Constraint-Based In-Station Train Dispatching

Andreas Schutt
Data61, CSIRO, Melbourne, Australia

Matteo Cardellini
DIBRIS, University of Genova, Italy

Jip J. Dekker
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia
ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Melbourne, Australia

Daniel Harabor
Department of Data Science and Artificial Intelligence, Monash University, Clayton, Australia

Marco Maratea
Department of Mathematics and Informatics, University of Calabria, Rende, Italy

Mauro Vallati
Department of Computer Science, University of Huddersfield, UK

Abstract

In-station dispatching is the problem of planning the movements of scheduled trains inside a railway station. Effective solutions for in-station dispatching are important for maximising the utilisation of railway infrastructure and for mitigating the impact of incidents and delays in the broader network.

In this paper, we explore a constraint-based approach to perform in-station train dispatching. Our extensive empirical analysis of multiple modelling, search strategy, and solver choices, performed over synthetically generated, yet realistic, data, shows that our method outperforms the existing planning-based state-of-the-art approach. In addition, we present different optimisation criteria, which can be effortless defined thanks to the constraint-based approach.

2012 ACM Subject Classification Applied computing → Operations research; Computing methodologies → Planning and scheduling

Keywords and phrases in-station train dispatching, train scheduling, railway scheduling, constraint programming, mixed-integer programming

Digital Object Identifier 10.4230/LIPIcs.CP.2025.33

Supplementary Material *Dataset*: `https://github.com/ShortestPathLab/train_dispatching_benchmark`

1 Introduction

Railways play a significant economical role in our society for transporting either goods or passengers, but the increasing volume of people and freight transported on railways is congesting the networks [4]. Train traffic control in railway networks deals with the problem of finding appropriate routes for trains to respect a given timetable, and is usually divided into two main areas: line dispatching [20] and in-station train dispatching [6].[1] The former considers the overall railway network and the routing of trains between different railway stations. The latter is focused on the routing of trains inside a specific station; to deal with delays and disruptions and minimise their overall negative impacts, such as cost penalties for the rail operator and inconveniences for passengers.

[1] We note that both or a combination of both problems are sometimes referred as train or railway scheduling problem in the operation research and constraint programming community.

Despite its importance, the in-station train dispatching problem is often handled manually by experienced operators in charge of one or more connected stations [21, 17, 6]. These operators monitor the situation inside a station and instruct train conductors about which path to follow and when. Operators receive limited support from railway control systems (e.g., RailSys, a popular software package originally developed for Deutsche Bahn) which focus mainly on operational performance indicators and operator support services, such as simulation and validation to measure the quality of human-driven decision-making (see, e.g., [12]). These limitations exist, in part, because in-station train dispatching is a hard complex problem falling in the class of job-shop scheduling problems [19]. Typically, there are three key decisions to be made for each train: what time it should enter the station, which path (including the platform) it should take, and how long it should dwell at the platform. A main complication is that each train needs to be routed through the station such that its occupation of track segments does not overlap with that of other trains.

Several notable works appear in the research literature, which attempt to address the in-station dispatching problem automatically. Mannino and Mascis [21] developed a specialised algorithm that offers real-time re-scheduling over a 1.5 hour planning horizon, but limited to just a few (≤ 8) trains. Their dataset is no longer available. Meanwhile, Kumar et al. [17] consider a CP-based optimisation method which tackles up to 23 trains and substantially larger station layouts. Unfortunately, this approach requires several minutes to solve one instance, decisions for most of the trains are already fixed a priori, and solutions are limited to a small time horizon (10 minutes). Their datasets are not available. Recently, Cardellini et al. [6] made a significant step with a method based on automated planning. Their approach models the problem using PDDL+ [13] and uses a modified version of the planning engine ENHSP [25, 26] for solving. This system can tackle more trains (30+) over a larger time horizon (2+ hours) with solutions computed in just seconds. For confidentiality reasons, their datasets are also not available.
In this paper, we make substantial new contributions to in-station train dispatching:

1. We develop a new constraint-based model using the modelling language MiniZinc [28]. MiniZinc has several compelling advantages compared to PDDL+: it is easier to understand, it does not change for every instance, and it offers the opportunity to explore a range of solution approaches, including different objectives, solving technologies and search strategies.

2. We generate and make available a new benchmark dataset, which is synthetic but based on real historical data. Modelled on a typical medium-sized Italian railway station, we generate a range of problem instances with up to 50 trains over a several hour time horizon – substantially larger than problems previously appearing in the literature.

3. We analyse the performance of our approach using a range of solvers and two different objectives: *makespan* and sum of (individual train) *endtimes*. Results show that our method is competitive in terms of runtime to the PDDL+ planning approach in [6]: we solve more problems overall, we can usually find a first solution with a comparable runtime of just seconds, and we solve most problems to optimality given a few minutes.

We report results for a wide range of solving technologies including constraint programming (CP) and mixed-integer programming (MIP) solvers, i.e., Chuffed, Google's OR-Tools solver CP-SAT, IBM ILOG CP Optimizer, and Gurobi. We believe this is the largest evaluation and empirical comparison yet undertaken for in-station train dispatching.

2 Related Work

In the broader topic of railway traffic control, most works deal with the problem of finding appropriate routes for trains, usually regarding a given official timetable [30]. By contrast, we focus on the problem of in-station train dispatching, where the main challenge is optimising the movement of trains inside stations. Given the hard combinatorial nature of the problem, many works have focused on related sub-problems or on a more abstract formulation of the overall problem. Rodriguez [24] formulated a CP model for performing train scheduling at a junction, which shares some characteristics of a station, but does not include platforms and stops. Other works such as [7, 3, 8] focused on the problem of assigning trains to available platforms, given the timetable and a set of operational constraints, without focusing on the paths that trains should follow to reach the platforms. Meanwhile, Caprara et al. [5] analysed a different perspective and focused on the identification and evaluation of recovery strategies to be applied in case of delays of one or more trains arriving at the controlled station. These strategies include actions such as the use of backup platforms or alternative paths, and are assumed to be readily applicable.

A few works have directly addressed the in-station train dispatching problem. We discussed these works in Section 1 but give more details here. Mannino and Mascis [21] introduced a MIP model for controlling a metro station. Their approach, evaluated on data from a medium-sized Milanese metro station, can effectively manage delays by re-scheduling trains in real-time (1 second per solve iteration). Unfortunately, the approach is limited to small numbers of trains. In Kumar et al. [17] authors consider a CP model for performing in-station train dispatching for a busy Indian terminal. Their approach can deal with a large railway station and more trains, but at the cost of considering only very short time horizons (less than 10 minutes). The approach depends on station-specific optimisations, which means the model and associated gains are not easily generalised to other settings. Another drawback is the approach used to model non-overlap constraints, which ensure paths assigned to trains are collision-free. Their proposed model employs sub-paths of train routes and platforms where a sub-path is a path from one signal/semaphore (see Section 3) to another one (not necessarily the next one) and, thus, it is a higher abstract level than a track segment. Using this approach, trains need to wait until an entire sub-path is released before they can use any track segment that is part of the sub-path. This eliminates some plans and could be avoided if (as in our approach) the non-overlapping constraints were modelled over track segments. A final difference is evaluation, where reported results are all produced by a single solver: IBM ILOG CP Optimizer.

The most recent and closer work to ours is from Cardellini et al. [6]. In this paper, as already mentioned in the introduction, the authors presented an approach based on automated planning with PDDL+, a mixed discrete-continuous extension of the standard PDDL language. A modified version of the PDDL+ planning engine ENHSP is then used for solving. The approach was validated on historical data of a medium-sized railway station from the North-West of Italy provided by Rete Ferroviaria Italiana (RFI), but because of confidentiality issues the authors could not share the data.

Other related CP methods are proposed in [14, 22]. Geske [14] proposed a CP approach for simulating the train timetable for parts of the German railway network, which combines line and in-station train dispatching. The CP model uses the global constraints cumulative and diffn for modelling the non-overlapping constraints over track segments, and the method stops once a solution is found. Masoud et al. [22] implemented a CP model in IBM ILOG CP Optimizer for scheduling coal trains between mines and a port in Australia using the

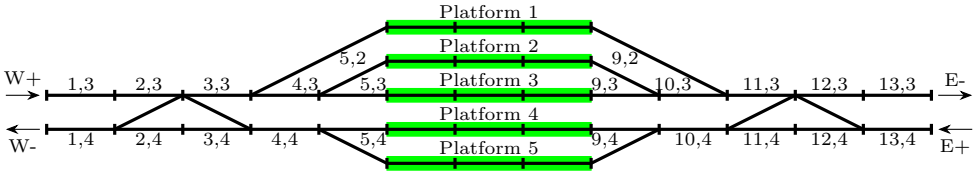


Figure 1 The diagram of the railway station under consideration (adapted from `https://github.com/matteocarde/icaps2021`).

global constraint disjunctive for modelling non-overlapping constraints on tracks between junction points. Their method minimises the makespan while also considering coal demand constraints at the port. A similar problem, and associated CP-model, is developed for freight-rail capacity-evaluation in [15]. We also note that the MiniZinc Challenge [28] has two CP models "train" and "train-scheduling" for line dispatching, but there is no information on their background and origin.

