UNIVERSITÀ DEGLI STUDI DI GENOVA
SCUOLA POLITECNICA
DIBRIS
DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

Università di Genova

MASTER'S DEGREE IN
COMPUTER ENGINEERING - ARTIFICIAL INTELLIGENCE AND HUMAN-CENTERED COMPUTING

Accademic Year 2020/2021

Artificial Intelligence Techniques for Solving the In-Station Train Dispatching Problem

Candidate
Matteo Cardellini

Supervisors
Prof. Marco Maratea
Prof. Mauro Vallati

Abstract

In railways, the in-station dispatching problem consists in planning the movements of trains inside a railway station, where a potentially large number of trains have to stop according to an official timetable. Managing the movements of trains inside a station in a correct way can help reduce the accumulation of delays, which can then have a beneficial knockout effect on the rest of the network. Despite the delicacy of this operation, in-station train dispatching is still largely handled manually by human operators without a decision support system able to provide helpful guidance and aid.

In this thesis, the author makes a step towards supporting the operator with some automatic tool, by describing an approach for performing in-station dispatching by means of artificial intelligence techniques. Given its mixed discrete-continuous nature, the problem is modelled with an AI planning language. To improve the performances and being able to scale to large flows of trains, a state-of-the-art AI planning engine is enhanced by domain-specific solving heuristics. The presented modelling and improvements are then tested and benchmarked using real-data of a station of the North West of Italy.

«Cosa aggiungere potrebbe un narratore
a quanto già narrato dall’attore»

Contents

Chapter 1

Introduction

1.1 Context and Motivations

Railways represent one of the most important and high-volumes means of transportation in Europe for transporting either goods or passengers. Rail transport is the most sustainable, whether in terms of CO_2 emissions, energy consumption, use of space, or noise levels [Givoni et al., 2009]. However, the increasing volume of people and freight transported on railways is congesting the networks [Bryan et al., 2007]. Railway stations, where platforms and rails can be seen as scarce resources in which numerous trains need to stop according to a given timetable, are probably the most critical points for interconnecting trains' paths. If not dealt carefully, the concrete risk of accumulating delays arises, which may result in cost penalties and inconveniences for passengers. The only fast and economically viable way to increase capacity is then to improve the efficiency of daily operations in order to be able to control a larger number of running trains without requiring massive public investments in new physical assets.

The main problems that need to be addressed in the context of a railway station are: (i) the initial generation of timetables, and (ii) the mitigation of delays and the resolution of conflicts which could arise in the station, with the aim of minimising the overall negative impact. The latter, formally defined as in-station train dispatching, is still nowadays handled manually by experienced operators in charge of a large set of connected stations, despite the fact that in-station train dispatching plays an important role in maximising the effective utilisation of available railway infrastructures. These operators, termed dispatchers, are in charge of monitoring the occupations of resources by all the trains in the station to ensure safety protocols: they provide instructions to train conductors, and control the railroad switches in order to make the train follow a path inside the station that will allow the train to enter the station, exit it or stop at a platform (if needed). Despite the paramount importance and complexity of their job, dispatchers are currently receiving very limited support to their decision. The current railway control systems only provide a sensor-based overview of the traffic conditions of the station, but they do not provide any suggestions for the correct movements of trains inside it.

The two aforementioned problems have different needs concerning the allowed time to find a viable solution. The task concerning the generation of timetables is an optimization problem combining a multitude of decision variables, complex cost functions and constraints. Despite this difficulty, the generation of timetables is an operation performed a few times a year. In Italy, usually, the timetable changes depending on the season (usually one for summer and one for the rest of the year) and some special timetables are put in place during the Christmas period, scaling in order to compensate for the higher number of passengers travelling during the holidays. The infrequency of generation of the timetables allows less care on the timing needed to produce a solution. The in-station dispatching problem, on the other hand, requires a more careful attention regarding the times needed to find a viable plan. The unexpected arise of maintenance necessities, causing the temporary unavailability of resources, or a high accumulation of delays requires a complete novel reschedule, invalidating the previously generated plans. For this reason, the time of execution of algorithms finding solutions for the in-station train dispatching problems have to be careful considered and tuned in order to find the solution in the fastest way possible.

1.2 Goals and contributions of this thesis

In this thesis, the in-station train dispatching problem is formalized and addressed for the purpose of making a significant step towards supporting the operator with a tool able to solve the problem in an automated way by means of automated planning.

The problem is modelled as an AI planning problem using `PDDL+` [Fox and Long, 2006a], a planning domain description language for modelling mixed discrete-continuous planning domains. The use of this particular language allows to manage several real-world complications: (i) the opportunity to define a model valid for several stations while keeping it as general as possible, (ii) the need to manage the continuous flow of time, and (iii) the necessity to manage external events coming from the outside world (i.e. the arrival of trains from the outside network).

Domain-specific enhancements that allow to quickly solve large and complex instances are introduced to the domain independent planning engine ENHSP [Scala et al., 2016, Scala et al., 2020], a state-of-the-art planner for `PDDL+` encoding. The enhancements, consisting of an advanced heuristic, a time-dependent discretisation of the time and a series of constraints pruning the search-space, allows the planner to solve large instances in a small amount of time, allowing the model to be used to quickly solve conflicts caused by the unexpected arise of maintenance necessities, causing the temporary unavailability of resources, or a high accumulation of delays.

In order to validate the modelling of the problem at hand, the plans produced by human operators are compared with the plans produced by the automated planning engine. This validation is performed with the aim of answering the question "Would the autonomous planning agent, given the same initial conditions, have made the same decision taken by the human dispatcher ?". The validation process answered the question positively, giving the proposed approach dignity to correctly model the real-world dynamics. This implies that planning-based tools can be correctly exploited, and the planning engine can provide an encompassing framework for comparing different strategies to deal with recurrent issues, and for testing new train dispatching solutions.

In the proposed approach, we considered real-world historical data of a medium-sized railway station from North-West of Italy provided by Rete Ferroviaria Italiana (RFI), and tested our approach in numerous scenarios. Results show the potential of our approach on a wide range of scenarios. In particular, the proposed approach demonstrates the ability to reduce delays and to better exploit the available infrastructure – with the potential of allowing a station to serve a larger number of trains without the need for structural modifications.

1.3 State-of-the-art

Automated approaches have been employed for solving variants of the in-station train dispatching problem. Mixed-integer linear programming models have been introduced in [Mannino and Mascis, 2009] 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 [Kumar et al., 2018] 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 [Abels et al., 2020] a hybrid approach that extends Answer Set Programming (ASP) [Lifschitz, 1999] is used to tackle real-world train scheduling problems, involving routing, scheduling, and optimisation.

Given the complexity of the train dispatching problem, many works focused on related subproblems or on a more abstract formulation of the overall problem. For example, [Rodriguez, 2007] formulated a constraint programming model for performing train scheduling at a junction, which shares some characteristics of a station, but does not include platforms and stops. Differently from [Rodriguez, 2007], a number of works [Cardillo and Mione, 1998, Billionnet, 2003, Chakroborty and Vikram, 2008] focused on the problem of assigning trains to available platforms, given the timetable and a set of operational constraints.

Taking another perspective, [Caprara et al., 2010] focused on the identification and evaluation of recovery strategies in case of delays.These strategies include actions such as the use of different platforms or alternative paths.

In [Li et al., 2021] a solution is presented to efficiently manage dense traffic on rail networks based on Multi-Agent Path Finding (MAPF) which can plan collision-free paths for thousands of trains, but doesn't take into account safety mechanism of mutual exclusion of resources or the need of a train to stop at platforms.

In Chapter 8 a more in depth analysis on the state-of the art will be performed covering also orthogonal problems like line dispatching and timetabling.

1.4 Structure of this thesis and Architecture

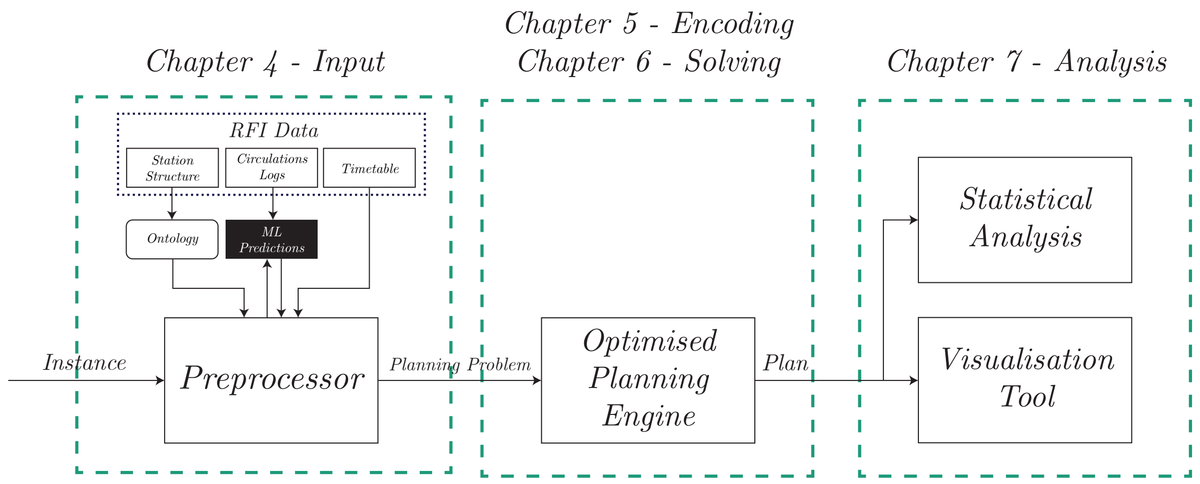


Figure 1.1: The architecture presented in this thesis aimed at solving the in-station train dispatching problem.

In Chapter 2 a formal representation of the structure of a railway station and a description of the mechanics by which a train moves inside a station are presented; afterwards the in-station train dispatching problem, with all its challenges and complications, is formally introduced. In Chapter 3 one of the most important languages for describing planning domains, the Planning Domain Definition Language (in short `PDDL`) is presented following its development history from the first version of `PDDL`, for classical planning, to `PDDL+` for modelling mixed discrete-continuous domains; the reasons behind the choice of `PDDL` as a modelling language together with a quick look at the algorithms behind the planning engine ENHSP are discussed at the end of the chapter.

In Figure 1.1 a block-diagram depicting the architecture presented in this thesis is shown. The first block represents the input of the problem (which will be thoroughly described in Chapter 4): in this phase the instance of the problem, provided by the human dispatcher, is reinforced by a preprocessor which augments it with additional information extrapolated from data provided by Rete Ferroviaria Italiana (RFI), the owner of Italy's railway network; this data mainly consists of (i) a JSON-like file containing a structured representation of the station's topology (ii) a Flat-File Database containing timestamped logs coming from the Rail Traffic Management System (TMS) which controls and monitors the movements of trains inside the station, and (iii) the timetable of the station which contains information about the expected times in which trains should arrive and depart from the station. The file representing the topology of the station is parsed and re-written under the form of an ontology, which enables to reason upon the status of each component in a simple way through a set of well-defined logic rules. In order to realistically plan the movements of trains inside a railway station, the time it takes for a train to move between points inside the station is of paramount importance; these times are predicted with Machine Learning's model which are trained on the data provided by RFI (the block is represented with a black box since the main work was produced in [Boleto et al., 2021] and the results were straightforwardly used in this thesis). The preprocessor uses all this accessory data to produce a planning problem written in the Planning Domain Definition Language (`PDDL`) which is described, in Chapter 5. This planning problem is then taken as input from a state-of-the-art planning engine, ENHSP [Scala et al., 2016, Scala et al., 2020] which is presented in Chapter 6. ENHSP is a modular planning engine, and includes a range of off-the-shelf search and heuristic techniques which proved to be capable of solving prototypical instances, hence demonstrating the feasibility of the approach. To allow ENHSP to solve large and complex in-station train dispatching problems, leveraging its modularity, the behaviour of the solver was specialised with domain-specific extensions which leveraged the prior knowledge of the domain in order to guide the search through the search-space. The produced plans are then analysed in several ways in Chapter 7: Firstly, a visualisation tool is introduced, which enabled domain experts to manually inspect the produced plans and validate that the planned movements capture correctly the expected behaviour of trains inside the station. Then, the automatic planner validates the plans produced by human dispatchers in the past, thus showing that the proposed AI dispatcher could have taken the same decision a human operator would have, given the same initial conditions. Then, an analysis is performed to evaluate the capacity of the proposed approach to choose plans that would minimise the total delay of trains in stations. Afterwards, the automatic planning approach is used to simulate a stress-test in which more and more trains are scheduled to pass through the station; this simulation provides an evaluation of the limit on how much of the railway station's available infrastructure can be exploited.

In Chapter 8 other state-of-the-art systems which models the in-station train dispatching are introduced and compared to the proposed approach of this thesis; additionally an excursus on orthogonal problems in the railway domain are introduced. In the final chapter (Chapter 9) an analysis on possible future works is presented and conclusions are drawn.

Chapter 2

Problem definition

This chapter is organized in three sections, in the first (Section 2.1) the topology of a railway station is introduced together with a mathematical formalization, then in Section 2.2 the movements that a train can perform are introduced and, based on the type of the train (i.e. Origin, Destination, Transit or Stop) the initial conditions and goals that will characterise the movement of the train inside the station are presented. Section 2.3 ends Chapter 2 by presenting the In-Station Train Dispatching Problem and by providing a hint on the difficulties that need to be tackled by an automatic planner.

2.1 Topology of a railway station

In [Moscarelli et al., 2017] a classification and statistics of Italian stations is provided, extrapolated from Rete Ferroviaria Italiana's Service Charter. Stations are classified in four categories: *Platinum*, *Gold*, *Silver* or *Bronze* on the base of four criteria: (i) station dimension, (ii) daily number of travellers (iii) connections with other means of transport and (iv) commercial offer. In this thesis, a formulation is presented based upon a real medium-sized station classified by RFI as *Gold*.

By modelling the complexity and nuances of a *Gold* station, stations of minor classifications like *Silver* and *Bronze* are also covered in the formalization. The number of stations in Italy is approximately 2500 containing 13 platinum stations, 103 gold, 850 silver and 1500 bronze. Consequently, numerous stations in Italy can be modelled as expressed in the following sections.

A railway *station* can be represented as a tuple $\mathcal{S} = \langle G, I, P, \mathcal{E}^+, \mathcal{E}^- \rangle$. 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 edges that defines the connections between them. Their status can be checked via track circuits, that provide information about occupation of the segment

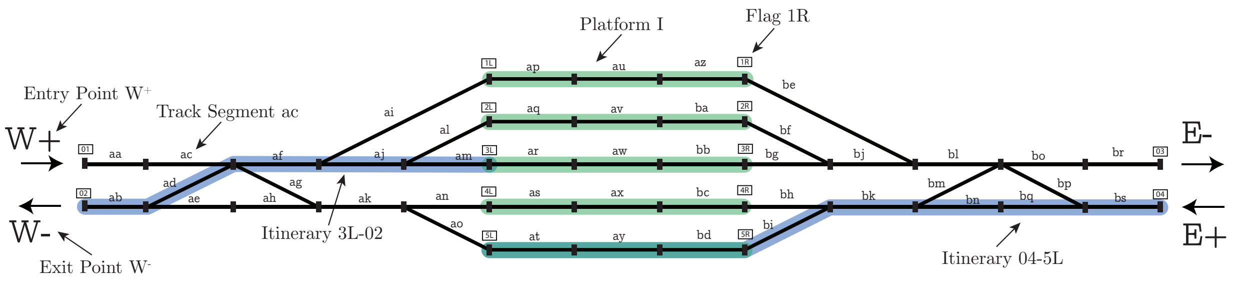


Figure 2.1: Schematic representation of a medium-sized (gold) station. Examples of itineraries are highlighted in a blue colour. One itinerary allows the train to move from the East entry point ($\mathtt{E}^{+}$) to the Platform `V`, the other brings a train from the Platform `III` to the West exit point ($\mathtt{W}^{-}$)

and about corresponding timings. Track segments are grouped in itineraries I.

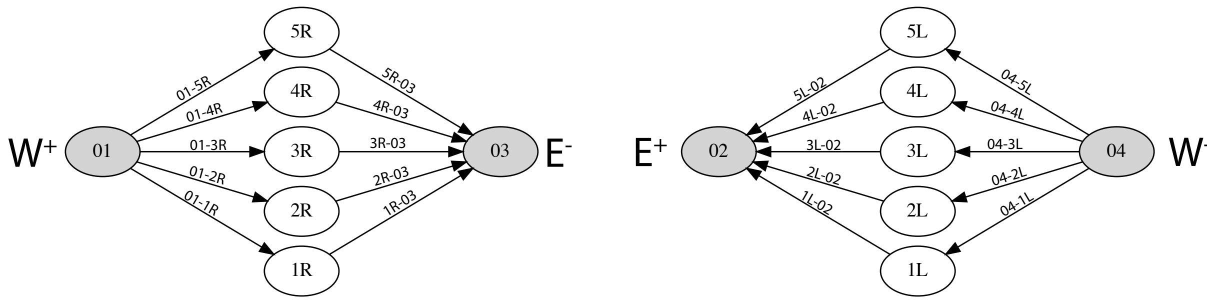


Figure 2.2: Graph of the itineraries. Nodes are flags and the connecting oriented edge are itineraries. Gray edges coincide with portals of the station.

Each itinerary $i = \langle s_{m1}, s_{m2}, ..., s_{mn} \rangle \in I$ is a path graph, i.e., a 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. Itineraries represent a simplification for human operators who are able to give instructions for the movements of a train inside the station in a more simple and concise way; for this reason itineraries explicitly describe a direction of movement inside the station by defining an ordered sequence of track segments and by connecting two *flags* together. Flags are positioned in strategic points of the station, usually near a *portal* or a *platform*, in order to easily manage the flow of trains inside the station. Flags are a graphic and logic constructs that allow train dispatchers to define the movements between points in a station; usually, flags are positioned near a traffic light which tells the train if it should proceed to move to the next itinerary or wait.

P is a set of *platforms* that can be used to embark/disembark the train. Each platform $p \in P$

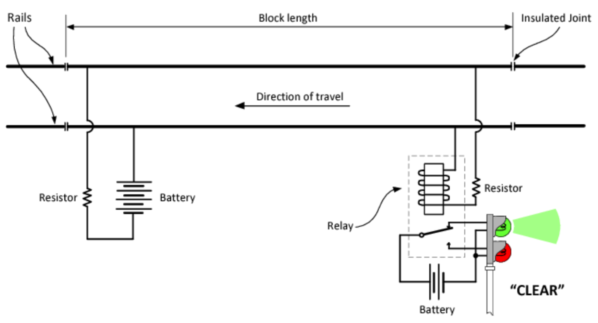
has an associated set of track segments $\{s_i, \ldots, s_j\}$ indicating that trains stopping at those track segments will have access to the platform.

$\mathcal{E}^+ = \{e^+\}$ is a set of entry points to the station from outside the railway network; similarly, $\mathcal{E}^- = \{e^-\}$ is a set of exit points of the station that allow the train to leave the station and enter the external railway network. Entry and exit points together represent portals that connect the station with the external rail network.

