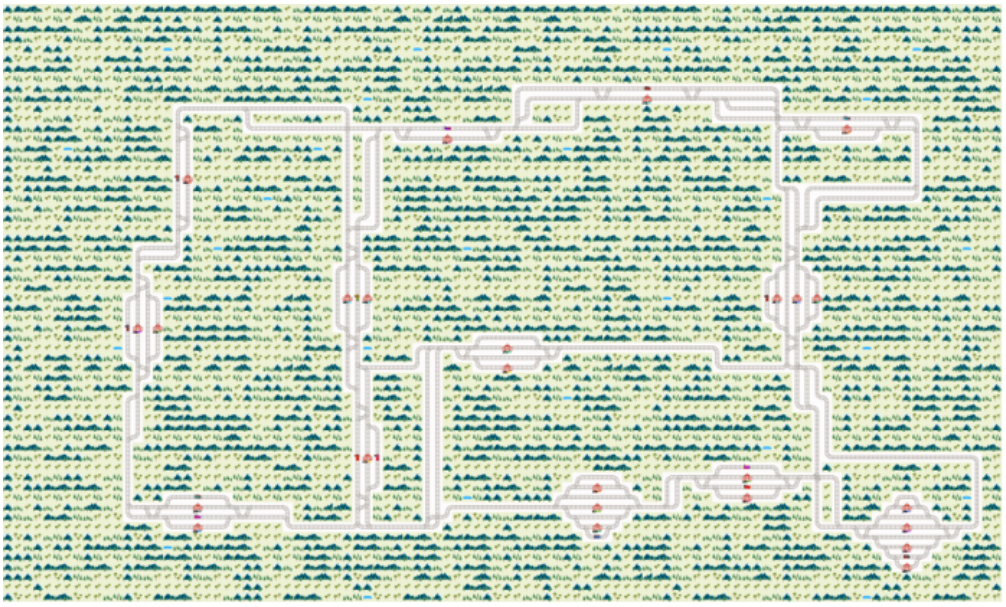


Figure 6: *Example infrastructure synthesized by the Flatland generator. It is composed of multiple cities, with randomised number of parallel tracks in cities and between cities.*

4.2 Schedule

For a given specific infrastructure, we generate a train schedule. We take a set of trains $\mathcal{A}$ as given, where each train $a \in \mathcal{A}$ has a single origin and destination in the grid and no intermediate stops. The schedule over time period $t = 1, \ldots, T$ defines, for each train, the origin departure time and passing time at each position in the grid (hence also the path in the grid). It is generated such that the sum of train times of all trains is minimized within a given chosen upper bound T. We make a number of assumptions: (i) as provided in the Flatland toolbox, each train has a specific but constant speed (ii) there are no time reserves in the schedule, meaning that the trains cannot reduce delay during their journey; (iii) trains appear in the grid when their schedule starts and disappear after reaching their target.

In reality, the schedules are, of course, more complex. For example, there are typically two different schedules: one published and one operational where trains must not depart earlier than published. Moreover, real schedules need to consider a number of aspects that we ignore, for example, respecting fleet management objectives and constraints.

Different schedules can be generated for each given infrastructure by varying the number of trains, their origin-destination pairs and the upper bound T.

4.3 Malfunction Generation

For a given infrastructure and schedule, we generate different problem instances by simulating malfunctions. In each instance there is one single malfunction $M = (t, d, a)$ and it is generated as follows: We draw a train $a \in \mathcal{A}$ at random, we then draw a malfunction time t at random from T. The malfunction duration d can be either fixed or drawn at random. An example is shown in Figure 7 where time is shown on the y-axis (from top to bottom) and space on the x-axis.

We make two strong simplifying assumptions. First, in reality, multiple malfunctions can occur during the time period T of interest, we assume that only one occurs. This is a strong assumption only if the time period is long or the area of interest is very large. Second, at the time of a malfunction, its duration is typically not known in reality while we assume that it is. These assumptions can be relaxed in the synthetic environment but would require different rescheduling models than the ones we consider in this exploratory work.

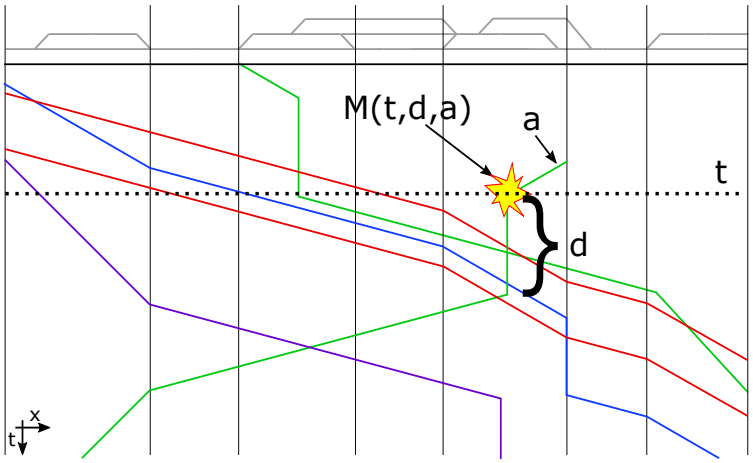


Figure 7: *Malfunction generation. This figure illustrates a simple operations plan with 6 trains on a small infrastructure. A single malfunction $M(t, d, a)$ occurs for train a at time t with a total duration of d time steps.*

5 Computational Results

In this section, we present the results from our numerical simulations. The aim of the experiments is to verify the existence of restricted (offline) scopers that allow for a large speed-up in solution time, as well as to investigate the speed-up of non-trivial online scopers. The simulations use different scopers for a large set of different schedule and infrastructure configurations. We outline the experimental design in the following section. We then analyze the results first focusing on speed-up (Section 5.2), and second, on solution quality (Section 5.3).

5.1 Experiment Design

The computational difficulty of any re-scheduling problem depends on the interplay of the infrastructure configuration, the planned railway schedule as well as the location and time of the disturbance. As it is hard to characterize what makes an instance hard to solve, we rate the difficulty the instances according to average required computation time to solve the full re-scheduling instance (online unrestricted scoper).

Using an hierarchical setup for problem instance generation, as shown in Table 2, we produce a diverse set of problem instances with varying levels of difficulty (as measured by computation time). The full parameter ranges are shown in Appendix B and the distribution of instance difficulty can be found in Appendix C.

When reporting the results in the following sections, we order instances according to the relative difficulty of the re-scheduling instances. While schedule and infrastructure remain mostly static in real world railway networks, we chose to generate different infrastructure configurations to get a wider distribution of problem instances. However, it turned out to be difficult to identify the relevant drivers for the instance difficulty. The resulting distribution is biased towards low computing time (more than 60% of experiments are in the shortest bin with computation times of less than 38 seconds).

All results were obtained by using the publicly available model of Abels et al. (2019, 2020)[2] using an Answer Set Programming (ASP) solver, without specific tuning of solver parameters. The solver objective penalizes delay and minimizes the number of changes in the rescheduling solution[3]. We used the same solver parameters for all instances, and the computations were run in a cloud environment.

Level	Element	Variables
0	infrastructure	Dimension of the grid as well as density of train tracks and number of trains.
1	schedule	Schedule times and paths for all trains.
2	malfunction	Malfunctioning train and malfunction onset.
3	solver runs	Each configuration was run multiple times with different seeds

Table 2: *Hierarchical experiment design. For any level, all subsequent levels generate unique problem instance configurations.*

5.2 Speed-Up

In this section, we analyze the speed-up, that is the reduction in computation time needed by the solver[4] after the scope restriction compared to the one without scope restriction. The results are shown in Figure 8 where we only analyze experiments in the range of interest for real-world applications of $[20s, 200s]$ computation time for the full problem instance. The results are binned in 10 equidistant bins for the analysis. We omit a comparison of absolute computational time for rescheduling and refer the interested reader to the Appendix C.

The *upper_bound scoper* is a sanity check for our approach. The results illustrate the maximum speed-up that can be achieved with the ASP-Solver. The speed-up is limited by solver specific pre-processing. This check is achieved by locking all variables to the final solution and letting the ASP-Solver build and solve the problem instance without any degrees of freedom. The computational cost of this is specific for the used ASP solver and can, of course, vary for other solvers.

Our estimate for maximum speed-up factors is represented by the *max speedup* scoper, which consistently achieves a factor above 2 and up to 10. Given that this scoper makes use of the full problem information to deduce the minimal core problem it constitutes a maximum in our setting.

The *baseline scoper* uses a much looser restriction on the scope. It restricts the scope to only the affected trains, but provides full routing and time flexibility. An accurate online scoper should hence be able to achieve this speed-up. We observe speed-up factors ranging between 2 and 5. These are large enough to have a positive impact on real-world operations.

[2] `https://github.com/potassco/train-scheduling-with-clingo-dl`

[3] More precisely, the objective for re-scheduling is a weighted sum of the delay at the target and a penalty for the number of nodes in the graph the re-scheduled path leaves to the scheduled path. The last term should help to avoid "flickering", the re-scheduled should not deviate from the scheduled path "without need".

[4] We measure elapsed time, including the grounding phase of the ASP solver, without optimizing the encoding to speed up neither the grounding nor the solving phase. The absolute computation times and relative speed-ups of the scope restriction may be different for other ASP encodings and other OR solvers. We only investigate the speed-up in this one setting.

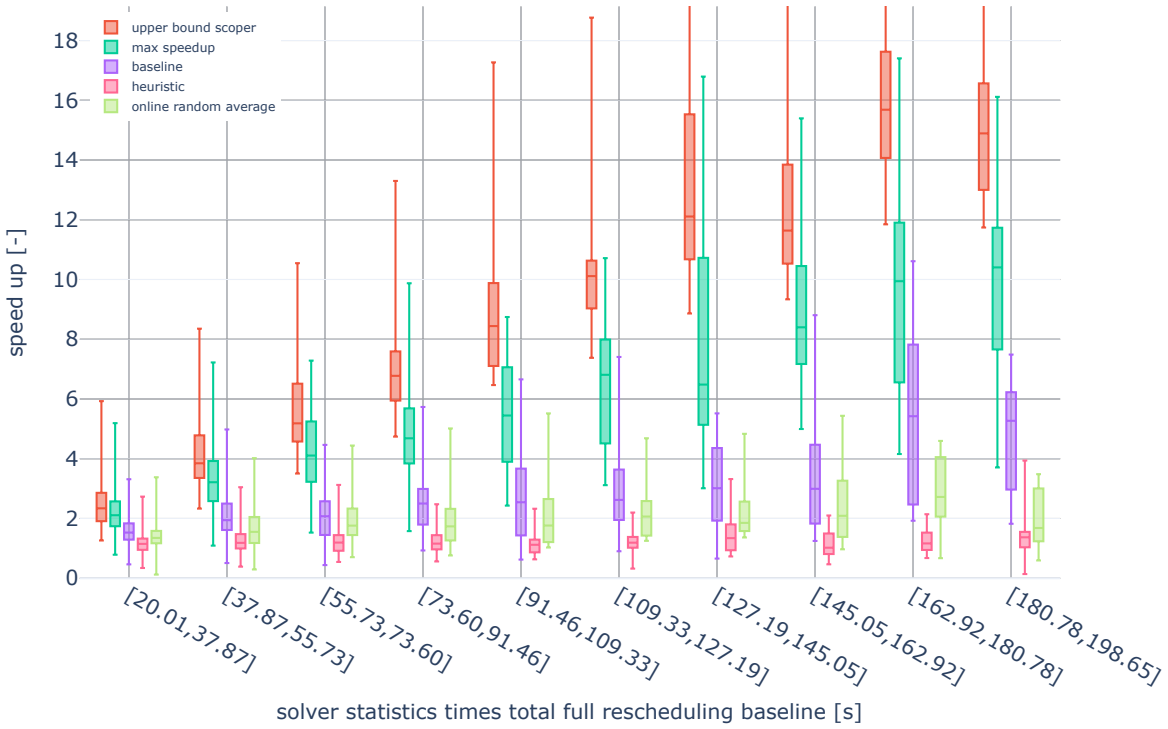


Figure 8: *Speed-up, defined as the the factor between the computation time needed to solve the full scope and the restricted scope (per scoper), where a higher number means more improvement.*

The *heuristic scoper* is desiged as a first attempt at an online scoper. It uses a simple delay propagation logic to define the core problem. Important in this context is the classical compromise between false positive and false negative rates. On the one hand, if the scoper has a high false positive rate, it will lead to little or no speed up. On the other hand, if the false negative rate is high, this can have a detrimental impact on solution quality which we analyze in the following section.

