Mixed-direction train shunting with numerical planning

Approach to support train departures at any time during the shunting plan

Ákos S. Sárkány

Supervisors: Sebastijan Dumančić, Issa K. Hanou

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 24, 2023

Name of the student: Ákos Soma Sárkány
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Issa Kalina Hanou, Rihan Hai

Abstract

In this paper, we discuss our novel approach to support mixed-direction shunting in the planning domain for tree-like shunting yards. We explain how we used numeric fluents to improve an existing PDDL domain to support such actions. We elaborate on the underlying conceptual model of our domain and argue how does it accurately represent shunting problems. Furthermore, we describe the experiment that we performed with the MetricFF planner to evaluate our domain and discuss the results of the experiment. Finally, we argue why the MetricFF planning system is not suitable to efficiently support our domain in real-life scenarios.

1 Introduction

The *Nederlandse Spoorwagen (NS)*, the largest railway operator in the Netherlands, manages hundreds of trains daily. However, the demand for rapid transportation varies throughout the day, thus rail operators have to adjust for this by storing the unused units for a shorter or longer period of time. To store the trains, shunting yards exist. These are large fields with multiple tracks, located in close proximity to a station. The routing of the trains from the station to the right shunting track, and matching the supply of the trains with the demand is called the *Train Unit Shunting Problem (TUSP)*, and it was first formulated mathematically by Freling et al., 2005.

Since the TUSP is an *NP-hard problem* it is difficult to find solutions to large-scale problems in reasonable time (Freling et al., 2005). To speed up the computation in practice, railway planners utilize algorithmic support to solve subproblems of real-life instances (Van Den Broek et al., 2022). Such algorithmic support can be provided by planning systems, which are algorithms that use heuristics to determine the sequence of actions that are necessary in order to reach a predefined goal.

In their work, Freling et al. describes one of the main characteristics of the TUSP as follows: "Arrivals and departures of train units might be mixed in time. This implies, within the planning horizon, the first departure might take place before the last arrival has taken place." This is a quite significant feature since this is the case in most real-life situations.

In this research, we focus on mixed-direction traffic in the routing subproblem of the TUSP. More precisely, we are interested in the following question:

Can a planning system efficiently support mixed-directional train shunting in the planning domain?

Our question consists of two parts. Therefore, to answer it, first, we had to answer these two sub-questions:

1. *Can the domain support mixed-direction planning?*
2. *If yes, then can a planning system find a solution for problem instances efficiently?*

In Section 3 we answer the first sub-question. For the second sub-question, we explain the experiment that we used evaluate the efficiency in Section 4, and discuss its results in Section 5.

While answering the questions of this study we only focused on *tree-like* shunting yards, with *last-in-first-out (LIFO)* tracks only (Freling et al., 2005). Another limitation is we considered the routing only from the entrance of the shunting yard and not from the station. Furthermore, we abstracted away from the actual physical length of the trains and assumed that shunting tracks can be split into equal-length "track parts", where each "track part" has the capacity to hold any type of train from the fleet.

2 Relevant Work

In this section, we introduce the relevant concepts to this study. We start with a literature review on the TUSP. Then we explain what planning systems we considered in this research, and elaborate on some techniques that they implement. We focus a short discussion on the MetricFF planning system since this planner is the most relevant for this study Finally, we present an example domain that models the basics of the TUSP in PDDL for the planner algorithm.

2.1 TUSP

In their work, Freling et al., 2005 capture a real-life instance of the shunting problem, and formulate it as a mathematical model that they call TUSP. They divide it into two subproblems, matching supply with demand and routing the train units. They describe the four main characteristics of the problem as follows:

1. **Mixed-direction traffic:** The first departure might take place before the last train arrived.
2. **Train sub-types:** Train units of the fleet can have different properties. This might limit which tracks different types can use.
3. **Multi-way arrival** Tracks in the yard might be approached from either end.
4. **Flexible arrivals and departures** Trains arrival to and departure from the shunt tracks are not exact.

In their study, they use integer programming to solve the problem, with metrics to minimize the cost of shunting. They successfully applied their model to the station of Zwolle and obtained promising computational results.