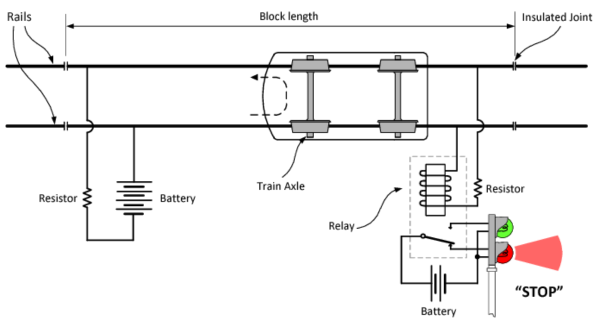
For the sake of this formalisation, entry points and exit points behave like buffers of infinite size which are connected to a single track segment that is the first (last) track segment the train will occupy after (before) entering (exiting) the station.

Figure 2.1 provides a schematic representation of a railway station. Flags are marked with a rectangle providing the identification of the flag. In medium-sized station, flags are usually located only near portals and near platforms, while in larger stations, flags usually split very long itineraries in order to provide more control on the station. Platforms, highlighted in green in the figure, usually are delimited by two flags; based on how the station is represented in the CAD files, the flags belonging to a platform are distinguished in *"left"* or *"right"* based on their position in respect to the platform. These two flags encapsulate a number of track segments that physically represent the platform and will be the track segments in which the train will rest when it's stopping at the platform. The identification number of the platform is given in Roman numbers starting from the top. Flags belonging to a platform are thus identified combining the number of the platform and the position of the flag (e.g. flag `3L` is the left flag of platform `III`). In the figure, two itineraries are highlighted with a blue colour; itinerary `3L-03` connects the flags `3L` and `03` allowing a train to move from platform `III` to the West exit point `W`$^-$. Itinerary `04-5L` brings a train from the East entry point `E`$^+$ to the platform `V`. As it can be seen in the figure, itineraries which brings to the platform are also composed by the track segments of the platform, itineraries which leave a platform start from the track segments immediately after the platform.

Figure 2.2 presents a higher level visualisation of the graph of itineraries of the station. The nodes of the graph represents the flags and the connecting edges represents the itinerary that allows movement between flags. Flags that corresponds to portals are highlighted in grey. As it can be seen, even if itineraries share a subset of track segments the abstract graph of itineraries only provide a sense of direction between the flags and can be seen as disjointed. Moreover, the figure shows that if the movement of the train happens from West to East the right flag (denoted with a R) allows a train to stop at (or pass by) a platform; in the other direction instead the left flag (denoted with L is used). The graph of itineraries results acyclic, thus excluding the possibility for a train to perform tight-turns, reverse its movement and exit from the same direction it entered.



(a) Track circuit off



(b) Track circuit on

Figure 2.3: A schema of a DC track circuit as shown in [Scalise, 2014]. The figure on the left depicts a track segment and the flow of current when no train is running on it. When a train approaches the block, its wheels and axles connect the two running rails together shorting the battery, thus signalling the presence of a train.

2.2 Movements of trains inside the station

A track segment can be occupied by a single train at the time. Track segments are a visual representation of a physical component called track circuit. As stated in [Scalise, 2014], the track circuit consists of a block section of rails electrically insulated with other adjacent pieces of rails by insulated joints at each edge. The signal source (in the DC case it's a battery, in the AC case a transmitter) is connected to the rails at one edge of the block section, while the receiver (a relay) is connected to the other edge. When a train approaches the block, its wheels and axles connect the two running rails together, shorting the circuit and thereby reducing to zero the current through the relay. This causes the relay to drop, turning off the green signal light and turning on the red light to indicate that the block is occupied by a train. Figure 2.3 depicts the two different state of a track circuit.

For safety reasons, a train is required to *reserve* an itinerary before moving on it, 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.

Figures 2.1 and 2.2 provide two different levels of detail on the station. The first one gives a graphical representation of the undirected graph G and represents the physical connections of track segments inside the station. This graph alone is unable to capture the allowed movement of the trains inside the station; given the limited ability of manoeuvre of a train, tight turns must not be allowed. For example, a train that is moving through the track segments `aa` and `ac` cannot move to the track segment `ad` next since the angle would be too tight; instead, the train could move through the segments `af` or `ag`. The choice between the two track segment is mechanically expressed by *switches* that are positioned at the intersection of the track segments. A switch can be in two positions: *normal* or *reverse*; the first position will allow the train to move to the track segment af, the latter to ag. The concept was not modelled in Section 2.1 because the position of the switches can be straightforwardly deduced by the itinerary chosen by the train. The choice of the next track segment after ac depends on the itinerary chosen by the operator for the train. If the operator chooses itinerary 01-5R, 01-4R or 01-3R then the train will move to the track segment af next; instead, if the chosen itinerary is one of 01-4R or 01-5R the next segment will be ag. As it can be seen from Figure 2.2 no itinerary exists that allows a train to move from the entry point W$^+$ (flag 01) to the exit point W$^-$ (flag 02). This forbids the tight turn to the track segment ad after ac.

A train t moving 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. 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.

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.
- *Destination*: the controlled station is the final destination of the train. The train is expected to reach the destination at a specific time 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 a specific time 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 a specific time via a given entry point, and is required to reach a specified exit point after a stop at a platform for embarking/disembarking passengers.

To every train that travels inside the railway network, an identifier number is assigned. This identifier number uniquely represents the route of a train. For example, the train that every day connects the station *Genova Piazza Principe* with the station *Roma Termini* has the unique identifier number of 8623 [Scalzo and Mangione, 2003]. This number will then be used every day to indicate a train with this route.

When a train reaches a destination, it can remain idle for some time, and then it can change the number and depart towards a new destination. At the station level this behaviour can be modelled joining the behaviour of two types of train: a train of type Destination will arrive from an entry point and stops at a platform, then, after a scheduled amount of time the train will change its number and become an origin train that will depart from a platform at a specific instant and leave the station from an exit point.

A *timetable* is the schedule that includes information about when trains should arrive at the controlled station, when they arrive at a platform (arrival time), and the time when they leave a platform (departure time). The timetable provides some constraints on the movement of trains; for example, a train can not depart from the station before the departure time defined in the timetable. If a train has a delay, and it arrives at the platform after the departure time scheduled in the timetable, it has to wait a minimum amount of time in order to allow the passengers to board or alight. The time in which the train can depart the station T_d is defined by the following equation:

$$T_d = max\{T_{dt}, T_a + T_{min}\} \tag{2.1}$$

where T_{dt} is the time of departure scheduled in the timetable, T_a is the arrival time of the train at the platform and T_{min} is the minimum time the train has to remain at the platform.

2.3 In-Station Train Dispatching

We are now in the position to define the in-station train dispatching problem as follows: Given a railway station $\mathcal{S}$, 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.

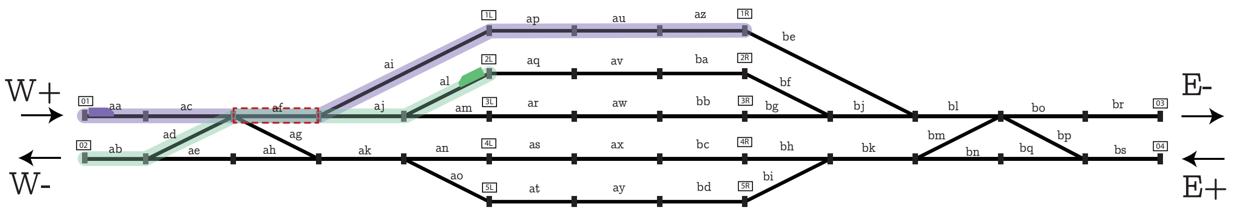


Figure 2.4: Example of incompatible itineraries. The itineraries `01-1R` and `2L-02` are incompatible because they share the track segment `af`.

The dispatching problem has a series of problems that need to be tackled by an automatic planner:

1. Find a route (a sequence of itineraries) between the originating and destination point of a train. For example for a train of type Stop a route must guide the train from the specified entry point to the exit point, passing through a platform in order to allow the train to stop. For an Origin train instead, it must be guided from the platform to the exit point.

2. Guarantee the mutual exclusion of the track segments between trains. When a train reserves an itinerary it has to also block all the track segments of the itinerary in order to avoid the occupation of the track segment by another train while moving, causing an incident. For this reason all the incompatible itineraries are to be tracked and blocked accordingly. In Figure 2.4 it is showed an example of a train that needs to move from the West entry point (`W`$^+$) to the platform `II` and another one that needs to move from the platform `I` to the West exit point (`W`$^-$). The two itineraries share a track segment, and so they are incompatible; for this reason a train needs to move before the other.

3. The mutual exclusion defined at the previous point will rapidly block all the track segments of the station with just a small amount of trains. Looking back at Figure 2.1 it can be seen that a lot of itineraries share a small subset of track segments. For example all the itineraries that bring from the West entry point (`W`$^+$) to all the platform share the track segment `aa`. By blocking all the segments of an itinerary until the train has completed the movement in it quickly brings a station to a halt. For this reason the planner has to allow other trains to move earlier by individuating the times after which it is safe for the train to move in an incompatible itinerary. Again in Figure 2.4 the train that moves first doesn't need to complete the whole itinerary in order for the second train to begin its movement, but it only needs to complete the itinerary until the track segment `af`, which is shared between the two itineraries.

4. Guarantee the constraints imposed by the timetable: a train cannot leave the station before the departure time expressed in Equation 2.1.

Chapter 3

Planning Domain Definition Language

In Artificial Intelligence, automated planning concerns the task of finding a sequence of steps (actions) to reach a predefined goal from a set of initial conditions. In the in-station train dispatching domain the initial conditions can be represented by the number and positions of trains inside the station, the goal could be the successful departure of all the trains from the station and the action are the possible movements that a train can perform inside a station (i.e., entering and leaving the station, stopping at a platform, reserve an itinerary). In this chapter the main concepts behind automated planning are presented following the development history of one of the most important languages for describing planning domains, i.e., the Planning Domain Definition Language (in short `PDDL)` [Mcdermott et al., 1998]. Examples will be provided in the in-station train dispatching domain in order to clarify how the language has been exploited to model the problem at hand. Section 3.1 introduces the definition of a Classical Planning problem which consists in finding a sequence of action that allow to reach a goal from some initial conditions; at the end of the section, some motivations on why Classical Planning is not able to successfully model the in-station train dispatching problem are discussed and followed by the introduction of Temporal and Hybrid Planning, in Section 3.2, which addresses and resolves these problems. In Section 3.3 a formal definition of `PDDL+` is introduced. Finally, in Section 3.4, an explanation is given on why `PDDL+` was chosen among other planning languages is presented.

A more complete and precise formulation of the encoding will be provided in Chapter 5. The domain of in-station train dispatching can be described as fully observable (meaning that the planning agent is able to fetch any information available) and deterministic (meaning that every action the autonomous agent can take have predetermined and certain results).

3.1 Classical Planning

In classical planning, the task of planning is defined as a search problem in a list of states. A state consists in a conjunction of ground[1] predicates (Boolean variables). For example,

$$hasEnteredStation(T1) \wedge trainInItinerary(T1, I01\text{-}4R) \\ \wedge trainIsAtEntryPoint(T2, E^{+})$$

represents the state in which the train `T1` has entered the station and ($\wedge$) is running through itinerary `I01-4R` while the train `T2` is waiting at the portal `E`$^{+}$. From this example it is clear that in this semantic a state doesn't represent the status of the train but instead the status of the complete planning world (the station). The *closed-world assumption* states that every predicate that is not mentioned in the state is assumed to be false. This simplification allows to better manage the transition between states, having to explicit only the things that *actually change* instead of having to say also which predicates *stay the same* (Frame Problem [McCarthy and Hayes, 1981]). An action can be represented by a schema consisting of the action name, the preconditions and the effects of that action. Preconditions and effects of an action are conjunctions of literals (positive or negative ground predicates). The precondition defines the states in which the action can be executed, and the effects define the consequences of applying that action on that state, i.e. how the state changes by choosing the action.

Consider this example of the action schema described in Figure 3.1: the action `entersStation_T1_I01-1R` allows the train `T1` to enter the station through the itinerary `I01-1R`. The precondition states that for this action to happen the train should be at the entry point `W`$^{+}$, should not have entered the station yet, and all the track segments of the itinerary `01-1R` should not be occupied by any train. The effects instead state that, if this action is chosen, the resulting state will no longer have a train at the endpoint `W`$^{+}$, the train should have entered the station (with the aim of avoiding an infinite applicability of this action), the train should reserve and enter the itinerary `01-1R` and all its track segment should be blocked. The action was specified in the Planning Domain Definition Language (`PDDL)` . The syntax is Lisp-like with fully parenthesized prefix notation. The use of the `not` keyword allow to specify a negative literal which in the preconditions means that a predicate should be false (or *undefined*, under the closed world assumption) in order for an action to be applicable and, in the effects, that the result of the application of the action should set the predicate to false.

The *initial state* is a conjunction of positive literals that are true at the beginning of the planning: this is the first state from which the planner will try to apply actions. A *goal state* instead is a conjunction of positive and negative literals that represents a test to assess if the planning should end. The goal state can be seen as a special action with only precondition that, if applicable,

[1] A ground predicate is a predicate in which no variable appears. For example `hasEnteredStation(?t)` could represent the concept for a generic train `t` to have entered the station, while `hasEnteredStation(T1)` states that the particular train `T1` has entered it.

```
(:action T1_entersStation_I01-1R
    :precondition (and
        (trainIsAtEntryPoint T1 W+)
        (not (trainHasEnteredStation T1))
        (not (trackSegBlocked aa))
        (not (trackSegBlocked ac))
        (not (trackSegBlocked af))
        (not (trackSegBlocked ai))
        ...
        (not (trackSegBlocked az))
    )
    :effect (and
        (not (trainIsAtEntryPoint T1 W+))
        (trainHasEnteredStation T1)
        (itineraryIsReserved I01-1R)
        (trainInItinerary T1 I01-1R)
        (trackSegBlocked aa)
        (trackSegBlocked ac)
        (trackSegBlocked af)
        (trackSegBlocked ai)
        ...
        (trackSegBlocked az)
    )
)
```

Figure 3.1: A schema of a PDDL action that allows a train T1 to enter the station through the itinerary I01_1R

is always preferred and terminates the planning immediately. The problem is solved when the planner can find a sequence of actions that transform the initial state in one that passes the condition expressed in the goal state.

In Figure 3.2 an example of initial and goal state is provided. The initial state in which a train T1 is at the entry-point W^+ while another train T2 is stopped at platform II. The goal condition is that the train T1 and T2 should have left the station from E^- and W^- respectively.

Classical planning solvers are thus able to find a sequence of actions that, starting from the initial conditions, lead to the goal state. However, classical planning is not able to completely represent the in-station train dispatching problem for the following reasons:

1. In classical planning actions are instantaneous; this means that the effects of an action are

```
(:init
    (trainIsAtEntryPoint T1 W+)
    (trainIsStoppingAtPlatform T2 S_II)
)

(:goal
    (and
        (trainExitsStationAtExitPoint T1 E-)
        (trainExitsStationAtExitPoint T2 W-)
    )
)
```

Figure 3.2: The initial state and the goal state expressed in the PDDL language

applied immediately if the preconditions of the action are met and the planner choose to apply that particular action. In the problem at hand, instead, some actions needs to define different instants in which the effects have to be applied to the world. Consider an action which describes a train stopping at a platform: this action needs to take some time in order to complete, and some effects have to be applied at the beginning and some other at the end of the action. For example at the beginning of the stop at the platform the train has to signal that it is stopping and reserving all the track segments of the platform, at the end instead it has to signal its readiness to continue its journey through the station thus signalling to the planner that it is able to reserve another itinerary for the purpose of leaving the station.

2. The planner is not obliged to perform an action, even if all its preconditions are met; this is because it is possible that the action could lead to a dead-end or there is a better action to apply (the definition of *quality of an action* and of cost functions will be covered in the next sections when discussing heuristics). Events, on the other hand, happens when their preconditions are met, independently of the decisions of the planner: in the in-station dispatching problem events have to be used in order to correctly model the arrival of trains at portals, which can not be delayed or overlooked by the planner. Classical planning has no constructs able to manage events in a simple way.

3.2 Temporal and Hybrid Planning with PDDL+

The modelling incapacity of Classical Planning presented at the end of the previous section were solved with the introduction of PDDL+ [Fox and Long, 2006a] which is able to describe *Temporal* and *Hybrid* problems: *Temporal* meaning that it has a native way of dealing with time (in reality with any type of numeric variable, called functions) and *Hybrid* meaning it can model mixed discrete-continuous domains, thus implementing events and processes. Syntactic elements able to model temporal domains was already introduced with `PDDL2.1` [Fox and Long, 2003], a more advanced version of `PDDL` (or `PDDL1.0`). Since `PDDL+` is derived from `PDDL2.1,` and it is strictly more expressive than it [Fox and Long, 2006a] the first was preferred for its natural support for dealing with exogenous events. Notably, `PDDL+` has been already successfully exploited in a number of application domains including UAV manoeuvring [Ramirez et al., 2018], battery management [Fox et al., 2012], and urban traffic control [McCluskey and Vallati, 2017].

`PDDL+` introduces two new native constructs: processes and events. Processes model a continuous change in numeric fluents and events allow modelling exogenous happenings that have to occur as soon as their preconditions are met. Before introducing the concepts of processes and events, a brief excursus on how the planner deals with continuous problems has to be taken (a more in depth analysis on how the planner works will be introduced in the next sections). In order to manage continuous time-dependent processes, the planner has to discretise the time in multiple discretisation steps (δ) that determines how often an action can be applied, or an event can be triggered. The value of δ can be given to the solver by the AI expert based on the knowledge on the domain and the granularity by which the solution has to be found. The choice of the correct discretisation step can be delicate: the choice of a very small δ can exponentially increase the solving time, since at every discretisation step several actions could be applied, enlarging the search-space. On the other hand, the selection of a large δ can invalidate the solution: consider for example an in-station dispatching problem in which a train has to leave the station before a certain time T; if all the possible plans allows the train to leave the station at most at an instant $t = T - \frac{\delta}{2} + \epsilon$ (with ϵ arbitrarily small) then no viable plan can be found since the planner would realise the achievement of the goal only after the time T has passed. In Chapter 6 an algorithm is presented for an adaptive delta that removes the necessity to pick a discretisation step beforehand.

In Figure 3.3 two examples of processes are presented. The process `incrementTime` models the continuos flow of time: having no precondition, the process is triggered at every discretisation step applying the effects of increasing the numeric function `time` of the quantity `#t` which is grounded by the planner with the defined value of δ. The process `incrementTrain-StopTime` instead takes care of counting the time in which a generic train `t` is stopping at any platform, increasing the numeric fluent `(`trainStopTime) of that particular train.

An event has the same schema as an action, but as previously mentioned, represents an occurrence in time that is out of the control of the planner and has to happen as soon as its preconditions are met.

