An Efficient Hybrid Planning Framework for In-Station Train Dispatching

An Efficient Hybrid Planning Framework for In-Station Train Dispatching

Matteo Cardellini[1][0000−0003−3788−9475], Marco Maratea[1][0000−0002−9034−2527], Mauro Vallati[2][0000−0002−8429−3570], Gianluca Boleto[1], and Luca Oneto[1][0000−0002−8445−395X]

[1] University of Genoa, Italy, {name.surname}@unige.it
[2] University of Huddersfield, UK, m.vallati@hud.ac.uk

Abstract. In-station train dispatching is the problem of optimising the effective utilisation of available railway infrastructures for mitigating incidents and delays. This is a fundamental problem for the whole railway network efficiency, and in turn for the transportation of goods and passengers, given that stations are among the most critical points in networks since a high number of interconnections of trains' routes holds therein. Despite such importance, nowadays in-station train dispatching is mainly managed manually by human operators.
In this paper we present a framework for solving in-station train dispatching problems, to support human operators in dealing with such task. We employ automated planning languages and tools for solving the task: PDDL+ for the specification of the problem, and the ENHSP planning engine, enhanced by domain-specific techniques, for solving the problem. We carry out a in-depth analysis using real data of a station of the North West of Italy, that shows the effectiveness of our approach and the contribution that domain-specific techniques may have in efficiently solving the various instances of the problem. Finally, we also present a visualisation tool for graphically inspecting the generated plans.

Keywords: Railway Station · In-Station Train Dispatching · Automated Planning · PDDL+.

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 [2]. In-station train dispatching is the problem of optimising the effective utilisation of available railway infrastructures for mitigating incidents and delays, and their impact. This is a fundamental problem for the whole railway network efficiency, and in turn for the transportation of goods and passengers, given that stations are among the most critical points in networks since a high number of interconnection of trains' routes holds therein. Despite such importance, nowadays in-station train dispatching is mainly managed manually by human operators, while automatic support would

be of great help for the wide railway network efficiency. The nature of real-world applications often necessitates the representation of the dynamics of the application in terms of mixed discrete/continuous behaviour, including effects, processes, exogenous events, and continuous activities.

In this paper we present a framework (shown in Figure 2 of Section 3) for solving in-station train dispatching problems, so to support human operators in dealing with such task. We employ automated planning languages and tools: in automated planning, knowledge is made explicit in a symbolic problem instance models. When using the dominant family of planning knowledge representation languages – PDDL [13], a problem is specified via an initial state and goal state, that need to be reached. States, represented via fluents, can be changed by applying actions, that are specified via preconditions and effects, represented as a Boolean combination of predicates. A *solution (plan)* is a sequence of actions such that a consecutive application of the actions in the plan (starting in the initial state) results in a state that satisfies the goal [7]. Such classical planning model has been recently extended, in order to handle mixed discrete/continuous behaviours, resulting in PDDL+ [5], which thus allows to model hybrid domains. In addition to actions, PDDL+ introduces *continuous processes* and *exogenous events*, that are triggered by changes in the environment. Processes are used to model continuous changes. PDDL+ applications in real-world domains include real-time UAV manoeuvring [14], efficient battery load management [6], and urban traffic control [12].

Our framework, that extends a research prototype [3], is organised in three main elements: a preprocessor, that takes as input a description of the in-station train dispatching problem, and outputs a PDDL+ instance model; a planning engine, able to solve the PDDL+ problem and computing a plan, and a visualisation tool. We present all such elements and focus on the planning engine, its input/output, and its performance when employing the ENHSP planning engine [15, 16], enhanced by domain-specific techniques. We carry out a detailed analysis using real data of a station of the North West of Italy, provided by Rete Ferroviaria Italiana (RFI), that shows the effectiveness of our approach and the contribution that the implemented domain-specific techniques may have in efficiently solving the various instances of the problem.

The paper is structured as follows. Section 2 introduces our target problem. Then, Section 3 presents our framework and its elements. The experimental analysis of our modified planning engine on the real data is shown in Section 4. The paper ends in Section 5 by discussing related work and drawing conclusions.

2 The In-Station Train Dispatching Problem

This section describes the in-station train dispatching problem, following the formalisation introduced in [9].