Lentink et al., 2006 proposed an algorithmic approach to solve the TUSP by dividing it into four sub-problems. Kroon et al., 2008 introduces a new model for the TUSP. This allows them to solve the matching and the parking subproblem in an integrated manner. Haahr et al., 2017 compare multiple methods of solving the TUSP and introduce three novel approaches: a constraint programming formulation, a column generation approach, and a randomized greedy heuristic.

Recently, Mulderij et al., 2020 introduced their approach, which uses *multi-agent path finding (MAPF)*, and it is capable to model a more advanced version of the TUSP: the *TUSP with servicing (TUSS)*. They model the tracks of the shunting yard as a graph $G = (L, E)$, where the nodes in L are the locations a train can be at, and edges in E connect these locations. A train unit is modeled as an "agent" $a_n \in A$, that can be moved through the graph.

They define a TUSP instance as the graph $G = (L, E)$, the vector of the initial locations for each agent $[li_a \in L]$ for all $a \in A$, and the goal locations $[lg_a \in L]$ for all agent $a \in A$. To solve the problem instance, they look for the set of paths p_a for each agent $a \in A$, which takes the agents from the initial location to their goal location conflict-free. This means, two agents cannot be on the same node ($l \in L$) or edge ($e \in E$) at the same time.

2.2 Planning systems

Since this is an open-source project, during the research we only focused on freely available planning systems. Most of such systems were developed for the *International Planning Competition (IPC)* or formerly for the *Artificial Intelligence Planning and Scheduling competition (AIPS)*. These competitions were organized by the *International Conference on Automated Planning and Scheduling (ICAPS)*[1] and they were focused on the presently relevant topics in the planning community. Over these competitions, several successful planning frameworks were developed.

One of the most significant among the early planner is the *Heuristic Search Planner (HSP)*, which was developed by Bonet et al., 1997. As its name suggests, this planner uses heuristics to solve problems, by estimating the distance from the goal and guiding the traversal of the relaxed graph towards states with more optimal values. To achieve relaxation in the search space, they ignore the *delete lists*, which means the negative effects of each applied action are omitted. New relaxed states are generated until all goal conditions are satisfied. To extract the final plan from the relaxation, HSP uses *greedy hill climbing* algorithm.

A successful successor of the HSP, the *Fast-Forward (FF)* system (Hoffmann, 2001) was the most successful planner in the AIPS'00 competition. It uses the same structure and basic principle, that is behind HSP. However, there are three important differences between the systems.

1. The FF heuristic function uses relaxed plan extraction, compared to HSP's difficulty weighing function. In most cases, this results in a better estimate.
2. HSP uses a common hill-climbing algorithm where the successor is chosen randomly, meanwhile in FF, the *enforced hill-climbing (EHC)* algorithm performs a complete breadth-first search, looking for a strictly better evaluation
3. FF uses *helpful action pruning* Hoffmann, 2001, while such techniques are not used in HSP.

It is also important to mention here, that if the EHC algorithm with the pruning fails on a problem, FF performs a complete A* search in order to find a solution.

Later, Helmert, 2006 came up with a different approach to implement the heuristics. In their work, they use *causal graph heuristics* for the search which is different from the FF and HSP approach. Instead of extending the search space forward by chaining actions and ignoring the side effects, the FD system constructs a *causal graph* from the dependencies of precondition and state transitions. This technique is more effective in general since it prevents the large-scale expansion of the search space with the increase of objects, and it depends more on the number of predicates.

[1] https://www.icaps-conference.org/

2.3 MetricFF

The main challenge for the participants of the 3^{rd} IPC was to handle numeric resources in the planning process. One of the two best-performing systems in the competition was the MetricFF planner. To develop the system, Hoffmann, 2003 extended his prior work on the FF framework (Hoffmann, 2001) such that it is capable of handling a restricted set of numeric problems. Hoffmann extends the *ignore delete lists* relaxation approach, such that it ignores the decreasing effects on numerical state variables.