Figure 3.4 presents the schema of an event that takes care of signalling the arrival of a train at an entry point of the station. The event `T1_arrivesAtEntryPoint_W+` triggers as soon the numeric function `time` reaches a predetermined value t_a which is the time of arrival of the train at the station, defined in the instance. The two conditions on the boolean predicates `train-HasEnteredStation` and `trainIsAtEntryPoint` allows the event to be triggered only

```
(:process incrementTime
    :parameters()
    :precondition()
    :effect (and
        (increase time #t )
    )
)
(:process incrementTrainStopTime
     :parameters(?t - train)
     :precondition (and
         (trainIsStopping ?t)
     )
     :effect (and
         (increase (trainStopTime ?t) #t )
     )
)
```

Figure 3.3: An example of two processes expressed in the `PDDL+` language. The process `incrementTime` models the continuous flow of time. The process `incrementTrainStopTime` counts the stopping time of a train at a platform.

once and signals that a train is waiting at the entry point of the station. Depending on the traffic inside the station, the planner could decide to have the train wait at the entry point for some time, before allowing it to enter the station; the coupling of the event `arrivesAtEntryPoint` and of the action `entersStation` allows to correctly model this behaviour by forcing the presence of the train at the instant t_a with an event but leaving to the planner the decision of when to allow the train to enter the station, with an action.

While in classical planning the result of the planning was a sequence of actions that had to be performed in sequence as a means to reach a goal from the initial conditions, in temporal planning the plan produces is a sequence of *timestamped* actions describing not only the sequence but also the instance in which the action has to be performed, thus waiting for events to manifest their effects in between actions.

3.3 PDDL+ formalization

In this section, a more formal definition of `PDDL+` is provided. An encoding in `PDDL+` is characterized of two parts: the domain and the problem. The domain contains all the information regarding how the modelled problem works independently on the instance provided; in this file

```
(:event T1_arrivesAtEntryPoint_W+
    :parameters()
    :precondition (and
        (>= time  253 )
        (not (trainHasEnteredStation T1))
        (not (trainIsAtEntryPoint T1 W+))
    )
    :effect (and
        (trainIsAtEntryPoint T1 W+)
        (trainHasArrivedAtStation T1)
    )
)
```

Figure 3.4: The schema of a `PDDL+` event that takes care of signalling the arrival of a train at an entry point of the station

is modelled, for example, how a train moves inside a station. The actual list of the trains that will move through the station, their initial conditions and goals are instead defined in the problem, together with the topology of the station (the interconnections between itineraries, the track segments that compose the itineraries and the platforms, etc). This decoupling between the behaviour of the model and the actual instance gives the possibility to specify a single domain that works for multiple scenarios and stations.

PDDL+ Domain Model. A PDDL+ planning domain model $\mathcal{D}$ is defined by the following tuple

$$\mathcal{D} : \langle \mathcal{T}, \mathcal{C}, \mathcal{F}, \mathcal{X}, \mathcal{A}, \mathcal{E}, \mathcal{P} \rangle$$

- $\mathcal{T}$ (Types) is a set of types (i.e. `train`, `itinerary`, `platform`, etc).
- $\mathcal{C}$ (Constants) is a set of typed objects, each of which is simply a name given to the object, and its type. For example `train T1` or `platform I`, etc.
- $\mathcal{F}$ and $\mathcal{X}$ are sets of propositional and numeric fluents, respectively. A more detailed list of all the predicates and numeric fluents will be introduced in Chapter 5.
- $\mathcal{A}$ (Actions), $\mathcal{E}$ (Events), and $\mathcal{P}$ (Processes) are sets of transition schema. A transition schema is the tuple $\langle \sigma, pre, eff \rangle$ where:
    - σ is a sequence of objects from $\mathcal{C}$ or variables typed in $\mathcal{T}$,
    - pre is a first order formula [Smullyan, 1995],
    - eff is a set of Boolean and numeric effects. Boolean effects are assignments $\langle p, \top, \bot \rangle$ with $p \in \mathcal{F}$ where numeric effects are assignments $\langle p, \xi \rangle$, with ξ being an arithmetical expression.

– Both pre and eff only mention fluents from σ or objects from $\mathcal{C}$.

PDDL+ Problem A planning problem is defined combining a planning domain model $\mathcal{D}$ with a set of typed objects, an initial state and a goal. A planning problem asks whether, given a planning domain model, a set of objects, an initial state and a goal, there is a plan that lets the agent achieves the goal from such an initial state, considering the constraints imposed by $\mathcal{D}$, and the actions that can be performed. More formally, let $\mathcal{D}$ be a PDDL+ domain model; a planning problem is the tuple $\Pi : \langle \mathcal{D}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ where:

- $\mathcal{O}$ is a set of typed objects (i.e., the objects that compose the instance: `train T1`, `platform I`, `itinerary I01_53`, etc).
- $\mathcal{I}$ represent the initial conditions and is a function that assigns i) a truth value to all ground propositional fluents in $\mathcal{F}$ over compatible objects from $\mathcal{C} \cup \mathcal{O}$ ii) a rational value to all ground numeric fluents obtained substituting all numeric fluents in $\mathcal{X}$ over compatible objects from $\mathcal{C} \cup \mathcal{O}$.
- $\mathcal{G}$ represent the goal to achieve and is a set of first order formulas over ground propositional and numeric fluents.

```
51,00:  T1_entersStation_IW1
83,00:  T2_entersStation_IE5
101,00: T1_beginStop_IW1_S_I
112,00: T1_overlaps_IW1_I1E
123,00: T1_endOverlap_IW1_I1E
131,00: T2_beginStop_IE5_S_V
141,00: T2_overlaps_IE5_I5W
157,00: T2_endOverlap_IE5_I5W
157,00: T3_entersStation_IE5
169,00: T1_exitsStation_I1E
190,00: T2_exitsStation_I5W
190,00: T4_beginsVoy_S_IV_I4W
201,00: T3_beginStop_IE5_S_V
243,00: T4_exitsStation_I4W
```

Figure 3.5: An example of a plan for four trains that move inside the station: train `T1` enters from the West entry-point and moves to the East exit-point after stopping at platform `I`, `T2` enters from east and exits at west stopping at platform `V`, train `T3` is a *Destination* train which enters from $\mathtt{E}^{+}$ and ends its voyage in platform `V` (after `T2` has left it), train `T4` is an *Origin* train which originates at platform `IV` and leaves the station at exit-point $\mathtt{W}^{-}$

PDDL+ Plan As previously mentioned, a PDDL+ plan is an ordered timestamped sequence of actions which have to be executed in those precise moments in order to reach a goal $\mathcal{G}$ from a set of initial conditions $\mathcal{I}$. The order of the action is total, meaning that even actions that should be applied at the same time need to be applied in order, this because the precondition of the subsequent action could require the effects of the actions which is applied before. More formally, a plan is a sequence $\mathcal{S} = (s_1, s_2, \ldots, s_n)$ where $s_i = \langle t_i, a_i \rangle$ where:

- $t_i \in \mathbb{R}$ is the timestamp of the action, i.e., the moment in which the action has to take place. As discussed in the previous section, a discretisation step has to be provided to the planning engine in order to deal with a finite amount of actions; for this reason t_i is always a multiple of δ.
- $a_i \in \mathcal{A}$ is the action that has to be applied at t_i. Processes and events are not under the direct control of the automatic planner, thus they are not present in the final plan.

In Figure 3.5 an example of a plan for four trains that moves inside the station: train T1 enters from the West entry-point and moves to the East exit-point after stopping at platform I, T2 enters from east and exits at west stopping at platform V, train T3 is a *Destination* train which enters from E$^+$ and ends its voyage in platform V (after T2 has left it), train T4 is an *Origin* train which originates at platform IV and leaves the station at exit-point W$^-$. As it can be seen, actions T2_-endOverlap_IE5_I5W and T3_entersStation_IE5 happens at the same time, but they are still ordered since the endOverlap frees the last track segments of itinerary IE5 which are precondition for the second action. In this case, the timestamp is always an integer since δ was chosen to be equal to 1.

3.4 Why PDDL+

In this section, an insight will be provided on the reasons why PDDL+ was chosen to model the problem at hand:

- As described in Section 3.3 a PDDL+ encoding is divided in two parts: the domain and the problem. This separation allows to describe the movements of trains in the station independently on the topology of the station and the initial position of the trains inside the station. This allows to define a set of rules valid for every (medium-sized) station and then use a different problem for every scenario and railway station. As it will be seen in Chapter 4, RFI specifies the topology of the station in a well-formed document which can be processed, and from which the PDDL+ problem formulation can be extrapolated.
- As shown in the examples of this chapter, the schemas of actions, events and processes are quite easy to read, even without a background in AI and logic. For this reason PDDL+ can

be shown to experts of the railway domain, who are not experts in AI, in order to validate the modelling choices made by the AI engineer. Other well-known formulations instead, like the Mixed Integer Linear Programming, are written in mathematical notation that can result hard to understand, even to people with some mathematical background. A basic MILP formulation will be introduced in the Related Work Chapter 8 in order to show the differences between the two approaches.

- The possibility to specify external events which are outside the control of the planner permits to natively model the flow of trains coming from the railway network outside the station.

- One of the most important challenges in knowledge representation is called the Frame Problem [McCarthy and Hayes, 1981]: the frame problem is the challenge of representing the effects of actions in logic without having to represent explicitly many intuitively obvious non-effects. To put it in prospective, the in-station train dispatching "world" has many variables, but, fortunately, each action changes no more than some small number of those variables. For this reason, actions are typically specified by what they change, with the implicit assumption that everything else (the frame) remains unchanged. `PDDL+` is an "action description language" which is able to elude the frame problem. As it will be seen in Chapter 6, ENHSP solves a `PDDL+` problem by copying the state of the world at every action and then applying the effects on this copied state in order to create a new state in which everything remained unchanged except for the effects of the action. In other planning/scheduling languages, like ASP [Lifschitz, 1999] or MILP, the frame problem has to be specifically addressed by the AI engineer, resulting in a harder-to-read encoding.

- In literature are available several planning engines, each with its own peculiarities, all of them open-sourced, highly customizable and tested against a high number of benchmarks. In this thesis the state-of-the-art planning engine ENHSP was utilized in order to solve the proposed `PDDL+` encoding. In the Related Work Chapter (Chapter 8) some of these other planning engines will be discussed.

Chapter 4

Input

In this chapter, the data needed in order to plan the correct movements of trains in a station is discussed. In Section 4.1 the data provided by Rete Ferroviaria Italiana (RFI), the owner of Italy's railway network, that provides signalling, maintenance and other services for the railway network, is presented. An ontology, which allows to formally describe the components of a station, and to reason upon it, in order to extract more knowledge on the status of every component, is presented in Section 4.2. Finally, in Section 4.3, a methodology to predict the duration of movements of a train inside a station is discussed.

4.1 Data provided by Rete Ferroviaria Italiana

The novelty of this thesis relies on the use of the structure and real data coming from a station in the North-West of Italy.[1] Rete Ferroviaria Italiana[2] provided four main files:

1. A CAD file with the drawings of the station complete with information about track segments, portals and platforms. The visualisation of the station in the drawings was helpful in order to better understand all the nuances and complications of the movements of the trains inside the station and most importantly for the correct debug of the planner (covered in Section 7.1). Figure 2.1 is a dummy station created for better giving visual support of some concepts of this thesis. Despite not representing a real station Figure 2.1 correctly represents the structure complexity of a medium-sized station.

2. A JSON-like file containing a structured representation of the station's topology, for exam-

[1]Because of confidentiality issues the author cannot report the name of the station and other details regarding the data that cannot be disclosed by RFI.

[2]The author sincerely thank Renzo Canepa (RFI) for his support and for providing the data.

ple, all the track segments that compose itineraries and platforms are listed together with information about flags, portals and track segments. The real modelled station includes 130 track segments (out of which 34 are track switches), 107 itineraries, 10 platforms, 3 entry points, and 3 exit points.

3. A Flat-File Database containing timestamped logs coming from the Rail Traffic Management System (TMS) which controls and monitors the movements of trains in a cluster of station in the North-West of Italy. In this file some information can be retrieved:

    (a) The instant of occupation and liberation of a track segment: as stated in Section 2.1 the status of a track segment is controlled with a track circuit. As stated in Chapter 2, a track circuit is a physical component that acts as an open circuit with the two rails of the track. When a rolling stock moves on top of a track segment, its axles shunt the rails together, closing the circuit and activating a relay which signals the occupation of the track segment. The track segment becomes free when the last axles of the train leave the track segment, opening the circuit. The TMS tracks the status of the relay and logs in the main database the timestamp in which the status of the track circuit changes.

    (b) The instant of reservation, occupation and liberation of an itinerary: as stated before, the itinerary represents a sequence of track segments which the train will move through. An itinerary needs to be reserved in advance in order to signal to the controlling system that all the track segments of that itinerary are to be left clear until the train has finished its movement. This will also give to the operators a clear picture of the future movements of the trains inside the station. The itineraries' logs are the only ones containing information about the trains which are moving in the station (in particular the trains' identification number). Track circuits instead, being physical components, are unable to identify the train but only signal its presence.

    (c) The start and end of a stop in a platform by a train. This information becomes helpful when analysing the time spent by the train in boarding/alighting operations.

    The logs captured the movements of trains in the period of February 2020 to May 2020. This period encapsulates the COVID-19-related lockdown in Italy which started in March 2020. The lockdown effects on the mobility are reflected in the logs: the average number of trains per day was 130 before the lockdown and 50 after movement restrictions were enforced in Italy.

4. The timetable of the station which contains information about the type of train (Stop, Transit, Destination, Origin), the nominal time of arrival at the platform and the minimum allowed time of departure from it.

4.2 Ontology

An ontology is a formal explicit description of concepts (classes) in a domain: it allows to list the properties of each concept, assigning to it various features and attributes. An ontology, together with a set of individual instances (i.e. instantiations of classes in to objects) constitutes a knowledge base, which represent what is currently known of the domain at hand. By defining a set of rules and restrictions which connects properties and classes together, it is possible to apply reasoning techniques in order to expand the knowledge base of the domain.

The introduction of an ontology, given the complicated structure of the station, is necessary in order to derive new knowledge on the state of the station's components given the rules of interactions between them, provided by experts of the domain. For example, the reservation of track segments can be deduced by the status of the itineraries, incompatibility between itineraries can be deduced by intersecting the set of track segments that compose them, the connection between itineraries allows to define a set of routes that connect an entry point to an exit point.

4.2.1 Classes

The classes of the ontology are the following:

- **Topology Component**: A topology component is an abstract parent class that contains all the components mentioned in Chapter 2: **Flag**, **Itinerary**, **Platform**, **Portal**, **Station** and **TrackSegment**. The entity Portal has two children, **EntryPoint** and **ExitPoint**.
- **Train**: A Train represents a physical object that will move through the station.

4.2.2 Object properties

- `allowsToStopAt(Itinerary i, Platform p)`: if an itinerary i brings to a platform p (i.e. the flag at the end of the itinerary is coincident with a platform).
- `belongsTo(TopologyComponent a, TopologyComponent b)`: if a component a is part of a component b (i.e. a track segment is a part of an itinerary or a platform, an itinerary is part of a station, etc). The inverse is `isComposedOf(b,a)`.
- `blocks(Train t, TopologyComponent c)`: if a train t is physically occupying a component (that can be a track segment, an itinerary or a platform). The inverse is `isBlockedBy(c,t)`.
- `bringsTo(Itinerary i, Flag f)`: if an itinerary i directs a train to a flag f. The inverse is `isDestinationOf(f,i)`.

- `startsFrom(Itinerary i, Flag f)`: if an itinerary i originates from a flag f. The inverse is `isOriginOf(f,i)`.
- `entersFrom(Train t, EntryPoint e)` (or the similar property `exitsFrom(t, ExitPoint e)`) if a train t will enter (or exit) the station from the EntryPoint (ExitPoint) e.
- `isConnectedWith(Itinerary a, Itinerary b)`: if the itinerary a is connected with the itinerary b (i.e. if the destination flag of a is the origin flag of b).
- `isIncompatibleWith(Itinerary a, Itinerary b)`: if the itinerary a shares a track segment with itinerary b.
- `reserves(Train t, TopologyComponent c)`: if a train t has reserved a component c. The reservation starts when the operator declares that the train will move through the component.
- `stopsAt(Train t, Platform p)`: if a train t needs to stop at the platform p for embarking/disembarking operations.
- `allowsToExitFrom(Itinerary i, ExitPoint e`: if an itinerary i is a viable route in order to exit the station through the exit point e.
- `reachableFrom(Itinerary i, EntryPoint e`: if an itinerary i can be reached from the entry point e.
- `canTravel(Train t, Itinerary i)`: if a train t, which needs to move from an entry point to an exit point, can use itinerary i in order to reach its goals.

4.2.3 Expanding the knowledge base with Semantic Web Rules

Semantic Web Rules [Horrocks et al., 2004] enables Horn-like rules to be combined with an OWL knowledge base in order to reason upon it. These rules are of the form of an implication between an antecedent (body) and consequent (head). The intended meaning can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. Both the antecedent (body) and consequent (head) consist of zero or more atoms. An empty antecedent is treated as trivially true (i.e. satisfied by every interpretation), so the consequent must also be satisfied by every interpretation; an empty consequent is treated as trivially false (i.e., not satisfied by any interpretation), so the antecedent must also not be satisfied by any interpretation. Multiple atoms are treated as a conjunction.

When specifying the properties of Section 4.2.2, some information can be deduced from other properties without having to manually specifying it. A set of Semantic Web Rules were introduced in order to increase the knowledge base of the ontology.