Based on the results reported in Figure 8 we clearly see the impact of a relatively high false negative rate. Our simple heuristic scoper does not achieve a significant speed up.

As a sanity check we also report results for *random average scoper*, which is the average solution time of using 5 randomly generated *core problems*. This scoper only shows a minor speed-up.

The results support our hypothesis that sufficient decrease in problem scope can lead to significant speed-up in computation time. These findings are expected in light of the literature on heuristic variable fixing. Furthermore, we see that, at least for the given simplified rail infrastructure and schedules, the core re-scheduling problem is much smaller than the full re-scheduling problem. This even holds true for comparably small railway networks and simple schedules. However, identifying the core problem is not a trivial task.

5.3 Solution Quality

In this section we briefly comment on solution quality as defined by the objective function. In our case it consists in minimizing the temporal delay compared to the original schedule. Given our simulation environment, we do not investigate solution quality in more general terms taking into account business rules and industry metrics.

Figure 9 shows an aggregate view of lateness (compared to a perfect rescheduling solution) for all instances. We note that all but the random scoper have high solution quality. By design, the baseline and max speed-up scopers yield a solution of the same cost as the optimal solution. The additional lateness in the case of the baseline scoper is

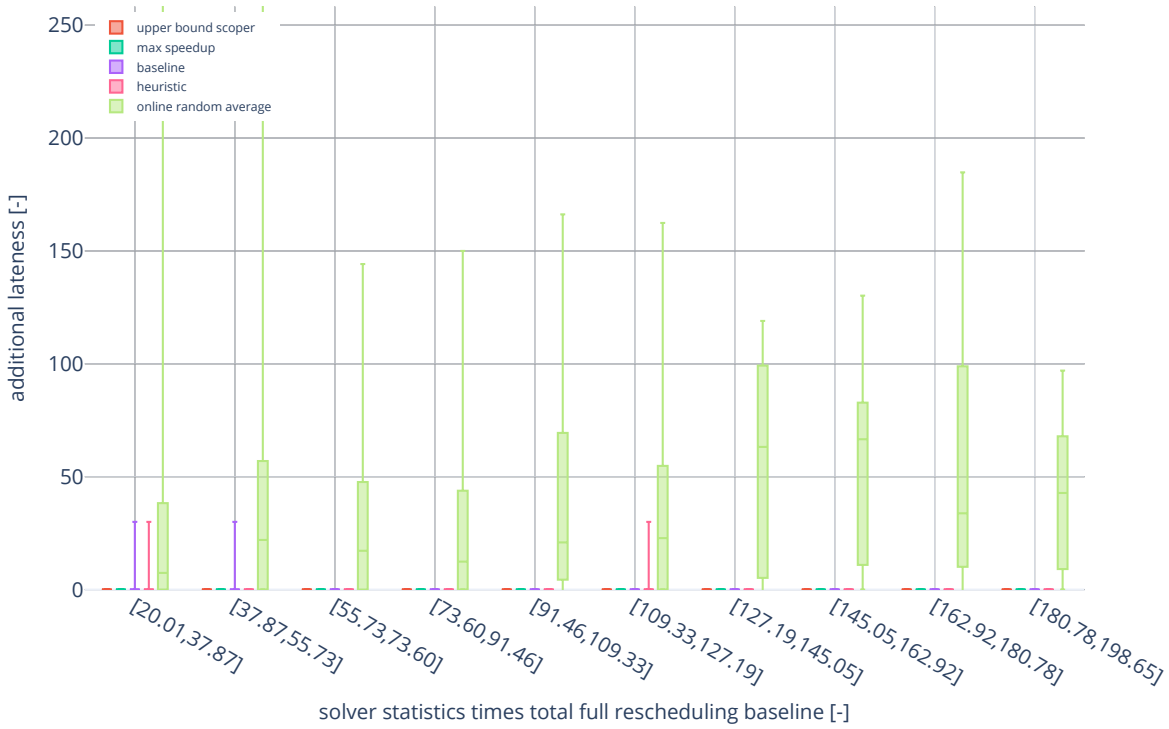


Figure 9: *Solution quality. This plot shows the additional lateness that was generated compared to the perfect rescheduling solution.*

compensated by fewer track changes (not shown). For all but a few instances, our heuristic scoper also yields the same high quality solutions. It can be explained by its low false negative rate. Recall, however, that this scoper did not yield any speed up due to its high false positive rate. As expected, the random scoper, achieves a poor solution quality.

These preliminary results show that, according to expectation, we can achieve a relatively high speed up by identifying trains affected by a malfunction with little negative effect on solution quality (baseline scoper). However, as we already stated, accurately defining the core problem is a non-trivial task.

6 Conclusion

Effective rail-time railway traffic management is essential for reliable train services. In practice, operators rely on a large extent on humans and their expertise to make rescheduling decisions. These human dispatchers heavily rely on direct human communication to coordinate their decisions. On the contrary, algorithmic solutions in industry are only implemented on restricted geographic scopes due to computational budget restrictions. In turn, the geographic decomposition introduces coordination challenges and is therefore difficult to scale.

Motivated by results in the literature and practical challenges, we argued that it should be possible to define a scoper (i.e., an algorithm) that can predict a reduced problem scope. We referred to this as the core problem which restricts the original problem in time and space by identifying train services that need to be rescheduled. The aim is to find high-quality traffic management solutions within a limited computing time budget and without introducing a coordination challenge.

We reported preliminary results from an exploratory study where we used the Flatland simulation environment to generate problem instances. Based on the results we concluded that, as expected, accurately identifying the core problem can lead to important speedups. However, identifying the core problem is a non-trivial task and a high-performing scoper needs to avoid false negatives (to ensure feasibility resp. solution quality) while not having too many false positives (to keep speed-up) when it comes to identifying the trains that are affected by a disturbance. This

was an exploratory study and much research is needed to turn these ideas into an operational approach. How to define an accurate scoper[5], how to best interact with a given solver, how to guarantee feasibility, and how to characterize uncertainty around the core problem are examples of four open and essential questions. Furthermore, not all of the specific business rules need to be reflected by the solver cost function – the solver and its metric might be part of a larger pipeline in an industrial setting.

The main purpose of sharing these preliminary ideas and results are to stimulate more work on this research topic. We believe that adequately defined machine learning algorithms trained on historical and/or simulated data should be able to accurately predict the core problem. The main reason is that infrastructure and train services are relatively stable over time. We hence face a recurrent problem that experienced humans are good at solving. We provide an extensible playground open-source implementation and our benchmarking instances and data in our GitHub repository (see *Data and Code Availability* Section below).

Data and Code Availability

Coda and data are available in our public GitHub repositories[6,7]. Also, the paper with an extended technical appendix can be found in the code repository.

Author Contributions Statement

Following CRediT (Contributor Roles Taxonomy[8]), the authors have contributed as follows.

Emma Frejinger Conceptualization, Writing (Literature Research), Writing (Review and Editing)

Erik Nygren Conceptualization, Methodology, Software (Analysis), Writing (Introduction, Computational Results, Review and Editing)

Christian Eichenberger Software (Pipeline and Analysis), Conceptualization, Methodology, Writing (First Integral Draft, Review and Editing)

Acknowledgement

This research project was only possible due to the generous support of both Mila and CIRRELT which provided access to working places, compute and the opportunity for great scientific exchanges.

We in particular want to thank Andrea Lodi and Yoshua Bengio who helped shape this idea through discussions and feedback.

Furthermore we want to thank the Swiss Federal Railway company for creating this opportunity for an intensive international research collaboration by providing resources as well as access to their core challenges. In particular we would like to thank Dirk Abels, Mathias Becher and Jürg Balsiger for supporting the research efforts and paving the way for an unobstructed research exchange between SBB, Mila and CIRRELT. Also we would like to thank for the fruitful discussions with colleagues at SBB (PFI, Flux, RTO) and University of Potsdam (Potassco).

References

Abbink, E., Bärmann, A., Bešinovic, N., Bohlin, M., Cacchiani, V., Caimi, G., de Fabris, S., Dollevoet, T., Fischer, F., Fügenschuh, A., Galli, L., Goverde, R. M., Hansmann, R., Homfeld, H., Huisman, D., Johann, M., Klug, T., Krasemann, J. T., Kroon, L., Lamorgese, L., Liers, F., Mannino, C., Medeossi, G., Pacciarelli, D., Reuther, M., Schlechte, T., Schmidt, M., Schöbel, A., Schülldorf, H., Stieber, A., Stiller, S., Toth, P., and Zimmermann, U. T. *Handbook of Optimization in the Railway Industry*, volume 268 of *International Series in Operations Research & Management Science*. Springer, 2018. ISBN 978-3-319-72152-1. doi: 10.1007/978-3-319-72153-8. Editors: Borndörfer, R., Klug, T., Lamorgese, L., Mannino, C., Reuther, M., Schlechte, Th.

[5] At a more theoretical level, how many equivalent core problems are there and how to decide between them?

[6] `https://github.com/SchweizerischeBundesbahnen/rsp`

[7] `https://github.com/SchweizerischeBundesbahnen/rsp-data`

[8] `https://credit.niso.org/`

Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., and Wanko, P. Train scheduling with hybrid ASP. In *LPNMR*, volume 11481 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2019. URL https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/lpnmr/AbelsJOSTW19.pdf.

Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., and Wanko, P. Train scheduling with hybrid answer set programming. *CoRR*, abs/2003.08598, 2020. URL https://www.cs.uni-potsdam.de/wv/publications/DBLP_journals/corr/abs-2003-08598.pdf.

Ahuja, R. K., Cunha, C. B., and Şahin, G. *Network Models in Railroad Planning and Scheduling*, chapter Chapter 3, pages 54–101. INFORMS, 2005.

Bengio, Y., Lodi, A., and Prouvost, A. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon, March 2020. URL http://arxiv.org/abs/1811.06128. arXiv:1811.06128 [cs, stat] version: 2.

Cacchiani, V., Galli, L., and Toth, P. A tutorial on non-periodic train timetabling and platforming problems. *EURO J Transp Logist*, 4:285–320, 2015.

Cacchiani, V. and Toth, P. Nominal and robust train timetabling problems. *European Journal of Operational Research*, 219(3):727 – 737, 2012.

Caimi, G., Fuchsberger, M., Laumanns, M., and Luethi, M. A model predictive control approach for discrete-time rescheduling in complex central railway station areas. *Computers & Operations Research*, 39:2578–2593, 11 2012. doi: 10.1016/j.cor.2012.01.003.

Caimi, G. C. *Algorithmic decision suport for train scheduling in a large and highly utilised railway network*. PhD thesis, ETH Zurich, Aachen, 2009. URL https://doi.org/10.3929/ethz-a-005947637.

Caprara, A., Fischetti, M., and Toth, P. Modeling and solving the train timetabling problem. *Operations Research*, 50 (5):851–861, 2002.

Cordeau, J.-F., Toth, P., and Vigo, D. A survey of optimization models for train routing and scheduling. *Transportation Science*, 32(4):380–404, 1998.

Corman, F., D'Ariano, A., Pacciarelli, D., and Pranzo, M. Optimal inter-area coordination of train rescheduling decisions. *Transportation Research Part E: Logistics and Transportation Review*, 48(1):71–88, 2012. Select Papers from the 19th International Symposium on Transportation and Traffic Theory.

Fischetti, M. and Monaci, M. Using a general-purpose mixed-integer linear programming solver for the practical solution of real-time train rescheduling. *European Journal of Operational Research*, 263(1):258–264, 2017.

Fuchsberger, M. Algorithms for traffic management in complex central station areas, 2012.

Goverde, R. M. A delay propagation algorithm for large-scale railway traffic networks. *Transportation Research Part C: Emerging Technologies*, 18(3):269–287, 2010.

Janhunen, T., Kaminski, R., Ostrowski, M., Schellhorn, S., Wanko, P., and Schaub, T. Clingo goes linear constraints over reals and integers. *TPLP*, 17(5-6):872–888, 2017.

Li, S., Yan, Z., and Wu, C. Learning to Delegate for Large-scale Vehicle Routing, October 2021. URL http://arxiv.org/abs/2107.04139. arXiv:2107.04139 [cs].

