Dealing with conflicting trains

Effectively avoiding and resolving conflicts during shunting

Mees Gribnau[1]

Supervisor(s): Sebastijan Dumancic[1]**, Issa Hanou**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Mees Gribnau
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumancic, Issa Hanou, Rihan Hai

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

A shunting yard is used to store trains between arrival and departure. A conflict arises in a shunting yard when one train obstructs another from leaving. Resolving a conflict is done by re-allocating the trains obstructing the departing train to other tracks in the shunting yard. However, these re-allocations complicate the problem at hand and incur high costs for train operators. Therefore, it is desirable to avoid conflicts whenever possible. The aim of this paper is to find an effective manner to deal with conflicts in a train shunting yard in an existing planner system. We propose the split into a portfolio planner, which first tries to find a solution without any re-allocations, and if that does not yield a solution will look for a solution with re-allocations. For both planners, a model is defined. Furthermore, the paper explores techniques to increase the speed of the first planner, namely heuristic search, a set partitioning approach, and constraint programming. An implementation of the latter approach has exhibited excellent performance across problems of all sizes.

1 Introduction

In 2019 the Dutch rail network facilitated more than 1.3 million journeys a day. An overwhelming majority of those journeys happen during the day, resulting in a big difference between the number of trains active during the day and during the night. This excess of trains is stored in *shunting yards* at train stations. To ease the strain on the train station workers, plans detailing where and how to park the trains are created in advance. As the size of the shunting yard and the number of trains requiring shunting increases, the manual formulation of such plans becomes progressively more challenging. Consequently, the development of algorithmic planners becomes necessary.

Space, time, maintenance, cleaning, and routing constraints all have to be considered during the planning stage. One of those constraints concerns the order in which the trains can be parked over the different *shunting tracks*. A *conflict* occurs when one train obstructs another from leaving. One of the main concerns during the creation of a shunting plan is the avoidance of conflicts, as they can be hard to resolve, costly for train operators, and can result in delays. A conflict can often be resolved by *re-allocating* previously parked trains to a second parking spot, however, this is a costly endeavor and should be avoided if possible.

There are two main types of shunting yards, namely *shuffleboards* and carousals. This paper only considers shuffleboards which consist of a multitude of Last-In-First-Out (LIFO) tracks, which operate similarly to a stack; the trains enter and exit on the same side of the track, and the other side is a dead end. These tracks are connected to each other in a tree-like structure.

The objective of this paper is to answer the following research question: *How can conflicts effectively be avoided and resolved during the creation of a train shunting plan?*. The following sub-questions will be answered to accomplish this:

- How can conflicts be detected?
- How can a plan with minimal conflicts effectively be found?
- How can conflicts be resolved effectively?

Multiple approaches are taken within this research, including a heuristic, set partitioning, and constraint programming.

The structure of the paper is as follows. Chapter 2 conducts a literature review focusing on related work in the field. Chapter 3 provides a description of the specific problem addressed in this paper. The contributions of this paper are outlined in chapters 4 and 5, while chapter 6 presents the evaluation of the obtained results. For a more in-depth understanding of replicating the findings, chapter 7 offers a detailed explanation. Chapter 8 provides a discussion, and finally, in Chapter 9, the conclusion and avenues for future research are presented.

2 Related work

For this paper, there are two areas of research of importance, research into planning and research into this specific problem.

2.1 Planning

Planning Domain Definition Language (PDDL) introduced by Ghallab et al. (1998) is a language that can be used to define planning problems. A complete planning problem in PDDL consists of two parts: a domain file and a problem file. In the domain file, the type of problem is defined, whereas the problem file is used to define the specifics of the problem instance. The most important parts of a PDDL domain are object types, predicates, and actions. A predicate is a Boolean value and applies to a set of objects given as an argument, for example, the predicate hasParked in the example below keeps track of which trains have been parked. An action can only be performed if its precondition evaluates to true, and it effects are expressed by changing the value of predicates. In the problem file the object instances, the initial state, and goal conditions are defined.

```
1  ...
2  (:types
3      train
4  )
5  (:predicates
6      (hasParked ?t - train)
7  )
8  (:action park-train
9      :parameters (?train - train ...)
10     :precondition (and (not (hasParked ?train
       ))
11     :effect (and (hasBeenParked ?train))
12 )
```

Listing 1: Example PDDL domain

PDDL was created to standardize planning problem definitions in the International Planning Competition (IPC), a competition whose goal is to promote planning research. In the competition a multitude of *planners* compete to find solutions, consisting of a set of actions with their arguments,

to a collection of problems as quickly as possible. For this research, one of the participants in the satisfying track of the classical part of the IPC2018 will be extended. Planners which entered into the satisfying track of the classical part of the IPC2018 include Saarland, Freelunch-Madagascar, Serialized Iterative Width, and BFS(f).