Incompatible Itineraries. If two itineraries i_1 and i_2 share a segment s then they are incompatible:

$$\begin{aligned} Itinerary(i_1) \wedge TrackSegment(s) \wedge Itinerary(i_2) \\ \wedge belongsTo(s, i_1) \wedge belongsTo(s, i_2) \rightarrow isIncompatibleWith(i_1, i_2) \end{aligned} \tag{4.1}$$

Incompatible Occupancy. If a train t has reserved an itinerary i then every other itinerary i' that is incompatible with i is blocked until the reservation is lifted:

$$\begin{aligned} Itinerary(i) \wedge Itinerary(i') \wedge Train(t) \wedge reserves(t, i) \\ \wedge isIncompatibleWith(i, i') \rightarrow block(t, i') \end{aligned} \tag{4.2}$$

Itinerary Connections. If an itinerary i_1 brings to a flag f and an itinerary i_2 starts from that flag, then they are connected:

$$\begin{aligned} Itinerary(i_1) \wedge Itinerary(i_2) \wedge Flag(f) \wedge bringsTo(i_1, f) \\ \wedge\ startsFrom(i_2, f) \rightarrow isConnectedWith(i_1, i_2) \end{aligned} \tag{4.3}$$

Platform. An itinerary i brings to a platform p if the itinerary brings to a flag f which belongs to the platform p. Iteratively, an itinerary i_1 brings to a flag f if it is connected to another itinerary i_2 which brings to the flag f itself.

$$\begin{aligned} Itinerary(i) \wedge Platform(p) \wedge Flag(f) \wedge bringsTo(i, f) \\ \wedge\ belongsTo(f, p) \rightarrow allowsToStopAt(i, p) \end{aligned} \tag{4.4}$$

$$\begin{aligned} Itinerary(i_1) \wedge Itinerary(i_2) \wedge Platform(p) \wedge allowsToStopAt(i_2, p) \\ \wedge\ isConnectedWith(i_1, i_2) \rightarrow allowsToStopAt(i_1, p) \end{aligned} \tag{4.5}$$

Reservation of an itinerary. If a train t reserves an itinerary i then it also reserves all the track segments s that are part of the itinerary:

$$\begin{aligned} Itinerary(i) \wedge Train(t) \wedge TrackSegment(s) \wedge reserves(t, i) \\ \wedge\ belongsTo(s, i) \rightarrow reserves(t, s) \end{aligned} \tag{4.6}$$

Itinerary brings to exit point. An itinerary i allows exiting from an entry point if it brings to a flag f which belongs to the exit point. Iteratively, an itinerary i_1 allows exiting from an entry point if it is connected to another itinerary which allows it.

$$\begin{aligned} Itinerary(i) \wedge Flag(f) \wedge ExitPoint(e) \wedge belongsTo(f, e) \\ \wedge\ bringsTo(i, f) \rightarrow allowsToExitFrom(i, e) \end{aligned} \tag{4.7}$$

$$\begin{aligned} Itinerary(i_1) \wedge Itinerary(i_2) \wedge ExitPoint(e) \wedge allowsToExitFrom(i_2, e) \\ \wedge\ isConnectedWith(i_1, i_2) \rightarrow allowsToExitFrom(i_1, e) \end{aligned} \tag{4.8}$$

Itinerary arrives from entry point. An itinerary i arrives from an entry point e if it's directly connected to it or it is iteratively connected to another itinerary which does.

$$\begin{aligned} &Itinerary(i) \wedge Flag(f) \wedge EntryPoint(e) \wedge belongsTo(f, e) \\ &\quad \wedge startsFrom(i, f) \rightarrow reachableFrom(i, e) \quad (4.9) \\ &Itinerary(i_1) \wedge Itinerary(i_2) \wedge EntryPoint(e) \wedge reachableFrom(i_1, e) \\ &\quad \wedge isConnectedWith(i_1, i_2) \rightarrow reachableFrom(i_2, e) \quad (4.10) \end{aligned}$$

Itineraries traversable by a train A train t, which moves from the entry point a to the exit point b, will only move through itineraries that brings to the exit point b and that arrives from the entry point a.

$$\begin{aligned} &Itinerary(i) \wedge Train(t) \wedge EntryPoint(a) \wedge ExitPoint(b) \\ &\wedge entersFrom(t, a) \wedge exitsFrom(t, b) \wedge allowsToExitFrom(i, b) \\ &\quad \wedge reachableFrom(i, a) \rightarrow canTravel(t, i) \quad (4.11) \end{aligned}$$

The ontology will result useful in the next chapter as a means to remove unhelpful actions and events from the encoding. For this reason, depending on the initial conditions and goal of every single train, the ontology is used to reason on the itineraries which will actually be useful to the planner in order to plan. All the actions that concerns itineraries that will not help a train in reaching his goals are automatically pruned.

4.3 Estimation of the duration of trains movements

In order to realistically plan the movements of trains inside a railway station, the time it takes for a train to move between points inside the station is of paramount importance. In the model presented in this thesis, since trains are not equipped with a GPS, these times were predicted based on the logs coming from the Rail Traffic Management System (TMS) which records the occupation and liberations of track segments in the station [Theeg and Vlasenko, 2009]. 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. These purely statistical methods are able to give an estimate on the time needed by the train to complete a track segment with a MAE of 19.3 ± 5.1. The high average error requires the need for the introduction of a deep-learning based multiscale model able to take into account the relationship between trains in the station, automatically learn the representation and improve the accuracy of the statistic model. This thesis straight-forwardly implements the models trained by [Boleto et al., 2021] which, in their paper, adopted a two-step approach to address the problem of predicting the duration of in-station train movements by firstly leveraging on state of the art shallow models based on Random Forests (RF), fed by domain experts (i.e., operators of RFI) with domain specific features, to improve the current predictive systems. Then, custom deep multiscale models are introduced based on Temporal Convolutional Network (TCN) able to automatically learn a rich and expressive representation directly from the data and to improve the accuracy of the shallow models. Benchmarks on real-world data coming from the same station discussed on this thesis shows an improvement to a MAE of 7.8 ± 2.5 for the shallow models and 6.9 ± 2.1 for the deep models.

Chapter 5

Encoding

In this chapter, we introduce and specify the `PDDL+` model designed for dealing with the problem described in the previous chapters.

In Section 5.1 a brief and more informal introduction to all the actions, events and processes of the encoding is presented in order to provide a basic idea on how the movements of the trains are modelled. In Section 5.2 the grounding in the preprocessor is introduced, which is able to reduce the search-space and make easier the specification of the encoding. In Section 5.3 all the fluents that compose the encoding are presented which will be used in Section 5.4 which thoroughly defines all the actions, events and processes of the encoding.

5.1 Introduction to the encoding

Figure 5.1 provides a graphical representation of how actions and events are interleaved in regard to different potential states of a train: Rectangles represent events and squared rectangles represent actions. Processes are omitted for clarity. Train states are not explicitly represented in the `PDDL+` model, but they can help to understand the structure and the dynamics of the encoding. As already defined in Section 2.2 a train can be: *Stop*, ready to enter the controlled station from an entry point, stop at a platform and exit from an exit point; *Origin*, a train that originates from the controlled station – the train is stopped at a platform and needs to be moved towards an exit point; *Destination*, the train enters from an entry point and then, after stopping at a platform, terminates its trip at the station.

The core of the proposed `PDDL+` encoding is the way in which the movement of trains in the railway station is modelled and controlled. The operators `EnterStation`, `BeginsVoyage`, and `BeginsOverlap` are all allowing the corresponding train to reserve an itinerary and to start moving on it. They have differences in terms of preconditions and effects, as they deal with

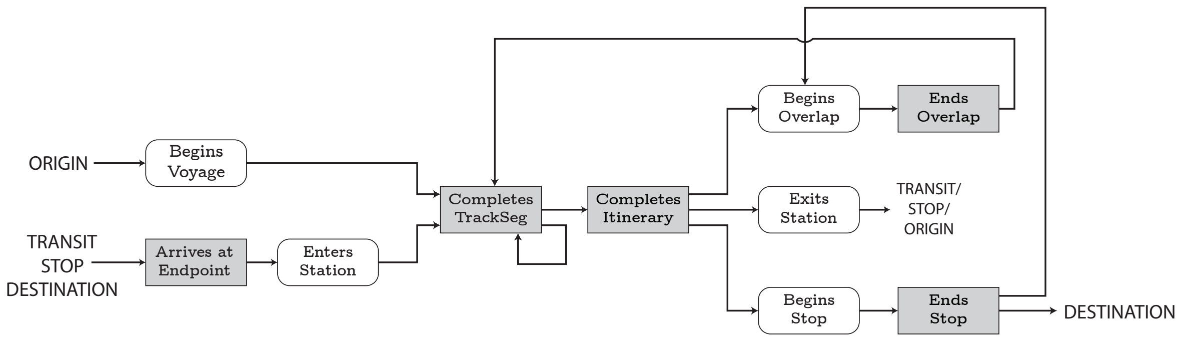


Figure 5.1: A flow chart showing the movements of the train inside the station based on its type. Rectangles represent events and squared rectangles represent actions. Processes are omitted for clarity.

trains in different logical states. For instance, the `EnterStation` operator can be used only by a train that is at an entry point: one of the effects is that the train is no longer approaching the train station, but is navigating through the controlled station. The use of these operators trigger a process that is used to model the time needed by the train to reach the end of the itinerary. Over time, track segments of the itinerary that are not occupied by the train any more, are released via the `CompletesTrackSegment` event. When the train has completed all the track segments of the itinerary, the event `CompletesItinerary` is triggered. Notably, when a train reaches the end of an itinerary, it is still occupying part of its segment tracks: the precise number depends on the type of train, and on the length of the segments. The `BeginOverlap` operator is used by a train moving between two subsequent itineraries of the station, to take into account the "overlapping" time needed by a train to completely leave the previous itinerary.

Figure 5.1 also shows the `ExitStation` operator, that is used to allow a train that reached an exit point to actually leave the station and release the occupied track segments, and the `Begin-Stop` operator, that is used to stop a train at a platform to allow the disembarking/embarking of passengers. The duration of the stop is variable: each type of train has a minimum time that is required to stop to safely allow the movement of passengers; further, a train is not allowed to leave a platform before its timetabled leaving time. This is encoded in the corresponding action using appropriate preconditions. Finally, if the train is of type *Destination*, after the event of `EndsStop` which signals that a train has completed the disembarking process, a predicate is used to model the fact that a destination train has terminated, and should not move any further.

Dedicated processes, not shown in Figure 5.1 for the sake of readability, are used to keep track of the time spent by a train: (i) navigating an itinerary; (ii) overlapping, i.e., moving from one itinerary to the next; and (iii) stopping at a platform. Further, in order to encode an explicit notion of the time that is passing, in our model a dedicated `timePassing` process is employed. This helps when dealing with the time-related aspects of the problems to be solved, by avoiding the need to use timed initial literals.

5.2 Preprocessor and Grounding

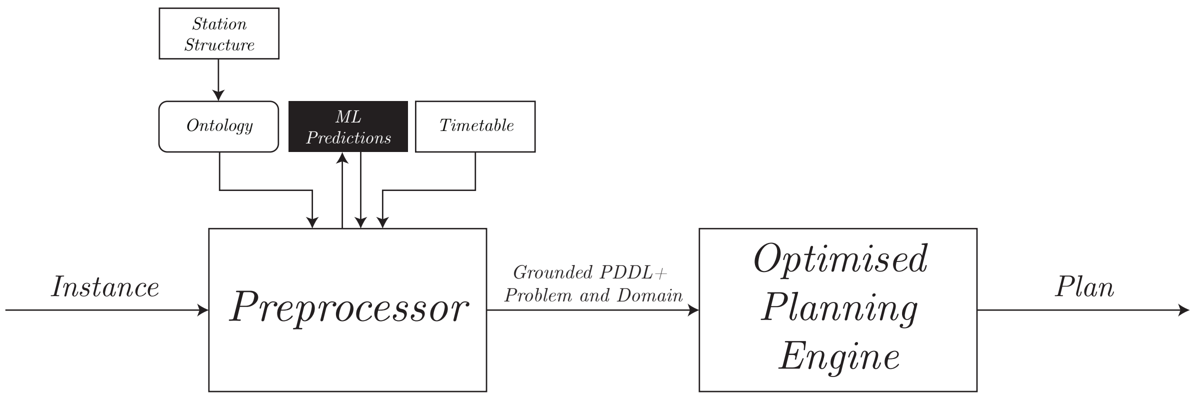


Figure 5.2: The architecture that produces the grounded `PDDL+` Domain and Problem

As discussed in Chapter 3 a `PDDL+` domain and problem are transformed internally by the planning engine in a search problem in a search-space. In order to achieve this a procedure, called *grounding*, has to be made in advance in order to substitute all the variables with the constants of the domain, for example by creating sets of actions for every train (e.g., the action `enter-Station` which deals with a generic train has to be created for every train). It's easy to see that with the number of trains inside the station growing, it also increases, in an exponential fashion, the dimension of the search-space in which a solution has to be searched. The curse of dimensionality [Bellman, 1966] is intrinsic in planning problems, but something can be done in order to cut waste and reduce the number of possible states that are already known to be unreachable. The planning engine ENHSP already has a grounding system inside it able to substitute the variables and create *grounded* actions, events and processes. Besides substituting variables with constants, the ENHSP grounder is able to reduce the set of actions by performing a static analysis method [Scala and Vallati, 2021] which is able to identify actions/events which preconditions will never be satisfied and remove them from the search-space. However, being a *static* analysis, the system is unable to identify actions/events that *could* in theory be activated but will never have their preconditions activated because of the structure of the problem at hand. A practical example is a train that needs to move from the West entry-point to the East exit-point; as it can be seen in Figure 2.2 some itineraries will never contribute to reaching the goal: for this reason the action that involves these itineraries can be straightforwardly pruned and removed from the search-space. In Figure 5.2 a block diagram of the architecture used to produce the grounded `PDDL+` problem and domain is shown. The preprocessor takes as input (i) the instance (i.e., the list of trains with their types, initial condition and goals), (ii) the station structure, formalized with the ontology presented in Chapter 4, (iii) the timetable of the day in which the planning take place. The preprocessor elaborates this data in the following way:

1. using the ontology, based on the initial conditions and goal of every single train, to reason on the itineraries which will actually be useful to the planner in order to plan. All the actions that concerns itineraries that will not help a train in reaching his goals are automatically pruned.
2. querying the machine learning model for predicting the time of occupancy of itineraries, track segments and stopping times at platforms for every train.
3. pruning all the actions that are not required by the type of the train (e.g., an *Origin* will not need an action `entersStation`).
4. listing all the track segments of the itineraries: when an itinerary is reserved all the track segments that compose it have to be blocked in order to achieve mutual exclusion; in order to do this in the `PDDL+` encoding all the track segments have to be explicitly stated (an example of this grounding can be seen in Figure 5.3a.

5.3 Fluents to model the flow of time

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

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

The fluents from 3 to 6 are given by the machine learning model which, based on the features of the train, predicts a duration of the action by the train. All these fluents are already known in the preprocessing phase and so are automatically grounded, leaving their integer value in the final encoding.

5.4 Actions, events and processes to model the movement of trains

In this section, we focus on the PDDL+ structures used to model the movement of trains per type. For the sake of readability, we say that a Boolean predicate is activated (de-activated) when its

```
(:action entersStation
  :parameters(
    T1 - train
    W+ - entryPoint
    I01-4R itinerary
  )
  :precondition (and
    (trainIsAtEP T1 W+)
    (not (trainHasExited T1))
    (not (trainHasEntered T1))
    (not (trackSegBlocked aa))
    ...
    (not (trackSegBlocked ba))
    )
  :effect (and
    (not (trainIsAtEP T1 W+))
    (itineraryIsReserved I01-4R)
    (trainInItinerary T1 I01-4R)
    (trainHasEnteredStation T1)
    (trackSegBlocked aa)
    ...
    (trackSegBlocked ba)
  )
)
```

(a) Action `entersStation`

```
(:process incrementTime
  :parameters()
  :precondition ()
  :effect (and
    (increase time  #t )
  )
)

(:event arrivesAtEntryPoint
  :parameters (
    T1 - train
    W+ - entryPoint
  )
  :precondition (and
    (>= time 242)
    (not (trainHasEnteredStation T1))
    (trainEntersFromEntryPoint T1 W+)
  )
  :effect (and
    (trainIsAtEntryPoint T1 W+)
    (trainHasArrivedAtStation T1)
  )
)
```

(b) Process `incrementTime` and event `arrivesAtEntryPoint`

Figure 5.3: The grounded action `entersStation` which enable a train `T1` to enter from an entry-point W^+ through itinerary `01-4R`. (right) The process `incrementTime` which keep track of the flowing of time and the grounded event `arrivesAtEntryPoint` which signals that a train `T1` has reached the endpoint W^+

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 (Figure 5.3b) encodes an explicit notion of passing time by increasing the fluent time
- An event arrivesAtEntryPoint(t,e^+) (Figure 5.3b) is triggered when the fluent time reaches the value of the predicate arrivalTime(t), and it requires that the predicate trainEntersFromEntryPoint(t,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,δ^+,i) (Figure 5.3a) can be used by the planning en-

```
(:event completeTrackSeg
  :parameters(
    T1 - train
    aa - trackSeg
    ac - trackSeg
    I01-4R itinerary
  )
  :precondition (and
    (>= (timeReservedIt I01-4R) 29)
    (trainInItinerary T1 I01-4R)
    (trackSegBlocked aa)
    (trackSegBlocked ac)
  )
  :effect (and
    (not (trackSegBlocked aa))
    (not (trackSegBlocked ac))
  )
)
```

(a) Event completeTrackSeg

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

(:process increaseReservedTimeIt
  :parameters(I01-4R - itinerary)
  :precondition (
    itineraryIsReserved I01-4R
  )
  :effect (
    increase (reservedTime I01-4R) #t)
  )
)
```

(b) Processes increaseTrainStopTime and increaseReservedTimeIt

Figure 5.4: (left) Grounded event completeTrackSeg which signals that a train T1 has finished his run through the track segments aa and ac. (right) The process increaseReservedTimeIt which keeps track of how long a train is reserving itinerary I01-4R and the process increaseTrainStopTime which counts the time passed for a train T1 which is stopping at a platform.

gine to let the train t enter the station from the entry-point δ^+, 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)` (Figure 5.5b) keeps track of how many seconds have passed since the reservation of an itinerary i by any train, updating the fluent `reservedTime(i)` accordingly.
- The event `completesTrackSeg(t,s,i)` (Figure 5.4a) is triggered when a train t has left the track segment s. This is triggered when `reservedTime(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 `re-`

```
(:action beginStop
  :parameters(
    T1 - train
    I04-3L - itinerary
    SIII - platform
  )
  :precondition (and
    (trainHasCompletedIt T1 I04-3L)
    (not (stopIsOccupied SIII))
    (not (trainIsStopping T1))
  )
  :effect (and
    (trainIsStoppingAtStop T1 SIII)
    (trainIsStopping T1)
    (assign (trainStopTime T1) 0 )
    (stopIsOccupied SIII)
    (not (itineraryIsReserved I04-3L))
    (not (trackSegBlocked aa))
    ...
    (not (trackSegBlocked am))
  )
)
```

(a) Action `beginStop`