A railway *station* can be represented as a tuple $\mathscr{S} = \langle G, I, P, E^+, E^- \rangle$. G is an undirected graph, $G = \langle S, C \rangle$, where S is the set of nodes representing the track segments, i.e., the minimal controllable rail units, and C is a set of

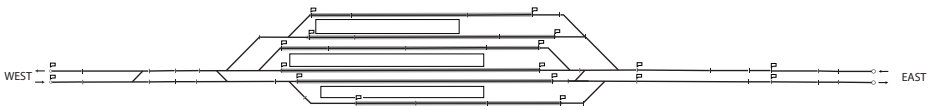

Fig. 1: A subsection of the real-world railway station considered in the experimental analysis. Flags are used to delimit itineraries; shorter segments represent track segments; bold indicates platforms.

edges that defines the connections between them. Their status can be checked via track circuits, that provide information about occupation of the segment and about corresponding timings. Track segments are grouped in itineraries I. Each itinerary $i = \langle s_{m1}, s_{m2}, ..., s_{mn} \rangle \in I$ is a path graph, subset of the graph G, representing a sequence of connected track segments. Track segments are grouped in itineraries manually by experts of the specific railway station, and a track segment can be included in more than one itinerary. While track segments are the minimal controllable units of a station, itineraries describe paths that the trains will follow in order to move inside the station.

P is a set of platforms that can be used to embark/disembark the train. Each platform $p \in P$ has an associated set of track segments $\{s_i, \ldots, s_j\}$ indicating that trains stopping at those track segments will have access to the platform.

E^+ is a set of entry points to the station from outside railway network; similarly, E^- is a set of exit points of the station that allow the train to leave the station and enter the outside railway network. For the sake of this formalisation, entry points and exit points behave like buffers of infinite size which are connected to a single track segment that is the first (last) track segment the train will occupy after (before) entering (exiting) the station.

Figure 1 provides a schematic representation of part of the Italian railway station we use in our analysis. In the figure, track segments and platforms are easily recognisable, and flags are used to indicate initial and end points of itineraries. 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 t going through the controlled railway station is running a route R_t in the station graph G, by reserving an itinerary and moving through the corresponding track segments. A route $R_t \subseteq I$ is a sequence of *connected disjointed itineraries* which the train t will travel in the station. To simplify the notation we define with $h(R_t) \in S$ the first track segment of the route and with $l(R_t) \in S$ the last.

Considering a single railway station, there are four possible types of train:

– *Origin*: the train originates at the controlled station. It is initially at a platform p, and it is required to leave the controlled station via a specified exit point.

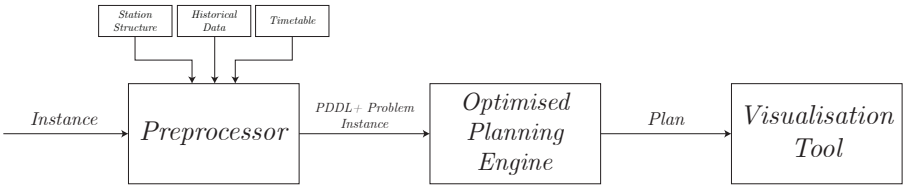

Fig. 2: Architecture of the proposed framework.

- *Destination*: the controlled station is the final destination of the train. The train is expected to reach the destination at time t_a via a given entry point, and is required to reach a platform to end its trip.
- *Transit*: the train is moving through the controlled station without making a stop. The train is expected to reach the destination at time t_a via a given entry point, and is required to reach a specified exit point without stopping at a platform.
- *Stop*: the train is making an intermediate stop at the controlled station. The train is expected to reach the destination at time t_a via a given entry point, and is required to reach a specified exit point after a stop at a platform for embarking/disembarking passengers.

The sequence of itineraries $R_t = (i_1, i_2, ..., i_n)$, with $i_k \in I$, which the train t will run across have to be connected in the graph G.

A *timetable* is the schedule that includes information about when trains should arrive at the controlled station, when they arrive at a platform, and the time when they leave a platform.

We are now in the position to define the in-station train dispatching problem as follows: Given a railway station $\mathscr{S}$, a set of trains T and their current position within the station or their time of arrival at the controlled station, find a route for every train that allows to respect the provided timetable, as much as possible.

3 Description of the Framework

The architecture of the proposed framework is shown in Figure 2. An instance can be defined using a provided interface, allowing a human operator to specify the trains that need to be controlled, their type (i.e., *Origin*, *Destination*, *Transit* or *Stop*) and the expected arrival/departure times, together with all the features needed to cluster the historical tracks' travel times (as explained in Section 3.1). The prepocessor generates a PDDL+ problem instance, that can then be parsed and reasoned upon by a dedicated optimised planning engine, by relying also on information about the structure of the controlled station, timetables, and historical data. The preprocessor uses the morphology of the station to compute all the possible routes that a train can move through in order to reduce the combinations that a plan may follow.