Saarland uses Star-Topology Decoupled State Space Search as described in Gnad and Hoffmann (2018). To combat the explosive nature of search spaces in planning problems it tries to split the domain into conditionally independent components.

Freelunch-Madagascar translates the problem to a Boolean satisfiability problem (SAT) and then uses an existing SAT solver to solve the problem.

Serialized Iterative Width, or SIW, is an algorithm that uses blind search. Its efficiency depends on heavy pruning which is done through a novelty test; if a newly generated state has gone too many actions without making a new combination of atoms true, it is pruned. In PDDL an *atom* translates to a single predicate in the goal definition. SIW might have trouble with domains that include dead-ends and larger size problems where it takes a higher quantity of actions before the value of atoms is changed, as it might prune too early and it has next to the novelty test no other means of distinguishing states.

BFS(f) is a best-first planner which uses lazy evaluation and multiple heuristics to decide which node to expand upon. The most important heuristic score nodes similar to the novelty test above. In the case of ties, a helpful action heuristic is used, which on beforehand analyses the problem to generate landmarks that every solution has to pass through to reach the goal state. Then to evaluate a state it counts the number of landmarks it has reached. If the nodes are still tied a third heuristic is employed, which either estimates the cost of achieving the most expensive atom or the combined cost of achieving every unachieved atom independently, depending on the configuration of the planner.

2.2 Train Unit Shunting Problem

Freling et al. (2005) introduced the Train Unit Shunting Problem (TUSP), with a given set of arriving train and departing trains, composed of a time and a train unit, where each train unit has a type and sub-type. The arriving trains have to be mapped to the departing trains such that the sub-types match. Because of the large size of the domain, they propose a two-step solution, where first each arriving shunt unit is matched to a departing shunt unit and then each train unit is assigned to a shunt track, the latter problem being referred to as the Track Assignment Problem (TAP). It is formulated as a Set Partition Problem. Every track is assigned a set of blocks, consisting of multiple train units. It is assumed that before the first departure all the arrivals take place and all the trains have to be parked. Assignments that result in a conflict are considered unfeasible, thus it does not allow for re-allocations. In this study, there is also a constraint where trains can only be moved to specific tracks suitable for their sub-type. Subsequent research often disregards this limitation. The Train Assignment Problem (TAP) bears a resemblance to the initial problem we are about to present. However, because the costs associated with parking a train on a track are taken into account, it transforms into a cost minimization problem, where our objective is to seek any conflict-free solution.

Gilg et al. (2018) introduces the *Train* Assignment Problem without this limitation and provides an integer programming approach to solve it. LIFO, FIFO, and FREE tracks are all included.

Stefano and Koči (2004) provide a graph theoretical approach for a similar problem without any length constraints. An online algorithm is provided that assigns a sequence of incoming trains to the minimum number of FIFO tracks in polynomial time. Although the fact that there are no length constraints, which will be part of our problem, completely transforms the nature of the problem, but some insights in this paper can still be used in our research to create feasibility checks.

Cornelsen and Stefano (2007) contribute a polynomial coloring algorithm for FREE tracks in instances of the problem with a timetable for departures and arrivals, both with and without re-allocation.

In the work of van den Akker et al. (2008) re-allocations are being considered in a version of the problem where departures can happen before the last arrival. A greedy algorithm and a dynamic programming solution are proposed. The greedy algorithm assigns trains to tracks in a round-robin manner, or according to a form of priority. The dynamic programming solution considers all the possible actions in all possible states. As the search space can be very large a few suggestions are given to exclude nodes from the search, such as removing symmetries and upper bounding the solution.

Wolfhagen (2017) introduces the Train Unit Shunting Problem with Re-allocation (TUSP-R) and formulates it as a Mixed Integer Programming Problem (MIP).