Compared to the CP models in the literature, we model the non-overlapping constraints over individual track segments, which allows a "tight" scheduling of trains leading to fewer delays. We also explore the usage of redundant constraints to strengthen the model, which has not been done before to the best of our knowledge in this context, and investigate the impact of three solution objectives and four solvers on the solving performance.

3 Problem Description and Model

First, we provide the necessary background for the considered in-station dispatching problem. Next, we introduce our basic constraint-based model and additional redundant constraints to strengthen propagation. Then we introduce different solution objectives and finally we discuss search strategies.

3.1 Problem Description

Figure 1 provides an example structure of a railway station. A railway station can be represented as a graph, composed by a set of connected track segments, i.e., the minimal controllable rail units. Their status can be checked via track circuits, that provide information about occupation of the segment and about corresponding timings. Track segments can be divided into two classes, stopping points and interlocking. A stopping point is a track segment in which a train can stop: this is possible if there is a connected platform, i.e., a point in the station used to embark/disembark the train (in green in Figure 1), or at the entrance and the exit points of the railway station (indicated as E and W in the figure). Entry points are segments where the train stops before being allowed to enter into the station (or queues behind other trains); similarly, exit points allow the train to leave the station and enter the outside railway network.

Sequences of connected track segments are organised in itineraries: this is manually done by experts at the specific railway station. While track segments are the minimal controllable units of a station, itineraries describe paths that the trains will follow in order to move within the station. Semaphores are positioned in the station at specific points, and that are the only points at which trains are allowed to stop, usually signalling the beginning and the end of an itinerary. Intuitively, trains are only allowed to stop at the end of an itinerary, where semaphores are placed. For example, all trains in Figure 1 consists of two itineraries. The first one from the station entrance to the end of the platform in their direction and the second one, the first track segment after the end of the platform till the station exit.

A track segment can be occupied by a single train at the time. For safety reasons, a train is required to reserve an itinerary, and this can be done only if the itinerary is currently not being used by another train. While a train is navigating the itinerary, the track segments left by the train are released. This is done to allow trains to early reserve itineraries even if they share a subset of the track segments. A train occupies a track segment for a duration that depends on many variables; e.g., type of the train, length of the train, position of the segment within the itinerary, and weather. Based on historical data, when available, corresponding values can be estimated for planning purposes. A train going through the controlled railway station is running a route in the station, by reserving an itinerary and moving through the corresponding track segments.

Finally, a timetable is the schedule that includes information about when trains arrive at the controlled station, when they arrive at a platform, and the time when they leave a platform. Figure 2 shows a non-overlapping schedule of three trains moving across the station from Figure 1. We show temporal reservations for itineraries leading to and leaving from each platform. In this example, the orange train overtakes the red train via Platform 3. Currently, human train dispatchers rely on tools to visualise the conditions of the controlled station. When making re-scheduling decisions they use data such as occupied track segments (similar to Figure 2), timetable information and extensive communication with train drivers and personnel in the station. Dispatchers operate in real-time and mostly on a reactive basis, according to the arrival time of trains. They are guided by an intuitive understanding about the quality of potential decisions and their knock-on effects.

3.2 Constraint-based Model

The in-station train dispatching problem can be characterised by the tuple (T, E), where $T = \{1, 2, \dots\}$ is the set of *trains* and $E = \{1, 2, \dots\}$ is the set of track *segments* in the station. A train $t \in T$ is specified by its earliest start time $\underline{s}_t^T \in \mathbb{Z}$, and a set of possible *routes* $R_t \subseteq \mathbb{Z}$ through the station.

For the purposes of our model, we distinguish between four train types $y_t^T \in \{\mathsf{std}, \mathsf{ori}, \mathsf{des}, \mathsf{van}\}$. Here std is a *standard* train that enters and exits the station, to and from the external network; ori is an *origin* train that start at a platform from the beginning of the planning horizon and subsequently exits the station; des is a *destination* train that enters the station, terminates at a platform, and occupies the platform for the remaining planning horizon, and; van is a *vanishing* train that enters the station, terminates at a platform, and "vanishes" from the station after completing its stop. A vanishing train models the case when the train leaves the station into the train yard or side-track, which is assumed to always have enough capacity.

A route $r \in R_t$ of a train t is a sequence of itineraries forming the path of the train through the station. Its grounded representation is modelled as a sequence of temporal *reservation blocks* $B_r = \{i, i+1, \dots, j\}$, where $i, j \in \mathbb{Z}$ and $i < j$, and the integers represent the indices of the blocks. A block $b \in B_r$ of route r specifies a track segment $e_b^B \in E$ and a reservation duration $d_b^B \in \mathbb{Z}_0^+$, excluding dwell time (defined below). A train occupies a block b starting from the sum of the end time of its previous block in the sequence B_r and a start offset time $o_b^B \in \mathbb{Z}$. If b is the first block in the sequence, the start offset time is zero.

When moving through a platform, a train has a minimal dwell time $\underline{l}_r^R \in \mathbb{Z}_0^+ = \{0, 1, \dots\}$. We note that even if a train does not take passengers at the station, it might be allowed to stop at the platform, e.g., waiting for a green signal or a passing of another train through a different platform. However, the minimal dwell time is zero. Similarly, we also define a minimal duration for a train to move through the station $d_r^R \in \mathbb{Z}$, excluding the dwell time.

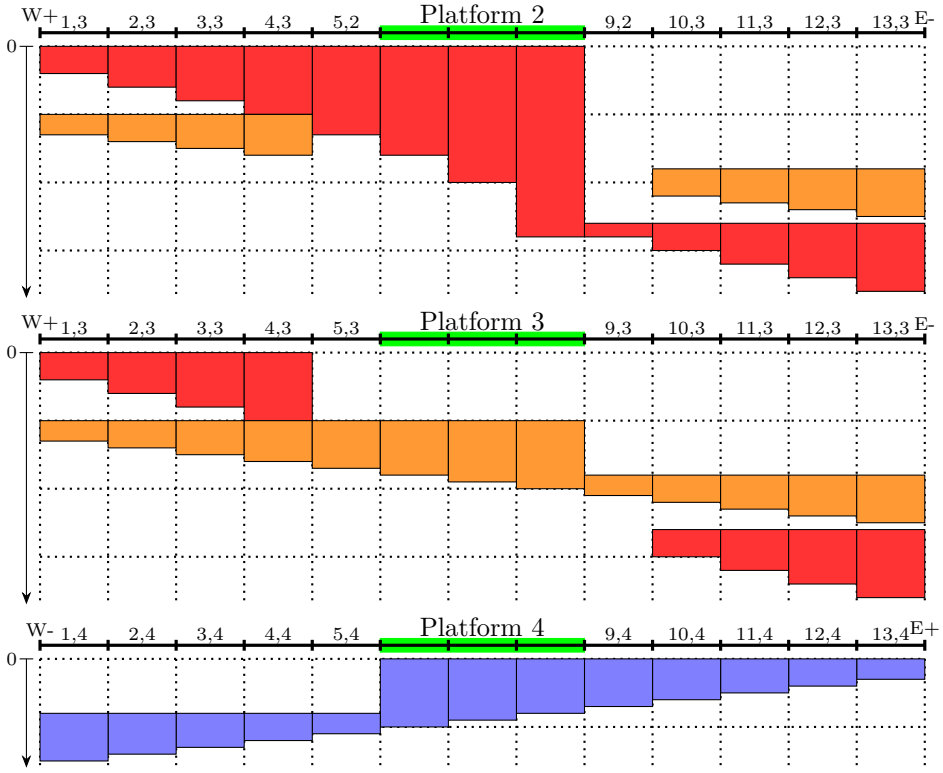


Figure 2 Three Gantt charts of a train schedule of three trains across the station shown in Figure 1, in which the x-axis corresponds to track segments of one route through the station and the y-axis the time. The red, orange, and blue train enters the station at W+, W+, and E+, exits at E-, E-, and W-, and drives via Platform 2, 3, and 4, respectively. The coloured rectangles show the reservation times of the corresponding track segment in the station shown in Figure 1 for each train.