The optimised planning engine is based on the well-known ENHSP [15, 16] system, enhanced with domain-specific optimisations to improve performance on in-station train dispatching instances. Finally, the generated plans can be graphically shown by a visualisation tool, that aims at supporting a human operator in checking and simulating the planned movements of trains inside the station.

3.1 Preprocessor

The preprocessor is in charge of translating a given problem instance into a PDDL+ problem instance, so that it can be solved by a PDDL+ planning engine. We can define a PDDL+ problem model as a t-uple $\langle I, G, A, O \rangle$. I is a description of the initial state, that characterises the initial state of the world. G is the list of goals that must be achieved. A is the set of actions, events, and processes that are used to model the evolution of the world. Actions are under the control of the planning engine, while events and processes are exogenous and are triggered automatically when the corresponding preconditions are met. Processes are used to model continuous changes, while events are instantaneous changes of the state of the world. Finally, O is the list of objects involved in the problem.

To perform its task, the preprocessor relies on additional data provided by Rete Ferroviaria Italiana (RFI). Here we introduce the provided data, and describe the structure of the PDDL+ model that is generated.

Data provided. RFI provided the access to the data of 5 months (January to May 2020) of train movements of one medium-sized railway station of the Liguria region, situated in the North West of Italy. A subsection of the station is shown in Figure 1: the station has been anonymised, and the complete structure can not be provided due to confidentiality issues. The modelled station includes 130 track segments (out of which 34 are track switches), 107 itineraries, 10 platforms, 3 entry points, and 3 exit points. The average number of trains per day was 130 before COVID-19-related lockdown and 50 after movement restrictions were enforced in Italy. The travel times information needed by the PDDL+ model, e.g., the time needed by each train to complete a track segment, to leave a platform, maximum times, etc., were calculated by leveraging on historical data provided by RFI. The whole dataset of historical tracks' travel times were clustered by a variety of features, such as train characteristics (e.g., passengers, freight, high speed, intercity, etc.), station transit characteristics (i.e., overall train trip inside the rail-network, and entry and exit points), weekdays, and weather conditions. According to the characteristics of the problem at hand, it is then possible to estimate the travel times of every train by assigning the average value of the times inside the corresponding cluster.

Times. The travel times information are encoded in the PDDL+ model using the following fluents:

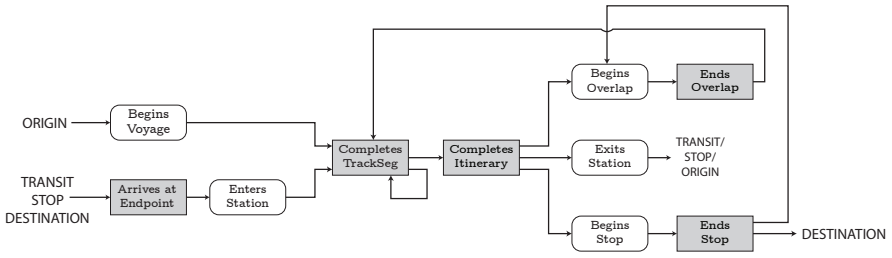

Fig. 3: A flow chart showing the movements of the train inside the station based on its type. Rectangles represent events and squared rectangles represent actions. Processes are omitted for clarity.

- `arrivalTime(t)`: the time a_t at which the train t arrives at the controlled station.
- `segmentLiberationTime(t,s,i)`: the amount of time it takes for train t to free segment $s \in i_n$ since it starts moving on itinerary i.
- `timeToRunItinerary(t,i)`: the amount of time it takes for train t to move through itinerary i.
- `stopTime(t,p)`: the expected amount of time it takes for train t to embark/disembark passengers or goods at platform p.
- `timeToOverlap(t,`$\mathtt{i_n}$`,`$\mathtt{i_m}$`)`: the time it takes for train t to completely leave itinerary i_n and move to itinerary i_m.
- `timetableArrivalTime(t)` (`timetableDepartureTime(t)`): the time in which the train t should arrive at (departure from) a platform, according to the official timetable.

Movement of trains. In this section we focus on the PDDL+ structures used to model the movement of trains per type. An overview of the events and actions used to model trains' movement is provided in Figure 3. For the sake of readability, we say that a Boolean predicate is activated (de-activated) when its value is made True (False).