3 Problem Description

This study aims to investigate how to adapt an existing PDDL domain and planner to effectively avoid and resolve conflicts. Re-allocations are expensive for train station operators and heighten challenges in seeking a solution, as they let the size of search space explode. For this reason, we propose the use of a portfolio planner, where the first planner tries to find a solution to the problem without re-allocations and the second planner tries to find a solution to the problem with re-allocations. The first planner should find a solution quickly if one exists because it has a relatively small search space. If no solution can be found the second planner can be run for a longer time period.

Both problems involve moving the trains from arrival to a shunting track to departure; in a valid solution, every train must have been parked before they depart. The problems also have properties similar to the TAP as described in Freling et al. (2005). Arriving trains have been matched to departing trains already, so the problem input contains a set of trains for which the arrival and departure order is known. We will refer to the position of a train in the arrival order as its arrival time. Furthermore, we adhere to the midnight constraint as introduced by Winter and Zimmermann (2000), which states that the first departure happens after the last arrival. In this paper we do not make a distinction between trains and train

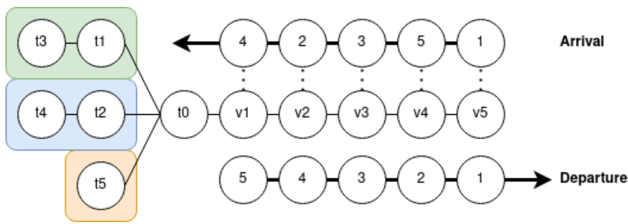

Figure 1: Example problem instance with three tracks.

units; every train has exactly one train unit. The shunting yard consists of *track-parts*, each of them fitting exactly one train. A shunting track, or track, consists of a number of connected track parts. Trains can be parked on tracks, but not on any other track-parts in the yard. A *conflict* or *crossing* refers to the scenario in which two trains are both stationed on the same track, where the train scheduled for an earlier departure cannot proceed unless the other train departs first. In a shuffleboard yard, a conflict arises when any two trains are parked on the same track, where the train which is parked on the track earlier is also the with a lower departure time. In our problem instances, it is impossible for a train parked on one track to obstruct a train on any of the other tracks.

To find a solution without re-allocations every train should move through exactly one track. A train moving through more than one track would indicate that a re-allocation has taken place, and no tracks would indicate the train has not been parked. In our problem with re-allocation, every train can only be re-allocated once. Thus in our problem with re-allocation, every train should move through either one or two tracks.

3.1 Formal problem definition

We can represent the shunting yard as a tree, with every node representing one track part. The tracks can be represented as set of sets of vertices. We define a move as transferring a train from its current position to an adjacent unoccupied vertex.

The Train Unit Shunting Plan Problem (from now on referred to as the TUSPP) consists of finding a sequence of moves which on a given tree T moves each of k trains numbered 1, 2, ..., k from its source vertex to its destination vertex. With a given set of sets of vertices R, each train moves through at least one element of in any of the sets in R and does not move through any two vertices such that both vertices are elements of different sets in R.

The Train Unit Shunting Plan Problem with Reallocation (from now on referred to as the TUSPP-R) consist of finding a sequence of moves which on a given tree T moves each of k trains numbered 1, 2, ..., k from its source vertex to its destination vertex. With a given set of sets of vertices R, each train moves through at least one element of in any of the sets in R and does not move through any three vertices such these vertices are elements in different sets in R.

Note that the TUSP and TAP only concern assigning every train to a track, while the TUSPP and TUSPP-R also include moving the trains over the shunting yard.

4 Conflict without reallocation

Extension of a provided domain is discussed in chapter 4.1, and improvements to an existing planner are explored in 4.2.

4.1 Extending the domain

The first part of this research entails transforming a provided domain. This section will give an overview of the changes made in this domain. The changes are made for two reasons: to disallow a conflict to occur and to reduce the search space.

Within the confines of the original domain, there exist three object types, namely trackpart, track, and trainunit. These remain unchanged. There are four actions, each of them describing a move of the train between track parts in different parts of the train station. For example, move-on-arrival is used to move the trains from the train station to a switch, whereas move-to-tree moves the trains from a switch to one of the tracks.