Liu, L. and Dessouky, M. Stochastic passenger train timetabling using a branch and bound approach. *Computers & Industrial Engineering*, 127:1223 – 1240, 2019.

Luan, X. *Traffic Management Optimization of Railway Networks*. PhD thesis, Delft University of Technology, 2019.

Luan, X., Schutter, B. D., van den Boom, T., Corman, F., and Lodewijks, G. Distributed optimization for real-time railway traffic management. *IFAC-PapersOnLine*, 51(9):106–111, 2018.

Lusby, R., Larsen, J., Ehrgott, M., and Ryan, D. Railway track allocation: models and methods. *OR Spectrum*, 33: 843–883, 2011.

Mohanty, S., Nygren, E., Laurent, F., Schneider, M., Scheller, C., Bhattacharya, N., Watson, J., Egli, A., Eichenberger, C., Baumberger, C., Vienken, G., Sturm, I., Sartoretti, G., and Spigler, G. Flatland-rl : Multi-agent reinforcement learning on trains, 2020.

Swiss Federal Railways. Rcs – Traffic management for europe's densest rail network. controlling and monitoring with swiss precision. https://bahninfrastruktur.sbb.ch/content/dam/internet/bahninfrastruktur/de/produkte-dienstleistungen/bahninformatiksysteme/bahnbetrieb/Broschuere-RCS.pdf.sbbdownload.pdf, 2020a. Accessed: 2020-07-15.

Swiss Federal Railways. The Swiss way to capacity optimization for traffic management. https://www.globalrailwayreview.com/wp-content/uploads/SBB-RCS-Whitepaper-NEW.pdf, 2020b. Accessed: 2020-07-15.

Van Thielen, S., Corman, F., and Vansteenwegen, P. Considering a dynamic impact zone for real-time railway traffic management. *Transportation Research Part B: Methodological*, 111:39–59, 2018. ISSN 0191-2615.

Appendix A Implementation Overview for the Hierarchical Experiment Setup

We now give an overview of the pipeline from an implementation perspective.

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.

Referring to Figures 10a and 10b, the pipeline decomposes into five top-level stages plus a post-experiment analysis stage:

1 Agenda Expansion The parameter ranges are given as input and expanded into infrastructure and schedule parameters to generate infrastructure and schedule.

2 Infrastructure Generation This generates railway topologies and places train start and targets in the infrastructure[9].

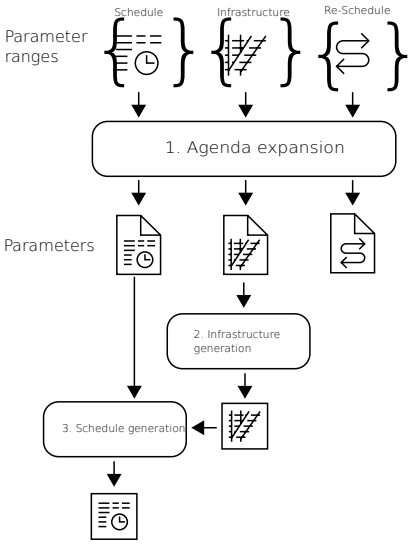
3 Schedule Generation This generates the exact conflict-free paths and times through the infrastructure for all trains.

4 Agenda Run From hierarchical directory of generated infrastructures, generated schedules and re-schedule parameters, an agenda of experiments is compiled and experiment are run sequentially.

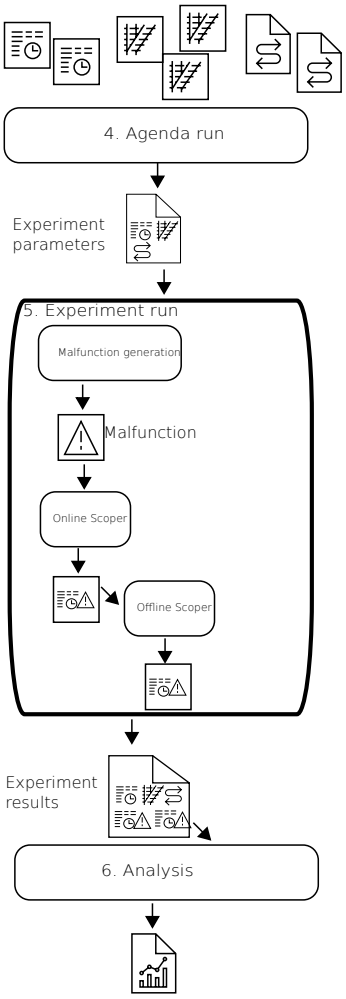
5 Experiment Run Here, the different scopers are run on the same problem instance as we know it from Figure 4.

6 Agenda and Experiment Analysis This stage aggregates experiment results from multiple experiment results and generates the plots in Section 5.

[9]Conceptually, the placement of trains 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 train placement.



(a) *Offline preparation for pipeline. The output is a hierarchical directory structure, where each infrastructure can have multiple schedule and each schedule multiple reschedule parameters. These files are generated by expanded initial parameter ranges for schedule, infrastructure and rescheduling.*



(b) *Agenda run starts from the offline preparation hierarchy of infrastructures, schedules and re-schedule parameters (see Figure 10a) from which a subset is chosen. An agenda run consists of multiple experiment runs. Each single experiment run gets experiment parameters as input and runs different scopers on them. The resulting re-schedules and performance measure are stored in Experiment Results. The results from multiple runs can be visualized by the analysis component.*

Figure 10: *Rescheduling experiments pipeline: preparation, experiments, analysis. Cornered rectangles represent files or data structures and rounded rectangles represent steps.*

Appendix B Experiment Parameters

Here we give more details on the experiments for the results of Section 5. We use 3264 experiments from a grid spanned by the parameter ranges below. We have 12 infrastructures (Level 0), 4 schedules per infrastructure (Level 1), $(50 + 62 + 74 + 86)$ malfunctions (Level 2), and only have one solver run per experiment (Level 3).[10] The value ranges $[a, b, n]$ have the following meaning:

- if $n = 1$, we take only a (ignoring b);
- if $n > 1$, we take n points from the half-open interval $[a, b)$ of step size $\lfloor \frac{b-a}{n} \rfloor$.

In the following tables, we abbreviate 100 instead for $[100, 100, 1]$ and $[0, 48]$ for $[0, 48, 48]$. Examples: $[8, 15, 3]$ expands to $8, 10, 12$, and $[1, 2, 2]$ expands to $1, 1$.

B.1 Infrastructure Parameters

Parameter	Symbol	Description	Value
`infra_id`	–	infrastructure id	$[0, 48]$
`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	[190,190,1]
`max_num_cities`	$\|S\|$	Maximum number of cities to be places in the environment. Cities are the only places where trains 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
`max_rail_in_city`	–	Maximum number of parallel tracks in the city	2
`number_of_agents`	$\|\mathcal{A}\|$		$[62, 86, 4]$
`speed_data`	$v : \mathcal{A} \to [0, 1]$	distrbution of speeds among trains	{ 1: 0.25, 1/2: 0.25, 1/3: 0.25, 1/4: 0.25 }
`number_of_shortest_paths_per_train`	$\approx (\mathcal{V}_a, \mathcal{E}_a)$	We compute shortest paths only once; should be larger than the number in scheduling and re-scheduling.	10

[10] The exposition of the parameter ranges slightly deviates from the structure in the data repository. Instead of 12 infrastructures with 4 schedules, technically, we have 48 infrastructures with 1 schedule each (using `flatland_seed_value`=$[190, 191, 4]$) . Note that, due to multi-threading, schedule generation is non-deterministic even when using the same ASP seed.

B.2 Schedule Parameters

Parameter	Symbol	Description	Value
infra_id	–	reference to infrastructure	$[0, 48]$
schedule_id	–	schedule id	$[0, 4]$
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_train_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_train_id	m_{train}	Which train is disturbed?	$[0, 86]$
number_of_shortest_paths_per_train	$\approx (S, L)$	Defines the route restrictions for the trains.	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.	60
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

Appendix C Further Experiment Result Details

C.1 Re-scheduling Times Distribution

Figure 11 shows that the hardness of re-scheduling is not purely determined by the infrastructure parameters (in particular grid size), but that there is a fat tail of run times depending on the malfunction configuration within the infrastructure.

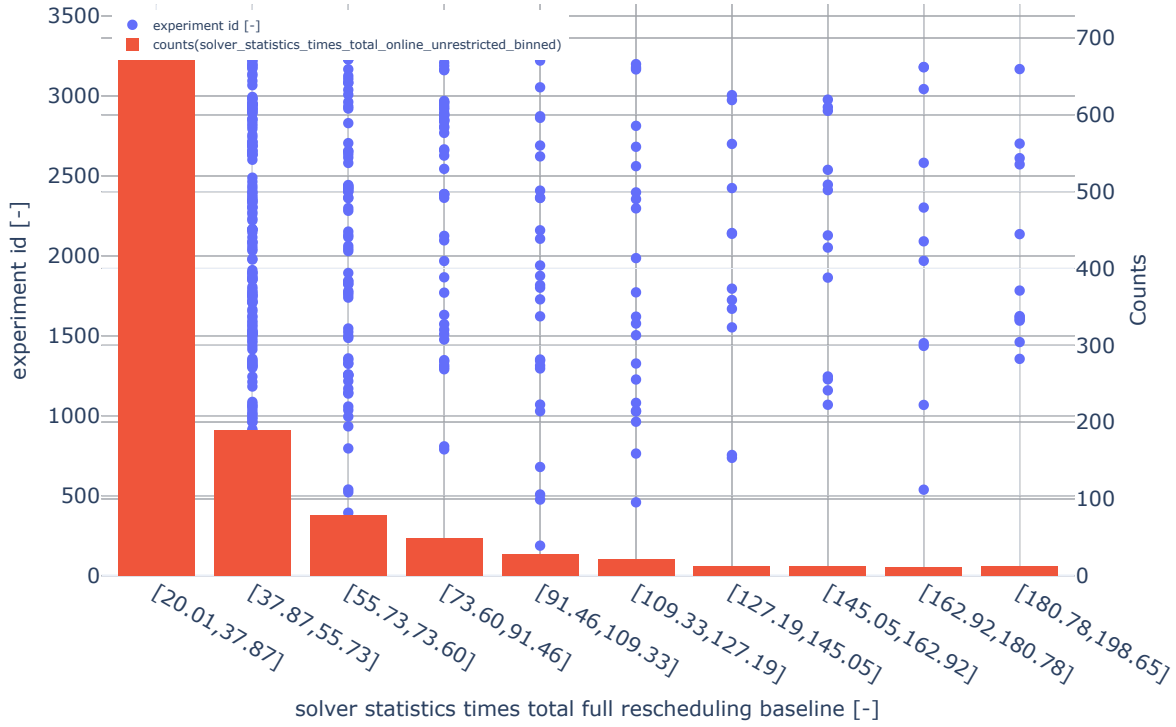


Figure 11: *Histogram of experiments in 10 equidistant bins, filtered on data with* `solver_statistics_times_total_online_unrestricted` *between* $20s$ *and* $200s$. *There is no correlation between experiment ID and computation time, meaning that no single factor (e.g. infrastructure, schedule or malfunction) is defining the complexity of a problem instance. The histogram further shows, that most problem instances have low complexity (short solving time) where our approach will have little impact.*

C.2 Scope Restriction Quality

In this section, we present the results about scope restriction quality, i.e. the ability of the scopers to identify the relevant parameters of the re-scheduling problem for a given problem instance. The quality of the scope restriction has a direct impact on the solution time speed-up and the solution quality, which will be discussed in the following sections.

To asses the quality of a scope, we start by defining the **core problem** as the differences between the original schedule and an optimal re-scheduling solution found with no scope restriction. Theoretically, there could exist multiple core problems with the same optimization score, but here we only focus on the core problem defined by the first optimal solution found by the optimizer.

The quality of any scoper is then defined by its accuracy expressed through the F_1-measure. It takes into account the number of trainruns that differ from the core problem, i.e number of false positives (trains not contained in core problem) and false negatives (trains contained in the core problem but not identified by the scoper).