Transit train. If a train is of type Transit it will traverse the station without stopping to embark/disembark passengers or goods at any of the platforms. The movement of the train through the station is regulated using the following PDDL+ constructs (listed in application's order):

- A process `incrementTime` encodes an explicit notion of passing time by increasing the fluent `time`.
- An event `arrivesAtEntryPoint(t,`$\mathtt{e^+}$`)` is triggered when the fluent `time` reaches the value of the predicate `arrivalTime(t)`, and it requires that the predicate `trainEntersFromEntryPoint(t,`$\mathtt{e^+}$`)`, indicating the entry point

for the train, is active. The event makes true the predicate `trainHasArrivedAtStation(t)` signalling that the train is at the gateway of the station ready to enter.
- An action `entersStation(t,e+,i)` can be used by the planning engine to let the train t enter the station from the entry-point e^+, using the itinerary i. The precondition that must hold in order for this action to be taken are: (i) the entry point is connected to the itinerary i, and (ii) the itinerary i is free, i.e., none of its track segments are occupied by a train. As a result of this action i is reserved by t by blocking all the track segments in it via dedicated predicates `trackSegBlocked(s)`, and by the predicate `trainInItinerary(t,i)`.
- A process `incrementTimeReservedItinerary(i)` keeps track of how many seconds have passed since the reservation of an itinerary i by any train, updating the fluent `tReservedItinerary(i)` accordingly.
- The event `completesTrackSeg(t,s,i)` is triggered when a train t has left the track segment s. This is triggered when `tReservedItinerary(i)` has reached the value specified in the fluent `segmentLiberationTime(t,s,i)`, and as a result the segment s is freed by de-activating the predicate `trackSegBlocked(s)`.
- As soon as the train has reached, with its head, the end of the itinerary (so when `tReservedItinerary(i)` is greater than `timeToRunItinerary(t,i)`), the event `completesItinerary(t,i)` is triggered. It activates a predicate `trainHasCompletedItinerary(t,i)`.
- An action `beginsOverlap(t,in,im)` can be used by the planning engine to encode the movement of train t from itinerary i_n to itinerary i_m. This action can be used only if `trainHasCompletedItinerary(t,in)` is active and all the track segments $s \in i_n$ are not occupied by another train.
- A process `incrementOverlapTime(t,in,im)` keeps track of the time a train t is taking to move all its carriages through the joint that connects i_n to i_m by increasing the value of the fluent `timeElapsedOverlapping(t,in,im)`.
- An event `endsOverlap(t,in,im)` is triggered when the fluent that counts the time of overlap `timeElapsedOverlapping(t,in,im)` is greater than the fluent `timeToOverlap(t,in,im)`. The event signals that the train is no longer on itinerary i_n deactivating the predicates `trainInItinerary(t,in)` and resetting the value of function `tReservedItinerary(in)` to 0.
- Finally, when a train has completed its routes of itineraries, and with the last itinerary i has reached the segment leading to an exit point of the station e^-, the action `exitsStation(t, i, e-)` allows the train to exit the station activating the predicate `trainHasExitedStation(t)`.

Part of the described constructs are shown in Figure 4. The goal for a train t of type *Transit* is to reach the state in which the predicate `trainHasExitedStation(t)` is active.

Stop train. A train of this type needs to stop at a platform before exiting the station, as it is making an intermediate stop at the controlled station. Three

```
(:action entersStation
:parameters
 (T1 - train EP1 - entryP
 I1-2 itinerary)
:precondition (and
 (trainIsAtEndpoint T1 EP1)
 ...
 (not (trainHasExitedStation T1))
 (not (trainHasEnteredStation T1))
 (not (trackSegBlocked cdb1))
 (not (trackSegBlocked cdb2))
 ...
 (not (trackSegBlocked cdb6)))
:effect (and
 (not (trainIsAtEndpoint T1 EP1))
 (itineraryIsReserved I1-2)
 (trainInItinerary T1 I1-2)
 (trainHasEnteredStation T1)
 (trackSegBlocked cdb1)
 (trackSegBlocked cdb2)
 ...
 (trackSegBlocked cdb6)))
```

```
(:event arrivesAtEntryPoint
:parameters (T1 - train EP1 - entryP)
:precondition (and
 (>= time (ArrivalTime T1))
 (not (trainHasEnteredStation T1))
 (trainEntersFromEntryPoint T1 EP1))
:effect (and (trainIsAtEndpoint T1 EP1)
 (trainHasArrivedAtStation T1)
 (assign (trainEntryIndex T1)
 (trainsArrivedAtEndpoint EP1))
 (increase (trainsArrivedAtEndpoint EP1) 1 )))

(:event completeTrackSeg
:parameters
 (T1 - train cb1 - trackSeg I1-2 itinerary)
:precondition (and
 (>= (timeReservedIt I1-2) 29)
 (trainInItinerary T1 I1-2)
 (trackSegBlocked cdb1))
:effect (and
 (not (trackSegBlocked cdb1))))

 (:process incrementTimeReservedItinerary
:parameters(I1-2 - itinerary)
:precondition (itineraryIsReserved I1-2)
:effect (increase
 (timeElapsedReservedItinerary I1-2) #t ))
```