Without re-allocations, a valid solution cannot contain a crossing at any point in time. To avoid expanding states which can never lead to a solution, we disallow trains from entering a track if that will lead to a conflict. We introduce a numeric fluent, departure, which specifies the departure time of every train. A numeric fluent in PDDL works similarly to a predicate, but instead of a Boolean value, it contains a number. Using this we check if the train conflicts with any of the trains already on the track by comparing the departure times. If the incoming train departs later than any of the trains already on the track, entering the track is not allowed, as it would lead to a conflict.

Trains can only move to track parts directly adjacent to the track part they currently occupy. The nextTo predicate, which specifies if two track parts are connected, is used to enforce this. It takes two arguments of the trackpart type, first the track where the train is moving from and the second where it is moving to. Due to the tree-like structure of the shunting yard, an efficient solution strictly involves moving trains toward the leaf nodes upon arrival and toward the root node upon departure. To reduce the search space of the problem this predicate can be split into two different predicates: nextToOnArrival and nextToOnDeparture. During the arrival phase, trains are only allowed to move in the nextToOnArrival direction, and vice versa for departure. The difference of the nextTo predicates between the original domain and the improved problem in figure 1 is shown in listings 2 and 3 below.

```
1    (nextTo t0 t1)
2    (nextTo t1 t0)
```

Listing 2: nextTo predicates in original domain

```
1    (nextToOnArrival t0 t1)
2    (nextToOnDeparture t1 t0)
```

Listing 3: nextTo predicates in improved domain

In the original domain, there exist no limitations for the order in which trains can be moved. In reality, one train should be parked before another can be moved, as there are limited workers who can drive the trains. We make the assumption that there is only one driver. To account for this we add a

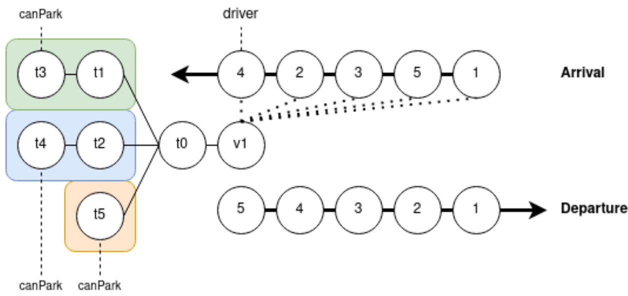

Figure 2: Improved domain of example problem 1.

driver predicate to the domain, which specifies which train the driver currently occupies. A driver can move between trains only directly after the train it is currently in is parked. A driver should move to the next train arriving or departure, so we add a nextTrainArriving and nextTrainDeparting predicate which details which train is next in line, functioning similarly to a linked list. To switch from the last arrival to the first departure, we also add two predicates detailing which trains these are, and add a driver switch action that checks for these trains in particular. This switch also sets the departing predicate to true, indicating that the departure phase has started. Where first only moves in the nextToOnArrival directions where allowed, now trains can only move using tracks connected by the nextToOnDeparture predicate.

The original domain considers a train parked as soon as it enters a track, and after parking the train can still be moved around. A problem arises when implementing the above-described improvements, as the driver switching trains as soon as it enters the first track-part in a track would block the rest of the track. To resolve this we add a canPark predicate, which describes for every track which track-part the trains can currently park on. It is set to the deepest free track-part in every track and moves one track-part up whenever a train is parked. An additional action is created to park the trains, which checks if the canPark predicate holds true for the track-part the train is currently occupying, and allows the driver to switch trains.

Finally, the track parts where the trains are located at the start of the problem are used as a means to establish the arrival and departure order. With the introduction of the driver, this has become redundant, as it is already guaranteed that the trains will be parked and depart in order. At the initial state, we place all the trains on a singular track part, which we will refer to as the train station. To make sure all trains are able to reach the train station when departing, we introduce a depart-train action, which removes the train from the domain and sets a new hasDeparted predicate. In the goal state we now only have to check if every train has departed. By doing this we eliminate track parts that in reality might not exist and keep solutions concise.

4.2 Improving the planner

The second part of this research entails improving an existing planner. Three different approaches have been taken. First, a heuristic which estimates the chance a conflict will occur in a given state. Second, a set partitioning which reasons over the combination of possible complete track assignments. At last, a constraint programming approach which constricts the assignments to neighboring track parts.