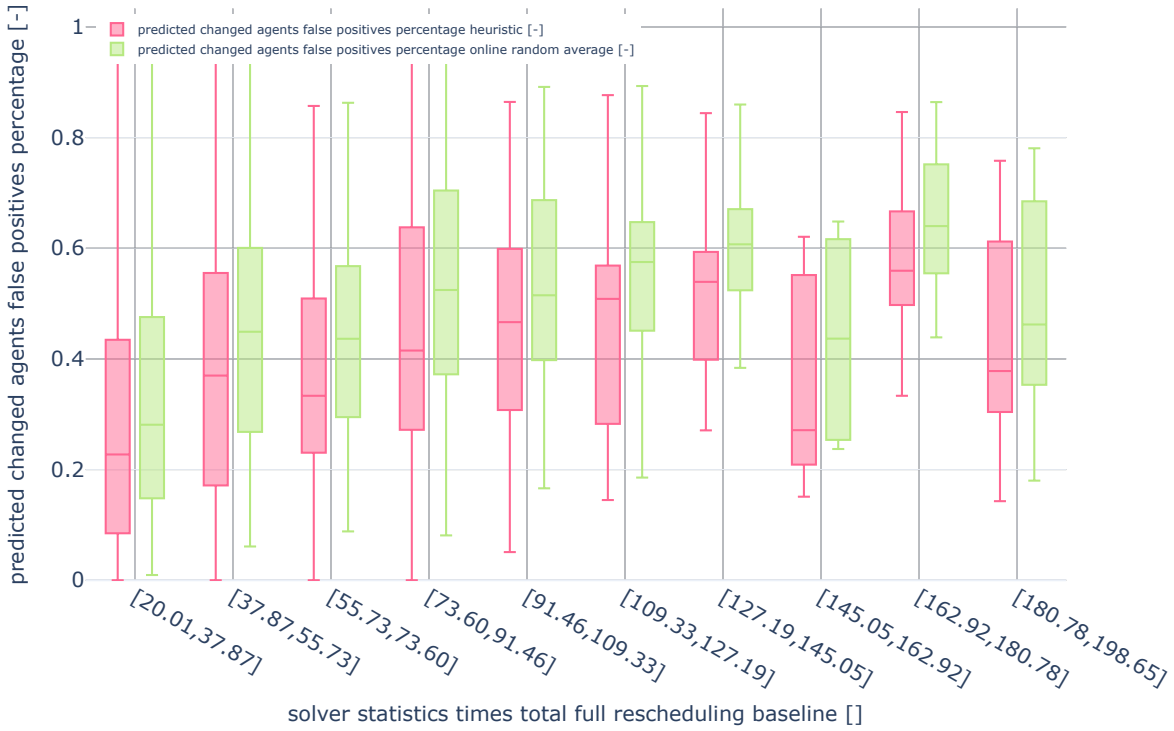


Figure 12: *Prediction quality: false positives and false negatives rate. Online transmission chains fully restricted is not shown as it is almost completely identical to online transmission chains route restricted.*

All but the **heuristic** and **random** scoper achieve the perfect quality score of 1, due to their construction. In Figure 15 we compare the F_1 score of these two scopers to asses their accuracy.

Detailed analysis of the F_1 score of the heuristic scoper reveals that there is a large number of false positives and almost no false negatives.

It becomes evident from the number false positives that the heuristic scoper generally overestimates the size of the core problem. The low number of false negatives in the heuristic scope restriction means that the core problem is contained within the predicted scope. This can be extracted from Figure 14.

Given the overestimation of the scope we only expect a minor speed-up for the heuristic scoper while the solution quality should be equivalent to best solution. In Figure 12 and Figure 13 in the appendix you find the quality as well as false negatives across all experiments.

The results show that it is difficult to define a heuristic scoper which reliably identifies the core problem without overestimation. This stems from the fact that false negatives lead to poor or even infeasible solutions. We identify this as a key challenge for our approach that needs to be overcome to benefit from the speed-up advantages.

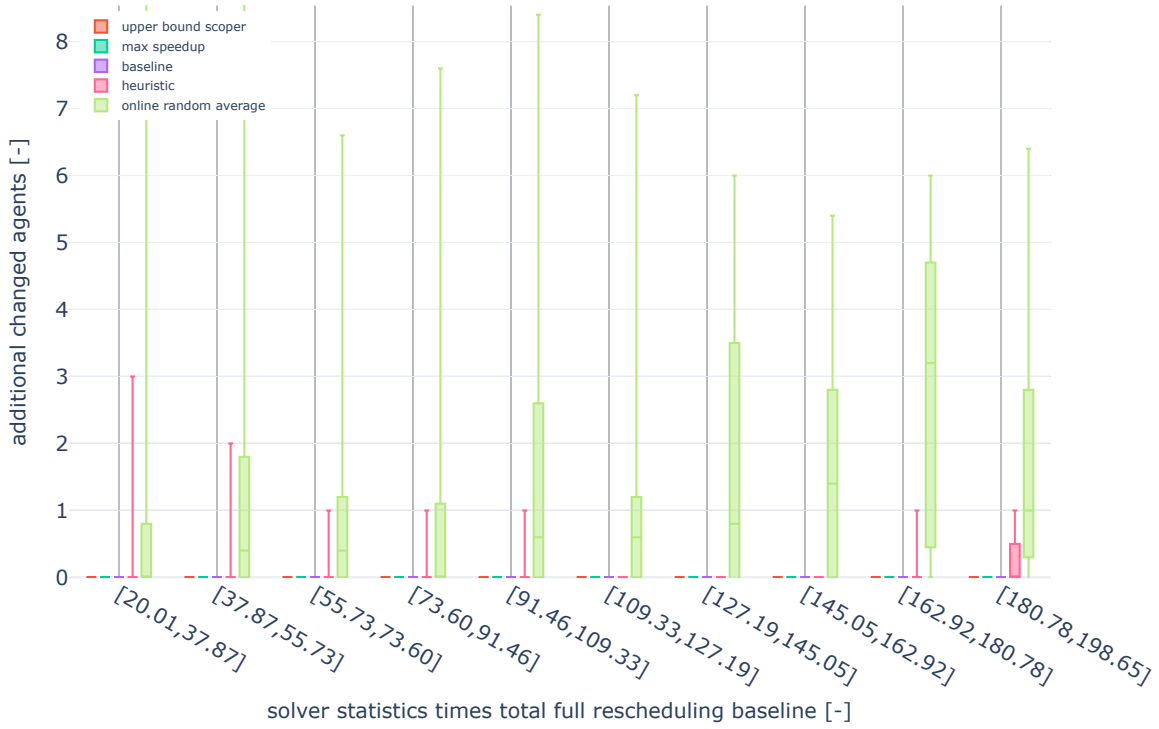


Figure 13: *Prediction quality: additional changed trains.*

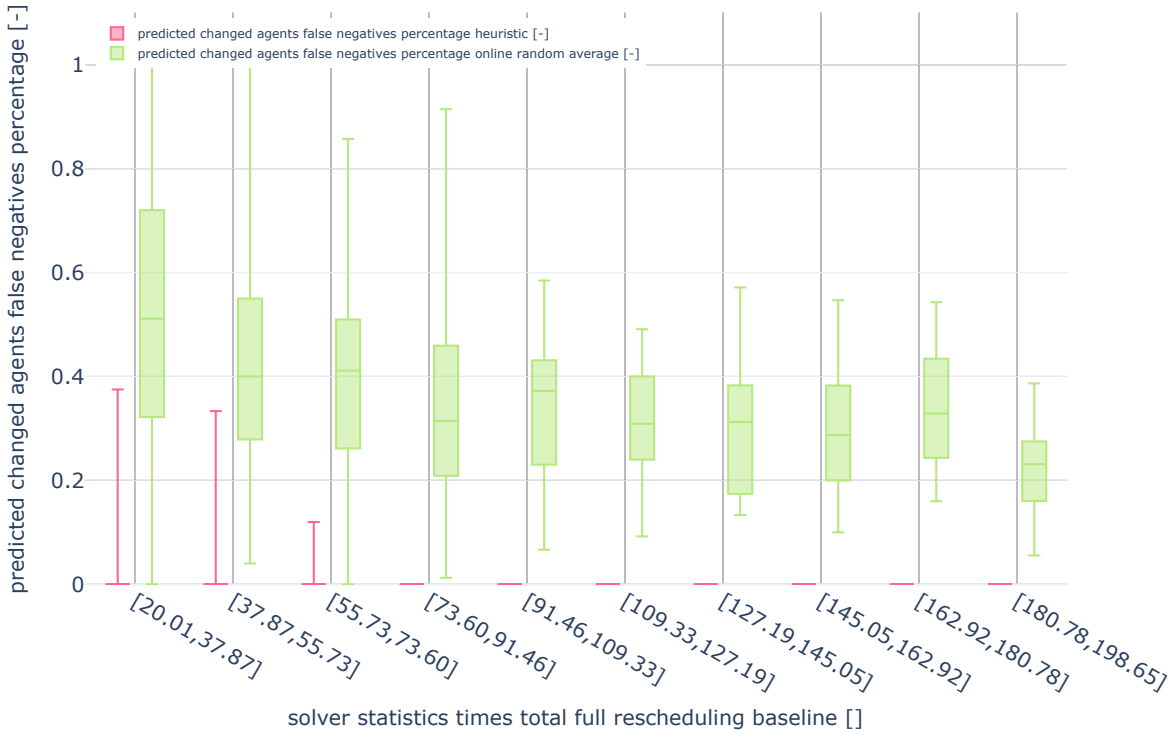


Figure 14: *This figure shows the distribution of the prediction quality across all experiments. False negative lead to worse solution quality or infeasible solutions. Only in a few cases did the heuristic scoper produce false negatives, and thus we expect solution quality to be on par with the optimal solution.*

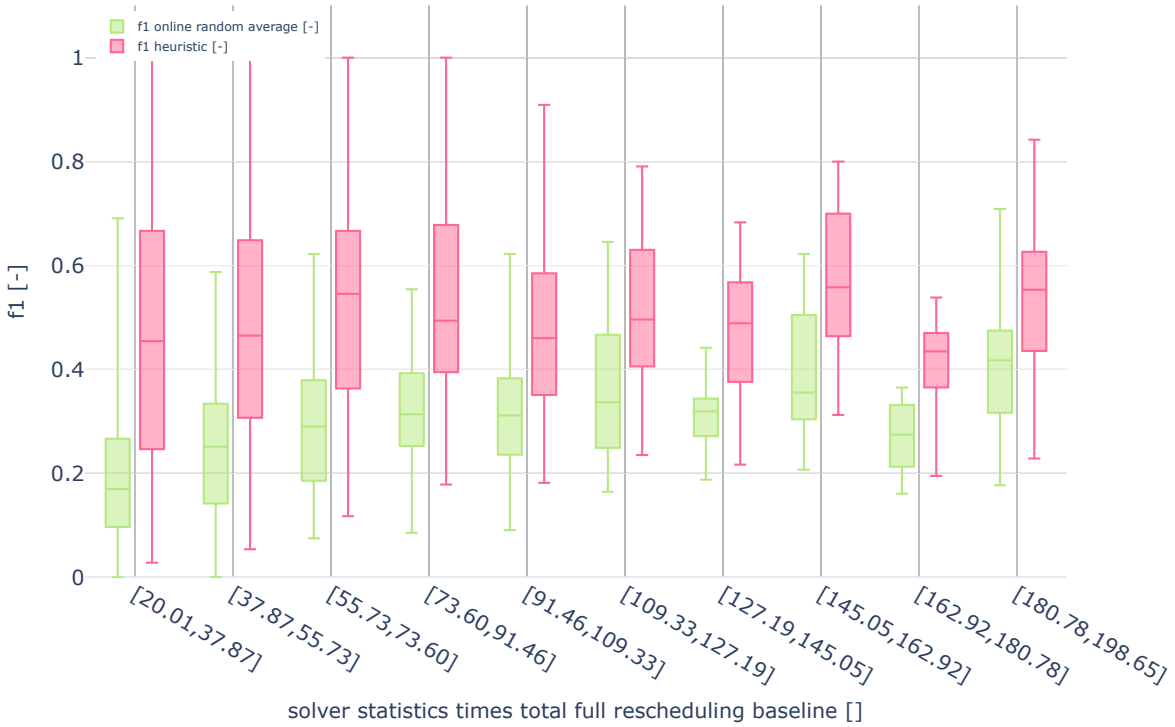


Figure 15: *Prediction quality of the heuristic and random scoper. This figure shows the F1 score of the heuristic and random scoper. The performance of the heuristic scoper is well above the random baseline. We lack further baselines to compare with and the heuristic baseline should serve as a benchmark for future improved heuristics or ML scopers.*

Internal Technical Notes, Work in Progress!!!! version May 8, 2023

A Formal Description of Infrastructure, Schedule, Malfunction and Re-Scheduling Objective

[[ChE: does the mixing of formal and implementation details make sense?]] This section gives a formal description of the problem instances of Section 4.