This introduces some limitations on the solvable instances since this approach can only support *strongly monotonic* problems (Hoffmann, 2003). In short, this means numeric conditions prefer higher values and effects should not decrease the value of the numeric variables.

To extract the plan from the relaxed search space, the system implements six search algorithms:

BFS: Applies Best-First search only. Best-First search is an A* strategy with all weights $w_g = 0$.

BFS + H: Applies Best-First search with FF's *helpful action pruning*.

EHC + BFS: Tries Enforced Hill-Climbing with pruning, if fails, restart with Best-First search.

weighted A*: Applies the A* search strategy with weight function $f(s) = w_g \cdot g(s) + w_h \cdot h(s)$.

A* epsilon: Applies the A* search algorithm but also allows the selection of sub-optimal nodes.

EHC + A* epsilon: Tries Enforced Hill-Climbing with pruning, if fails, restart with A* epsilon search.

2.4 PDDL domain for TUSP

For defining the planning domain planners use the *Planning Domain Definition Language (PDDL)*. The language was developed by McDermott and the AIPS-98 Planning Competition Committee, 1998 for the AIPS-98 planning competition to use it for domain and problem specification.

PDDL uses objects, actions, and predicates to define the domain. Objects express the real-life entities the planning problem working with, while predicates are used to describe a temporary state S of the system. Actions operate on objects and predicates, and they describe the transformation of the system from state S_n to S_{n+1}. An action is viable if all of its *preconditions* hold in state S_n, where the action is considered. Once an action is executed in S_n, it affects the value of the predicates, changing them for state S_{n+1}. The new values are defined as the *effects* of the action.

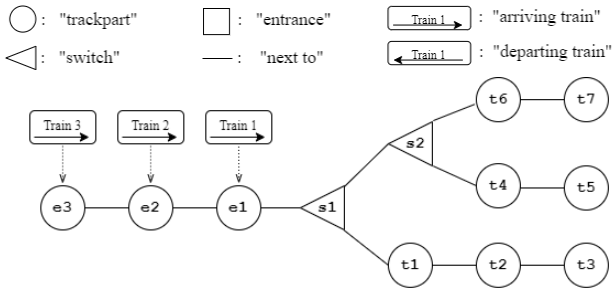
$$S_n + A_{p,e} \rightarrow S_{n+1}$$

Besides specifying the domain, PDDL is also used to define the problem itself. The initial state of the system and the achievable goal is defined in the problem file. The aim of

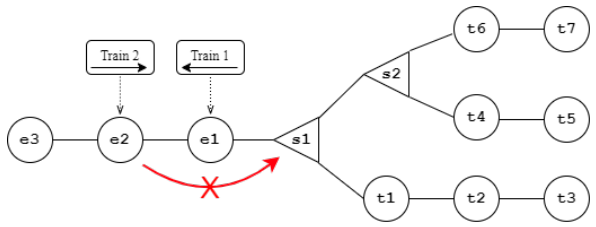
the planning systems is to find the sequence of k *actions* that transforms the initial state to the goal state:

$$S_0 + \sum_{i}^{k} A_{p_i,e_i} \rightarrow S_g$$

In order to successfully execute the previously mentioned planning on a problem instance, both the domain and problem files are needed. Therefore, in the following, we refer to both of them as ”domain”.



(a) Schedule on the example domain. The initial order of the trains on the ”arrival path” (e1, e2, e3) determines their arrival sequence. Here Train 1 will arrive first, while Train 3 last.



(b) Illustration of the limiting factor in the example domain. If a train departs before all the other trains are in the yard, it *blocks* all future arrivals since arriving trains cannot ”jump over” it.

Figure 1: Models of states in the example domain. **(a)** shows a general overview of the domain, while **(b)** highlights the constraint that limits the traffic into one direction.

For this study, we considered an existing PDDL domain that models the TUSP and is compatible with *tree-like* shunting yard layouts. In order to solve the TUSP in this domain, the planning system has to find the sequence of domain *actions* that moves all the trains in the yard, parks them at any track, and moves them out from the yard, all of this in the correct order. This domain was provided by the supervisors of this research and the complete file is presented in Appendix A. Throughout the study, we will refer to this as the ”example domain”.