Approach 1: Heuristic

The nature of PDDL results in most planners using forward search to find a solution. As it defines a start state, a goal state, and a set of actions to transform from one state to another, without further knowledge about the problem this is the most natural, if not only approach. To increase the performance of a planner, we can build a heuristic to judge the generated states.

Typically in heuristic search, the heuristic estimate the distance to a goal state, but in this problem it is also important to estimate the chance that a goal state is reachable from a current state. To balance between exploitation and exploration, both of these notions must be included. Generally, we do not want to expand states if we are almost certain that a conflict will occur, but if almost all trains have been parked already, we might as well check if we can reach a solution. To include the latter notion, we need to estimate how likely it is a conflict will pop up in a given state. Thus our heuristic function can be written as $f(S) = h(S) - c(S)$, with S being a state, f the heuristic score function, h the function which estimates the distance the goal and c being the function estimating the chance of a conflict.

A first intuition to create a heuristic might come from the fact that trains departing at a later stage should in general be parked deeper into the shunting tracks because this allows a larger amount of trains to park in front of it. Similarly, trains departing earlier should in general be parked closer to the beginning of the track. To translate this into a heuristic, we create a sorted array with the depth of every track. We map this to an array of all trains sorted on departure order to get the *expected depth* of every train. The closer a train is parked to its expected depth, the less likely it is that this train will be part of a conflict.

Our h function is the amount of unparked trains multiplied by a constant. The c function checks for all parked trains the difference is between its actual depth and expected depth. If this difference is low, the heuristic decreases, as a well-placed train decreases the likelihood that a conflict will occur. When the train is parked in a bad spot, we increase the heuristic.

Approach 2: Set partitioning

The aforementioned approach suffers from one major flaw: it fails to exploit the knowledge about the order in which the future trains will arrive. Before parking any train we can already tell if a combination of trains will be able to park on the same track.

To find a solution for the TUSPP it might be easier to first figure out on which track to park every train, and only then look at how to route the trains over the train station from arrival to parking to departure. We add a new predicate to the domain as described in 4.1. In the new domain, trains are only allowed to enter one track, the track for which this predicate is set. We create a preprocessor that finds a track assignment and transforms the problem definition by adding this predicate accordingly.

After finding the track assignment for each train. Instead of dividing the trains over the parking spots one by one, we can also reason over the possible sets of trains that can be assigned over any track for every length. We can generate all the possible combinations of assignments for every track, and then select one assignment for every track such that all the assignments together contain all the trains.

Obviously, all trains can be parked on a track by themselves without causing a conflict on that track. To generate all the possible combinations of a pair of trains on the same track one train has to arrive earlier and depart later than the other train. Now a dynamic programming approach can be used to generate possible combinations for the sets of higher cardinality; the generate the possible track configurations of three trains we combine two sets of length two where the last arriving train from one set is equal to the first arriving train of the other set. We do this until we have covered the track length for every track inside of the problem instance.

Now we can find out on which tracks the trains should be parked by selecting one of the sets for every track, such that the chosen sets for all tracks together contain all the trains. Assuming that the amount of parking spots is exactly equal to the number of trains to be parked, we are looking for a combination of sets such that no train is part of multiple sets. Thus the union between any of the chosen sets should be the empty set in a valid solution.

In some instances, we can fill in entire tracks before we have to make any decision. For example, take a problem instance with exactly as many trains as parking spots. If some track has a length of 5, and there exists only one combination of five trains that can fit together on the same track, if a valid solution exists it will always involve parking this combination on that track.

To effectively search for a solution, we want to find conflicts as soon as possible. To do this we represent every possible track assignment as a node. For every combination of two tracks, we create an edge between a node in both tracks if the union is empty. This represents a valid combination of assignments. To reduce similarities, for tracks of equal length we can remove any combination where the index of the second node is higher than the index of the first node.

To find the track we will pick an assignment for first, we look at the coverage for each of the tracks onto each other. The track of length 1 fully covers the tracks of length 1, whereas the second track of length 2 only covers 1/3 of the nodes in the other set of length 2.

This approach for as now only works when the number of trains is exactly equal to the number of parking spots. To also be able to use this approach in problems in instances where there are more parking spots than trains to be parked, we introduce the notion of ghost trains. A ghost train is a train that does not exist in the actual problem but is included during the set partitioning. To reduce the former problem to the latter problem, we add a ghost train for every empty spot. These ghost trains are added to the end of the arrival order and the beginning of the departure order. They will never create a conflict.