```
(:event endStop
  :parameters(
    T1 - train
    S_III - platform
  )
  :precondition (and
    (>= (trainStopTime T1) 300)
    (>= time 421)
    (trainIsStoppingAtStop T1 SIII)
    (stopIsOccupied SIII)
  )
  :effect (and
    (not (trainIsStoppingAtStop T1 SIII))
    (not (trainIsStopping T1))
    (trainHasStoppedAtStop T1 SIII)
    (trainHasStopped T1)
    (not (stopIsOccupied SIII))
  )
)
```

(b) Process `increaseTrainStopTime` and event `endStop`

Figure 5.5: (left) The grounded action `beginStop` which allows the train to stop at platform `SIII` coming from itinerary `I04-3L`. (right) Ground event `endStop` which, when the time has come, signals that the stop has come to an end.

servedTime(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,$\mathrm{i_n}$,$\mathrm{i_m}$) 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,$\mathrm{i_n}$) is active and all the track segments $s \in i_n$ are not occupied by another train.

- A process incrementOverlapTime(t,$\mathrm{i_n}$,$\mathrm{i_m}$) 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,$\mathrm{i_n}$,$\mathrm{i_m}$).

- An event endsOverlap(t,$\mathrm{i_n}$,$\mathrm{i_m}$) is triggered when the fluent that counts the time of overlap timeElapsedOverlapping(t,$\mathrm{i_n}$,$\mathrm{i_m}$) is greater than the fluent timeToOverlap(t,$\mathrm{i_n}$,$\mathrm{i_m}$). The event signals that the train is no longer on itinerary i_n deactivating the predicates trainInItinerary(t,$\mathrm{i_n}$) and resetting the value of function timeToRunItinerary($\mathrm{i_n}$) 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 δ^-, the action exitsStation(t, i, $\mathrm{e^-}$) allows the train to exit the station activating the predicate trainHasExitedStation(t).

The goal for a train t of type *Transit* is to reach the state in which the predicate trainHasExited(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 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) (Figure 5.5a) 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) (Figure 5.5b) 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)` (Figure 5.5b) 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)`.

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 subsequently connected itinerary. 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 `timetableDepartureTime(t)`.

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

Destination train. A train of type *Destination* behaves like a *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 as its goal the need to reach a state in which the predicate `trainIsStopping(t)` is active, meaning so that the train t has reached its stop.

The interested reader can find the whole `PDDL+` problem and domain definition for instances of different complexities at `https://github.com/matteocarde/icaps2021`.

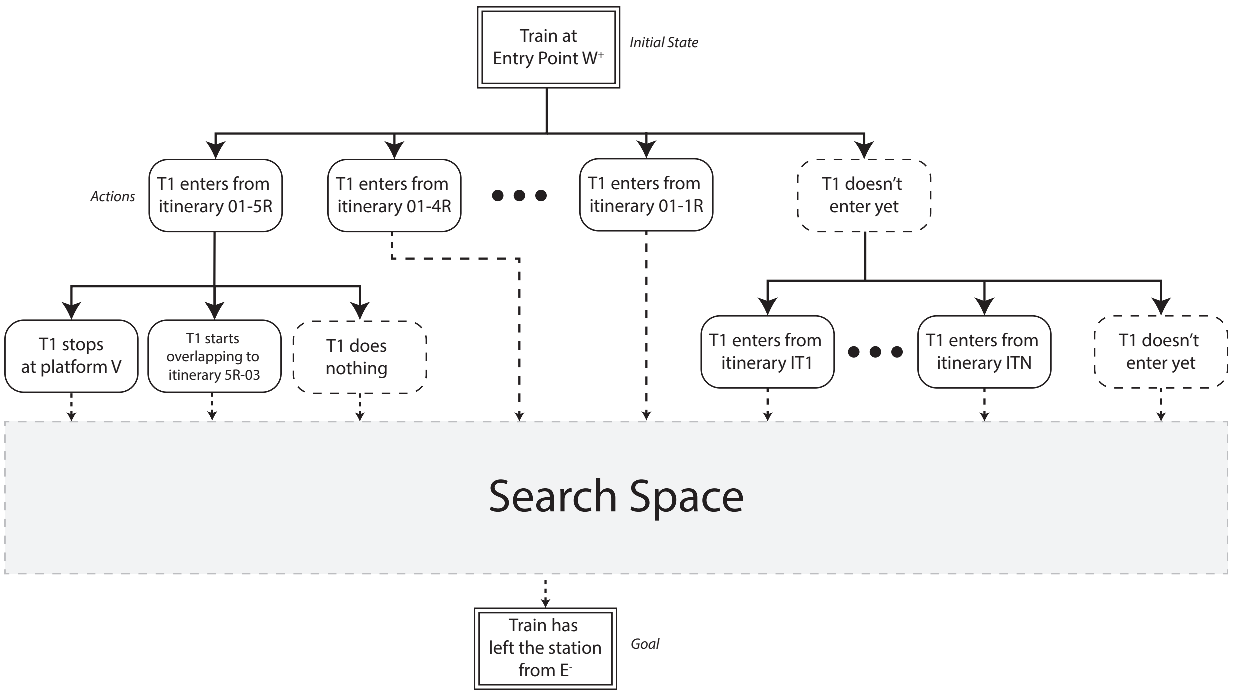
Chapter 6

Solving

The open-source[1] ENHSP planning engine [Scala et al., 2016, Scala et al., 2020] has been used to solve in-station train dispatching problems, encoded in `PDDL+`. ENHSP is a modular planning engine, and includes a range of off-the-shelf search and heuristic techniques; it deals with continuous processes using the Discretise and Validate approach, where the continuous model is initially discretised, then solved, and finally the found solution is validated against the original continuous model. Discretisation is done on the basis of a given delta, that controls the execution, planning, and validation processes of the planning engine. The delta for execution and planning is used to define, respectively, how often the planning engine is updating the state of the world checking for events and processes, and how often the planning engine is allowed to take a planning decision. In Section 6.1 a pseudo-algorithm is introduced that shows how the ENHSP planning engines solves `PDDL+` planning problems; afterwords a brief description of domain-independent heuristics are introduced. The off-the-shelf version of ENHSP, with domain-independent heuristics and with a Discretize and Validate approach, proved to be capable of solving prototypical instances, hence demonstrating the feasibility of the approach. To allow ENHSP to solve large and complex `PDDL+` in-station train dispatching problems, leveraging its modularity, the behaviour of the solver was specialised with domain-specific extensions which leveraged the prior knowledge of the domain in order to guide the search through the search-space. These extensions are presented in Section 6.2

In the next chapter, an analysis on the contributions made by every modification will be analysed and compared.

[1]available at `https://gitlab.com/enricos83/ENHSP-Public`



Train at Entry Point W+
Initial State
Actions
T1 enters from itinerary 01-5R
T1 enters from itinerary 01-4R
T1 enters from itinerary 01-1R
T1 doesn't enter yet
T1 stops at platform V
T1 starts overlapping to itinerary 5R-03
T1 does nothing
T1 enters from itinerary IT1
T1 enters from itinerary ITN
T1 doesn't enter yet
Search Space
Train has left the station from E-
Goal

Figure 6.1: An example of a search tree of a dispatching problem in which a train T1 enters from the Entry Point $\mathtt{W}^{+}$ and has to exit from the Exit Point $\mathtt{E}^{-}$. Rounded rectangles represent possible states following an action. States dashed border are a visual representation of the result of applying the *no-op* action in which no action is chosen.

6.1 Planning As Search

In this section, an algorithm is presented to provide some insights on how a solution for a planning problem is found in the state-of-the-art PDDL+ planner ENHSP. As previously described in Chapter 3, a plan consists of a sequence of (timestamped) actions that leads from the initial state to the goal state. Since at every possible stage of the plan multiple actions could be chosen, the most natural way to represent a search problem is introducing the concept of a *search tree*. In a search tree the root of the tree represents the initial state, nodes correspond to states in the state space of the problem and the arcs that connect two nodes represent an applicable action that brings from a state to another. The additional constructs of processes and events introduced in PDDL+ are not present in the search tree since they are not under the direct control of the planner. In Figure 6.1 a visual representation of a search tree is displayed.

The solution of a planning problem can thus be seen as an edge-path in the tree that connects an initial state to a goal state. The structure of a search tree provides with a native algorithm for searching for a solution: starting from the initial state *expand* the search tree by applying one after the other all the actions applicable in the state adding the new generated states in a list

Algorithm 1 Algorithm for finding a PDDL+ plan

```
1:  Input: Domain $\mathcal{D}$, Problem $\Pi$ and Discretization Step $\delta$
2:  Output: A plan that leads from the initial state to the goal state or $\emptyset$ if no viable plan can be found
3:  function FINDPLAN($\mathcal{D}, \Pi, \delta$)
4:      $Frontier \leftarrow$ COLLECTION()
5:      ADD($Frontier, \Pi.\mathcal{I}$)
6:      while ISNOTEMPTY($Frontier$) do
7:          $node \leftarrow$ CHOOSE($Frontier$)
8:          APPLYPROCESSES($node, \delta, \mathcal{D}.\mathcal{P}$)
9:          APPLYEVENTS($node, \mathcal{D}.\mathcal{E}$)
10:         if STATE($node$) $\models \Pi.\mathcal{G}$ then
11:             return EXTRACTPLAN($node$)
12:         for all $action \in$ APPLICABLEACTIONS($node, \mathcal{D}.\mathcal{A}$) $\cup \{NoOp\}$ do
13:             $expNode \leftarrow$ APPLYACTION($node, action$)
14:             ADD($Frontier, expNode$)
15:     return $\emptyset$
```

(called*frontier* or *open-list*), then *choose* in this list the next state to expand and repeat the process iteratively until a goal state is reached. This method of building a search tree is called *forward (progression) state-space search* solution. This algorithm also implicitly provides a procedure for *back-tracking*, when no actions are applicable in a particular state, simply by picking a state from the frontier which was the result of previous expansions.

In Algorithm 1 a pseudo-code is presented for the FINDPLAN() procedure extended with structure to manage PDDL+ constructs like processes and events. The functions APPLYPROCESSES() and APPLYEVENTS() check between all the processes of the domain $\mathcal{D}.\mathcal{P}$ and events $\mathcal{D}.\mathcal{E}$ which preconditions are met and apply the effects to $node$; the two functions are to be called in this precise order since APPLYPROCESSES() changes the value of all the numeric functions substituting the value of `#t` with the value expressed in δ and then APPLYEVENTS() checks the preconditions of the events based on the updated numeric functions' values. APPLYACTION() instead, does not change the state of $node$ but creates a new state $expNode$ which is the result of the expansion with $action$ picked from the set of APPLICABLEACTIONS() in $node$. Since it is possible that it is more convenient to not apply any action and wait for events to be triggered, a special action $NoOp$ is added to the list of applicable actions which does not change the state of $node$. When a node is found that entails the goal the search algorithm terminates by calling the function EXTRACTPLAN() on the node which reached the goal node; in fact, since possible backtracks could jump between states in an unordered fashion a simple and fast way to store plans is to keep a parenthood relation between nodes and then extract the plan by looking iteratively at all the parents of the goal reaching node.

The most generic function of Algorithm 1, which actually defines the search strategy of the algorithm, is CHOOSE(). Different implementations of this function can produce different plans with very different planning times. The search strategies are mainly classified in two categories: *uninformed search* (also called blind search) and *informed search* (or heuristic search). In uninformed search strategies, no additional information is provided beyond that provided in the problem definition and the algorithm simply expands nodes with no knowledge if the expanded nodes contribute to reaching the final goal or not. In informed search, instead, the search algorithm tries to pick nodes that look "more promising" in reaching a solution faster.

Examples of uninformed search algorithms are the Breadth-First Search (BFS) or Depth-First Search (DFS). These algorithms differ simply from the choice of which collection to use to represent the frontier. For example, if the frontier is represented by a Queue then the CHOOSE() function implements the functionality of DEQUEUE() and will always retrieve the node with the lowest level, thus implementing a BFS approach. If instead the frontier is represented with a Stack, and CHOOSE() is equivalent to POP(), then a DFS strategy is performed. These strategies work best with a small search-space and with a small branching factor, but when dealing with huge search-spaces a more intelligent approach is need to guide the planner towards a goal.

For this reason, heuristics have been developed in order to provide a measure on the *quality* of the node that estimates how much the node is "near" to a goal state. This quality is specified using an *evaluation function* $f(n)$ that maps a state to a real number, with lower values of $f(n)$ representing states with better quality. The algorithm will then choose the highest quality nodes first (for example, implementing the frontier with a PriorityQueue ordered on the value of $f(n)$). Of course, heuristics do not provide perfect estimations of the distance to the goal states (otherwise the implementation of heuristics will always produce a plan at the first try), but can reduce drastically the computation times avoiding spending a lot of time blindly expanding states which are less probable to lead to a goal state. The heuristics can be of two types:

1. *Domain Independent Heuristics* which try to fetch information about the distance from the goal by simply looking at the structure of the problem. In general heuristic functions are computed in parallel search trees created from *relaxed problems* which are problems with fewer restrictions on the actions. In the literature, several relaxations are presented which are able to provide huge boosts to the discovery of plans. For classical planning, one of the most famous and pivotal algorithms is GRAPHPLAN() [Blum and Furst, 1997]. For PDDL+ the interested reader is referred to [Scala et al., 2016, Piotrowski et al., 2016]

2. *Domain Dependent Heuristics* which instead introduce some additional knowledge on the domain which cannot be deduced from the planning problem alone. For example in the in-station train dispatching problem, based on the prior knowledge of the structure of the station and on the interconnections between track segments, an itinerary can be preferred to another since it can lead to an exit portal quicker than the others. This type of heuristic has been successfully implemented in the solver of the in-station train dispatching problem

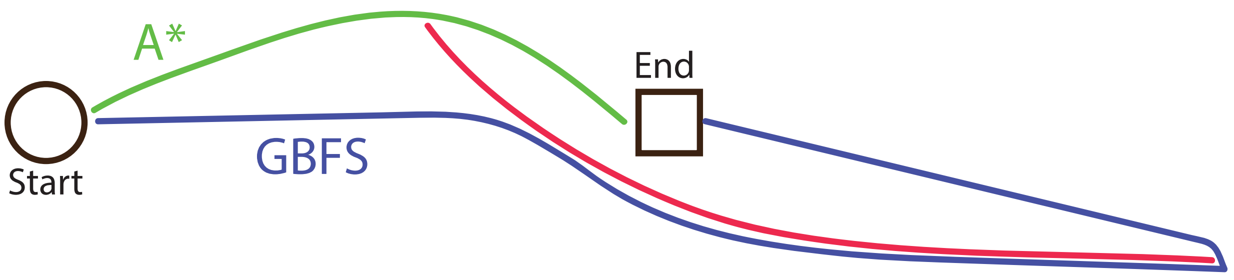


Figure 6.2: Example on how two different informed search algorithms, A^* and Greedy Best First Search (GBFS), find plans in a euclidean space

and will be covered in the next section.

Two well-known informed search algorithms are the Greedy Best-First Search (GBFS) and the A^* algorithm. The GBFS approach simply picks the nodes which look more promising and nearest to the goal (i.e., with the lowest $f(n)$) in a greedy fashion. This algorithm, like all greedy algorithms, can lead to a solution very fast but usually produces longer plans since it doesn't take into account the number of already expanded nodes. In Figure 6.2 a visual explanation on how GBFS (Blue) deals with obstacles is presented: suppose a planning agent has to find a plan of moving from the circle to the square, which lies behind the red wall. A heuristic is provided which is simply the euclidean distance between the square (goal) and the current position of the agent. The GBFS algorithm will quickly move in the direction of the square, but as soon as it tumbles against the wall it will try to bypass it in a direction which will actually make the path longer. In the A^* algorithm instead, the number of steps taken are considered while choosing the next node to expand. The evaluation function is in fact computed as $f(n) = g(n) + h(n)$ where $g(n)$ is the number of steps taken from the initial state to the node n and $h(n)$ is the heuristic (or estimated cost) of going from the node n to the goal state. In this way, as presented in Figure 6.2, the A^* algorithm will actually prefer a path that is initially longer, but that will produce a plan in fewer steps.

6.2 Domain-Specific Improvements

To allow ENHSP to solve large and complex `PDDL+` in-station train dispatching problems, leveraging its modularity, the behaviour of the solver was specialised in three ways: through the design of an adaptive notion of execution and planning delta (Section 6.2.1), the introduction of a domain-specific specialized heuristic (Section 6.2.2), and the insertion of three constraints to prune unpromising areas of the search-space (Section 6.2.3)

Algorithm 2 Adaptive Delta Queue