Fig. 4: The partially grounded action `entersStation(t,e+,in)` (left), events `arrivesAtEntryPoint(t,e+)` and `completesTrackSeg(t,s,in)`, and process `incrementTimeReservedItinerary(in)` (right).

other PDDL+ constructs are added with respect to a train of type `Transit` in order to model this behaviour:

- The action `beginStop(t,i,p)` is used to allow the planning engine to stop the train t at a platform p after having completed itinerary i, and having the platform at the end of the itinerary. The effect of this action is to signal that the train is at the platform by activating the predicate `trainIsStoppingAtStop(t,p)` and all the track segments $s \in p$ are blocked in order to achieve mutual exclusion on the platform.
- A process `increaseTrainStopTime(t)` keeps track of the time spent by the train at a platform. This is done by increasing the value of the function `trainStopTime(t)` over time.
- The train t can begin its route towards the exit point only after a time `stopTime(t)` has passed – this it to allow passengers or goods to embark / disembark. A train cannot leave the platform before the timetabled departure, set via the function `timetableDepartureTime(t)`. For this reason the event `endStop(t,i,p)` that signals that the train is ready to leave the platform can be triggered only if the fluent `trainStopTime(t)` has reached the value set in the predicate `stopTime(t)` and the fluent `time` is greater than `timetableDepartureTime(t)`. This event activates the predicate `trainHasStopped(t)`.

```
(:action beginStop
:parameters
(T1 - train I3 - itinerary
 S1 - platform)
:precondition (and
 (trainHasCompletedItinerary T1 I3)
 (trainInItinerary T1 I3)
 (not (trainIsOverlapping T1))
 (not (stopIsOccupied S1))
 (not (trainIsStopping T1))
 (not (trainHasStoppedAtStop T1 S1))
...)
:effect (and
 (trainIsStoppingAtStop T1 S1)
 (trainIsStopping T1)
 (assign (trainStopTime T1) 0 )
 (stopIsOccupied S1)
 (not (itineraryIsReserved I3))
 (not (trackSegBlockedtrackSegC))
...
 (not (trackSegBlockedtrackSegL))))
```

```
(:process increaseTrainStopTime
:parameters(T1 - train)
:precondition (trainIsStopping T1)
:effect
 (increase (trainStopTime T1) #t ))

(:event endStop
:parameters
 (T1 - train I3 - itinerary S1 - platform)
:precondition (and
 (>= (trainStopTime T1) 5 )
 (>= time (DepartTime T1))
 (trainInItinerary T1 I3)
 (trainIsStoppingAtStop T1 S1)
 (stopIsOccupied S1))
:effect (and
(not (trainIsStoppingAtStop T1 S1))
(not (trainIsStopping T1))
(trainHasStoppedAtStop T1 S1)
(trainHasStopped T1)
(not (stopIsOccupied S1))))
```

Fig. 5: The partially grounded action `beginStop(t,i_n,p_k)` (left), process `increaseTrainStopTime(t)` and event `endStop(t,i_n,p_k)` (right).

The train t will then begin its trip towards the exit point with the action `beginsOverlap(t,i_n,i_m)` where i_n is the itinerary where the platform is located, and i_m is a subsequent connected itinerary. Part of the described constructs are shown in Figure 5. For a train of type *Stop* the goal is to reach a state in which the predicates `trainHasExitedStation(t)` and `trainHasStopped(t)` are both active.

Origin train. A train of type *Origin* needs to specify the platform the train is parked at. For this reason the predicates `trainIsStoppingAtStop(t,p)` and `trainHasStopped(t)` are activated at the initial state of the problem, indicating the train t is departing from platform p. This type of train resemble the train of type *Stop* but without the possibility to enter the station, since it is already inside it at the beginning of the plan. Only one action is introduced in order to model the behaviour of an *Origin* train:

– An action `beginVoyage(t,p,i)` can be used by the planning engine to allow the departure of train t from platform p via itinerary i. This action can not be executed before the timetabled departure time `timetableDeparture-Time(t)`.

The goal of a train of type *Origin* are the same of a train of type *Transit*: it must have exited the station.