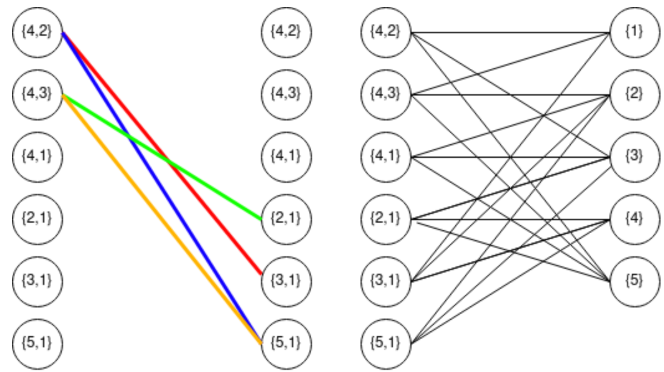

Figure 3: Possible assignment combination between tracks of length [2,2] and track of length [2,1] of problem in figure 1.

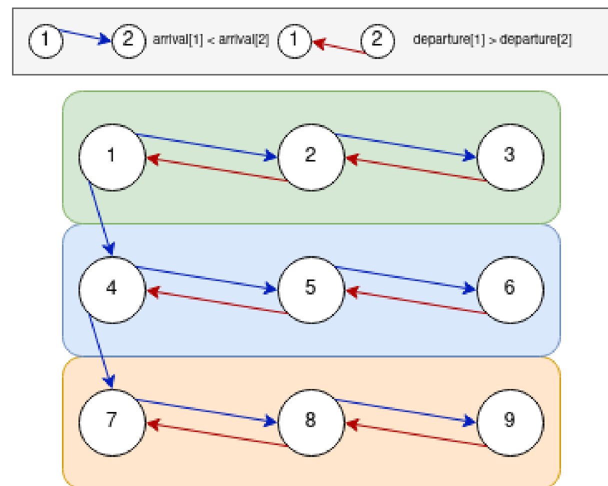

Figure 4: Constraints between track-parts in a [3,3,3] shunting yard.

Approach 3: Constraint programming

Instead of analyzing all the possible sets of allocation for every track, it is also possible to let a constraint programming solver intrinsically find out if a combination of allocations is possible. To do this we define the array, where each cell represents a track part. The element a cell contains represent the train that will be parked on this track part. Each element must hold an integer value between 1 and the number of trains. To make sure the solver does not assign the same trains to multiple track parts, we define an all-different constraint over the entire array. To make sure there are no conflicts in the assignments, we add two constraints for each connected shunting track part in a track:

1. arrival time of the train on the deeper track part is lower
2. departure time of the train on the deeper track part is higher

To break symmetries we can add a constraint between tracks of equal length which ensures that the arrival time of the train parked at the last position of the first track is always lower than the arrival time of the train parked. An overview of the entire model is shown in figure 4.

5 Conflicts with reallocation

To create a model which allows for reallocation, we build on top of the model proposed in 4.1, without the constraint that a train can only enter a track if it does not directly result in a conflict. In order to keep track of whether a train has been reallocated or not, we introduce a hasReallocated predicate. Now we create an extra action that allows drivers, to switch to any of the trains which have not been reallocated yet and do not have any trains ahead of them. To allow the train to move to another track we introduce a isReallocating and reallocatingToStation predicate. The isReallocating simply tells us if a driver is currently reallocating a train. A reallocation consists of two phases, the first phase moves the train back toward the train station, and the second phase then moves it into another track. The predicate reallocatingToStation will be used to denote which phase of the reallocation the driver is currently in. To be able to go to the next train in order after a reallocation, it is also necessary to explicitly keep track which train the driver should go to after it is done with reallocating.

5.1 Detecting unsolvable conflicts with reallocation

We define an unsolvable conflict as a conflict that can not be resolved with any number of re-allocations. Unsolvable conflict occurs when a train has to leave and the trains ahead on the track are unable to re-allocate themselves over the other tracks. By the pigeonhole principle, if the number of trains ahead of a departing train t_a is higher than the number of free spaces on other tracks s_f, there exists an unsolvable conflict, as for the train to leave $t_a + t_o$ trains have to fit into $s_u + s_f$ spaces, with:
t_a being the number of trains ahead of the departing train,
t_o being the number of trains on other tracks,
s_u being the number of other track spaces in use on other track,
s_f being the number of other track spaces free on other tracks.