```
1:  function GenerateInitialQueue($E, A, I$)
2:      $Q$ = InitQueue()
3:      for $e$ in $E \mid e = "AtEntryPoint"$ do
4:          enqueue(Q, $e.time$)
5:      for $a$ in $A \mid a = "LeaveOrigin"$ do
6:          enqueue(Q, $a.earlyTime$)
7:      for $p$ in getActiveProcesses($I$) do
8:          for $e$ in identifyEvents($p$) do
9:              enqueue($e.time$)
10:     return sort($Q$)
11:
12: function ExtendQueue($Q_p, a$)
13:     $Q = Q_p$
14:     $TE$ = calcTrigEvents($a$)
15:     for $e$ in $TE$ do
16:         enqueue($e.time$)
17:     return sort($Q$)
```

6.2.1 Adaptive Delta

As discussed in Chapter 3 in order to manage continuous time-dependent processes, the planner has to discretise time in multiple discretisation steps (δ) that determines how often an action can be applied, or an event can be triggered. The value of δ can be given to the solver by the AI expert based on the knowledge on the domain and the granularity by which the solution has to be found. The choice of the correct discretisation step can be delicate; the choice of a very small δ can exponentially increase the solving time, since at every discretisation step several actions could be applied, enlarging the search-space. On the other hand, the selection of a large δ can invalidate the solution. In the proposed `PDDL+` model, it is possible to know a priori when events will be triggered. For *Approaching* trains, the moment in which the *At EntryPoint* event will be triggered is given in the ground formulation. All the other events are the result of actions executed by the planning engine, and the moment in which they will be triggered is only related to the time in which the corresponding action has been executed. This implies that there is no need to use a fixed execution and planning delta, but the delta value can be adjusted according to when the next event will be triggered, or an action will be available. From a planning perspective, it is useless to consider all the steps in between, because the state of the world does not change. This is similar in principle to the approach exploited by decision-epoch planning engines for dealing with temporal planning problems [Cushing et al., 2007].

An *Adaptive Delta Queue* was designed in order to manage this behaviour, that is initialised as shown in Algorithm 2. Taking into account the set of ground events E, ground actions A, and the initial state description I, it is possible to identify when to stop the execution to take decisions and update the state of the world. Given the proposed PDDL+ model, in the function GENERATEINITIALQUEUE() it is straightforward to extract the time at which an *At EntryPoint* event will be triggered, and the earliest time at which an action *Leave Origin* will be available. Similarly, it is straightforward to check the presence of trains already in the controlled station, and identify the time of the corresponding events, if any. When an action is executed, a new queue Q is created, and attached to the resulting search state. The function EXTENDQUEUE(), shown in Algorithm 2, is used to extend the delta queue of the parent state Q_p taking into account the events TE that will be triggered by the applied action a, and their trigger time.

In each state, the next delta step is identified by considering the corresponding delta queue, and by picking up the time of the next event that will be triggered. Between two events, nothing will happen, so there is no need to generate and assess additional states.

6.2.2 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 a 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:

$$h(z) = \sum_{t \in T(z)} \rho_t(z) + \pi_t(z) \tag{6.1}$$

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:

$$\rho_t(z) = \max_{R \in \mathcal{R}_t(z)} \sum_{i_n \in R} \texttt{timeToRunItinerary(t, i}_\texttt{n}\texttt{)} \tag{6.2}$$

where $\mathcal{R}_t(z)$ 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 function $\pi_t(z)$ 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.

6.2.3 Constraints

Finally, the implemented constraints focus on the time spent by the trains. In particular, for every train t of a given problem to solve, the following constraints are enforced.

$$\texttt{stayInStationTime}(t) < MaxStayInStation \tag{6.3}$$

$$\texttt{fromArrivalTime}(t) < MaxFromArrival \tag{6.4}$$

$$\texttt{stoppingTime}(t) < MaxStoppingTime \tag{6.5}$$

Equation 6.3 indicates that a train is not allowed to stay in the station more than a given maximum value. Similarly, Equations 6.4 and 6.5 constraint, respectively, the time passed from the arrival of the train in the station, and the time spent stopping at a platform. The idea behind such constraints is to avoid 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. Such constraints are also used to implement an anytime planning framework, able to generate a solution of increasing quality over time. Starting from an initial value corresponding to worst case scenarios observed in historical data, the constraints' value are then reduced if a plan is generated: this process is repeated until either the cut-off time is reached, or the planning engine returns that no solution can be found.

Chapter 7

Evaluation and Analysis

In this chapter, an analysis of the performances and an evaluation of the capacity of the proposed approach to deal with real problems are presented. Firstly, in Section 7.1 a visualisation tool is introduced which enables to inspect the topology of the station, debug the produced plans and visualise the movements of trains inside the station. Then, in Section 7.2 a double validation is performed by having human dispatchers validate the plans produce by the automatic planner and then having the automatic planner validate the plans produced by human dispatchers in the past, thus showing that the proposed AI dispatcher could have taken the same decision a human operator would have, given the same initial conditions. Then, in Section 7.3, an analysis is presented to evaluate the capacity of the proposed approach to choose plans that would minimise the total delay of trains in stations. In Section 7.4, the automatic planning approach is used to simulate a stress-test in which more and more trains are scheduled to pass through the station; this simulation provides an evaluation of the limit on how much of the railway station's available infrastructure can be exploited and the maximum number of trains over which a renovation of the station, with the insertion of additional tracks, is required in order to avoid the station to become a bottle-neck for the whole railway network. Lastly, in Section 7.5, an analysis of the improvements and domain-specific extensions introduced in Chapter 6 is presented to see whether and to what extents they are beneficial in terms of planning time.

All the tests were run on a 2.5GHz Intel Core i7 Quad-processors laptop with 16 GB of memory made available and a macOS operating system. The cut-off time was set to 5 CPU-time minutes, but plans are usually generated in less than 30 seconds. These specifications, which nowadays is common in most computers, show that the proposed approach can run on a simple laptop and without the need of a great computing power.

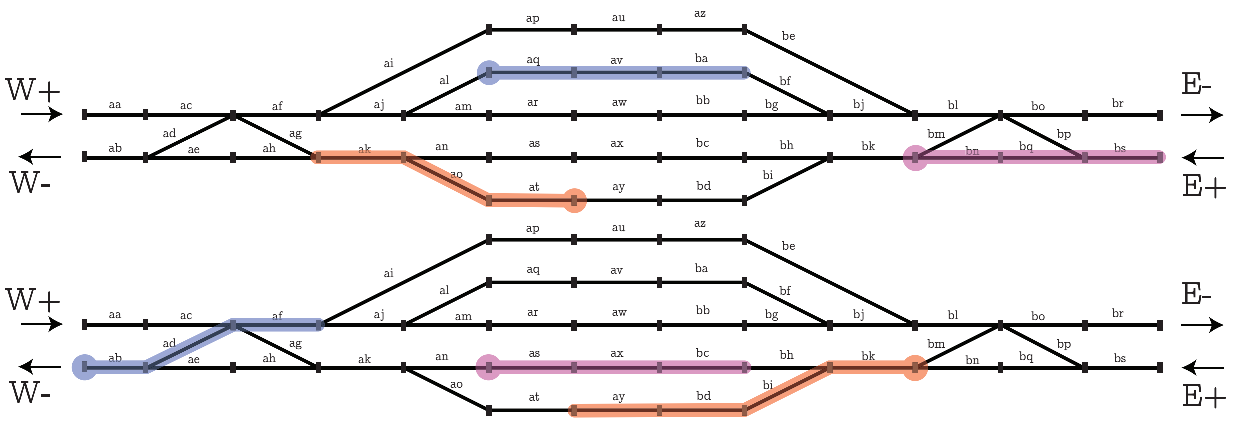


Figure 7.1: Two screenshots from the visualisation tool capturing the itineraries occupied by three trains at different time steps. A train is represented by a coloured line, the "head" of a train is represented with a circle to indicate the direction of the train. Top: The red train is moving from the West entry point to platform `V`, the blue train is stopped at platform `II` and the purple train is arriving from the East entry point. Bottom: The red train has stopped at the platform and is moving towards the East exit point, the blue is leaving the station from the West exit point. The purple train has stopped at platform `IV`

7.1 Visualisation

As presented in Section 4.1 the CAD file of the station was of particular use in order to understand the movement of the trains inside the station and to debug the proposed approach discussed in the previous chapters. For this reason, a visualisation tool was constructed using the state-of-the-art technology for the visualisation of data: the framework D3[1]. The visualisation tool build has the following functionalities:

1. Search the components (i.e., track segments, itineraries, flags, platforms) inside the station via a search-bar, highlighting the component on the map.
2. Highlight the itineraries and see the path of track segments it contains.
3. Show the movements of the trains inside the station by highlighting the occupations of itineraries described in the logs in order to better visualise how a train moves and how the movements of multiple trains are coordinated.
4. Display the presence of rolling stocks simultaneously present in the station by visualising the occupancy of single track segments with the aim of analysing how they interact with each other.

[1]Data-Driven Documents `https://d3js.org/`

5. Display the result of the AI planned movements inside the station (see Chapter 5) in order to debug and assess the accuracy of the plans. In Figure 7.1, two screenshots from the visualisation tool are presented which shows the movements of three trains as planned from the planning engine.

7.2 Validation Against Historical Data

Validation consists in analysing, inspecting and debugging the plans produced by an AI planning engine to understand if the proposed approach can successfully model real-world instances of the in-station train dispatching problem. This validation was performed in two ways: a manual inspection and an automatic one. For the manual inspection, the visualisation tool presented in the previous section was utilized for showing the plan to experts of the domain for the purpose of understanding if the planned movements of the train inside the station are correct; then special instances, known to be difficult to manage by human dispatchers, were constructed ad-hoc and the resulted plan inspected and validated.

A second approach was to be able to automatically validate the proposed approach through the analysis of historical plans produced by human operators. The logs provided by RFI, and presented in Chapter 4, allows to reconstruct the plan of the trains chosen by human operators. This plan, together with the `PDDL+` domain and problem, is then given as input to the ENHSP internal validator, which answer the question "Would this plan be a solution of this problem and domain ?" or, in other words, if the `PDDL+` formulation is capable of representing how the addressed in-station train dispatching problem is currently faced by the human operators. This step is fundamental for at least two reasons: (i) to ensure that the `PDDL+` formulation is realistic and can capture all the nuances of the real-world application, and (ii) to support the use of historical plans as baseline for validation and comparison purposes.

To perform this analysis, a day was selected – 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 operators. The recorded happenings of that day were recorded under the form of a single `PDDL+` plan, using the operators introduced in the corresponding section. These plans were then successfully validated, using the ENHSP validator, against the proposed `PDDL+` model. Notably, the fact that the `PDDL+` model can correctly model the real-world dynamics, implies that planning-based tools can be straightforwardly exploited, and the planning engine can provide an encompassing framework for comparing different strategies to deal with recurrent issues, and for testing new train dispatching solutions. This result already represents a significant leap forward for the state of the art of the application field.

In Algorithm 3 a pseudo-code of the internal ENHSP validator, which was used to perform this analysis, is presented. The function VALIDATE() takes as input the discretization step δ, the

PDDL+ Domain and Problem together with the plan (which in this case is the plan chosen by human operators, extracted from the logs); At the beginning, a variable $state$ is initialized with the initial conditions of the problem. Until the plan is not finished, the algorithm increments the time of the state by the discretisation step and applies all the appliable processes in order to bring forward all the time-dependent fluents (i.e., the time of occupation of an itinerary, the stop time, etc). Then, the events are applied calling APPLYEVENTSUNTILFIXPOINT(), which applies all events until no more events are applicable, this because cascading events have to be taken into

Algorithm 3 Algorithm for validating a PDDL+ plan