A.1 Infrastructure Generation

Infrastructure Generation is given width and height of an environment and then places cities on this grid. The cities are then populated with rail tracks and the cities are connected by rail tracks. This procedure is controlled by 3 parameter (maximum number of cities, maximum number of parallel tracks in the city and maximum number of parallel tracks in a city). Infrastructure Generation outputs a grid of cells with a directed graph on top. The directed graph represents the routing possibilities in the infrastructure. Each edge enters through one of the cell borders to the north, east, south or west to a different and traverses the cell. In this way, we can identify each edge with the entering border of a cell in the grid. Each edge has a cell, which it traverses.

A *railway infrastructure* is 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.

A.1.1 Implementation Details

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_trains:** 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 trains 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

A.2 General Train Scheduling Problem

We use the abstract model from Abels et al. (2020) train scheduling on a directed graph with resources and discrete time steps.

The infrastructure part of the model features resources and blocking times. Resources need to be reserved exclusively for a train at each time step. In our setting, each edge has exactly one resource, identified by the cell it traverses (the abstract model is more general). Blocking times are the times a resource cannot be used by another

train after it is completely released by a train, which is constant in our case, reflecting the same behaviour in FLATland.

The scheduling part of the model features for each train the sub-graph usable by the train, time window constraints for passing through vertices and minimum travel times. We only use the shortest path for each train for scheduling in order to speed-up schedule generation In our setting, each train has an earliest departure time at the start node and a latest arrival time at its destination; the time window constraints are then set the earliest and latest possible values which still allow feasibility for the train. Minimum travel times are the inverse of the train's FLATland speed. Technically, the abstract model associates minimum travel times with edges, so each train has its own edge with the corresponding speed profile and the cell associated, ensuring mutual exclusive usage.

We do not use additional features such as waiting times, connections and collision-free resource points (used for splitting and merging of trains).

We introduce the abstract model from Abels et al. (2020), 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 A.3 and A.5, 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 4 in a more formal setting.

According to Abels et al. (2020), 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,
  - $m : E \to \mathbb{N}$ 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 train 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 Abels et al. (2020) 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 train a in the schedule (P, A), and, similarly $\sigma(a, P) = v_1$ for the source node of train 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.

A.3 Schedule Generation

We start schedule generation from infrastructure generation as described in Section A.1; 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 $(\mathcal{C}, \mathcal{V}, \epsilon, \mathcal{E})$
- $\mathcal{A}$ of trains;
  - each train $a \in \operatorname{dom} \mathcal{A}$ has a source $\sigma(a) \in \mathcal{V}$ and some target in $\tau(a) \subseteq \mathcal{V}$[11]

[11]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.

  - a speed $v(a) \in [0, 1]$
  - a path restriction represented by acyclic subgraph $(\mathcal{V}_a, \mathcal{E}_a)$
- a release time r[12], 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.
- an upper bound U for all discrete time steps

In summary, we start from $\mathcal{C}, \mathcal{V}, \mathfrak{c}, \mathcal{E}, \mathcal{A}, \sigma, \tau, v, r, \{ (\mathcal{V}_a, \mathcal{E}_a) \}_{a \in \mathcal{A}}, U$. 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
  - $S = \mathcal{V}_a$
  - $L = \mathcal{E}_a$
  - $e(v) = \min_{p:\sigma(a)\text{–}v \text{ path in } (\mathcal{V}_a, \mathcal{E}_a)} |p| \cdot v(a)^{-1}$
  - $l(v) = \max_{p:v\text{–}\tau(a) \text{ path in } (\mathcal{V}_a, \mathcal{E}_a)} U - |p| \cdot v(a)^{-1}$
  - $w(e) = v(a)^{-1}$ [13];
- $N = (V, E, R, m, a, b)$ 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 trains $a \in \mathrm{dom}(\mathcal{A})$)
- 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.

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 16 and 17, 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 $e(v) > l(v)$.

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

[13] 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.

Algorithm 1 *propagate_earliest*

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

$Open \leftarrow F$
for $v \in Open$ **do**
  for $v' \in S - F : (v, v') \in L$ **do**
    if $v' \notin \mathrm{dom}(e)$ **then**
      $e(v') \leftarrow \infty$
    end if
    $e(v) \leftarrow \min\{\, e(v'), e(v) + mrt \,\}$
    $Open \leftarrow Open \cup \{\, v' \,\}$
  end for
  $Open \leftarrow Open - \{\, v \,\}$
end for

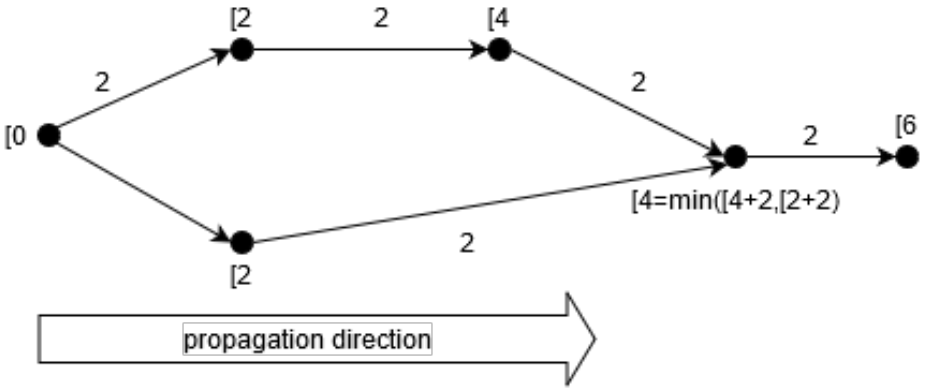


Figure 16: *Illustration of $propagate\_earliest$ for a train a with speed $v(a) = \frac{1}{2}$ (i.e. $mrt = 2$).*

Algorithm 2 *propagate_latest*

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

$Open \leftarrow F$
for $v \in Open$ **do**
  for $v' \in S - F : (v', v) \in L$ **do**
    if $v' \notin \mathrm{dom}(l)$ **then**
      $l(v') \leftarrow -\infty$
    end if
    $l(v) \leftarrow \max\{\, l(v'), l(v) - mrt \,\}$
    $Open \leftarrow Open \cup \{\, v' \,\}$
  end for
  $Open \leftarrow Open - \{\, v \,\}$
end for

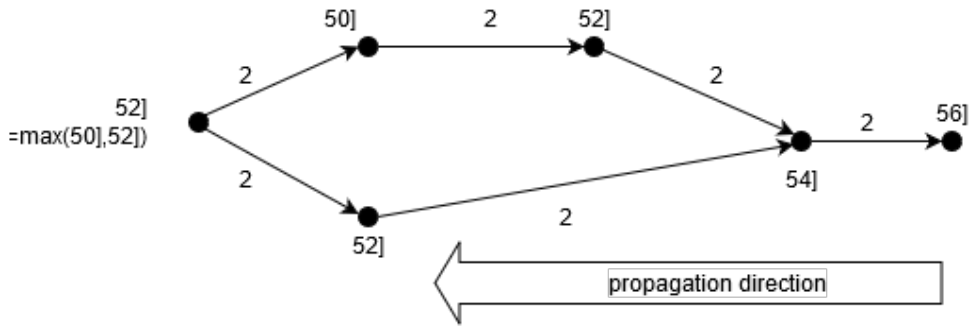


Figure 17: *Illustration of $propagate\_latest$ for a train a with speed $v(a) = \frac{1}{2}$ (i.e. $mrt = 2$) and for upper bound $U = 56$.*

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

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

and restricting

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

A.3.1 Implementation Details

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 trains per city, hence the sum represents the time required for the last train at the station if they all start one after the other and have constant speed; δ represents the minimum time required per cell for a train 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 train 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.

A.4 Malfunction Generation

Given a schedule, we generate one malfunction for a train during its run. Three parameters control this: which train, how long affected and earliest time step after the scheduled departure.

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;
- $m_{train} \in \mathrm{dom}(\mathcal{A})$: the id of the train that is concerned.

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

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

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

A.5 Objective for Re-Scheduling

[[EF: Given equivalent solutions, the last term will favour staying with the previous (possibly rescheduled paths). So only resechedule if necessary.]]

> The objective for re-scheduling is a weighted sum of the delay at the target and a penalty for the number of times the re-scheduled path leaves to the scheduled path. The last term should help to avoid "flickering", the re-scheduled should not deviate from the scheduled path "without need".

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 .

B Detailed Definition of Scopers

[[ChE: We need to use same terminology as in main text and proof-read in detail the formal description and pseudo-codes.]]

We now look at the different scopes of Section 3.2 in more detail.

All these scopers have the same interface, i.e. take the same parameters:

- an indexed set $\mathcal{A}$ of trains $\mathcal{A}(a) = (S, L, e, l, w)$ 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 $M = (m_{time\_step}, m_{duration}, m_{train})$;
- train 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 trains,

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

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:

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

B.1 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 $M = (m_{time\_step}, m_{duration}, m_{train})$ 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.

Algorithm 3 $scoper\_online\_unrestricted$ for train $a \in \mathrm{dom}(\mathcal{A})$

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

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

Before describing the third case of the malfunction happening while the train is running (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 a train can reach the vertex and that l reflects the last possible time the train 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.

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).

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

B.2 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.

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
$S \leftarrow \{ v \in S : e(v) \leq l(v) \}$, $L \leftarrow \{ (v, v') \in L : v, v' \in S \}$

Algorithm 5 $scoper\_online\_unrestricted\_running$ for running train $a \in \mathrm{dom}(\mathcal{A})$

Input: (S, L), (P, A), $M = (m_{time\_step}, m_{duration}, m_{train})$, 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 $a = m_{train}$ **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
$e, l, (S, L) \leftarrow propagate(e, l, (S, L), F_e, F_l, \{ v_1, v_2 \}, mrt, U, c)$

Algorithm 6 $scoper\_offline\_fully\_restricted$ for train $a \in \mathrm{dom}(\mathcal{A})$

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

$S \leftarrow \{ v : v \in P \}$
$L \leftarrow \{ (v, v') \in P : v, v' \in S \}$
$e \leftarrow \{ (v, P(v) : v \in S \}$
$l \leftarrow \{ (v, P(v) : v \in S \}$

B.3 max_speedup

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 a train and let (P_S, A_S) be a solution to the re-scheduling problem. Algorithm 7 defines the "perfect" problem scope for a running train: 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 $scoper\_max\_speedup$ for running train $a \in \mathrm{dom}(\mathcal{A})$

Input: (S, L), (P_{S_0}, A_{S_0}), $M = (m_{time\_step}, m_{duration}, m_{train})$, 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: $L_1 \leftarrow \{\, (v, v') \in P_S$ 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** $a = m_{train}$ **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.

B.4 baseline

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 train which have any difference between S and S_0. 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 trains are determined (line 1) and then the scope is reduced by Algorithm 9: all trains 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 $scoper\_baseline$

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

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

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**

B.5 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), $M = (m_{time\_step}, m_{duration}, m_{train})$, 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: $e, l, (S, L) \leftarrow propagate(e, l, (S, L), \emptyset, \emptyset, \emptyset, mrt, U, c)$

B.6 heuristic, restricted_heuristic (Transmission Chain Propagation)

This is an implementation of a simple heuristic scope reducing algorithm. It can instantly reduce the scope of a re-scheduling problem by assuming that all trains stay on their planned path and by propagating the caused delay forward in time, passing it on to any scheduled train within a temporal distance smaller than the malfunction duration. Let $S_0 = (P, A)$ be a schedule and let $P(a) = (v_1, \ldots, v_n)$ the scheduled path for a train 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 trains 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 trains reached, initialized with the malfunction train (line 3). We initialize the queue and the changed trains with the malfunction train by pushing all vertices that are passed in the schedule after the malfunction; since the schedule is constructed such that the trains never stop, the delay is not reduced (lines 4–6) [14]. Then, we look at the next train after the queued element (lines 8–14). If the delay cannot be absorbed by the time difference between the queued element's departure and the next train entry (lines 16–17), then we mark the next train as reached (line 18), compute the transmitted delay (line 19). If the there is a delay $d' > 0$ propagated to a', then a' cannot enter $R(v)$ only d' later, i.e. the occupation of the previous call $R(v'')$ is prolonged by d', which may cause other trains to wait for a''s delayed departure. Finally, we enqueue all vertices scheduled after it is "hit" by the propagated delay (lines 24–25), starting at the cell before $R(v)$. As an illustration, take another train is scheduled to go through the malfunctioning train as it stands still but the other train 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.