This domain uses the following objects:

Object: *trackpart, track, trainunit*

Trainunit: *icm, virm, sng, slt*

The *trackpart* and *track* objects are used to define the physical layout of the shunting problem. The domain defines the *nextTo* predicate to indicate links between *tracparts* and the *switch* predicate to distinguish *trackparts* where two *track* converges. Figure 1a illustrates how these predicates are used to model a shunting yard on this domain.

It also shows three trains in an example initial state. It is important to note here, that the schedule for the trains is defined by the order of the *trainunits* on the arrival path (e1, e2, e3). Arrivals are defined by the initial state of the units. Meanwhile, in the goal, the final location for each *train type*[2] is set to define their departure order. This means, both the arrivals and the departures are defined over the same ordered list, the arrival path.

The physics of the domain forbids trains to pass by each other on any single track (here ”track” refers to any segment of consecutive *tracparts*). Trains can only pass by each other when they are on different tracks. From this, it is easy to see that when the first train departs, it blocks any further arrivals since no *trainunit* on the arrival path will be able to ”jump over” it and get into the shunting yard. Figure 1b illustrates this issue.

3 Domain extensions: Domain for mixed-directional shunting

Once we analyzed the example domain, we concluded it cannot support mixed arrivals and departures, since trains cannot pass each other on the arrival path. We realized that this part of the domain does not model reality accurately. In this section, we explain our novel approach to model the TUSP for *tree-like* shunting yards, with support for mixed arrival and departure times. First, we explain the concept behind our model, then we discuss how we managed to implement it in PDDL with the use of numeric fluents. We present our domain in complete form in Appendix B.

3.1 Abstract model

The main goal of our new model was to overcome the limitation presented by the unidirectional track that is used in the example domain to define the schedule. Our approach was to keep the core idea for the scheduling while defining the arrival path as set instead of an ordered list. In Figure 2 we illustrate the step-by-step abstraction of the *physical* domain.

The arrival path in the example domain is a convenient way to define the schedule, however, it fails to accurately model reality. In real life, the shunting yard is connected to the main railway network, where trains can mix freely. Therefore, we decided to model this behavior, by allowing a selected *trackpart* to hold multiple train units. As shown in Figure 2a, we named this node ”entrance”. In our model, all trains are initialized to be at this *trackpart*, and at the end of the plan, all trains should be back here.

Furthermore, we realized that explicit switch nodes are irrelevant to our question. Since we only focus on *tree-like* shunting yards, all switches are located after the entrance and before the start of the shunt tracks, thus they can be visualized as one ”n-way-switch” (Figure 2b). Here, it can easily be seen that an n-way-switch adds no additional flexibility or

[2]In general practice, train units of the same types can be used interchangeably.

constraints to the domain. Omitting the switch only implies that moving units between tracks can be done in one step instead of two or more. Therefore, in our final model (Figure 2c), we discarded it.

In our approach, we kept the main principle of the schedule definition, by defining the arrival and departure sequence based on the order of the trains. However, instead of using the order of the trains on the "arrival path" (since it was omitted), we define the order using integer numbers. The used numbers range between 0 and $n-1$, where n is equal to double the number of trains defined in the domain. This allows us to define mixed arrivals and departures since the numbers used to define both are selected from the same set.

Table 1 shows how the newly introduced numeric approach defines a mixed-direction schedule. This way *Train 1* should depart before *Train 3* arrives.

Unit	Type	Arrive	Depart
Train 1	slt	0	2*
Train 2	slt	1	4*
Train 3	sng	3	5

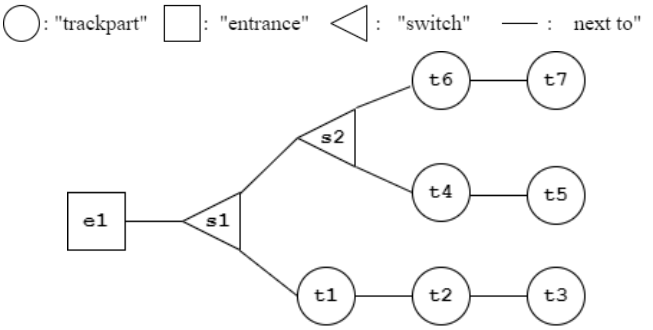
Table 1: Example definition of schedule in the new domain. $*$: since the units with the same type can be used interchangeably, these two departures can be switched freely.