Because $s_u = t_o$, with $t_o > s_f$, finding a solution is infeasible, as one track part now should have more than one train for the train to be able to depart. If we find out that state will lead to an unsolvable conflict, it can be pruned.

6 Experimental Setup and Results

This chapter explains the experimental setup and results. Due to time limitations, the implementation of the model and planner for the TUSSP-R was not possible. Consequently, the entirety of this section focuses on TUSSP.

6.1 Planner comparison

The proposed improvements will be implemented in an existing planner to test their effectiveness. This planner is picked from IPC2018 participants in the classical satisfying track. To decide which planner should be extended they have been compared on beforehand. Most of them failed to run on the given domain and problem. The most promising candidates were Saarland, Freelunch-Madagascar, SIW and BFS(F). A small explanation of how these planners work is given in section 2.1.

The domains used in these tests include one disallowing conflicts with numeric fluents as described in section 4.1, one which disallowed conflicts with a predicate, and the original domain which does allow conflicts. The other described improvements were not included in these tests.

Saarland did not perform well on the tested domains, possibly because the only dynamic objects are trains it was not able to split the domain effectively. Freelunch-Madagascar was overall the fastest planner, but is difficult to extend because it completely transforms the domain and SAT solvers themselves are already very efficient. SIW performs the fastest on small problem instances but has problems with finding solutions in bigger instances.

The BFS(f) planner performed decently in smaller problems and was the fastest in bigger problems, and due to its relatively easy-to-understand underlying algorithm and mechanisms most likely the easiest to extend. For these reasons, I choose to improve this planner for our domain.

6.2 Implemented improvements

I build a heuristic that replaces the existing heuristics inside of BFS(f) planner. The improved planner in some instances performs a little bit faster than the base planner, but the result was not significant. The heuristic implemented entailed assessing whether the projected depth matched the actual depth, resulting in a deduction of 5 from the heuristic value in cases of exact correspondence, whereas an addition of 3 was made in instances of disparity. In problem 1 it was about twice as fast as the original planner, I did not test it on any other problem. It is likely that as is it would not improve speed in large domains, a small difference in depth is expected in larger tracks.

The final version of the planner employs the constraint programming approach to construct a preprocessor responsible for the allocation of trains to tracks. The model was defined using MiniZinc. To accomplish this, a Python script was utilized to extract pertinent details from the original PDDL problem file, generating a MiniZinc data file representing the problem instance. Even when dealing with substantial instances (50+ trains) featuring numerous potential conflicts, this approach consistently yielded solutions within a second.

6.3 Generating problem instances

The implemented improvements were tested on five problems with a varying amount of trains. The first problem is problem 1 as shown in figure 1. The other problems have been generated by a script that takes the length of each of the shunting tracks as input. It uses descending arrival order and generates a random departure order. The MiniZinc model to check if this combination of shunting tracks and departure order is feasible. If it does not yield a solution, it starts over and goes through a maximum of 25 iterations. I started out with large tracks, splitting them semi-arbitrarily into more tracks until a feasible problem could be found. An overview of the problems is shown in table 1.

6.4 Final results

A full overview of the results is shown in table 2. The BFS(f) planner is an unchanged version of the planner. BFS(f) 1 uses

	Trains	Tracks
Problem 1	5	[2,2,1]
Problem 2	15	[1,2,5,7]
Problem 3	25	[1,2,5,7,5,5]
Problem 4	40	[5,5,5,5,2,8,3,3,3,1]
Problem 5	50	[5,5,5,5,2,8,3,3,3,1,10]

Table 1: Problem instances used to test improvements

a version of the domain which does not allow for conflicts, BFS(f) 2 uses the full improvements of the domain. CP refers to the final planner, CP:Pre is the preprocessor part of the planner, and CP: Plan is the part of the planner which finds the plan.

7 Responsible Research

All code has been executed on a shared server of the TU Delft, the performance of planners during the tests could have slightly been affected by the current load on this server, however, it is unlikely that big differences in execution time will be found when reproduced as difference in performance between different planners and improvements was order of magnitudes. All code used during the project has been uploaded to the TU Delft Repository.