[14] 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

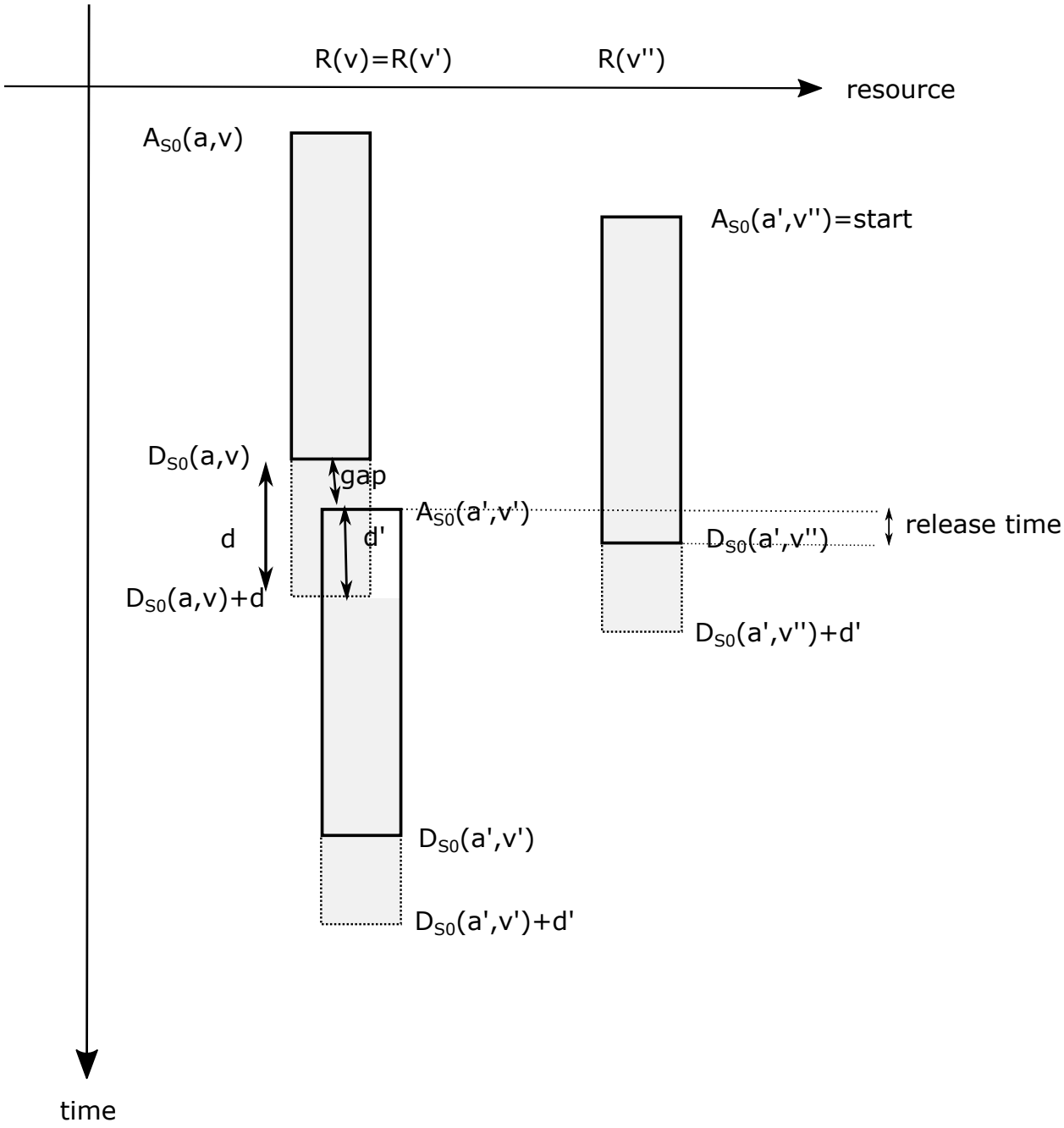


Figure 18: *Illustration of transmission chains algorithms. The plain boxes represent the resource occupation of the schedule S_0, the dotted boxes represent the propagated delays, and the shaded areas represent the resource occupations with the propagated delay included. We assume that trains a' can stop immediately in the the cell $R(v''$ just before they are hit at $R(v) = R(v')$ enter through v' by the malfunction and that they do not decelerate earlier. Furthermore, we assume that trains a' on their course after $start$ can never compensate the propagated delay, i.e. that there are on time reserves that could decrease the propagated delay.*

Algorithm 11 *changed_transmission_chains*

Input: $\mathcal{A}$, (P, A_{S_0}), $M = (m_{time\_step}, m_{duration}, m_{train})$
Output: $\mathcal{A}^* \subseteq \mathcal{A}$
1: $q = queue()$
2: $done = \emptyset$
3: $\mathcal{A}^* \leftarrow \{ m_{train} \}$
4: **for** $v \in P_{S_0}(m_{train}) : D_{S_0}(m_{train}, v) > m_{time\_step}$ **do**
5: $q.push((m_{train}, v, m_{duration}))$
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: $dep \leftarrow D_{S_0}(a, v)$
14: $a' \leftarrow \arg\min_{a' \in \mathcal{A}, v' \in P_{S_0}(a', v'), R(v') = r, A_{S_0}(a', v) \in [dep, dep+d]} A_{S_0}(a', v)$
15: **if** a' is defined **then**
16: $gap \leftarrow A_{S_0}(a', v') - dep$
17: **if** $gap < d$ **then**
18: $\mathcal{A}^* \leftarrow \mathcal{A}^* \cup \{ a' \}$
19: $d' \leftarrow d - gap$
20: $start \leftarrow \max\{ A_{S_0}(a', v'') : A_{S_0}(a', v'') < A_{S_0}(a', v') \}$
21: **if** $start$ is not defined **then**
22: $start \leftarrow A_{S_0}(a', v')$
23: **end if**
24: **for** $v'' \in P_{S_0}(a') : A_{S_0}(a', v'') \geq start$ **do**
25: $q.push((a', v'', d'))$
26: **end for**
27: **end if**
28: **end if**
29: **end while**

Again, we conjecture[15] 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 train.

Our approach is similar to delay propagation in Abbink et al. (2018) 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 a train at a resource is in the queue; we have refrained from this to keep the exposition simple.

B.7 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 trains 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 train is contained (lines 3–7).

[15] We do not prove it formally. The rationale is that every train 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, see Figure 18.

Algorithm 12 *scoper_random*

Input: $\mathcal{A}$, (P_{S_0}, A_{S_0}),$M = (m_{time\_step}, m_{duration}, m_{train})$,$v$,$U$,$U$,$c$, (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).

C Illustration of Cost Equivalence

We want to illustrate that there are cost equivalent solutions in our setting. We pick an experiment where we find that the sum of the lateness over all trains and the some over all route section penalties of all trains differ in two scopes, but sum up to the some total costs of the solver. We show such a constellation in Figure 19.

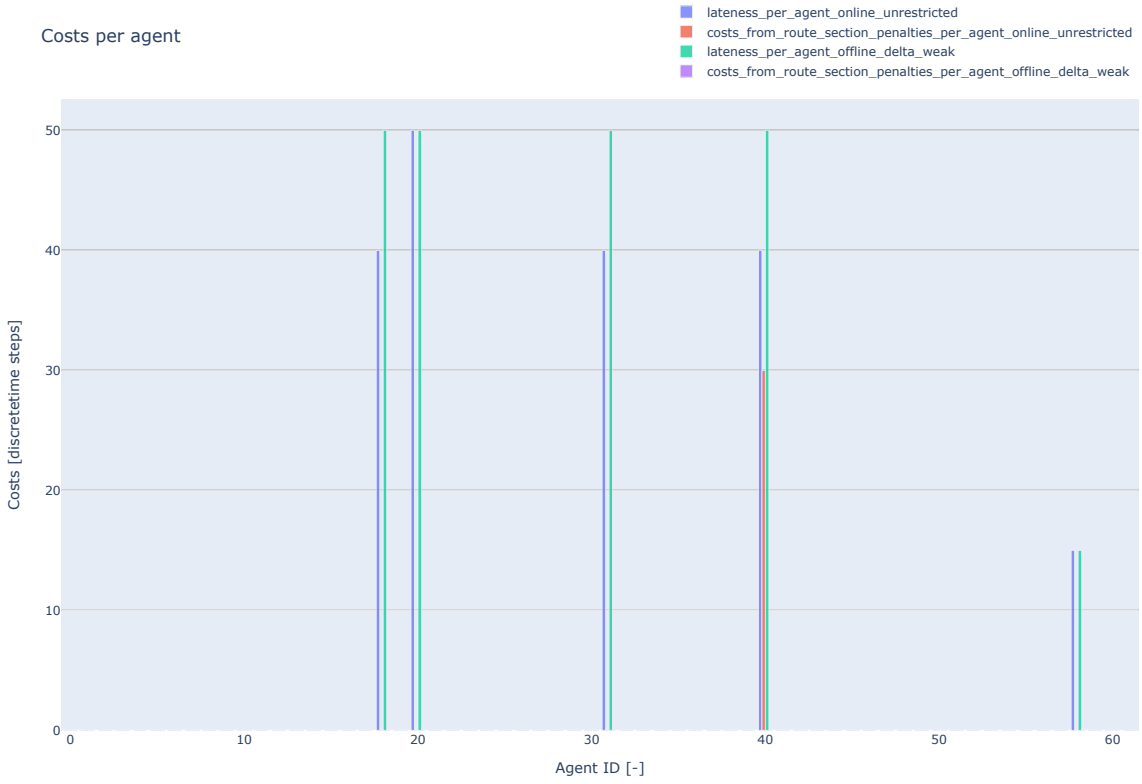


Figure 19: *Illustration of cost equivalence, here one run of experiment* 342 *of our agenda (multiple runs may yield different cost equivalent results). train* 40 *changes route in online unrestricted (penalty* 30*), but not in online route restricted, where not changing routes is compensated by a penalty of* 10 *for trains* 18*,* 31 *and* 40*.*

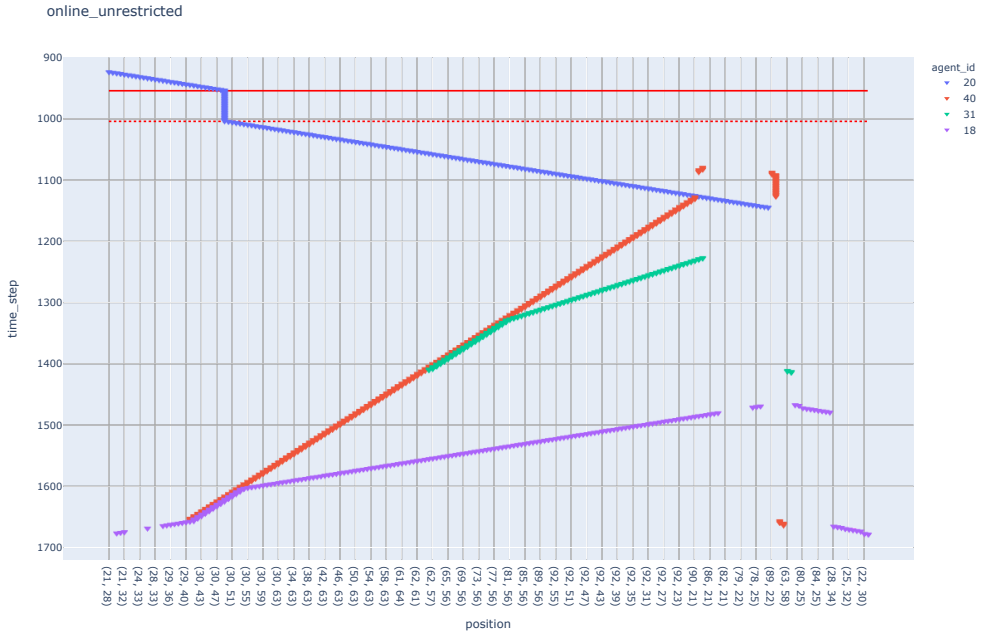
The same information is shown in the following table for scopes online_unrestricted (o.u.) and online_route_restricted (o.r.r.):

train	costs route section penalties		online_route_restricted	
	o.u.	o.r.r.	o.u.	o.r.r.
18	40	50	0	0
31	40	50	0	0
40	40	50	30	0