3.2 Implementation with PDDL 2.1

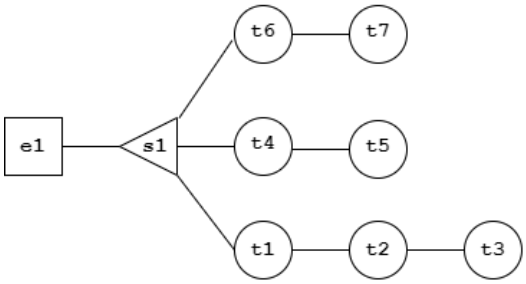
To model the tracks inside the yard we adhered to the example domain. We used the *nextTo* predicate to chain *trackparts*. To avoid collisions on the tracks, each *trackpart* is labeled with a *free* predicate only if no train is located over it. Trains can only be moved to *free trackparts*. To achieve a set-like behavior on the *entrance*, we omitted the use of the *free* predicate when moving trains to and from the *entrance*. This way there is no limit on how many trains can be on the entrance node.

To implement the ordering we used numeric fluents. This feature of PDDL was introduced by Fox and Long, 2003 for the 3^{rd} IPC, to model non-binary resources. To keep track of the order of the trains, we introduce a global counter function, the *timestep*. The value of this function gets incremented every time a train arrives at or departs from the yard. The actual enforcement of the schedule happens in two distinct ways:

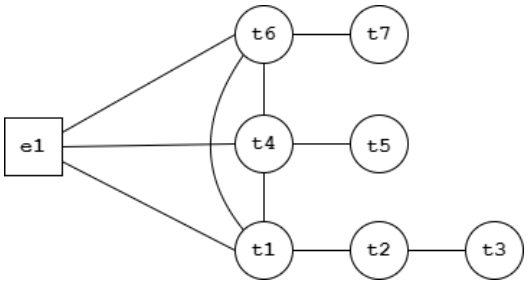
1. Each train is assigned an arrival number as its *arrive* function. The preconditions for the arrival action in the domain require the *timestep* to be equal with the *arrive* function of the given train when the action takes place.
2. To define the departures another technique is used. Trains only get their *departed* value assigned when they moved out from the yard with the corresponding action. However, when defining the goal for the problem, we specify the desired value for each *departed* function. This way the planning system aims to find the correct sequence of actions which results in the assignment of the correct value. It is important to note here, that since *trainunits* of the same type can be used interchangeably, we only specify the train type for the departures.



(a) Since the "arrival path" is connected to the main railway network where trains can mix freely, it can be represented as a set: "entrance".



(b) Since we only consider *tree-like* shunting yards, all switches are between the entrance and the start of the tracks. Therefore, the effect of switches can be combined to an *n-way* switch.



(c) Trains cannot be parked (or even stored) on switches, their only role was to connect the distinct *tracks*. Tracks are now connected directly.

Figure 2: The three layers of abstraction in the new domain. In the final model, we use the model represented in **(c)**

This concludes a positive answer to the first sub-question of this study. Yes, it is possible to support mixed-direction shunting in the planning domain. However, to determine its efficiency we had to perform further experimentation.

3.3 Plan Optimization

Thanks to the new, numeric nation of the domain, we could introduce a new plan optimization metric, the *cost* function. In practice, moving trains between tracks is a more time-consuming and costly operation than moving trains along a single track. Our domain reflects this property by increasing the *cost* more for such actions.

To minimize track changes in the final plan as much as possible, we try to minimize the value of *cost* for the optimal

plans. Therefore, the planner algorithm prefers plans with fewer movements between tracks. However, this metric can only be used with planning systems that support optimization metrics.