Destination train. A train of type *Destination* behaves like an *Stop* train but, after having reached the platform, it will remain parked there. For this reason no additional PDDL+ constructs are needed. A train of type *Destination* will simply have in its goal the need to reach a state in which the predicate `trainIs-Stopping(t)` is active, meaning so that the train t has reached its stop.

3.2 Optimised Planning Engine

The considered domain-independent PDDL+ planning engine ENHSP is based on a forward search approach on the discretised PDDL+ planning instance. In other words, the planning engine deals with time-related aspects by discretising it using a given delta value. We enhanced ENHSP in three main ways: adaptive delta, domain-specific heuristic, and constraints.

Adaptive Delta. Traditionally, PDDL+ planning engines exploit a fixed time discretisation step for solving a given instance. In the proposed model, it is possible to know a-priori when events will be triggered, so it is possible to exactly predict when the world will change. It is therefore possible to exploit a dynamic time discretisation. When nothing happens, there is no need to discretise time. This is similar in principle to the approach exploited by decision-epoch planning engines for dealing with temporal planning problems [4].

Specialised Heuristic. Following the traditional A* search settings, the cost of a search state z is calculated as $f(z) = g(z) + h(z)$, where $g(z)$ represents the cost to reach z, while $h(z)$ provides an heuristic estimation of the cost needed to reach a goal state from z. In our specialisation, $g(z)$ is calculated as the elapsed modelled time from the initial state to z. $h(z)$ is a domain-specific heuristic calculated according to the following equation:

$$h(z) = \sum_{t \in T(z)} \rho_t(z) + \pi_t(z) \tag{1}$$

where $T(z)$ is the set of trains of the given problem that did not yet achieve their goals at z. $\rho_t(z)$ is a quantity that measures the time that, starting from the current position, the considered train needs to reach its final destination and is computed as follows:

$$\rho_t(z) = \max_{R \in \mathscr{R}_t(z)} \sum_{i_n \in R} \texttt{timeToRunItinerary(t, i}_\texttt{n}\texttt{)} \tag{2}$$

where $\mathscr{R}_t(z)$ is the set containing all the possible routes R, as sequences of itineraries, that a train t can run across in order to reach its final destination (i.e., an exit point or a platform) from the state z. Since the initial and final position of every train is known a-priori and based upon its type (*Origin, Destination, Stop, Transit*) the set can be computed beforehand and used in the search phase.

The penalisation method $\pi_t(z)$ gives a very high penalisation value P for each goal specified for the considered train t that has not yet been satisfied at state z. For instance, if a train of type *Stop* has not yet entered the station, a penalisation of $2 \times P$ is given to the heuristic since there are two goals related to the train that has not been satisfied in the initial state, i.e., stopping at a platform and leaving the station from an exit-point.

Constraints. The planning engine has been extended by explicitly considering 3 set of constraints. The first set limits the time a train is allowed to stay in the controlled station. The other sets limit, respectively, the time passed from the arrival of the train in the station, and the time spent stopping at a platform.

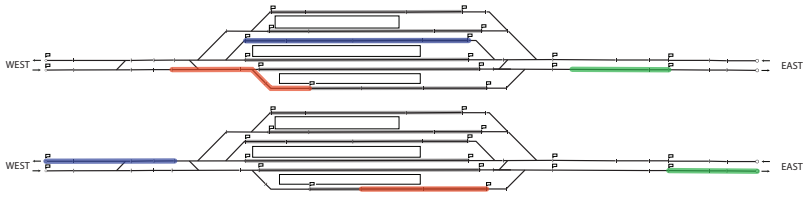

Fig. 6: Two screenshots from the visualisation tool capturing the itineraries occupied by three trains at different time steps. Top: The red train is moving from the West entry point to platform 1 (counting from the bottom), the blue train is stopped at platform 4 and the green train is approaching the East exit point. Bottom: The red train has stopped at the platform and is moving towards the East exit point, the blue and the green trains are leaving the station from the West and East exit points, respectively.

The idea behind such constraints is to avoiding situations where trains are left waiting for long periods of time, occupying valuable resources. The maximum times are calculated a-priori, according to historical data, and depends on the structure of the railway station.

3.3 Visualisation Tool

The last element of the framework shown in Figure 2 is the visualisation tool. This is a pivotal tool, that allows to visually inspect the generated plans and can be used by domain experts to compare alternative solutions, and by AI experts to validate the plans.