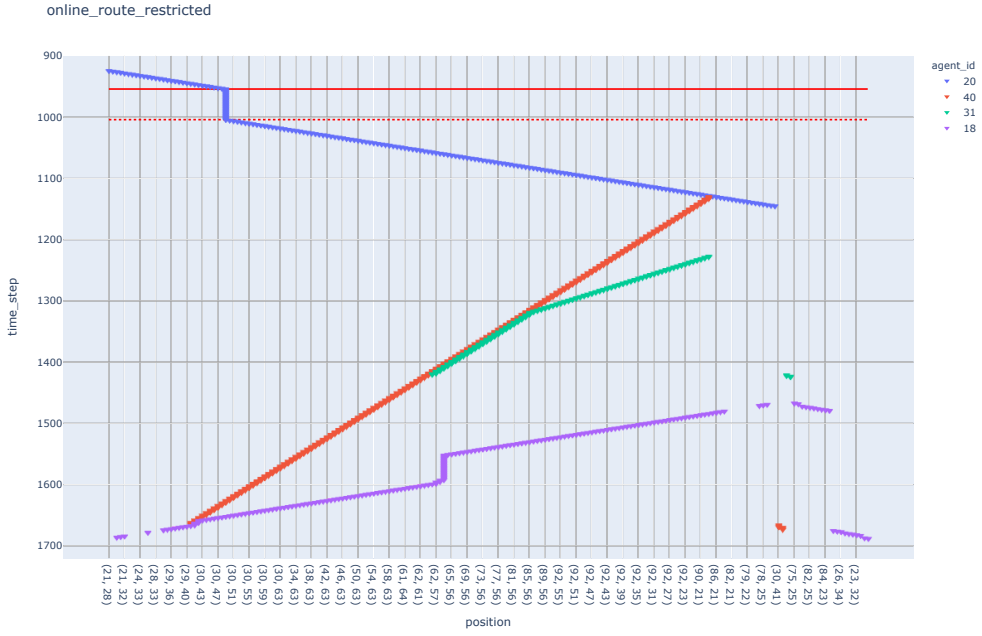
We see that train 40 changes route with respect to the initial schedule in o.u., but not in o.r.r.; on the other hand, in o.r.r. all three trains have an additional delay of 10 time steps compared to o.u. The time resource diagrams for these two situations are shown and discussed in Figure 20.

D More Computation Times

[[ChE: Let's decide what to do with this section after Erik has finalized the computational results in the main text.]]

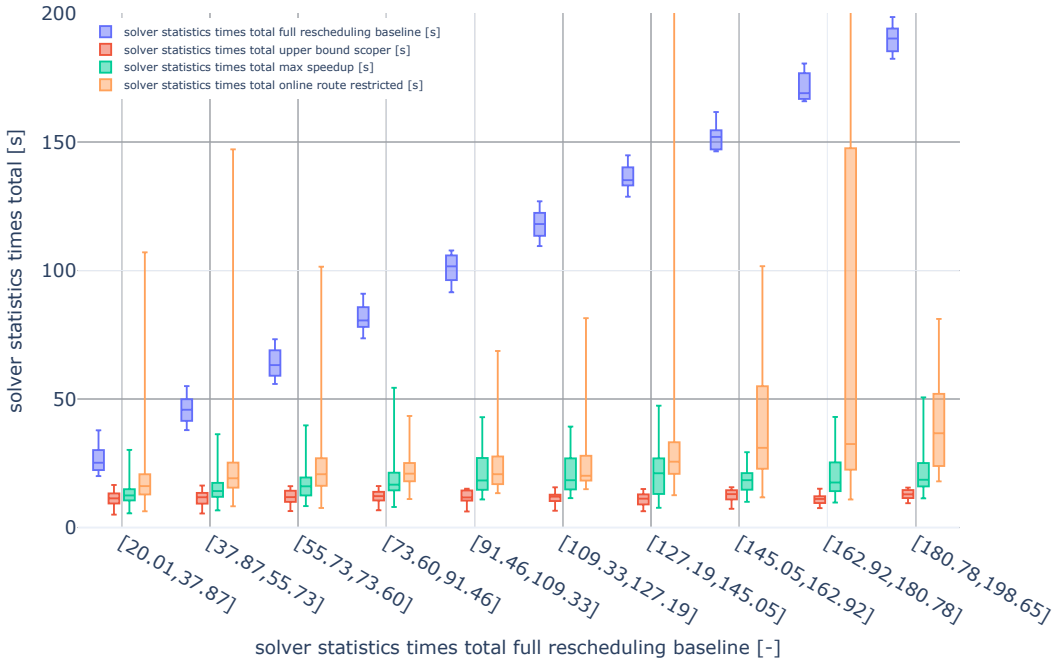


(a) *Time resource diagram for time_resource_online_unrestricted.*

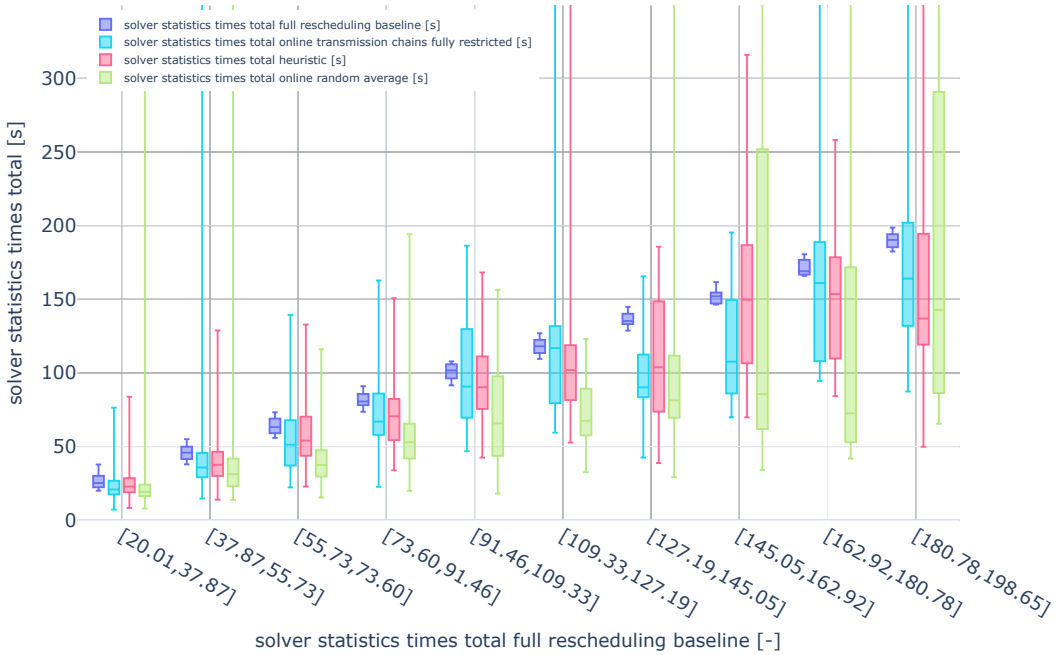


(b) *Time resource diagram for time_resource_online_route_restricted.*

Figure 20: *Time resource diagram two scopes illustrating cost equivalence: train* 20 *(blue) is the train having the malfunction at time step* 954 *for* 50 *time steps (horizontal red line and dashed horizontal red line for malfunction start and end). We see that train* 40 *(red) changes route to start earlier, being delayed but to arrive* 10 *time steps earlier in o.u. than in o.r.r. The secondary delay of* 10 *time steps of train* 40 *in o.r.r. causes tertiary delay in trains* 31 *(green) and* 18 *(purple) of* 10 *time steps; these two trains are running faster than the red one and catch up. In addition, train* 18 *(purple) has a different behaviour: it does not catch up fully onto train* 40 *(red), but pauses and resumes during its run in o.r.r., staying within time windows of size* 60*. The trajectories look as if they overlapped when trains catch up; however, zooming in would should that they do not overlap, the effect being due to the markers for the occupations not being scaled properly.*



(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.*

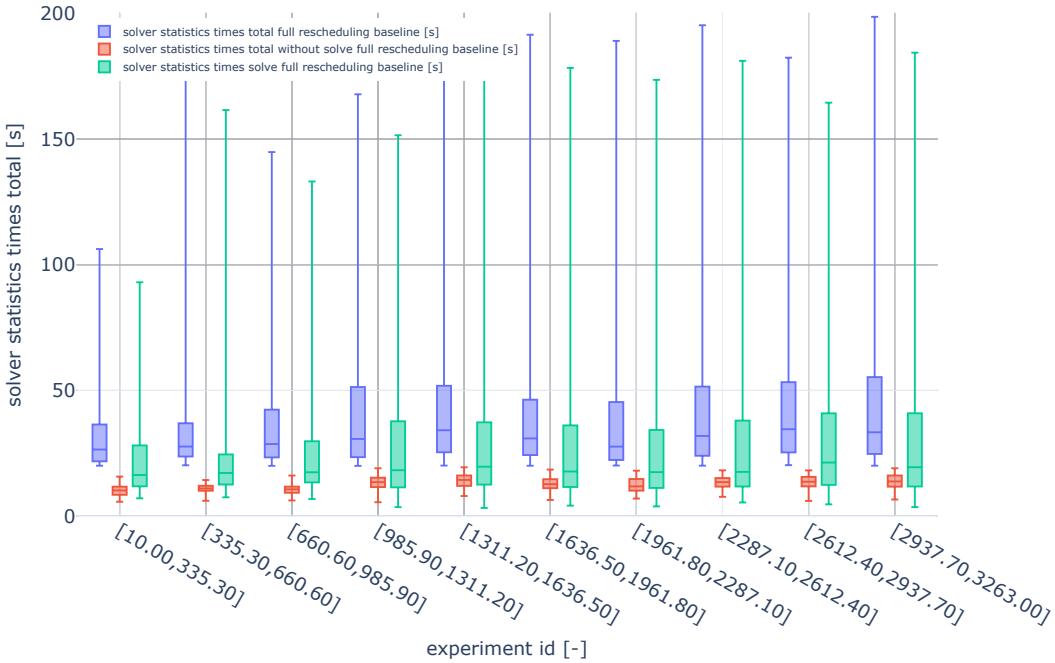
Figure 21: *Re-scheduling times against full re-scheduling time.*

E Solver-related Computational Details

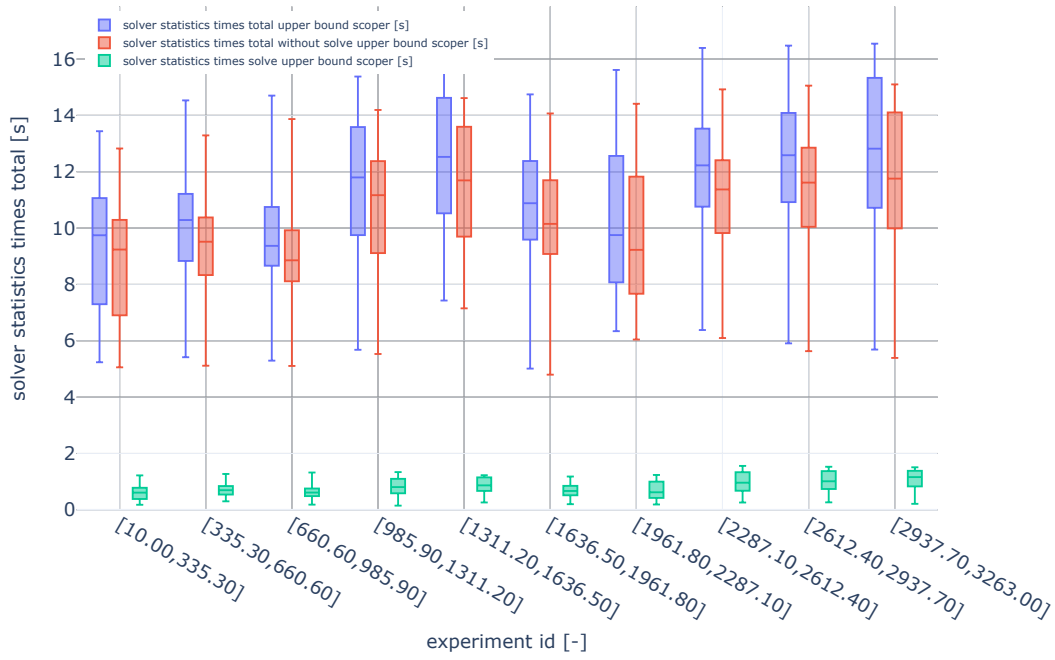
[[ChE: this section needs harmonization with the final scoper terminology - do we need all of them?]] [[ChE: ASP citations]] [[ChE: check with Potsdam]] [[ChE: EN: Experiment id unclear - redo plots in better way?]] This section gives some ASP specific details on the chosen settings of the implementation.