```
 1: Input: Domain $\mathcal{D}$, Problem $\Pi$, Plan $\mathcal{S}$ and a Discretisation Step $\delta$
 2: Output: True if the plan is valid, False otherwise
 3: function VALIDATE($\mathcal{D}, \Pi, \mathcal{S}, \delta$)
 4:     $state \leftarrow \Pi.\mathcal{I}$
 5:     while ISNOTEMPTY($\mathcal{S}$) do
 6:         $state.time \leftarrow state.time + \delta$
 7:         $state \leftarrow$ APPLYPROCESSES($state, \mathcal{D}.\mathcal{P}$)
 8:         $state \leftarrow$ APPLYEVENTSUNTILFIXPOINT($state, \mathcal{D}.\mathcal{E}$)
 9:         $nextAction \leftarrow$ PEEK($S$)
10:         if TIME($nextAction$) $> t$ then
11:             continue
12:         if PRECONDITIONS($nextAction$) $\not\models state$ then
13:             return false
14:         $state \leftarrow$ APPLYACTION($state, nextAction$)
15:         POP($\mathcal{S}$)
16:     if $state \models \Pi.\mathcal{G}$ then
17:         return true
18:     else
19:         return false
20:
21: Input: $state$ in which the event should be applied and all the events in the domain $\mathcal{E}$
22: Output: a new state in which all the applicable events have been applied
23: function APPLYEVENTSUNTILFIXPOINT($state, \mathcal{E}$)
24:     $newState \leftarrow state$
25:     for all $e \in$ APPLICABLEEVENTS($newState, \mathcal{E}$) do
26:         $newState \leftarrow$ APPLYEVENT($state, e$)
27:     if $newState \neq state$ then
28:         return APPLYEVENTSUNTILFIXPOINT($newState, \mathcal{E}$)
29:     else
30:         return $newState$
```

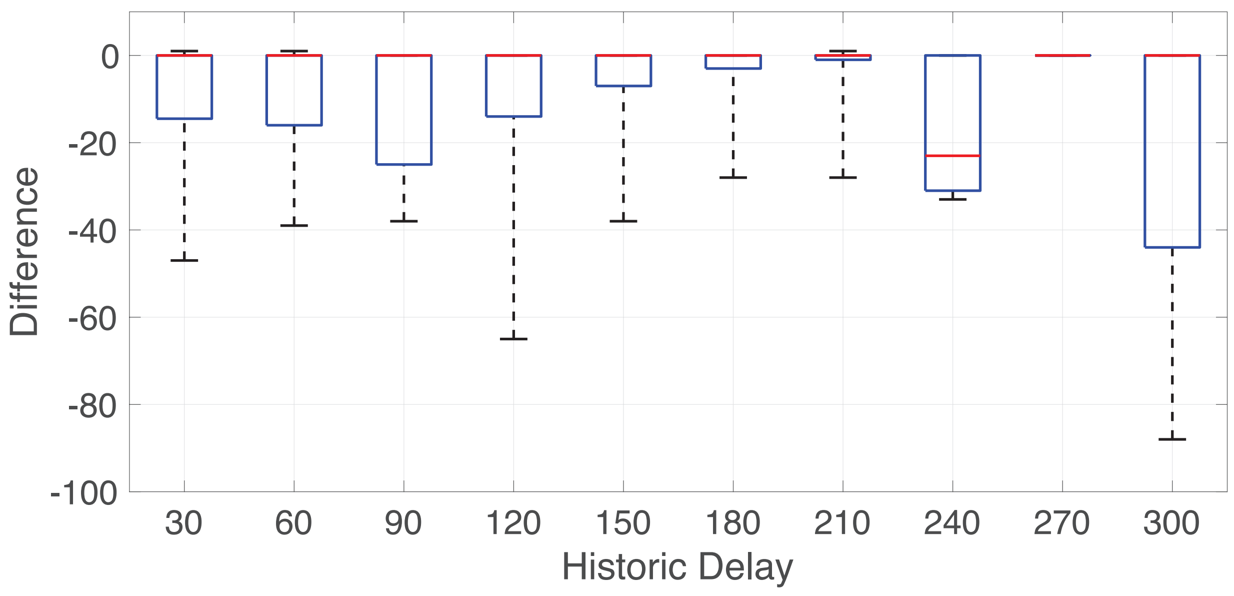


Figure 7.2: A box and whisker representation of the delay reduction achieved by our approach (y-axis), with regard to the recorded historical delay (x-axis). Whiskers refer to the highest and lowest reduction in delay, while the box (in blue) indicates the first, second, and third quartiles. Red indicates the median value. Negative y-axis values indicate that the planning approach reduced the delay, with regard to historical records.

account, in which the effect of one event make the preconditions true of another event. Then, as soon the time of the state reaches the time of the next action, its preconditions are tested against the current state; if the precondition cannot be applied, then it means that the plan is not valid since that action could have not been taken at that moment, otherwise the effects of the action are applied, and the state is updated with the result. After checking all the preconditions of actions in the plan, the goal has to be tested for the validity of the plan.

7.3 Minimisation of Delays

The next step is to focus on the benefits that the proposed `PDDL+`-based approach can deliver when dealing with delayed trains. Here, two different sets of experiments were performed: firstly, it was assessed how the plans generated with the proposed technique compares with the strategies currently implemented in the considered train station, in order to assess the improvement over the state of the art; secondly, an extensive analysis was performed aimed at quantifying the ability of the approach to cope with increasingly large, widespread delay. In order to deal with the first aspect, the data collected between January and March, prior to the COVID-19 related lockdown, was examined and all cases where at least one train was delayed with regard to the official timetable were identified. For the planning scenario, a time window centred on the delayed train(s) and bounded by the first time in which no trains are in movement in the controlled station was considered. What fell inside such time window was encoded in the considered planning problem. This allows to consider the entirety of the context in which the delayed trains have to operate, thus maximising the comparability of the results. In total, 311 scenarios, including one or more delayed trains, were identified using the described technique. Overall results are presented in Figure 7.2, where scenarios are clustered, in bins of 30 seconds, according to the average recorded delay (x-axis). The clusters' size vary between 2 (300 seconds delay) and 59 (30 seconds delay on average) instances; the number of trains per instance ranges between 1 and 12. Whiskers refer to the highest and lowest reduction in delay from the relative clustered recorded delay, while the box indicates the first, second, and third quartiles. Red is used to represent the median value. A value of 0 indicates that our approach performed exactly as the human operator in the recorded historical data; negative values on the y-axis indicate that we improved over the historical record. Results indicate that the PDDL+-based approach is never worse than the current strategy exploited at the controlled railway station, and that instead it is usually able to reduce the average delay.

For instance, taking the clustered bin of 90 seconds, i.e., historical cases where trains were delayed by 90 seconds, the proposed approach reduced the delay by up to 40 seconds. This is remarkable if it is also taken into account that the delay can be reduced only for trains that terminate or do an intermediate stop at the controlled station: there is no way to reduce the delay of trains that originate at the station – as they leave the platform late. By looking at the generated plans, the intuition is that the delay reduction is due to a better use of the available infrastructure, leading to shorter waiting times for trains entering the station.

To quantify the ability of the proposed approach to handle delays, the following peak hours time slots for the controlled railway station were considered: (i) 06.30-09.30, with 24 moving trains, (ii) 12.00-14.30 with 17 trains, and (iii) 17.00-20.30 with 29 trains. Time slots (i) and (iii) include the commuters trains, while (ii) has numerous trains used by students going home from school. Focusing on these three time slots, a day was selected, before the COVID-19 travel restrictions were put in operation, with the minimum mean squared deviation of recorded timings from the official timetable to be sure to have a levelled ground for performing the analysis. After that, every train was delayed of the considered time slot using a Gaussian-distributed delay $N(\mu, \sigma)$ with an increasing mean μ and a fixed standard deviation σ of 5 minutes. μ ranged from -10 to $+30$ minutes, with 5 minutes steps; in the analysis, a negative delay is used to model a train that is early with regard to the official timetable. For each value of μ, 10 planning instances were randomly generated using a Monte Carlo approach to inject delays to all the trains of the instance. Average results over the 10 instances are then considered. Figure 7.3 shows the achieved performance, as the relationship between the injected delay and the final observed average delay, on the time slot 17.00-20.30. Results on the other time slots are analogous. The behaviour can be interpolated by a straight line with an angular coefficient of 0.72; in other words, the proposed approach is able to plan the movement of trains absorbing 28% of the injected delay.

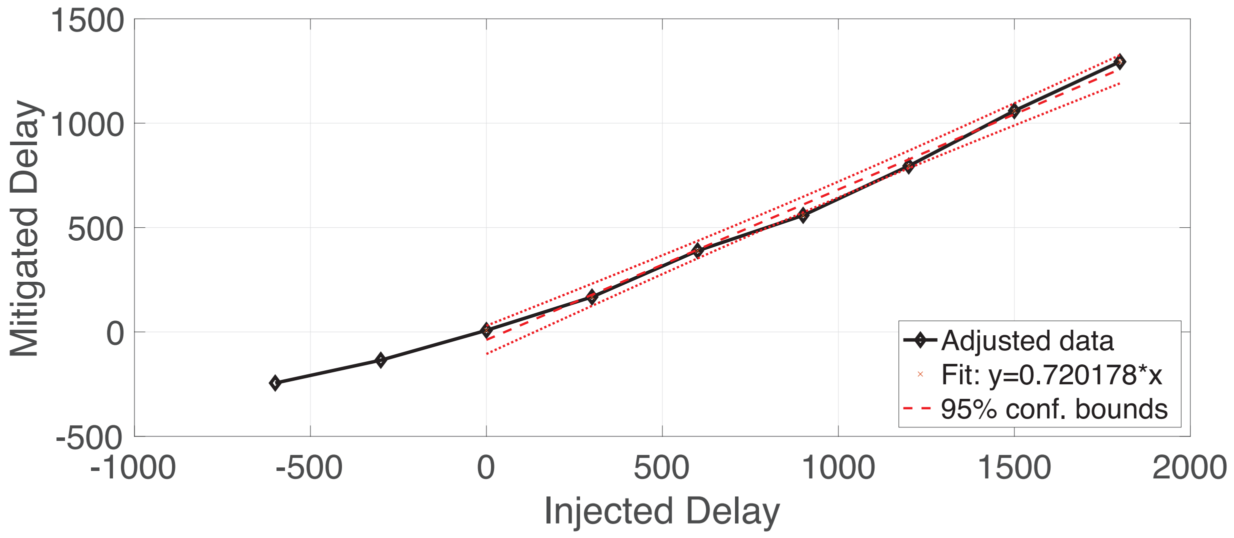


Figure 7.3: The relation between the injected Gaussian delay (x-axis) and the average mitigated delay obtained by exploiting the generated plans (y-axis), when considering the evening peak hours (17.00-20.30) on the controlled station.

For example, considering an injected Gaussian delay of 500 seconds, the final measured average delay is of 350 seconds: the proposed approach is able to absorb almost two minutes of the initial delay. These results are of particular relevance since in the considered scenarios all the trains are delayed (which is a worst-case scenario) and the approach is still capable of reducing the average delay by almost a third.

7.4 Increment of the Railway Station Capacity

For this scenario, a stress test was conducted with the aim of understanding if the proposed approach can lead to an increment of the railway station capacity. Using the same settings exploited for the previous scenario, an increasingly large number of synthetically generated trains was added as follows: (i) considering historical data, a train is selected that originates, terminates, or does an intermediate stop at the station, (ii) the arrival/departure time of this synthetically generated train is then scheduled in between of existing trains in the time slot, (iii) the already discussed (in the previous section) Gaussian distributed delay is injected, (iv) the instance is tested and the relationship between the injected delay and the final observed average delay is logged. This process is repeated ten times. Afterwords, another train is synthetically generated and added, restarting the aforementioned procedure.

The results of this set of experiments are presented in Figure 7.4. Surprisingly, results presented

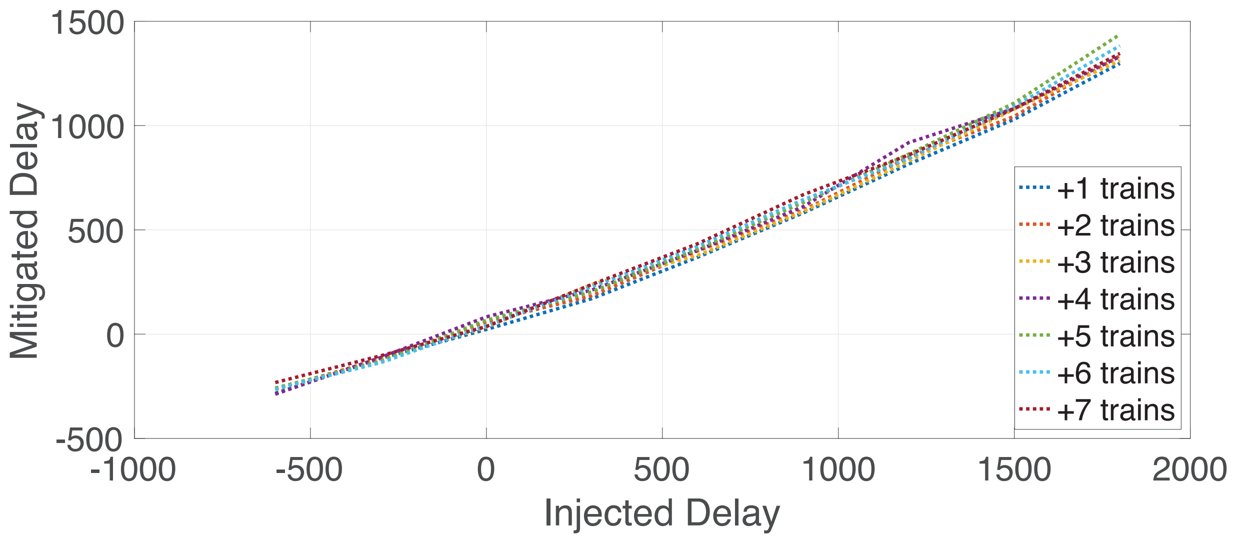


Figure 7.4: The relation between the injected Gaussian delay (x-axis) and the average mitigated delay obtained by exploiting the generated plans, in the presence of an increasingly large number of additional trains, when considering the evening peak hours (17.00-20.30) on the controlled railway station.

in Figure 7.4 are very similar to those shown in Figure 7.3: the fact that the station is serving up to 25% more trains does not result in a reduced capacity of the proposed approach of mitigating the injected delay. Consequently, the proposed approach seems to be very robust. When more than 7 trains are added in the considered time slot, the limited availability of entry/exit points and platforms makes it impossible to generate in-station dispatching plans that allows to satisfy even very loose constraints on the maximum time a train is allowed to stay in the station, or to wait at an entry point. Even a visual inspection performed by RFI experts, and some attempts to manually generate some reasonably timed dispatching plans, did not lead to the generation of any sensible solution. This suggests that the approach is able to exploit the available infrastructure up to a very high level, possibly close to the physical limit of the railway station.

7.5 Importance of the Domain-specific Extension

As presented in Chapter 6, some domain-specific extensions were introduced in the planning engine ENHSP to increase the capacity of the planner to deal a large amount of trains. A natural question that needs an answer is: what is the impact of the designed domain-specific extensions on ENHSP? For this purpose, the performance improvement that can be obtained by using every introduced domain-specific optimisations considering planning tasks involving an increasing number of trains to be controlled needs to be analysed. To perform this analysis, a day was

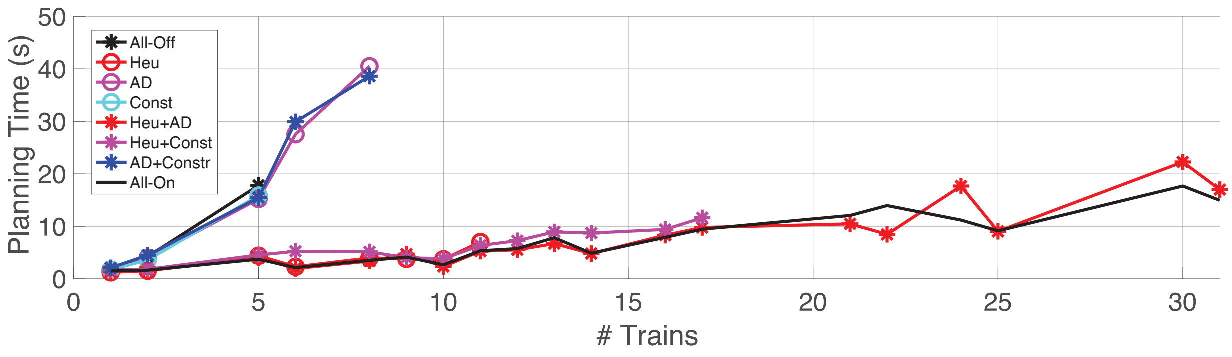


Figure 7.5: The CPU-time needed for planning instances with an increasing number of trains using different combinations of domain-specific optimisation techniques.

selected – 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. The peak hour of the day, $17{:}00$-$20{:}00$, when commuters return home was considered in which 31 trains move through the station. Different time windows of the considered 3-hours period were considered for the purpose of generating instances with an 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 7.5 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 thesis. 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.

According to the results in Figure 7.5, 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 waste 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 in which 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 regard to the shape of the generated solutions, 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.

Chapter 8

Related Work

In this chapter, an overview of other research work dealing with the in-station train dispatching problem is presented. Then, other problems in the railway domain are listed together with research works aimed at solving them. Lastly, some other domains which were solved with PDDL+ are described, showing that the language can also model a vast range of problems.

8.1 In-Station Train Dispatching

For what concerns in-station train dispatching, [Mannino and Mascis, 2009] introduced a mixed-integer linear programming (MILP) model for controlling a metro station. Their 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 the much larger and more complex railway stations. More recently, [Kumar et al., 2018] introduced a constraint programming model for performing in-station train dispatching in a large Indian terminal: this approach demonstrates to be able to deal with a large railway station, at the cost of considering very short time horizons (less than 10 minutes) and station-specific optimisations.

Given the complexity of the train dispatching problem, many works focused on related sub-problems or on a more abstract formulation of the overall problem. For example, [Rodriguez, 2007] formulated a constraint programming model for performing train scheduling at a junction, which shares some characteristics of a station, but does not include platforms and stops. Differently from [Rodriguez, 2007], a number of works [Cardillo and Mione, 1998, Billionnet, 2003, Chakroborty and Vikram, 2008] focused on the problem of assigning trains to available platforms, given the timetable and a set of operational constraints. Taking another perspective, [Caprara et al., 2010] focused on the identification and evaluation of recovery strategies in case of delays. These strategies include actions such as the use of different platforms or alternative paths. In [Li et al., 2021] a solution is presented to efficiently manage dense traffic on rail networks based on Multi-Agent Path Finding (MAPF) which can plan collision-free paths for thousands of trains, but doesn't take into account safety mechanism of mutual exclusion of resources or the need of a train to stop at platforms.

8.2 Line Dispatching

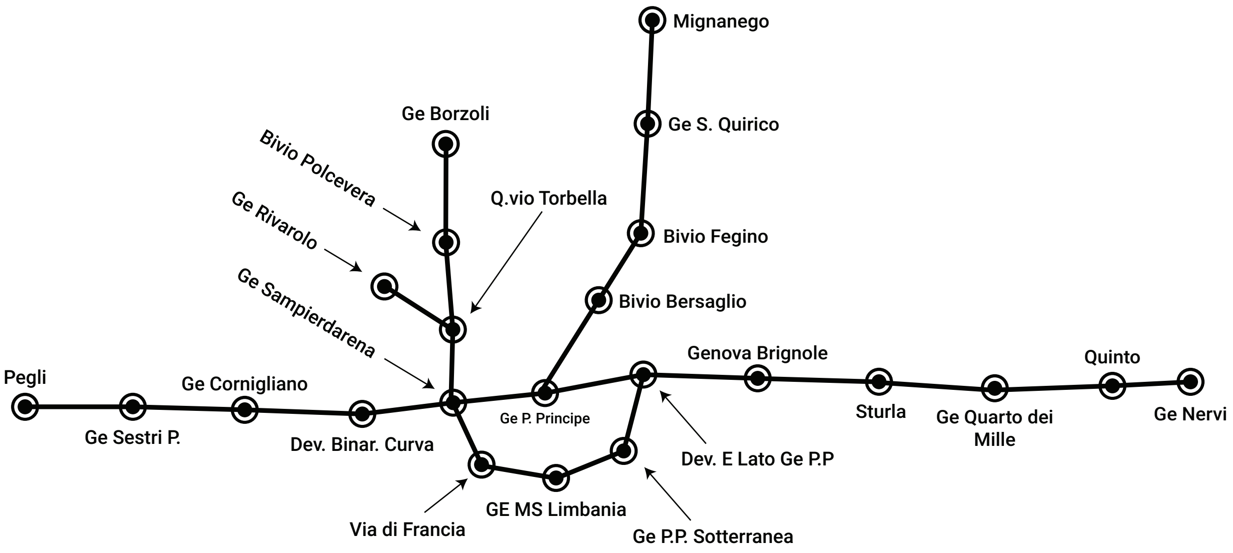


Figure 8.1: The graph representing the railway network around the Metropolitan City of Genova. Stations are nodes and edges represents connection between them.

Line dispatching (also called Line Planning Problem or Train Pathing) considers the overall railway network, or a subset of it (called railway nodes) and consists in establishing routes and precedence between trains in order to cope with normal operations but, also, to recover from deviations from the timetable and minimise the delays. In Figure 8.1 a graph illustrating the railway network around the Metropolitan City of Genova is shown. Stations are represented by nodes, and edges are railway tracks that connect them. As it can be seen by the figure, the majority of stations have only two entry points and two exit points (as the modelled station in Figure 2.1), for this reason, since the stations are connected by single tracks (one track for one direction and another for the opposite direction), overtaking can only happen inside a station. In line dispatching, a large amount of trains needs to move through the network respecting the timetable and the safety constraints.

With respect to line dispatching, [Lee and Chen, 2009] introduced a heuristic-based approach for tackling the problem of finding routes for trains while generating an overall timetable. [Böcker et al., 2001] explored the use of a multi-agent scheduling system, considering the railway transport system as a case study. On the topic of multi-agent, more recently [Atzmon et al., 2019] exploited multi-agent path finding to search for a route for a set of trains from a given origin point to a required individual destination. A different line of work takes advantage from the decomposition of the problem, where a master-slave algorithm is used to control the train traffic of large railway networks [Lamorgese and Mannino, 2013, Lamorgese and Mannino, 2015, Lamorgese et al., 2016].

8.3 Timetabling

In-Station Train Dispatching Problems and Line Dispatching Problems all concern moving trains inside a station, or the railway network, in order to respect an already known timetable constructed beforehand. For this reason, in the proposed approach of this thesis and in the related works, the peak hours of the day, in which a vast amount of passengers need to take a train in the station, were not modelled, since they simply signify a large or small number of trains which are scheduled to move inside the station (or network), based on the flow of passengers. The Timetabling Problem consists instead in finding a timetable which (i) is able to guarantee a correct amount of trains in order to supply to the peak hours of the day (ii) is resistant, to a certain degree, to possible delays that can occur during the day (iii) allows the passenger to move through the network taking multiple connecting trains without having to wait long times in station.

The problem of generating train timetables can so be divided into two different levels. The first one, the planning level, consists in generating the railway network timetable over a long period of time (in general seasonal), these are the timetables that, once planned, are then visible to the passengers when booking a travel; The second one, the operational level, focus on timetable rescheduling which is the daily task of the operators of adjusting the timetable in case of disruptions, maintenance, or addition of trains (usually freight trains). Problems at the first level can be naturally expressed as constraint satisfaction problems. The vast majority of works on this topic exploits MILP models, while considering different set of constraints and different levels of detail [Caprara et al., 2002, Barrena et al., 2014, Cacchiani et al., 2016]. A number of domain-specific approaches have also been introduced, and the interested reader is referred to [Cacchiani and Toth, 2012] for an extensive review of the field.

The literature on the second level is much more variegated but less extensive, given the multi-faceted nature of the problem to be addressed. [D'ariano et al., 2007] introduced a branch and bound algorithm to recompute a conflict-free and feasible timetable, given the current network conditions. A method to avoid conflicts and to early identify unfeasible timetable schedules is presented by [D'Ariano et al., 2007]. Finally, given a timetable and a current set of delayed trains, [Corman et al., 2012] focused on algorithms to explore the trade-off between cancelling trains reducing congestion and delays, and the inconvenience caused to passengers due to missed connections and limited service. In [Abels et al., 2020] a hybrid approach that extends Answer Set Programming (ASP) [Lifschitz, 1999] is used to tackle real-world train scheduling problems, involving routing, scheduling, and optimisation.

8.4 Other domain's problems solved with PDDL+

PDDL+ has been already successfully exploited in a number of application domains, ranging from defence, physics, healthcare and transportation. For example, in [Ramirez et al., 2018] an encoding is proposed to manage the automatic generation of tactical intercepts for Unmanned Aerial Vehicles (UAV) in air combat. In [Fox et al., 2012] a PDDL+ model is able to manage the efficient use of multiple batteries which led to construction of policies that, in simulation, significantly outperform other systems. In the domain of healthcare, [Alaboud and Coles, 2019] describe an application of planning with the aim of scheduling patient's medication needs and daily activities; in this domain, the usage of PDDL+ allowed to solve a problem which was unsolvable by previous state-of-the-art planners. In [McCluskey and Vallati, 2017] a PDDL+ formulation of urban traffic control, where continuous processes are used to model flows of cars, shows how planning can be used to efficiently reduce congestion of specified roads by controlling traffic light green phases. Several works utilise also the planning engine ENHSP for the production of plans formalized in PDDL+. For example, in [León et al., 2020] a model is presented for the automatic path planning for ultralight aircraft; and in [Kiam et al., 2020] a PDDL+-based planning framework is introduced for planning missions for multiple high-altitude pseudo-satellites.

8.5 Other PDDL+ planning engines besides ENHSP

Since the definition of PDDL+ in [Fox and Long, 2006b] several planning engines have been presented in order to solve temporal and hybrid planning domains. The first planning engines introduced to tackle the solving of PDDL+ planning problems are the TM-LPSAT [Shin and Davis, 2005] and UPPAAL/TIGA [Behrmann et al., 2007] which could solve PDDL+ problems only on linear domains. The first planning engine actually able to solve complex mixed continuos-discrete instances is called UPMurphi [Della Penna et al., 2009] and it introduced the use of explicit model-checking based techniques to solve universal planning problems on hardly-approachable domains like hybrid systems and nonlinear systems. UPMurphi was the first to introduce an internal parser able to decode native PDDL+ domain and problem specifications, without the need to translate them manually in a different formalism. Following a different approach, dREAL [Bryce et al., 2015] tried to solve PDDL+ plans in which action (but not events) preconditions and effect are transformed into SAT formulas and then applying the well-known techniques of resolution of SAT formulas (like DPLL [Davis et al., 1962]) and heuristics developed in the Satisfiability Modulo Theories field [Barrett and Tinelli, 2018]. Then, in 2016, several other approaches emerged to move forward the research on `PDDL+` universal planning engines: the planner used on this thesis, ENHSP, was presented in [Scala et al., 2016] and introduced a new heuristic called Additive Interval-Based Relaxation (AIBR); the same year the planner DiNo [Piotrowski et al., 2016], built on top of UPMurphi, introduced a heuristic as well, called Staged Relaxed Planning Graph+ (SRPG+). The new heuristics allowed both planners to scale and solve more complicated instances, and both outperformed UPMurphi in all instances. In [Cashmore et al., 2016] the planner SMTPlan was introduced, which built on top of the idea of dREAL of solving `PDDL+` via means of SMT, but extending it with an automatic compilation from `PDDL+` to SMT logic and dealing with events, which were not covered in dREAL. In [Balduccini et al., 2017] an encoding from PDDL+ to CASP is presented; CASP [Baselice et al., 2005] is an extension of ASP [Lifschitz, 1999] that allows the modelling of numerical constraints. The paper introduced extensions to the ezcsp CASP solver which allowed solving of CASP programs arising from PDDL+ domains. It is of notable mention, even if it's not a planning engine per se, the validator VAL [Howey et al., 2004] which, for example, is internally used by UPMurphi to check the validity of the produced plans. The capabilities of VAL can be critical in understanding the structures of large plans, with its visualisation and reporting facilities. VAL can additionally report if the plan is flawed, and give advice on how the plan should be fixed.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, an automatic solution to the in-station train dispatching problem was presented. The problem was modelled in `PDDL+`, and a set of domain-specific enhancements were designed which allow the ENHSP planning engine to quickly solve large and complex instances. Results on real-world historical data of a medium-sized railway station from North-West of Italy, provided by RFI, show the potential of the presented approach on a wide range of scenarios. In particular, the proposed approach demonstrated the ability to reduce delays and to better exploit the available infrastructure – with the potential of allowing a station to serve a larger number of trains without the need for structural modifications.