As dwell times are optional, we use a Boolean parameter $p_b^B \in \{\top, \bot\}$ which is true ($\top$) if the block b can contain a non-zero dwell time (i.e., if the train can stop at the block's segment e_b^B) and false ($\bot$) otherwise; the parameter $t_b^B \in T$ refers to the corresponding train. We denote by B the union of all blocks, i.e., $\bigcup_{t \in T} \bigcup_{r \in R_t} B_r$.

Decision Variables. The problem has three variables for each train t, which are the start time variable $\mathbf{s}_t^T$ modelling when trains enter the station or depart from the platform for origin trains, the route variable $\mathbf{r}_t^T$ for selecting the route through the station, and the dwell time variable $\mathbf{w}_t^T$ for modelling the duration a train stops at the platform. In addition, we have optional start time variables $\mathbf{s}_b^B$ and duration variables $\mathbf{d}_b^B$ for each block b, which are used for modelling non-overlapping constraints for each segment in the station. An optional variable is a variable that might be a part of a solution or not. It can be understood as a pair of the integer variables and a Boolean variable that indicates whether the variable is present or absent in a solution. In our case, whether a train traverses a particular segment depends solely on which route has been selected in a solution. Thus, related block variables to that route should be present in the solution while the others absent. All variables are (optional) integer variables.

Constraints. Equations (1–20) show all the constraints of our basic model: (1–5) are the core constraints, (6–11) link the different decision and block variables together, and (12–20) impose bounds on the variables. The constants $\underline{H}$ and $\overline{H}$ represent the start and end time of the planning horizon. The start time is simply $\min_{t \in T} \underline{s}_t^T$, whereas for the end time we compute a trivial upper bound by scheduling the trains on a single track starting with origin trains and ending with destination trains.

$$\mathsf{disjunctive}(Start, Dur) \qquad \forall e \in E : Start = \{\tilde{\mathbf{s}}_b^B \mid b \in B : e_b^B = e\}, \; Dur = \{\tilde{\mathbf{d}}_b^B \mid b \in B : e_b^B = e\} \tag{1}$$

$$\tilde{\mathbf{s}}_b^B = \mathrm{occurs}(\mathbf{s}_b^B) \odot \underline{H} \quad \tilde{\mathbf{d}}_b^B = \mathbf{d}_b^B + (\mathbf{s}_b^B - \underline{H}) \qquad \forall b \in B : p_b^B = \top \wedge y^T_{t_b^B} = \mathsf{ori} \tag{2}$$

$$\tilde{\mathbf{s}}_b^B = \mathbf{s}_b^B \quad \tilde{\mathbf{d}}_b^B = \mathbf{d}_b^B + \overline{H} \qquad \forall b \in B : p_b^B = \top \wedge y^T_{t_b^B} = \mathsf{des} \tag{3}$$

$$\tilde{\mathbf{s}}_b^B = \mathbf{s}_b^B \quad \tilde{\mathbf{d}}_b^B = \mathbf{d}_b^B \qquad \forall b \in B : p_b^B = \bot \vee y^T_{t_b^B} \in \{\mathsf{std}, \mathsf{van}\} \tag{4}$$

$$\mathbf{s}_{t_1}^T \leq \mathbf{s}_{t_2}^T \qquad \forall (t_1, t_2) \in O \tag{5}$$

$$\mathbf{r}_t^T = r \rightarrow \mathbf{s}_t^T = \mathbf{s}^B_{\min(B_r)} \qquad \forall t \in T, \forall r \in R_t \tag{6}$$

$$\mathbf{r}_t^T = r \leftrightarrow \mathrm{occurs}(\mathbf{s}^B_{min(B_r)}) \qquad \forall t \in T, \forall r \in R_t \tag{7}$$

$$\mathbf{d}_b^B = d_b^B \qquad \forall b \in B : p_b^B = \bot \tag{8}$$

$$\mathbf{d}_b^B = d_b^B + \mathbf{w}^T_{t_b^B} \qquad \forall b \in B : p_b^B = \top \tag{9}$$

$$\mathbf{s}_b^B = \mathbf{s}_{b-1}^B \oplus d_{b-1}^B \oplus o_b^B \qquad \forall t \in T, \forall r \in R_t, \forall b \in B_r \setminus \{\min(B_r)\} : \neg(p_{b-1}^B = \top \wedge p_b^B = \bot) \tag{10}$$

$$\mathbf{s}_b^B = \mathbf{s}_{b-1}^B \oplus d_{b-1}^B \oplus o_b^B \oplus \mathbf{w}_t^T \qquad \forall t \in T, \forall r \in R_t, \forall b \in B_r \setminus \{\min(B_r)\} : p_{b-1}^B = \top \wedge p_b^B = \bot \tag{11}$$

$$\mathbf{r}_t^T = r \rightarrow \underline{l}_r^R \leq \mathbf{w}_t^T \qquad \forall t \in T, \forall r \in R_t : p_r^R = \top \tag{12}$$

$$\mathbf{r}_t^T = r \rightarrow \mathbf{w}_t^T = 0 \qquad \forall t \in T, \forall r \in R_t : p_r^R = \bot \tag{13}$$

$$\mathbf{w}_t^T \leq \max_{r \in R_t} \underline{l}_r^R \qquad \forall t \in T : y_t^T = \mathsf{van} \tag{14}$$

$$\mathbf{w}_t^T = 0 \qquad \forall t \in T : y_t^T = \mathsf{ori} \tag{15}$$

$$\underline{s}_t^T \leq \mathbf{s}_t^T \leq \overline{H} \qquad \forall t \in T \tag{16}$$

$$\min(R_t) \leq \mathbf{r}_t^T \leq \max(R_t) \qquad \forall t \in T \tag{17}$$

$$0 \leq \mathbf{w}_t^T \leq \overline{H} - \underline{H} \qquad \forall t \in T \tag{18}$$

$$\underline{H} \leq \mathbf{s}_b^B \leq \overline{H} \qquad \forall b \in B \tag{19}$$

$$0 \leq \mathbf{d}_b^B \leq \overline{H} - \underline{H} \qquad \forall b \in B \tag{20}$$

Constraints (1–4) ensure train reservations are non-overlapping. We model this using the global constraint disjunctive for each segment $e \in E$ in our network (1). The disjunctive constraint guarantees a non-overlapping execution of tasks specified by their start time variables and their duration variables. In our case, we create an optional task by optional start time variables $\tilde{\mathbf{s}}_b^B$ and duration variables $\tilde{\mathbf{d}}_b^B$ for each block b that corresponds to the segment e. Both variables are assigned using their block variables $\mathbf{s}_b^B$ and $\mathbf{d}_b^B$ accounting for the type of train and whether there is a stop at the block. There are three cases to consider, depending on the type of train.

- The first case (Constraint 2) covers an origin train at the stop, that is, at the platform. In this case, the corresponding segment must be blocked from the beginning of the planning horizon $\underline{H}$ until the train leaves it. Note that the return value of the function occurs($\mathbf{s}_b^B$)

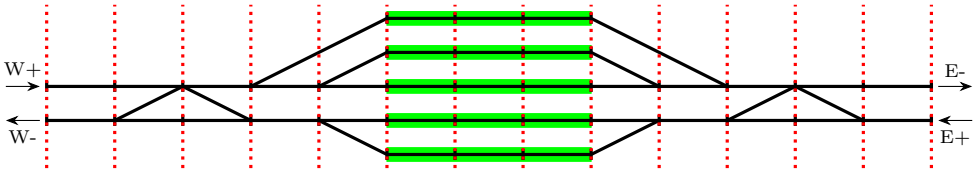


Figure 3 The diagram shows 13 columns of "parallel" track segments separated by red vertical lines of the railway station shown in Figure 1.

is 1 if the variable $\mathbf{s}_b^B$ is present, and absent otherwise. Additionally, we note that the result of the multiplication operator $\odot$ is only present if both multipliers are present. Thus, $\tilde{\mathbf{s}}_b^B$ is only present if $\mathbf{s}_b^B$ is present.

- The second case (Constraint 3) models destination trains having their stop, for which we artificially extend their task's duration by the end of the planning horizon $\overline{H}$, so that the platform is blocked for the remaining planning horizon.
- The last case (Constraint 4) covers all other cases.