Figure 6 shows two screenshots from the visualisation tool capturing the itineraries occupied by three trains (red, green, blue) in two subsequent time instants. The tool can parse a solution plan, represented using the PDDL+ standard, and extracts the movements of trains and their position at each time step. The itineraries are then coloured based on the train that is currently moving on them. The tool provides an interface that allows the user to move forward or backward in time, to see how the network conditions evolve. In the example shown in Figure 6, the red train is moving from the West entry point to platform 1 (counting from the bottom), the blue train is stopped at platform 4 and will soon move to the West exit point and the green train is approaching the East exit point.

4 Experimental Analysis

The aim of this experimental analysis is to assess the importance of the domain-specific techniques implemented in the planning engine. As a first step, to understand that the proposed framework can accurately model the dynamics of the in-station train dispatching problem, we validated on historical data.

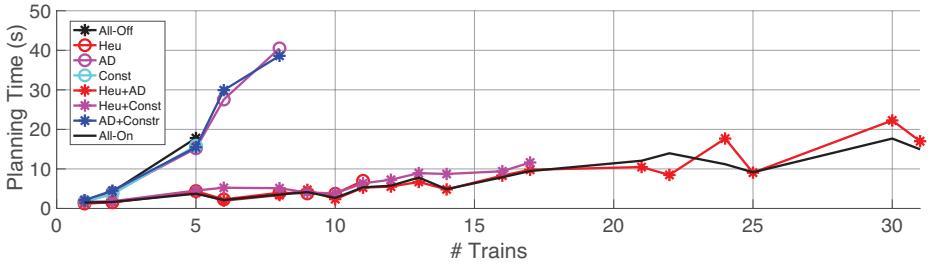

Fig. 7: The CPU-time needed for planning instances with an increasing number of trains using different combinations of domain-specific optimisation techniques.

To perform this analysis, we considered the 5 months of historical data, and encoded all the registered movements of trains under the form of PDDL+ plans – as if the movements were generated using our approach. We then validated the plans against our PDDL+ model with a PDDL+ validator. This step checks if the PDDL+ model presented in the paper is capable of representing the observed movements. Remarkably, the validation process showed that all the movements observed in the 5 months of historical data were correctly validated. This result also indicates that the proposed modelisation, and the overall framework, can provide an encompassing framework for comparing different strategies to deal with recurrent issues, and for testing new train dispatching solutions.

We can now turn our attention to assessing the usefulness of the improvements proposed in Section 3.2. To perform this analysis, we selected the day – in February 2020, before the start of the COVID-19 lockdown in Italy – with the minimum mean squared deviation of recorded train timings from the official timetable. This was done to guarantee that no emergency operations were executed by the human operators. We considered the evening peak hour of the day, 17:00-20:00, when commuters return at home. During the selected time-slot 31 trains move through the station. We considered different time windows of the considered 3-hours period to generate instances with a increasing number of trains to be controlled. These instances are then tested using the planning engine with different combinations of the domain-specific optimisations techniques. Figure 7 shows the CPU planning time required by the 8 combinations to deal with increasingly large instances. The combination labelled *All-Off* considers ENHSP run using the default settings, and none of the optimisation techniques introduced in this paper. The *All-On* label indicates instead the planning engine run using all the optimisations. Finally, labels *AD*, *Heu*, and *Const* are used to indicate the use of, respectively, adaptive delta, domain-specific heuristic, and constraints. The analysis was performed with a cut-off time of 60 CPU-time seconds, on a 2.5GHz Intel Core i7 Quad-processors with 16GB of memory made available and a Mac OS operating system.

According to the results in Figure 7, the *All-Off* combination can plan no more than 5 trains. Inspecting the instances with more that 5 trains it can be

seen that, while the first 5 trains are close to each other, the $6th$ train arrives at the station a couple of minutes after the others. This gap of time, in which nothing happens, causes the planning engine to wast a significant amount of time in trying to identify actions to apply to reach the goal, at every discretisation delta. The use of the adaptive delta allows the planning engine to skip between times where nothing happens, therefore it significantly reduces the amount of computations made by the planning engine, solving up to 8 trains. The use of the domain-specific heuristic alone can solve instances up to 11 trains included. Notably, there is a significant synergy between the heuristic and the adaptive delta. When these two optimisations are activated, the planning engine performance are significantly boosted, given that with these two techniques enabled we are able to solve all instances, as for the *All-On* label. On the other hand, the use of the constraints can provide some performance improvement, but limited. With regards to the shape of the generated solutions, we observed that all the approaches lead to similar solutions: there are no major differences from this perspective. We further experimented with other peak hours (07:30-10:00 and 12:00-14:30), and results are similar to the ones shown.