The present state of the code should not be used to generate shunting plans in for real-life use without thorough examination, as the absolute assurance of plan validity cannot be guaranteed with complete certainty.

8 Discussion

PDDL offers notable advantages in terms of quickly formulating problem definitions, and the decoupling between problem definitions and planners allows for seamless extensions of existing problems without disrupting the solvers' functionality. However, the standardization of the problem definition also comes with limitations which result in the fact that using PDDL to solve a specific problem is often not the most expeditious approach.

9 Conclusions and Future Work

In this paper, a technique is given to identify conflicts for both examined problems. A methodology has been provided to avoid conflicts whenever possible. An introduced model offers a means of resolving conflicts, although further research is required to explore how to effectively use this model in a planner.

P	BFS(f) 1	BFS(f) 2	CP: Total	CP: Pre	CP: Plan
1	1238	4	201	199	2
2	DNF	204	215	212	3
3	DNF	57152	228	218	10
4	DNF	DNF	708	249	459
5	DNF	DNF	1064	258	806

Table 2: Execution time in milliseconds of different planner configurations

During the exploration of conflict detection techniques that detect conflicts earlier in time in the context of heuristic search, I determined that set-partitioning and constraint programming offer more efficient approaches for resolving parking space assignments when we do not allow for reallocations.

As a result, I have approached the problem from a lot of different angles. It could be possible that one of the insights in one of the approaches could also be beneficial in one of the other approaches. For example, it is possible to reduce the domain of possible trains which can be assigned to each track part before starting the running of the constraint programming solver, the generation of possible. Or the track assignment combinations could be translated into a constraint programming problem or SAT.

After finding the parking track allocation in the preprocessor, with the current set of limitations, the rest of the problem has effectively been reduced to multiple path-finding problems. Using a path-finding algorithm instead of the PDDL solver could greatly increase the speed of the planning part in the constraint programming approach.

References

Cornelsen, S., & Stefano, G. D. (2007). Track assignment [Cited by: 29; All Open Access, Bronze Open Access]. *Journal of Discrete Algorithms*, *5*, 250–261. https://doi.org/10.1016/j.jda.2006.05.001

Freling, R., Lentink, R. M., Kroon, L. G., & Huisman, D. (2005). Shunting of passenger train units in a railway station. *Transportation Science*, *39*. https://doi.org/10.1287/trsc.1030.0076

Ghallab, M., Knoblock, C., Wilkins, D., Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Smith, D., Sun, Y., & Weld, D. (1998). Pddl - the planning domain definition language.

Gilg, B., Klug, T., Martienssen, R., Paat, J., Schlechte, T., Schulz, C., Seymen, S., & Tesch, A. (2018). Conflict-free railway track assignment at depots. *Journal of Rail Transport Planning and Management*, *8*. https://doi.org/10.1016/j.jrtpm.2017.12.004

Gnad, D., & Hoffmann, J. (2018). Star-topology decoupled state space search [Cited by: 18; All Open Access, Bronze Open Access]. *Artificial Intelligence*, *257*, 24–60. https://doi.org/10.1016/j.artint.2017.12.004

Stefano, G. D., & Kočí, M. L. (2004). A graph theoretical approach to the shunting problem. *Electronic Notes in Theoretical Computer Science*, *92*. https://doi.org/10.1016/j.entcs.2003.12.020

van den Akker, J. M., Baarsma, H., Hurink, J. L., Modelski, M. S., Paulus, J. J., Reijnen, I. C., Roozemond, D. A., & Schreuder, J. (2008). Shunting passenger trains : Getting ready for departure (O. Bokhove, J. L. Hurink, G. Meinsma, C. C. Stolk, & M. H. Vellekoop, Eds.). *Proceedings of the 63rd European Study Group Mathematics with Industry (SWI 2008, Enschede, The Netherlands, January 28-February 1, 2008)*, 1–19.

Winter, T., & Zimmermann, U. T. (2000). Real-time dispatch of trams in storage yards [Cited by: 61]. *Annals of Operations Research*, *96*(1-4), 287–315. https://doi.org/10.1023/a:1018907720194

Wolfhagen. (2017). The train unit shunting problem with reallocation.