Constraint (5) imposes a partial order on trains, in which order they can enter the station from the same entry point (e.g., W+ and E+ in Figure 1), where O is the set of all train pairs (t_1, t_2), for which an order must be imposed. For example, if two trains t_1 and t_2 are entering the station at the same entry point, then t_1 must enter before t_2 if $\underline{s}_{t_1}^T \leq \underline{s}_{t_2}^T$.

Constraints (6–7) ensure that if a route r is chosen (i.e., $\mathbf{r}_r^T = r$) then the start time of the first block of the route $\mathbf{s}_b^B$ equals to the train's start time and $\mathbf{s}_b^B$ is only present in this case. Constraints (8–9) model their duration, which is the reservation time if there is no stop or the reservation time plus the train's dwell time otherwise. Constraints (10–11) connect the start time variables of blocks in the same route where $b-1$ is the immediate predecessor of b and the addition operator $\oplus$ only returns the sum if both summands are presents; otherwise, the result is absent.

Constraint (12) refines the lower bound on the trains' dwell time in the case the route has a stop, while (13) assigns it to zero in the case that there is no stop. Constraint (14) imposes an upper bound on the train's dwell time if the train is vanishing, whereas the dwell time is set to zero for origin trains (15). Constraints (16–20) provide general lower and upper bounds on the decision and block's variables.

3.3 Redundant Constraints

Redundant constraints are additions to a model that do not change the solution space, but which might provide additional propagation for a CP solver. A potential weakness of our model are the non-overlapping constraints (1), for which the start time variables are optional when the train has multiple routes through the station. To introduce redundant constraints we observe that railway stations have a "regular" pattern (e.g., tracks are parallel, "parallel" track segments have similar length and similar position in their itinerary). We model these "parallel" track segments using the global constraint cumulative [1]. For instance, Figure 3 shows a possible separation of track segments by vertical lines. These vertical lines split the station in 13 *columns*, for which one can impose a redundant constraint for each. We note that a segment can be part of multiple columns.

Let C be the set of columns, $E_c^C \subseteq E$ a set of segments in the column c and $T_c^C \subseteq T$ the set of trains that have at least one route through the column. Then, we model the redundant constraints as follows for a column $c \in C$ with $|T_c^C| > |E_c^C|$.

$$\mathsf{alternative}(\mathbf{s}_t^c, \mathbf{d}_t^c, S, D) \qquad \begin{aligned} \forall t \in T_c^C : S = \{\mathbf{s}_b^B \mid b \in B : e_b^B \in E_c^C \wedge t_b^B = t\}, \\ D = \{\mathbf{d}_b^B \mid b \in B : e_b^B \in E_c^C \wedge t_b^B = t\} \end{aligned} \tag{21}$$

$$\min D \leq \mathbf{d}_t^c \qquad \begin{aligned} \forall t \in T_c^C : D = \{d_b^B + (p_b^B = \top \ ? \ \mathbf{w}_t^T : 0) \\ \mid b \in B : e_b^B \in E_c^C \wedge t_b^B = t\} \end{aligned} \tag{22}$$

$$\mathsf{cumulative}(S, D, U, |E_c^C|) \qquad \begin{aligned} S = \{\tilde{\mathbf{s}}_t^c \mid t \in T_c^C\}, \quad D = \{\tilde{\mathbf{d}}_t^c \mid t \in T_c^C\}, \\ U = \{1 \mid t \in T_c^C\} \end{aligned} \tag{23}$$

$$\tilde{\mathbf{s}}_t^c = \mathrm{occurs}(\mathbf{s}_t^c) \odot \underline{H} \qquad \tilde{\mathbf{d}}_t^c = \mathbf{d}_t^c + (\mathbf{s}_t^c - \underline{H}) \qquad \forall t \in T_c^C : (\exists b \in B : t_b^B = t \wedge p_b^B = \top) \wedge y_t^T = \mathsf{ori} \tag{24}$$

$$\tilde{\mathbf{s}}_t^c = \mathbf{s}_t^c \qquad \tilde{\mathbf{d}}_t^c = \mathbf{d}_t^c + \overline{H} \qquad \forall t \in T_c^C : (\exists b \in B : t_b^B = t \wedge p_b^B = \top) \wedge y_t^T = \mathsf{des} \tag{25}$$

$$\tilde{\mathbf{s}}_t^c = \mathbf{s}_t^c \qquad \tilde{\mathbf{d}}_t^c = \mathbf{d}_t^c \qquad \begin{aligned} \forall t \in T_c^C : (\forall b \in B : t_b^B \neq t \vee p_b^B = \bot) \\ \vee \quad y_t^T \in \{\mathsf{std}, \mathsf{van}\} \end{aligned} \tag{26}$$

Constraint (21) creates one mandatory "master" task with the start time variable $\mathbf{s}_t^c$ and duration variable $\mathbf{d}_t^c$ for each train $t \in T_c^C$ passing through the column c by linking both variables to their corresponding blocks' optional start time variables and duration variables from the different routes of the train t. The global constraint alternative is used, which ensures that at most one of the blocks' optional start time variable is present in a solution and, if so, then $\mathbf{s}_t^c = \mathbf{s}_{b'}^B$ and $\mathbf{d}_t^c = \mathbf{d}_{b'}^B$ where b' is the "present" block (i.e., the block belonging to the chosen route); otherwise, $\mathbf{s}_t^c$ is absent. In our case, $\mathbf{s}_t^c$ is present, which forces that exactly one of the blocks' start time variable is present. Constraint (22) imposes a minimum duration on the "master" task duration, but it might be redundant to (21) if the solver natively supports the global constraint alternative. Constraint (23) ensures that at most $|E_c^C|$ trains use one of the segments in E_c^C at any time by setting a task with a start variable $\tilde{\mathbf{s}}_t^c$, duration variable $\tilde{\mathbf{d}}_t^c$, and a resource requirement of 1 for each train t. These variables are then connected to their "master" task's variables $\mathbf{s}_t^c$ and $\mathbf{d}_t^c$ in the constraints (24–26) depending whether the train can have a stop in the column and its train type similar to the constraints (2–4) on page 7.

So far, we characterise a column c via its set of "parallel" segments E_c^C, but we did not specify how we determine this set because it may not contain all "parallel" segments. There are cases in which "parallel" segments belong to different columns because there is no overlap in trains using these "parallel" segments. We determine the set of columns by computing the connected components in an undirected graph for each set of "parallel" segments, where a node represents one "parallel" segment and two nodes are connected if there exists a train having two routes using the corresponding "parallel" segments. Such a partitioning of "parallel" segments leads to stronger redundant constraints.

3.4 Solution Objectives

While the planning method in [6] is tailored to only provide a good first solution and cannot be easily adapted to an optimisation method, for limits of both modelling and solving phases, our model is flexible in this sense, and we can explore different objectives while using the same constraint-based model. We explore three distinct objectives.

Listing 1 The standard search strategy.

```
ann: std = seq_search([
    int_search(start, smallest, indomain_min),
    int_search(route, smallest, indomain_min),
    int_search(dwell, smallest, indomain_min)  ]);
```

Listing 2 The fixed-order search strategy where the array `est` contains the trains' t earliest start times $\underline{s}_t^T$.

```
array [int] of int: order = arg_sort(est);
ann: fixed = int_search(
    [[start[order[i]], route[order[i]], dwell[order[i]]][j]
    | i in index_set(order), j 1..3],
    input_order, indomain_min );
```

Listing 3 The priority search strategy.

```
ann: prio = priority_search( start,
    [ int_search([start[t], route[t], dwell[t]],
                                  input_order, indomain_min)
    | t in T],   smallest, indomain_min );
```

satis first solution subjected to (1–20) and the redundant constraints (21–26) for columns under consideration

makespan $\min \max_{t \in T}(\mathbf{s}_t^T + d^R_{\mathbf{r}_t^T} + \mathbf{w}_t^T)$ subjected to (1–20) and the redundant constraints (21–26) for columns under consideration

endtimes $\min \sum_{t \in T}(\mathbf{s}_t^T + d^R_{\mathbf{r}_t^T} + \mathbf{w}_t^T)$ subjected to (1–20) and the redundant constraints (21–26) for columns under consideration

While the objective **satis** is the same as the planning method, **makespan** optimises the trains at the end of the planning horizon so that all trains exit the station as earliest as possible with less interference to the next planning horizon, and **endtimes** optimises the total amount of end times, which relates to minimising the total amount of delay. We highlighted that we only need to change one line of MiniZinc to change the solution objective. Thus, it would be easy to explore more objectives, e.g., minimising delays and the lexicographical optimisation of minimising the makespan and then the sum of end times.