Parts of this thesis has been published on [Cardellini et al., 2021a] and [Cardellini et al., 2021b]. In [Cardellini et al., 2021b] the main objective of the paper was to show the feasibility of the approach giving more focus on the results and analysis which are, in this thesis, presented in Chapter 7. In [Cardellini et al., 2021a] instead, a more in-depth formulation of the in-station train dispatching problem was provided together with a thorough description of the framework which was used to solve it; in addition, in this paper the analysis on the contribution of the different optimizations techniques (described here in Chapter 6) were presented.

9.2 Future Work

This thesis moves a step forward in the resolution of the in-station dispatching problem and more in general in the introduction of artificial intelligence deductive methods in the railway domain. In this section, some open problems which remain to be tackled are introduced. The author plans to work on some or all of these problems in the future.

9.2.1 Modelling of more realistic aspects

Some aspects of real life in-station train dispatching were left out of this thesis for the purpose of reducing the complexity of the problem. These problems need to be addressed and further analysed in order to be able to solve more complicated scenarios and to increase the realism level of the modelled solution.

Shunting operations. In railway operations, shunting consists of two different classes: (i) the process to move a whole train inside the station with the aim of freeing some resources, like platforms, as a means to avoid congesting the station (this process is also called manoeuvring) (ii) the process of sorting items of rolling stock into complete trains or to disassemble a train in multiple parts. it is easy to see that the separation of the shunting process in these two classes reflects a different complexity in the movements of the train inside the station: in the proposed modelling of the problem at hand, the train is always considered as an indivisible unit. For this reason the two problems can be tackled separately and in different orders: the process of manoeuvring a rolling stock throughout the station, once modelled, can indeed be reused for moving the disjointed rolling stocks after the separation of a train (or before the union) covered in the more complex shunting operation.

Shunting is a pivotal operation in a railway station for its ability to greatly increase the capacity of a station (and sequentially reduce the delay) by moving idle trains outside platforms (which in stations are scarce resources).

Long-distance train interconnections. A railway connection is formed when two long-distance passenger trains intersect their route in a station. To support passengers that needs to travel in both of these connecting trains, it is important that the trains stops in adjacent platforms. This because passengers travelling long distances are more prone to carry heavy bags and luggage which are difficult to transport throughout the station, moving from the arriving platform to the departure one. In the model presented in this thesis, a train is able to stop in all the platforms which are free when the train arrives at the station. The introduction of a coupling between trains that forces them to stop at adjacent platforms doesn't change the structure of the search-space in which a solution is searched but drastically invalidates a lot of solutions (i.e., all the solutions in which the coupled train doesn't stop at an adjacent stop). For this reason, the planning time of a solution risks exploding even with a small set of trains, since a lot of time would be invested in finding solutions that doesn't take care of the coupling between the train. For this reason, a new set of heuristic has to be investigated and developed with the aim of guiding two coupled trains in reaching two adjacent platforms without exploring all the possible combinations of invalid platforms.

9.2.2 Extension to bigger stations

As seen in Chapter 2 the station analysed in this thesis is a medium-sized station in the North West of Italy, classified as *Gold* by RFI. While the proposed formalization of a railway station covers a vast amount of stations in Italy (almost 95% according to [Moscarelli et al., 2017] are classified as *Gold*, *Silver* or *Bronze*) some larger stations can fall outside the proposed modelling. Unfortunately these big stations, with a high flow of traffic and passengers, are the most responsible for the delaying of trains and, for this, are the one more in need of automatic support. When dealing with bigger stations, some modelling decision which has been taken by inspecting the movements of trains in a real medium-sized station may fall. For instance, the hypothesis that the itinerary graph of the station is acyclic (as depicted in Figure 2.2) has induced the author to not introduce some guards and complications in the modellings in order to avoid the possibility of trains moving in loop inside the station. In bigger stations, classified by RFI as *Platinum*, some assumptions may fall, and some different constraints should be introduced with the aim of fitting these stations in the proposed modelling.

9.2.3 Explainability

Explainable Artificial Intelligence (XAI) is a branch of AI that takes care of building, improving or supporting AI algorithms in which the solutions can be understood by humans. In this thesis the models provided, and the tools presented, were built with the aim of *supporting* human operators in their job of managing the dispatching of trains inside a station. In order to allow the dispatcher to better understand the generated plans and trust the system, the AI has to provide some sort of insights on the reasoning process undertaken in order to find a viable solution. Furthermore, it is possible that, in special conditions, the AI agent is unable to find a solution that meets all the constraints; this result could arise if, for example, an instance is provided to the planner with a substantial number of trains, or with a considerable amount of blocked resources. If the planning fails and no viable solutions is found, it would be helpful to provide to the human dispatcher some insight on the possible reasons that caused the failure and how to adjust the instance in order for a solution to be found.

Moreover, in the future, it is possible that an AI planning agent would not only support the human dispatcher by *suggesting* some plans but actually *acting* on the resources of the station (i.e. switches, signals and traffic lights) in order to improve the traffic flow in the station autonomously. For safety reasons it is then paramount that the AI agent planning is able to provide some insight on its decisions and actions in order to be trusted by human dispatchers, train operators, railway companies and the public.

In [Fox et al., 2017] Explainable Planning (XAIP) is presented together with a survey of questions that an explainable planning agent should be able to answer. For example:

- 'Why did you do that ?': In very long plans is difficult to immediately see why a decision of the planner is beneficial to the whole plan. For example a dispatcher could ask: 'Why did you force this train to wait at this entry point ?'. The planner should be able to answer the question with something like 'I had it wait a bit so that this other train could leave earlier and thus reducing the total delay of x%'.
- 'Why can't you do that ?': The expert dispatcher, which is used to deal with the in-station train dispatching daily, could have some prior bias on how a train should behave inside the station. This could be determined by previous experiences on how situations have been dealt in the past. For this reason the human dispatcher could ask why an action was not performed. The agent should be able to answer that the action could not be taken (and the reasons why) or prove that, if the action had been chosen, it would have produced a worse plan than the one provided.

9.2.4 Expansion to group of stations

The work presented in this thesis focuses on a single station and the outside network is modelled as buffers of infinite sizes from which the trains enter or exit in the station at an instant provided by the instance given by the operator. The formalization presented can be applied to several stations separately in order to optimise and better manage the overall flow of trains. This approach would undoubtedly improve the flow of station in general, but better results could be achieved by looking at groups of stations as a whole and manage their interaction through lines.

As proposed in [Lamorgese and Mannino, 2015] it is possible to decompose the problem into smaller sub-problems associated with the line and the stations, managing and keeping track of the trains inside the lines (the line dispatching problem) and managing the more complex problem of the train dispatching problem as a sub problem which can be tackled in isolation.

Acknowledgements

Questa tesi è frutto di un anno e mezzo di duro lavoro e voglio usare questo spazio per ringraziare tutte le persone che mi sono state vicine in questo periodo.

Il più grande ringraziamento va ai miei genitori che mi hanno supportato (e sopportato) da sempre, mi hanno cresciuto, spronato, ed è grazie a loro che sono la persona che sono adesso; ringrazio mia madre per esserci sempre nei momenti difficili con una parola di conforto, una carezza o semplicemente per pensare ad altro, ringrazio mio padre per essere la mia guida e punto di riferimento e per aiutarmi sempre a ragionare con chiarezza sulle cose. Ringrazio mio nonno, a cui ho ancora la fortuna di poter leggere questa dedica, per avermi trasmesso la testa dura di andare sempre avanti nella vita a testa alta.

Ringrazio il Prof. Marco Maratea per essersi dimostrato un abilissimo mentore, che dal secondo anno della triennale, quando gli chiesi informazioni sul piano di studio, non ha mai smesso di guidarmi e mi ha sempre spronato a migliorarmi.

Ringrazio il Prof. Mauro Vallati per essere stato un insegnante esperto e un paziente relatore, per aver sempre risposto alle mie mail (sempre troppo lunghe) in maniera precisa, puntuale e in tempi brevissimi.

Ringrazio il Prof. Luca Oneto, senza il quale questa tesi non sarebbe stata possibile, per avermi formato su come funziona il mondo ferroviario e avermi guidato, con il suo spirito critico, nella risoluzione dei vari problemi.

Ringrazio Davide, amico fraterno, che in questi dodici anni in cui siamo cresciuti insieme mi è sempre stato vicino e con il quale ho condiviso praticamente tutti i momenti più importanti della mia vita, compreso questo.

Ringrazio Elena e Picci, per esserci sempre, per sapermi consigliare, per riuscire sempre a farmi ridere e per aver condiviso con me sia le cose belle che quelle brutte.

Ringrazio Gian, che mi ha accompagnato in questa tesi, per esserci sempre stato per scambiare consigli e opinioni e, soprattutto, per sdrammatizzare quando le cose non funzionavano come avrebbero dovuto.

Ringrazio Leo, che considero ormai mio fratello maggiore, per avermi guidato e sostenuto, sia nel mondo lavorativo, sia nella vita.

Ringrazio Adile per avermi sopportato fin da quando eravamo compagni di banco e per essersi dimostrata un ottima amica anche dopo (э анкэ пэр авэрми инсэняту ил чирилику).

Ringrazio la Gioi per tutti i bei momenti passati insieme e per tutti i momenti di spensieratezza.

Ringrazio Riccardo, Enrico, Daniele e Gian (di nuovo) (a.k.a Ex Dipartimento IT Secondhand) per avermi insegnato cosa vuol dire essere una squadra e saper affrontare i problemi con determinazione e con impegno.

Ringrazio i miei colleghi in Secondhand: Milena, Andrea, Mattia, Lorena, Stefano e Marcello, per avermi fatto capire cosa vuol dire lavorare con persone competenti e su cui poter contare.

List of Figures

Bibliography

[Abels et al., 2020] Abels, D., Jordi, J., Ostrowski, M., Schaub, T., Toletti, A., and Wanko, P. (2020). Train scheduling with hybrid answer set programming. *Theory and Practice of Logic Programming*, pages 1–31.

[Alaboud and Coles, 2019] Alaboud, F. K. and Coles, A. (2019). Personalized medication and activity planning in pddl+. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 492–500.

[Atzmon et al., 2019] Atzmon, D., Diei, A., and Rave, D. (2019). Multi-train path finding. In *Proceedings of the Symposium on Combinatorial Search*, pages 125–129.

[Balduccini et al., 2017] Balduccini, M., Magazzeni, D., Maratea, M., and LeBlanc, E. (2017). Casp solutions for planning in hybrid domains. *arXiv preprint arXiv:1704.03574*.

[Barrena et al., 2014] Barrena, E., Canca, D., Coelho, L. C., and Laporte, G. (2014). Exact formulations and algorithm for the train timetabling problem with dynamic demand. *Computers & Operations Research*, 44:66–74.

[Barrett and Tinelli, 2018] Barrett, C. and Tinelli, C. (2018). Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer.

[Baselice et al., 2005] Baselice, S., Bonatti, P. A., and Gelfond, M. (2005). Towards an integration of answer set and constraint solving. In *Proceedings of the International Conference on Logic Programming*, pages 52–66. Springer.

[Behrmann et al., 2007] Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K. G., and Lime, D. (2007). Uppaal-tiga: Time for playing games! In *Proceedings of the International Conference on Computer Aided Verification*, pages 121–125. Springer.

[Bellman, 1966] Bellman, R. (1966). Dynamic programming. *Science*, 153(3731):34–37.

[Billionnet, 2003] Billionnet, A. (2003). Using integer programming to solve the train-platforming problem. *Transportation Science*, 37:213–222.

[Blum and Furst, 1997] Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300.

[Böcker et al., 2001] Böcker, J., Lind, J., and Zirkler, B. (2001). Using a multi-agent approach to optimise the train coupling and sharing system. *European Journal of Operational Research*, 131:242–252.

[Boleto et al., 2021] Boleto, G., Oneto, L., Cardellini, M., Maratea, M., Vallati, M., Canepa, R., and Anguita, D. (2021). In-station train movements prediction: from shallow to deep multi scale models. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*.

[Bryan et al., 2007] Bryan, J., Weisbrod, G. E., and Martland, C. D. (2007). *Rail freight solutions to roadway congestion: final report and guidebook*, volume 586. Transportation Research Board.

[Bryce et al., 2015] Bryce, D., Gao, S., Musliner, D., and Goldman, R. (2015). Smt-based nonlinear pddl+ planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29.

[Cacchiani et al., 2016] Cacchiani, V., Furini, F., and Kidd, M. P. (2016). Approaches to a real-world train timetabling problem in a railway node. *Omega*, 58:97–110.

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

[Caprara et al., 2002] Caprara, A., Fischetti, M., and Toth, P. (2002). Modeling and solving the train timetabling problem. *Operations research*, 50:851–861.

[Caprara et al., 2010] Caprara, A., Galli, L., Kroon, L., Maróti, G., and Toth, P. (2010). Robust train routing and online re-scheduling. In *Proceedings of the workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, pages 24–33.

[Cardellini et al., 2021a] Cardellini, M., Maratea, M., Vallati, M., Boleto, G., and Oneto, L. (2021a). An efficient hybrid planning framework for in-station train dispatching. In *Proceedings of the International Conference on Computational Science*. Springer Nature Switzerland AG.

[Cardellini et al., 2021b] Cardellini, M., Maratea, M., Vallati, M., Boleto, G., and Oneto, L. (2021b). In-station train dispatching: A PDDL+ planning approach. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 450–458.

[Cardillo and Mione, 1998] Cardillo, D. D. L. and Mione, N. (1998). k l-list λ colouring of graphs. *European Journal of Operational Research*, 106:160–164.

[Cashmore et al., 2016] Cashmore, M., Fox, M., Long, D., and Magazzeni, D. (2016). A compilation of the full pddl+ language into smt. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 26.

[Chakroborty and Vikram, 2008] Chakroborty, P. and Vikram, D. (2008). Optimum assignment of trains to platforms under partial schedule compliance. *Transportation Research Part B: Methodological*, 42:169–184.

[Corman et al., 2012] Corman, F., D'Ariano, A., Pacciarelli, D., and Pranzo, M. (2012). Bi-objective conflict detection and resolution in railway traffic management. *Transportation Research Part C: Emerging Technologies*, 20:79–94.

[Cushing et al., 2007] Cushing, W., Kambhampati, S., and Weld, D. S. (2007). When is temporal planning really temporal? In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1852–1859.

[D'Ariano et al., 2007] D'Ariano, A., Pranzo, M., and Hansen, I. A. (2007). Conflict resolution and train speed coordination for solving real-time timetable perturbations. *IEEE Transactions on Intelligent Transportation Systems*, 8:208–222.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

[Della Penna et al., 2009] Della Penna, G., Magazzeni, D., Mercorio, F., and Intrigila, B. (2009). Upmurphi: A tool for universal planning on pddl+ problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19.

[D'ariano et al., 2007] D'ariano, A., Pacciarelli, D., and Pranzo, M. (2007). A branch and bound algorithm for scheduling trains in a railway network. *European Journal of Operational Research*, 183:643–657.

[Fox and Long, 2003] Fox, M. and Long, D. (2003). Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

[Fox and Long, 2006a] Fox, M. and Long, D. (2006a). Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297.

[Fox and Long, 2006b] Fox, M. and Long, D. (2006b). Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297.

[Fox et al., 2012] Fox, M., Long, D., and Magazzeni, D. (2012). Plan-based policies for efficient multiple battery load management. *Journal of Artificial Intelligence Research*, 44:335–382.

[Fox et al., 2017] Fox, M., Long, D., and Magazzeni, D. (2017). Explainable planning. *arXiv preprint arXiv:1709.10256*.

[Givoni et al., 2009] Givoni, M., Brand, C., and Watkiss, P. (2009). Are railways climate friendly? *Built Environment*, 35(1):70–86.

[Horrocks et al., 2004] Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., and Dean, M. (2004). Swrl: A semantic web rule language combining owl and ruleml. W3c member submission, World Wide Web Consortium.

[Howey et al., 2004] Howey, R., Long, D., and Fox, M. (2004). Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301. IEEE.

[Kiam et al., 2020] Kiam, J. J., Scala, E., Javega, M. R., and Schulte, A. (2020). An ai-based planning framework for haps in a time-varying environment. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 412–420.

[Kumar et al., 2018] Kumar, R., Sen, G., Kar, S., and Tiwari, M. K. (2018). Station dispatching problem for a large terminal: A constraint programming approach. *Interfaces*, 48:510–528.

[Lamorgese and Mannino, 2013] Lamorgese, L. and Mannino, C. (2013). The track formulation for the train dispatching problem. *Electronic Notes in Discrete Mathematics*, 41:559–566.

[Lamorgese and Mannino, 2015] Lamorgese, L. and Mannino, C. (2015). An exact decomposition approach for the real-time train dispatching problem. *Operations Research*, 63:48–64.

[Lamorgese et al., 2016] Lamorgese, L., Mannino, C., and Piacentini, M. (2016). Optimal train dispatching by benders'-like reformulation. *Transportation Science*, 50:910–925.

[Lee and Chen, 2009] Lee, Y. and Chen, C.-Y. (2009). A heuristic for the train pathing and timetabling problem. *Transportation Research Part B: Methodological*, 43:837–851.

[León et al., 2020] León, B. S., Kiam, J. J., and Schulte, A. (2020). A fault-tolerant automated flight path planning system for an ultralight aircraft. In Baldoni, M. and Bandini, S., editors, *AIxIA 2020 - Advances in Artificial Intelligence - XIXth International Conference of the Italian Association for Artificial Intelligence, Virtual Event, November 25-27, 2020, Revised Selected Papers*, volume 12414 of *Lecture Notes in Computer Science*, pages 175–190. Springer.

[Li et al., 2021] Li, J., Chen, Z., Zheng, Y., Chan, S.-H., Harabor, D., Stuckey, P. J., Ma, H., and Koenig, S. (2021). Scalable rail planning and replanning: Winning the 2020 flatland challenge. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 477–485.

[Lifschitz, 1999] Lifschitz, V. (1999). Answer set planning. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 373–374. Springer.

[Mannino and Mascis, 2009] Mannino, C. and Mascis, A. (2009). Optimal real-time traffic control in metro stations. *Operations Research*, 57:1026–1039.

[McCarthy and Hayes, 1981] McCarthy, J. and Hayes, P. J. (1981). Some philosophical problems from the standpoint of artificial intelligence. In *Readings in Artificial Intelligence*, pages 431–450. Elsevier.

[McCluskey and Vallati, 2017] McCluskey, T. L. and Vallati, M. (2017). Embedding automated planning within urban traffic management operations. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 391–399.

[Mcdermott et al., 1998] Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control.

[Moscarelli et al., 2017] Moscarelli, R., Pileri, P., and Giacomel, A. (2017). Regenerating small and medium sized stations in italian inland areas by the opportunity of the cycle tourism, as territorial infrastructure. *City, Territory and Architecture*, 4(1):1–14.

[Piotrowski et al., 2016] Piotrowski, W. M., Fox, M., Long, D., Magazzeni, D., and Mercorio, F. (2016). Heuristic planning for pddl+ domains. In *AAAI Workshop: Planning for Hybrid Systems*, volume 16, page 12.

[Ramirez et al., 2018] Ramirez, M., Papasimeon, M., Lipovetzky, N., Benke, L., Miller, T., Pearce, A. R., Scala, E., and Zamani, M. (2018). Integrated hybrid planning and programmed control for real time uav maneuvering. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 1318–1326.

[Rodriguez, 2007] Rodriguez, J. (2007). A constraint programming model for real-time train scheduling at junctions. *Transportation Research Part B: Methodological*, 41:231–245.

[Scala et al., 2016] Scala, E., Haslum, P., Thiébaux, S., and Ramirez, M. (2016). Interval-based relaxation for general numeric planning. In *Proceedings of the European Conference on Artificial Intelligence*, pages 655–663.

[Scala et al., 2020] Scala, E., Haslum, P., Thiébaux, S., and Ramírez, M. (2020). Subgoaling techniques for satisficing and optimal numeric planning. *Journal of Artificial Intelligence Research*, 68:691–752.

[Scala and Vallati, 2021] Scala, E. and Vallati, M. (2021). Effective grounding for hybrid planning problems represented in PDDL+. *The Knowledge Engineering Review*, 36.

[Scalise, 2014] Scalise, J. (2014). How track circuits detect and protect trains. In *Railw. Walk Rail Talk*, volume 1, pages 1–7.

[Scalzo and Mangione, 2003] Scalzo, A. and Mangione, C. (2003). Frecciabianca 8623 da torino porta nuova a roma termini. `https://www.e656.net/orario/treno/8623.html`. Accessed: 2021-05-23.

[Shin and Davis, 2005] Shin, J.-A. and Davis, E. (2005). Processes and continuous change in a sat-based planner. *Artificial Intelligence*, 166(1-2):194–253.

[Smullyan, 1995] Smullyan, R. M. (1995). *First-order logic*. Courier Corporation.

[Theeg and Vlasenko, 2009] Theeg, G. and Vlasenko, S. (2009). Railway signalling & interlocking. *International Compendium. Hamburg, Eurail-press Publ*, 448.