Summarising, the performed experiments indicate that the use of the specialised heuristic is the single most important component of the domain-specific planning engine. The adaptive delta plays an important role as well, but their synergic combination allows to solve all evaluated instances. The use of constraints provides some improvement, but not as significant as the other elements.

5 Related Work and Conclusions

Automated approaches have been employed for solving variants of the in-station train dispatching problem. Mixed-integer linear programming models have been introduced in [11] for controlling a metro station. The experimental analysis demonstrated the ability of the proposed technique to effectively control a metro station but also highlighted scalability issues when it comes to control larger and more complex railway stations. More recently, constraint programming models have been employed in [8] for performing in-station train dispatching in a large Indian terminal: this approach is able to deal with a large railway station, but only for very short time horizons, i.e., less than 10 minutes. In [1] a hybrid approach that extends Answer Set Programming (ASP) [10] is used to tackle real-world train scheduling problems, involving routing, scheduling, and optimisation.

In this paper, we presented a framework for solving in-station train dispatching, focused on modelling the problem with the PDDL+ language, able to specify mixed discrete/continuous behaviour of railway systems, and solving with the ENHSP planning engine enhanced with domain-specific techniques. Results on real data of a station of the North West of Italy, provided by Rete Ferroviaria Italia, show that our solution is efficient, scalable with regards to the number of trains and can handle large time horizons, for the station at hand. Future work will focus on modelling additional stations of the North of Italy, and to run field tests of the proposed framework.

Acknowledgements

This work has been partially funded by Hitachi Rail STS through the RAIDLab (Railway Artificial Intelligence and Data Analysis Laboratory), a joint laboratory between Hitachi Rail STS and University of Genoa. This work has been supported by RFI (Rete Ferroviaria Italiana) who provided the data for the analysis (we sincerely thank Renzo Canepa for his support). Mauro Vallati was supported by a UKRI Future Leaders Fellowship [grant number MR/T041196/1].

References

1. Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., Wanko, P.: Train scheduling with hybrid answer set programming. Theory and Practice of Logic Programming pp. 1–31 (2020)
2. Bryan, J., Weisbrod, G.E., Martland, C.D.: Rail freight solutions to roadway congestion: final report and guidebook. Transportation Research Board (2007)
3. Cardellini, M., Maratea, M., Vallati, M., Boleto, G., Oneto, L.: In-station train dispatching: A PDDL+ planning approach. In: Proc. of ICAPS (2021)
4. Cushing, W., Kambhampati, S., Weld, D.S.: When is temporal planning really temporal? In: Proc. of IJCAI. pp. 1852–1859 (2007)
5. Fox, M., Long, D.: Modelling mixed discrete-continuous domains for planning. J. Artif. Intell. Res. **27**, 235–297 (2006)
6. Fox, M., Long, D., Magazzeni, D.: Plan-based policies for efficient multiple battery load management. J. Artif. Intell. Res. **44**, 335–382 (2012)
7. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers (2004)
8. Kumar, R., Sen, G., Kar, S., Tiwari, M.K.: Station dispatching problem for a large terminal: A constraint programming approach. Interfaces **48**(6), 510–528 (2018)
9. Lamorgese, L., Mannino, C.: An exact decomposition approach for the real-time train dispatching problem. Operations Research **63**(1), 48–64 (2015)
10. Lifschitz, V.: Answer set planning. In: International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 373–374. Springer (1999)
11. Mannino, C., Mascis, A.: Optimal real-time traffic control in metro stations. Operations Research **57**(4), 1026–1039 (2009)
12. McCluskey, T.L., Vallati, M.: Embedding automated planning within urban traffic management operations. In: Proc. of ICAPS. pp. 391–399 (2017)
13. Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL - The Planning Domain Definition Language. Tech. rep., Yale Center for Computational Vision and Control (1998)
14. Ramírez, M., Papasimeon, M., Lipovetzky, N., Benke, L., Miller, T., Pearce, A.R., Scala, E., Zamani, M.: Integrated hybrid planning and programmed control for real time UAV maneuvering. In: Proc. of AAMAS. pp. 1318–1326 (2018)
15. Scala, E., Haslum, P., Thiébaux, S., Ramírez, M.: Interval-based relaxation for general numeric planning. In: Proc. of ECAI. pp. 655–663 (2016)
16. Scala, E., Haslum, P., Thiébaux, S., Ramírez, M.: Subgoaling techniques for satisficing and optimal numeric planning. J. Artif. Intell. Res. **68**, 691–752 (2020)