3.5 Search Strategies

In CP, search strategies can have a significant impact on the solution quality and the solving efficiency. We explore different search strategies only over all the decision variables. For readability, we describe the strategies in the MiniZinc language as search annotations, in which the MiniZinc arrays `start`, `route`, and `dwell` contain the start time ($\mathbf{s}_t^T$), the route ($\mathbf{r}_t^T$), and the dwell time ($\mathbf{w}_t^T$) variables for all trains t, respectively.

standard The standard search (Listing 1) assigns all start time variables before assigning the route and dwell time variables. The search incentivises the earliest start of all trains while keeping the route and dwell time flexible by selecting the train with the earliest possible start time in $\mathbf{s}_t^T$ and assigning this time to the variable.

fixed-order The fixed-order search (Listing 2) groups the three decision variables of a train together and assigns them first in the order of start time, route, and dwell time before fixing the variables of the next train. The trains are sorted in non-descending order of

the trains' earliest start time $\underline{s}_t^T$. Compared to the standard search, it fixes all variables of train, meaning that all corresponding block start time variables become present, and a solver can perform stronger propagation of the non-overlapping constraints earlier in the search. However, the fixed order of the trains does not incentivise the earliest start of all trains as the standard search.

priority The priority search (Listing 3) combines the standard and fixed-order search by grouping the train's decision variables as in the fixed-order search and selecting the train with the earliest possible start time in $\mathbf{s}_t^T$. However, the priority search [11] is not part of MiniZinc distribution and is only supported by Chuffed among the four considered solvers.

free The "free" search is a solver-specific generic search, which differs between solvers. For some solvers, it can be combined with a user-defined search at the solver's discretion (e.g., Chuffed [9, 10] switches between both searches when restarting).

restart For learning solvers as, e.g., Chuffed, a search can be combined with a restart policy, in which the solver restarts the search after a given number of failures are encountered in the search. We explore a geometry restart policy with a factor 1.5 and a base 100, in which the first restart is triggered after 100 failures and the n-th restart after $100 \cdot 1.5^{n-1}$.

4 Experiments

We carried out numerous experiments to test different modelling and solver choices, and compare the planning method in [6] on instances of up to 50 trains. We describe our setup in terms of hardware, instance generation procedure and solver selection and configuration.

Hardware. Our model was implemented in MiniZinc 2.9.2 and all solvers were executed using MiniZinc with default parameters unless otherwise stated. All experiments were conducted on a computational cluster, where each method was given exclusive access to a single core of an Intel Xeon Platinum 8260 CPU at 2.4GHz, 16 GB of RAM. Due to confidentiality, the planning method was run on a single core of a MacBook Pro Mid-2015 machine having a 2.5GHz Intel Core i7 (4870HQ) with 16 GB of RAM. According to the CPU Benchmark at `www.cpubenchmark.net` the MacBook Pro processor achieves a 10% lower mark for single thread computations compared to our cluster's processors. Thus, we conservatively assume the cluster processor is twice as fast as the other one for runtime comparison. We imposed a runtime limit of 300 seconds. The full list of our experimental results are available in Section A.

Instance Data. We took the railway station as depicted in Figure 1, which represents a typical medium-size Italian railway station. For this station, we created 141 realistic instances derived from real-world instances, ranging from one train up to 50 trains. These instances are available at `https://github.com/ShortestPathLab/train_dispatching_benchmark`. For each train, we uniformly chose the train type, the entry/exit points, the platform at which an origin starts, the arrival time of the train, selected from the integer interval $[0, 200 \cdot |T|]$ where a time unit represents one second, and whether or not the train has a stop at the platform taking passengers. The number of origin, destination, and vanishing trains are capped at 5 each. Their sizes are distributed as follows:

1–19 trains: (114 instances) We generate six instances for each number of trains. The maximal optimal makespan for these instances is about 70 minutes.

20–50 trains: (27 instances) We generate three instances for each number of trains with a step size of 5 (i.e., 20, 25, 30, etc.) and a further three instances with 21 and 22 trains. The maximal optimal makespan for these instances is about 170 minutes (almost 3 hours).

In addition, we collect 9 instances of between 1 and 5 trains, available from `https://github.com/matteocarde/icaps2021`. These instances were created by the authors of [6] for the same station (they do not appear in the paper). The maximal optimal makespan for these instances is about 7 minutes.

Solvers. Since our model is written in MiniZinc, we only chose solvers with a MiniZinc interface. We selected Google's OR-Tools solver CP-SAT version 9.12.4544 [23], the best performing CP solvers in recent MiniZinc challenges [28], Chuffed version 0.13.2 [9, 10], a CP solvers known for performing well on scheduling problems (see e.g. [16, 27, 29]), IBM ILOG CP Optimizer version 12.0.1 [18], a commercial CP solver also known for performing very well on scheduling problems, especially when optional tasks are involved, and its usage in the related work [22, 17], and Gurobi version 12.0.1 as the best performing MIP solver in recent MiniZinc challenges. We note that modelling in the solver's interface and having the full control over it may lead to a better performance, but at the cost of implementing equivalent models in different solver interfaces, and maintaining each of them accordingly, which is onerous, and requires a certain expertise level in each of these solvers. Moreover, for MIP solvers, MiniZinc uses advanced linearisation techniques producing models, which are often on-par with a tailored approach [2], and, for CP solvers, MiniZinc's high-level modelling language is close to the native ones in general. Thus, we do not expect a significant performance difference for Chuffed, CP-SAT and Gurobi. For CP Optimizer, this may not be the case, because this solver has a limited MiniZinc interface which does not support e.g., its global constraint alternative.

4.1 Search Strategies and Redundant Constraints

Tables 1 and 2 show the results of different combinations of solvers and search strategies, and different options for using redundant constraints. Table 1 lists the results for the objective **makespan** while Table 2 for **endtimes**. If a search strategy has the postfix "+" then it was combined with the restart policy **restart**. For each combination (a row in the table), we highlight the best performing option for the redundant constraints in *italic*, where an option is better than another one if it finds solutions for more instances (i.e., the sum of optimal and suboptimal solutions) or its average runtime is faster in the case of a tie. If this combination is also the best performing across all search strategies for a solver, then it is shown in **bold**. We test these five options for redundant constraints: (*none*) no redundant constraints, (*border*) only redundant constraints on columns bordering the external railway network, (*platform*) only redundant constraints on columns with track segments at platforms, (*b+p*) options (*border*) and (*platform*), and (*all*) redundant constraints on all columns.

Search Strategies. CP-SAT's free search is faster on average and optimally solved more instances than the standard and fixed-order search, independent of the redundant constraint option and solution objective. We note that CP-SAT pre-terminated with an error in the instance parsing stage for five instances, while it could find a solution for all other 145 instances. For Chuffed, there are more search options and a different performance outcome compared to CP-SAT. Only the fixed-order search with/without restart and priority search with restart can find a solution for every instance in the given time limit, and their performance is similar for the different redundant constraint options and solution objectives.

Table 1 Comparison of the of redundant constraints when minimising the *makespan*. For each method, we present the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), and the average runtime of all instances in seconds (time).

		none			border			platform			b + p			all		
Solver	Search	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time
Planner			137	28												
Chuffed	standard	110	116	84	*111*	*116*	*82*	109	115	86	110	115	85	110	115	86
	standard+	*110*	*117*	*84*	111	116	82	109	115	86	110	115	85	110	115	86
	fixed-order	136	150	37	**140**	**150**	**30**	135	150	39	138	150	31	138	150	32
	fixed-order+	135	150	38	*139*	*150*	*31*	134	150	39	138	150	31	138	150	32
	priority	126	139	55	*132*	*140*	*46*	126	139	55	131	140	46	130	139	47
	priority+	133	150	44	*138*	*150*	*34*	133	150	46	137	150	35	137	150	36
	free	*137*	*142*	*44*	132	137	51	135	140	47	132	137	54	126	136	62
CP Opt.	free	**139**	**150**	**32**	131	142	47	42	42	218	42	42	218	40	40	228
CP-SAT	standard	*132*	*141*	*44*	129	133	49	129	134	53	128	132	52	128	132	55
	fixed-order	134	145	39	*138*	*145*	*33*	134	145	40	138	145	34	137	145	35
	free	142	145	26	**142**	**145**	**25**	141	145	31	142	145	26	142	145	28
Gurobi	free	136	140	36	**138**	**140**	**32**	132	136	46	135	137	44	129	131	65

To our surprise, priority search without restart did not find a solution for all the instances, which indicates that Chuffed's propagation might not be strong enough to propagate the decisions on one train to other trains. Chuffed's free search optimally solved about 20 instances more than any of its other searches when minimising the sum of end times, but had problems finding a solution for some instances. Moreover, Chuffed has a wide variability on the performance for the different search strategies for minimising the makespan, so that Chuffed's virtual best solver optimally proves 145 instances and only requires an average runtime of 15 seconds while only using redundant constraints on border columns (full results are available in Section A). It shows that no search strategy dominates the other ones in this case. We note that this virtual best solver significantly outperforms any other solver.