The published solver parameters were found after checking on the SEQ heuristic Abels et al. (2019), with delay model resolution of 2, 5, 10 and the effect of different propagation options of Clingo-dl, which is a grounder and solver for solving ASP modulo Difference Constraints Janhunen et al. (2017).

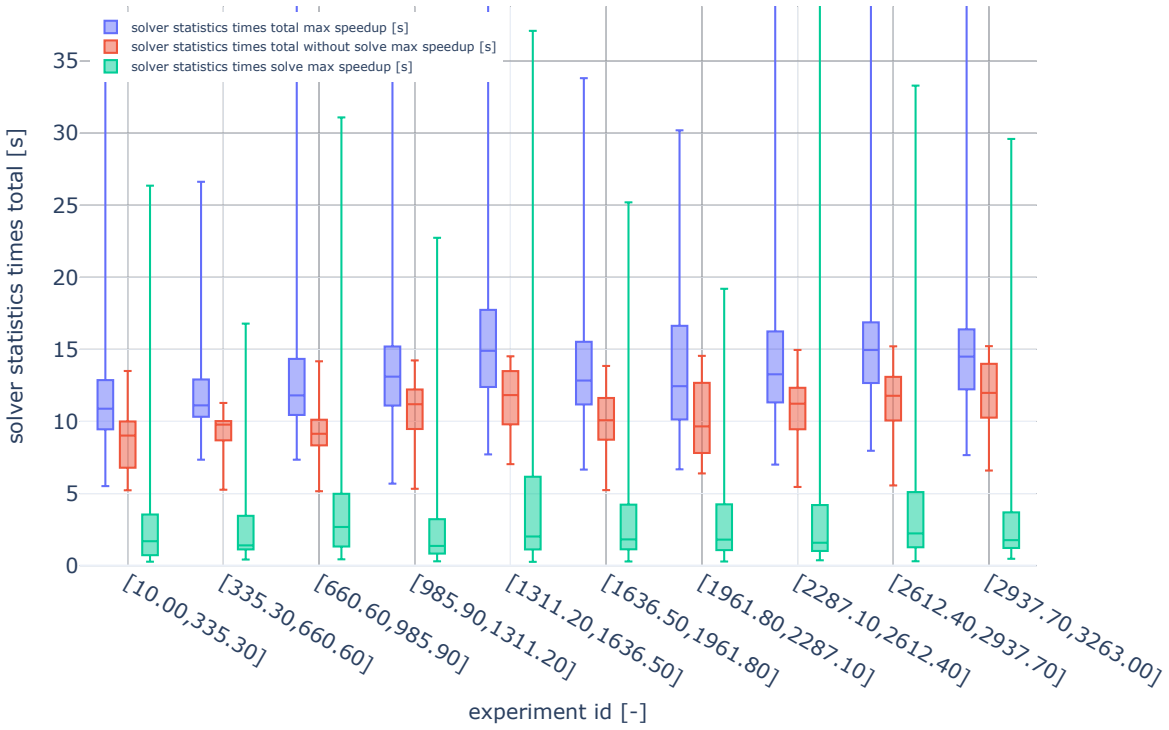
ASP heavily relies on grounding. Therefore, Figure 22 shows the time spent for solving (constraint propagation and conflict resolution) and non-solving (mainly grounding, the pre-processing) phase for the different scopers. It shows that the grounding times seem constant whereas the solve times increase. For a fully online scenario, the grounding times are prohibitive. It has to be said that we have not invested in optimizing the model formulation in this regard.



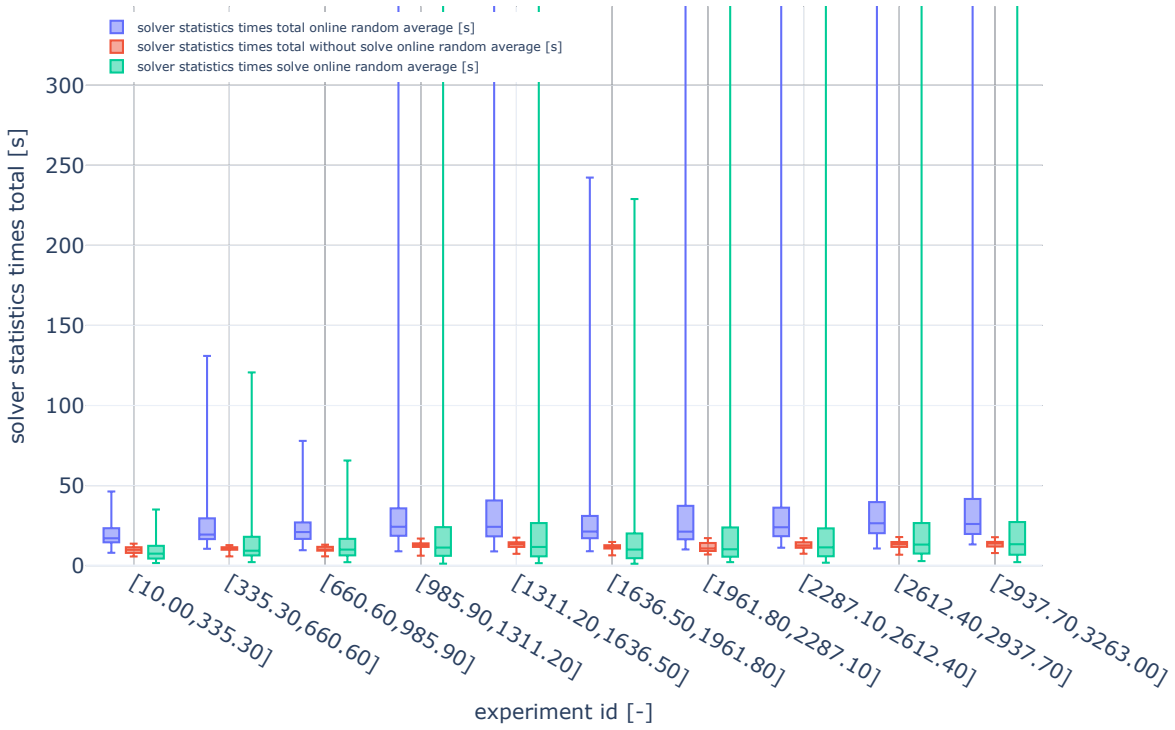
(a) *Total solver time, solve time and non-solve time for online unrestricted*



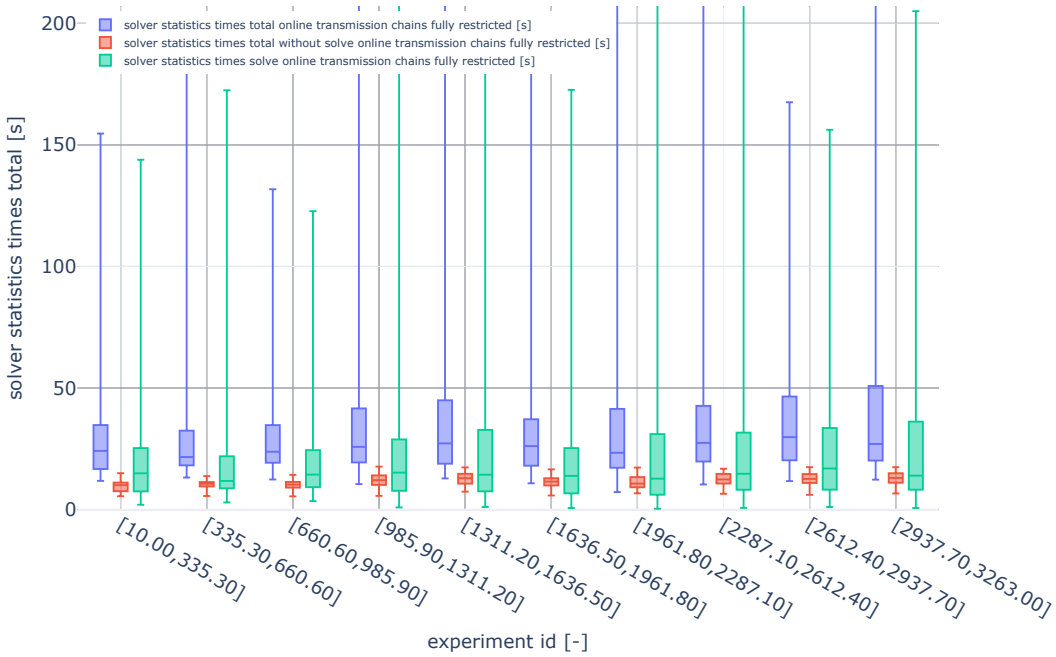
(b) *Total solver time, solve time and non-solve time for online fully restricted*

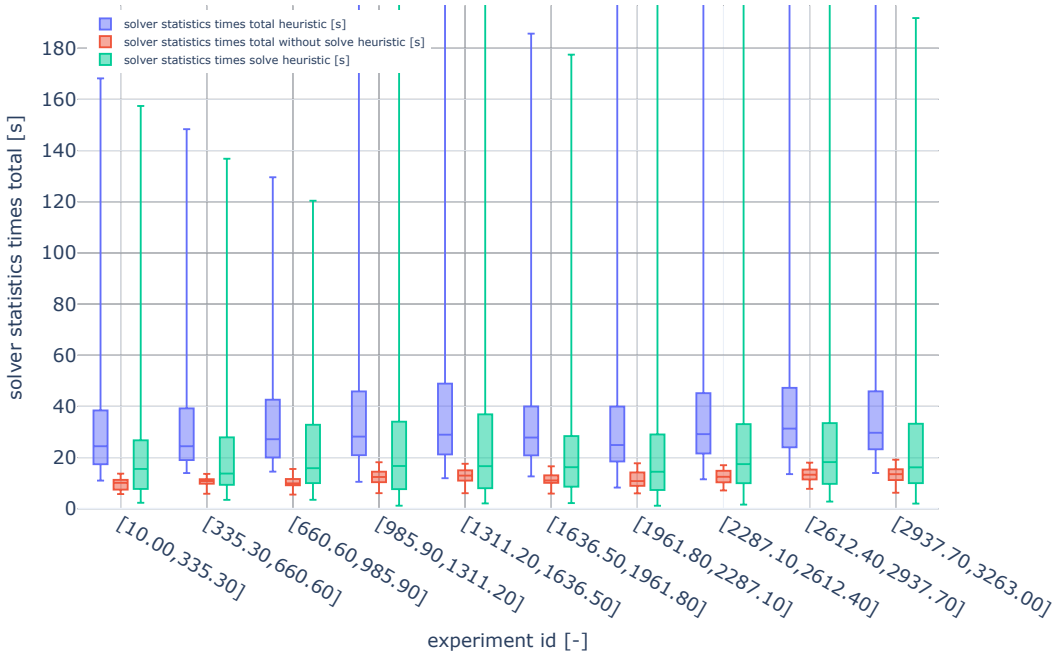


(c) *Total solver time, solve time and non-solve time for offline delta*



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





(e) *Total solver time, solve time and non-solve time for online transmission chains fully and route restricted*

Figure 22: *Absolute total solver time, solve time and non-solve time of the ASP solver per experiment id.*