Redundant Constraints. It is not the best choice for any solver to have them on all columns or only on the platform columns. This indicates that redundant constraints are not beneficial on all columns, which means that are not bottlenecks as, e.g., platform columns, in dispatching trains. We note that each additional redundant constraint comes with a cost for the solver, which needs to handle the additional variables and constraints during the solving process. For example, the flattening time that MiniZinc needs to convert the model with an instance increased by about 30% and 50% for CP solvers and Gurobi, respectively, which normally correlates to an increase in the problem size. Interestingly, what option is the best depends on solver and solution objectives, which, we suspect, is related to the different propagation strength of the solvers' global constraints and their implemented free search. For CP Optimizer, we see drastic performance deterioration when more redundant constraints are used, which points to that the additional variables created by the redundant constraints negatively impact its search decisions. There is a clear picture across the remaining solvers when minimising the makespan, that having the redundant constraints only for border columns will lead to the best performance of a solver. There is no uniform picture when

Table 2 Comparison of the of redundant constraints when minimising the *sum of end times*. For each method, we present the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), and the average runtime of all instances in seconds (time).

		none			border			platform			b + p			all		
Solver	Search	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time	opt	sat	time
Planner			137	28												
Chuffed	standard	97	117	110	*101*	*117*	*106*	98	115	109	101	115	106	99	115	107
	standard+	97	117	110	*101*	*117*	*107*	98	116	109	101	115	106	99	114	107
	fixed-order	103	150	101	*109*	*150*	*89*	104	150	100	*109*	*150*	*89*	108	150	92
	fixed-order+	103	150	101	108	150	90	104	150	101	*109*	*150*	*90*	107	150	92
	priority	102	139	101	108	140	92	103	139	99	*108*	*140*	*90*	107	139	91
	priority+	102	150	101	108	150	90	103	150	100	**110**	**150**	**87**	109	150	89
	free	122	141	72	128	137	61	*125*	*141*	*70*	126	134	66	121	132	71
CP Opt.	free	**106**	**150**	**107**	107	139	108	41	42	220	42	42	220	39	40	228
CP-SAT	standard	*112*	*141*	*82*	116	133	74	111	133	86	114	132	78	114	132	82
	fixed-order	111	145	87	*117*	*145*	*72*	109	145	91	116	145	76	115	145	79
	free	126	145	56	**126**	**145**	**55**	125	145	59	126	145	57	126	145	59
Gurobi	free	**133**	**140**	**45**	136	138	38	129	137	54	132	135	47	130	132	60

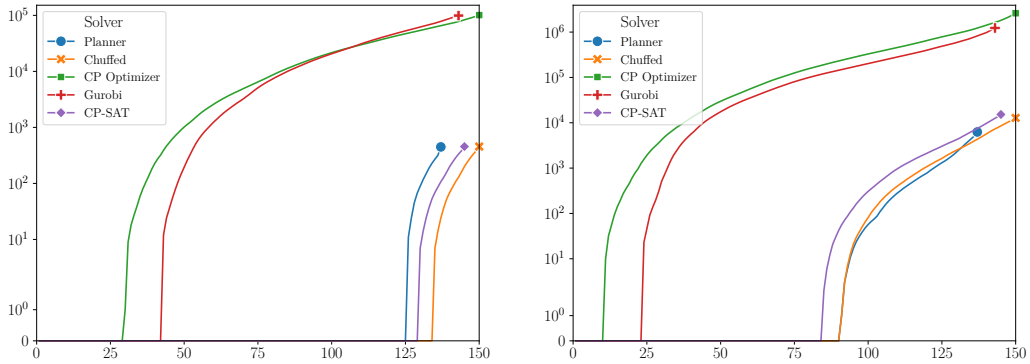
minimising the sum of end times, The best option is (*b+p*) for Chuffed, (*border*) for CP-SAT and Gurobi, and (*none*) for CP Optimizer, which shows that the differences in the solvers' architectures can differently impact their performance for different modelling choices.

Other Observations. The state-of-the-art planning approach in [6] was only able to find solutions for 137 instances whereas Chuffed found solutions for all instances and closed 147 across all cofigurations. CP solutions are competitive to the planning approach in terms of the makespan, sum of end times and computational runtime (see Appendix A for full details). As expected, the problem of minimising the sum of end times is more difficult to solve, which is reflected by the lower number of proven optimal solutions and higher average runtime for any solver because minimising the makespan only minimises the end time of the last train in the schedule.

4.2 Comparison to the State of the Art

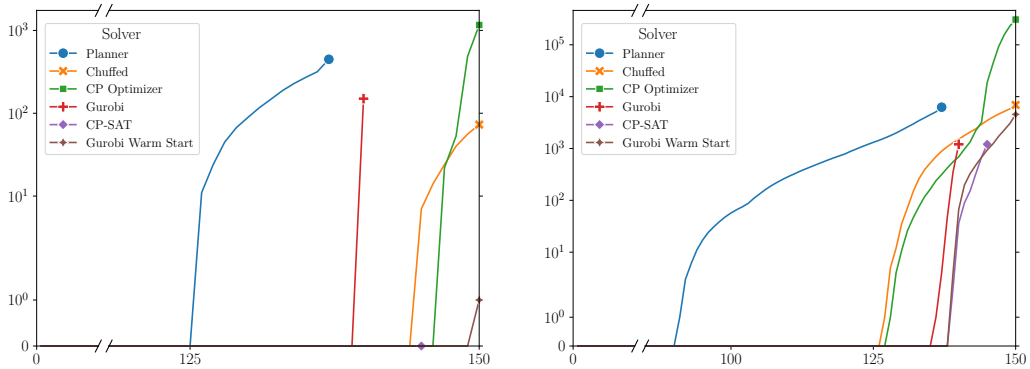
The planner [6] applies a dedicated search heuristic method for finding a first solution with a minimal flow-on impact beyond the station. First, we compare the planner to the best solver combination on the quality of the first solution found regarding their makespan and their sum of end times, and then to the best solution found within the runtime limit.

For the first comparison, the solvers used the solution objective **satis**, and these solver combinations: Chuffed and CP-SAT with fixed-order search and no redundant constraints, and CP Optimizer and Gurobi with free search and no redundant constraints. Figure 4 shows the cumulative differences between the best known objective value for the makespan (Figure 4a) and the sum of end times (Figure 4b) and the solvers' first solution. While the planner does not find a solution for 13 instances, when it found a solution it was always so good as the solutions of other solvers and better than the other solutions for some instances in terms of the makespan and the sum of end times. Unsurprisingly, Chuffed and CP-SAT using the fixed-order search, which incentivised scheduling trains as earliest as possible,



(a) Comparing the makespan. **(b)** Comparing the sum of end times.

Figure 4 The cumulative difference (y-axis) between the first solution found by each method and the best known solution for each solution objective. The x-axis shows the number of instances.



(a) Minimising the makespan. **(b)** Minimising the sum of end times.

Figure 5 The cumulative difference (y-axis) after 5 minutes of optimisation by each method and the best known solution for each solution objective. The x-axis shows the number of instances.

are competitive to the planner. While CP Optimizer and Gurobi first solution quality are understandingly much worse. Overall, Chuffed performs the best in comparison to the other solvers and planner, because it quickly finds a solution for each instance while incurring sometimes a small cost in the solution quality compared to the planner. We note that when only considering the instances for that CP-SAT does not terminate in an error, CP-SAT is the best performing solver.

The considered planner, as all systems solving PDDL+ planning instances, works in satisfying mode, i.e. it terminates after finding the first solution. It cannot be easily modified to operate in the so-called anytime mode, where better solutions are found over time. In our approach, we only have to change the solution objective to one of the two minimisation objectives to find better solutions over time. We used the best solver combinations identified in the previous section, which are highlighted in **bold** in Tables 1 and 2. In addition to these combinations, we also use Gurobi's option to warm start their search by passing in the first solution found by Chuffed using the search strategy *fixed-order* and no redundant constraints to overcome Gurobi's difficulty to finding a solution for all instances. Figure 5 shows plots of the cumulative differences between the best known objective value for the makespan (Figure 5a) and the sum of end times (Figure 5b) for the solvers' best found

solution within five minutes. It is clear that providing more solving time, all solvers find much better solutions than the planner on the harder instances. This highlights that a significant reduction of train delays can be made when the search continues after a first solution is found. Warm starting Gurobi with Chuffed's first solution has not only solved the issue with finding a solution for all instances, but also significantly increased its performance so that it is the best performing solver for both objectives. For minimising the makespan (the sum of end times) it optimally solved 146 (137) instances and found a feasible solution for the remaining 4 (13) instances while having an average runtime of 18 (36) seconds.

Discussion. The solutions obtained show that the solution found by the planning approach are sub-optimal for harder instances indicating the potential in the reduction of the flow-on impacts of delays beyond a station. The best solver for optimising both objective was a combination of Chuffed and Gurobi: Chuffed finds the first solution, which is then injected to Gurobi as a warm start. In terms of redundant constraints, they are an overhead when the solver is only asked to find a solution. When optimisation is required, then imposing redundant constraints on the track segments bordering to the railway network beyond the station is the most beneficial in terms of proving optimality and runtime.

5 Conclusion

We present a constraint-based model for solving the in-station train dispatching problem and explore different options for redundant constraints to strengthen the model, which has not been explored in this context before. Written in the solver-independent modelling language MiniZinc, our model allows us to evaluate multiple leading CP solvers including Google's OR-Tools, CP-SAT, Chuffed, IBM ILOG CP Optimizer as well as the MIP solver Gurobi. In comparison to our approach previous works suffered one or more short-comings: (i) inflexible or train specific models, which are difficult to modify constraints or the solution objective, (ii) not a fine granularity for modelling the non-overlapping constraints for trains (for reducing the problem complexity), and (iii) limited experimental studies on only one solver.

We evaluate our model on synthetic instances generated from real-world data of a railway station in Italy, containing up to 50 trains over a planning horizon of up to almost three hours on the larger instances. Experimental results show that our CP-based approach (Chuffed) can produce first solutions in seconds (similar to SOTA), which is suitable for close-to-real-time decision-making. If more time is available then warm-starting the search (Gurobi) with the first solution (from Chuffed) improves solution cost, up to optimal in just a few minutes for most instances. To our best knowledge, we conduct the largest comparison of solvers for this problem, and optimally solved the largest number of trains yet reported in the literature. Our benchmark dataset is available at `https://github.com/ShortestPathLab/train_dispatching_benchmark` to further work in the area.

References

1 Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and computer modelling*, 17(7):57–73, 1993.

2 Gleb Belov, Peter J. Stuckey, Guido Tack, and Mark Wallace. Improved linearization of constraint programming models. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 49–65, Cham, 2016. Springer International Publishing. `doi:10.1007/978-3-319-44953-1_4`.

3 Alain Billionnet. Using integer programming to solve the train-platforming problem. *Transportation Science*, 37:213–222, 2003. doi:10.1287/TRSC.37.2.213.15250.

4 Joseph Bryan, Glen Elliot Weisbrod, and Carl Douglas Martland. *Rail freight solutions to roadway congestion: final report and guidebook*, volume 586. Transportation Research Board, 2007.

5 Alberto Caprara, Laura Galli, Leo Kroon, Gábor Maróti, and Paolo Toth. Robust train routing and online re-scheduling. In *Proceedings of ATMOS workshop*, pages 24–33, 2010. doi:10.4230/OASICS.ATMOS.2010.24.

6 Matteo Cardellini, Marco Maratea, Mauro Vallati, Gianluca Boleto, and Luca Oneto. In-station train dispatching: A PDDL+ planning approach. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 450–458, 2021. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/15991.

7 Dorotea De Luca Cardillo and Nicola Mione. k l-list λ colouring of graphs. *European Journal of Operational Research*, 106:160–164, 1998. doi:10.1016/S0377-2217(98)00299-9.

8 Partha Chakroborty and Durgesh Vikram. Optimum assignment of trains to platforms under partial schedule compliance. *Transportation Research Part B: Methodological*, 42:169–184, 2008.

9 Geoffrey Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

10 Geoffrey Chu, Peter J Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver, 2018. URL: https://github.com/chuffed/chuffed.

11 Thibaut Feydy, Adrian Goldwaser, Andreas Schutt, Peter J Stuckey, and Kenneth D Young. Priority search with MiniZinc. In *ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.

12 Ian Fox et al. The network modelling, timetabling and fuel saving computer programs on the market. Technical report, Australian Rail Track Corporation, 2017.

13 Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.*, 27:235–297, 2006. doi:10.1613/JAIR.2044.

14 Ulrich Geske. Railway scheduling with declarative constraint programming. In Masanobu Umeda, Armin Wolf, Oskar Bartenstein, Ulrich Geske, Dietmar Seipel, and Osamu Takata, editors, *Declarative Programming for Knowledge Management*, pages 117–134, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

15 Daniel Harabor and Peter J Stuckey. Rail capacity modelling with constraint programming. In *Integration of AI and OR Techniques in Constraint Programming: 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29-June 1, 2016, Proceedings 13*, pages 170–186. Springer, 2016. doi:10.1007/978-3-319-33954-2_13.

16 Stefan Kreter, Andreas Schutt, and Peter J Stuckey. Using constraint programming for solving RCPSP/max-cal. *Constraints*, 22:432–462, 2017. doi:10.1007/S10601-016-9266-6.

17 Rajnish Kumar, Goutam Sen, Samarjit Kar, and Manoj Kumar Tiwari. Station dispatching problem for a large terminal: A constraint programming approach. *Interfaces*, 48(6):510–528, 2018. doi:10.1287/inte.2018.0950.

18 Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23:210–250, 2018. doi:10.1007/S10601-018-9281-X.

19 Leonardo Lamorgese and Carlo Mannino. An exact decomposition approach for the real-time train dispatching problem. *Operations Research*, 63(1):48–64, 2015. doi:10.1287/OPRE.2014.1327.

20 Leonardo Lamorgese, Carlo Mannino, and Mauro Piacentini. Optimal train dispatching by benders'-like reformulation. *Transportation Science*, 50:910–925, 2016. doi:10.1287/TRSC.2015.0605.

21 Carlo Mannino and Alessandro Mascis. Optimal real-time traffic control in metro stations. *Operations Research*, 57:1026–1039, 2009. doi:10.1287/OPRE.1080.0642.

22 Mahmoud Masoud, Erhan Kozan, Geoff Kent, and Shi Qiang Liu. A new constraint programming approach for optimising a coal rail system. *Optimization Letters*, 11:725–738, 2017. doi:10.1007/s11590-016-1041-5.

23 Laurent Perron. Operations research and constraint programming at Google. In *International conference on principles and practice of constraint programming*, pages 2–2. Springer, 2011.

24 Joaquín Rodriguez. A constraint programming model for real-time train scheduling at junctions. *Transportation Research Part B: Methodological*, 41:231–245, 2007.

25 Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez. Interval-based relaxation for general numeric planning. In *Proceedings of ECAI*, pages 655–663, 2016. doi:10.3233/978-1-61499-672-9-655.

26 Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramírez. Subgoaling techniques for satisficing and optimal numeric planning. *J. Artif. Intell. Res.*, 68:691–752, 2020. doi:10.1613/JAIR.1.11875.

27 Andreas Schutt, Thibaut Feydy, Peter J Stuckey, and Mark G Wallace. Solving RCPSP/max by lazy clause generation. *Journal of scheduling*, 16:273–289, 2013. doi:10.1007/S10951-012-0285-X.

28 Peter J Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, 2014. doi:10.1609/AIMAG.V35I2.2539.

29 Ria Szeredi and Andreas Schutt. Modelling and solving multi-mode resource-constrained project scheduling. In *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*, pages 483–492. Springer, 2016. doi:10.1007/978-3-319-44953-1_31.

30 Ruifan Tang, Lorenzo De Donato, Nikola Bešinović, Francesco Flammini, Rob M.P. Goverde, Zhiyuan Lin, Ronghui Liu, Tianli Tang, Valeria Vittorini, and Ziyulong Wang. A literature review of artificial intelligence applications in railway systems. *Transportation Research Part C: Emerging Technologies*, 140:103679, 2022.

A Full Results

The following three tables show the full list of our experimental results. Table 3 lists all results for the objective **satis**, i.e., finding a solution. Table 4 lists the results for the objective **makespan**, whereas Table 5 the results for **endtimes**.

Table 3 Comparison of all solving methods when instructed to find the *first solution*. For each method, the number of instances for which a solution is found (sat), the average runtime of all instances in seconds (time), the average makespan, and the average sum of end times.

solver	search	column	sat	time	makespan	end sum
Planner		none	137	28	2219	26844
Chuffed	fixed-order	none	150	5	2699	37449
		border	150	6	2699	37449
		platform	150	6	2699	37449
		b + p	150	6	2699	37449
		all	150	7	2699	37449
		virt. best	150	5	2699	37449
	priority	none	139	5	2291	24957
		border	141	9	2405	28829
		platform	139	6	2291	24957
		b + p	140	7	2349	26797
		all	139	6	2294	25023
		virt. best	142	8	2416	28987
	priority+	none	150	7	2700	37432
		border	150	7	2700	37435
		platform	150	8	2700	37434
		b + p	150	7	2700	37433
		all	150	7	2700	37433
		virt. best	150	7	2700	37430
	free	none	143	8	2836	40346
		border	137	11	2410	27614
		platform	142	8	2763	37174
		b + p	136	13	2307	24669
		all	136	12	2315	24958
		virt. best	143	15	2612	33174
	virt. best	none	150	8	2699	37418
		border	150	7	2699	37421
		platform	150	9	2699	37419
		b + p	150	7	2699	37420
		all	150	7	2699	37420
		virt. best	150	7	2699	37418
CP Opt.	free	none	150	5	3370	54742
		border	140	13	2743	32029
		platform	42	6	637	1937
		b + p	42	6	640	1980
		all	40	30	648	2000

CP-SAT	fixed-order	none	145	7	2770	38680
		border	145	8	2770	38680
		platform	145	8	2770	38680
		b + p	145	9	2770	38680
		all	145	10	2770	38680
		virt. best	145	7	2770	38680
	free	none	145	7	2847	44652
		border	145	8	2898	46322
		platform	145	8	2899	46328
		b + p	145	9	2898	46322
		all	145	12	2990	42546
		virt. best	145	10	2792	39730
	virt. best	none	145	7	2770	38679
		border	145	8	2769	38679
		platform	145	8	2769	38679
		b + p	145	9	2769	38679
		all	145	10	2770	38678
		virt. best	145	8	2770	38676
Gurobi	free	none	143	13	3064	35304
		border	142	12	2970	33647
		platform	135	7	2702	25345
		b + p	137	11	2792	26519
		all	132	21	2566	23028

Table 4 Comparison of all solving methods when optimsing the *makespan*. For each method, the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), the average runtime of all instances in seconds (time), and the average makespan for the solved instances.

solver	search	column	opt	sat	time	makespan
Planner		none	0	137	28	2219
Chuffed	standard	none	110	116	84	1884
		border	111	116	82	1884
		platform	109	115	86	1874
		b + p	110	115	85	1874
		all	110	115	86	1874
		virt. best	111	116	81	1884
	standard+	none	110	117	84	1903
		border	111	116	82	1884
		platform	109	115	86	1874
		b + p	110	115	85	1874
		all	110	115	86	1874
		virt. best	111	117	81	1903
	fixed-order	none	136	150	37	2696
		border	140	150	30	2697
		platform	135	150	39	2696

		b + p	138	150	31	2697
		all	138	150	32	2697
		virt. best	142	150	27	2696
	fixed-order+	none	135	150	38	2696
		border	139	150	31	2697
		platform	134	150	39	2696
		b + p	138	150	31	2697
		all	138	150	32	2697
		virt. best	141	150	28	2696
	priority	none	126	139	55	2288
		border	132	140	46	2345
		platform	126	139	55	2288
		b + p	131	140	46	2345
		all	130	139	47	2290
		virt. best	132	141	45	2358
	priority+	none	133	150	44	2697
		border	138	150	34	2697
		platform	133	150	46	2697
		b + p	137	150	35	2697
		all	137	150	36	2697
		virt. best	138	150	34	2697
	free	none	137	142	44	2384
		border	132	137	51	2250
		platform	135	140	47	2325
		b + p	132	137	54	2247
		all	126	136	62	2256
		virt. best	139	142	39	2381
	virt. best	none	145	150	19	2696
		border	147	150	15	2696
		platform	145	150	20	2696
		b + p	147	150	17	2696
		all	146	150	18	2696
		virt. best	148	150	12	2696
CP Opt.	free	none	139	150	32	2704
		border	131	142	47	2342
		platform	42	42	218	572
		b + p	42	42	218	572
		all	40	40	228	591
CP-SAT	standard	none	132	141	44	2567
		border	129	133	49	2279
		platform	129	134	53	2307
		b + p	128	132	52	2264
		all	128	132	55	2264
		virt. best	132	141	44	2567
	fixed-order	none	134	145	39	2767
		border	138	145	33	2767
		platform	134	145	40	2767
		b + p	138	145	34	2767

		all	137	145	35	2767
		virt. best	138	145	32	2767
	free	none	142	145	26	2767
		border	142	145	25	2782
		platform	141	145	31	2780
		b + p	142	145	26	2784
		all	142	145	28	2781
		virt. best	143	145	22	2767
	virt. best	none	142	145	25	2767
		border	143	145	22	2767
		platform	142	145	28	2767
		b + p	143	145	23	2767
		all	143	145	24	2767
		virt. best	143	145	21	2767
Gurobi	free	none	136	140	36	2259
		border	138	140	32	2253
		platform	132	136	46	2132
		b + p	135	137	44	2163
		all	129	131	65	2012
	warm start	none	144	150	23	2696
		border	146	150	18	2696

Table 5 Comparison of all solving methods when optimising the *sum of end times*. For each method, the number of proven optimal instances (opt), the number of instances for which a solution is found (sat), the average runtime of all instances in seconds (time), and the average sum of end times for the solved instances.

solver	search	column	opt	sat	time	end sum
Planner		none	0	137	28	26844
Chuffed	standard	none	97	117	110	16906
		border	101	117	106	16906
		platform	98	115	109	16419
		b + p	101	115	106	16419
		all	99	115	107	16419
		virt. best	103	117	103	16906
	standard+	none	97	117	110	16906
		border	101	117	107	16906
		platform	98	116	109	16726
		b + p	101	115	106	16419
		all	99	114	107	16404
		virt. best	103	117	104	16906
	fixed-order	none	103	150	101	37421
		border	109	150	89	37417
		platform	104	150	100	37421
		b + p	109	150	89	37417
		all	108	150	92	37417

		virt. best	113	150	82	37416
	fixed-order+	none	103	150	101	37421
		border	108	150	90	37417
		platform	104	150	101	37421
		b + p	109	150	90	37417
		all	107	150	92	37417
		virt. best	113	150	83	37416
	priority	none	102	139	101	24936
		border	108	140	92	26759
		platform	103	139	99	24936
		b + p	108	140	90	26759
		all	107	139	91	24985
		virt. best	110	141	86	26934
	priority+	none	102	150	101	37402
		border	108	150	90	37393
		platform	103	150	100	37402
		b + p	110	150	87	37394
		all	109	150	89	37396
		virt. best	110	150	86	37391
	free	none	122	141	72	26609
		border	128	137	61	24411
		platform	125	141	70	26403
		b + p	126	134	66	21366
		all	121	132	71	20913
		virt. best	130	141	56	25999
	virt. best	none	125	150	65	37386
		border	128	150	58	37381
		platform	127	150	64	37386
		b + p	127	150	58	37382
		all	124	150	62	37383
		virt. best	130	150	53	37380
CP Opt.	free	none	106	150	107	39396
		border	107	139	108	24375
		platform	41	42	220	1582
		b + p	42	42	220	1582
		all	39	40	228	1707
CP-SAT	standard	none	112	141	82	31811
		border	116	133	74	23440
		platform	111	133	86	23439
		b + p	114	132	78	23192
		all	114	132	82	23192
		virt. best	116	141	73	31811
	fixed-order	none	111	145	87	38652
		border	117	145	72	38645
		platform	109	145	91	38652
		b + p	116	145	76	38645
		all	115	145	79	38646
		virt. best	117	145	72	38645

	free	none	126	145	56	38584
		border	126	145	55	38586
		platform	125	145	59	38794
		b + p	126	145	57	38678
		all	126	145	59	38590
		virt. best	127	145	53	38579
	virt. best	none	126	145	56	38584
		border	126	145	55	38586
		platform	125	145	59	38591
		b + p	126	145	57	38590
		all	126	145	59	38590
		virt. best	127	145	53	38579
Gurobi	free	none	133	140	45	23632
		border	136	138	38	21880
		platform	129	137	54	21109
		b + p	132	135	47	20053
		all	130	132	60	18786
	warm start	none	134	150	43	37379
		border	137	150	36	37377