4 Evaluation of planning efficiency

In order to successfully answer the other half of our main research question we had to determine if a planning system can *efficiently* support our domain. In this section, we discuss the limitation that the new domain introduced on which planner is suitable and how it affected our decisions in choosing the planner for the experiment. Then, we explain the metrics that we used during the experiment to define the *efficiency* of a search and how they influenced the setup of the experiment. We also elaborate on the problem files that we used for the experiment and discuss how we estimated their difficulties. Finally, we discuss what methodology we used for the execution and present the results.

4.1 Numerical planning with MetricFF

As we discussed in Section 3.2, we had to introduce *numeric fluents* to our new domain to enforce the schedule in our model. This limited the number of planning systems that are suitable to solve problem instances of the new domain. Therefore, it was necessary to use a *numerical planner* for the experiments.

For reproducibility reasons, we had to use a planner that is freely available. Since there was no numerical track on the IPCs since 2003, there are a limited number of open-source numerical planners. Since MetricFF was one of the two best-performing planning systems in the IPC'03, we decided to use it for our experiments.

4.2 Metrics on Efficiency

To determine the efficiency of the planners we decided to use two metrics: the execution speed of the planning algorithm and the cost of the resulting plan. We derived our metrics from practical applications, where it is crucial to get the least expensive (shortest) plan as fast as possible. To record both values, we stored the output of the planners as a text file. From this file we could extract both the *total plan cost* and the *total elapsed time*.

Since we were interested in the efficiency of the planners, we decided to introduce a 30-minute timeout on the planner. In practice, the TUSP is usually solved for a 24-hour period, station by station (Lentink et al., 2006). Compared to that, our examples were reasonably simpler. Therefore, we decided, if an algorithm cannot find a plan for a problem at this time we do not consider it efficient.

4.3 Difficulty of the problems

To get a wider range of evaluation samples, we decided to run the search algorithms on several problem files. Each problem file (except the one with three trains) has the same yard layout: three *tracks*, each with three *trackparts*. The only difference between them is the number of trains and their schedule.

The difficulty of a TUSP increases as the shunting yard approaches its maximal capacity (Mulderij et al., 2020). Therefore, we chose to use the ratio between the number of *trainunits* and the number of *trackparts* to define the difficulty of a problem. However, since all problems have the same amount of *trackparts*, we omitted it, and only used the number of *trainunits* to estimate the difficulty. Based on this, we divided the problems into three categories:

EASY

- Problem3: Only 6 *trackparts*, 3 *trainunits*, maximum 2 trains are in the yard concurrently.
- Problem5: 5 *trainunits*, maximum 4 trains are in the yard concurrently.

MEDIUM

- Problem7 (a-c): 7 *trainunits*, maximum 5 trains are in the yard concurrently.
- Problem8 (a-d): 8 *trainunits*, maximum 6 trains are in the yard concurrently.

HARD

- Problem9 (a-c): 9 *trainunits*, maximum 7 trains are in the yard concurrently.

Originally we intended to introduce problems with more trains, however, since none of the algorithms finished on problems with more than 9 trains, we decided not to include them in our results.

	Problem3		**Problem5**		**Problem7**		**Problem8**		**Problem9**	
	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost
EHC+BFS	0.005s	4	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
BFS	0.006s	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
BFS+H	0.007s	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
weighted A*	0.006s	0	0.076s	6	1.881s	4	n/a	n/a	n/a	n/a
A* epsilon	0.005s	0	55.44s	8	6.438s	8	6.389s	16	n/a	n/a
EHC+A*eps	0.006s	4	57.628s*	8	6.856s*	8	9.354s*	16	n/a	n/a

Table 2: Results of the first round of the experiment. N/a shows where the 30-minute timeout was reached before the algorithm could finish. $*$ shows where the EHC failed and the alternative search method was applied.

	Problem7b		Problem7c		Problem8b		Problem8c		Problem8d		Problem9b		Problem9c	
	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost
weighted A*	0.563s	4	0.102s	16	85.757s	10	23.990s	12	1581s	6	n/a	n/a	n/a	n/a
A* epsilon	1.994s	19	0.818s	16	19.489s	17	11.629s	15	7.151s	14	n/a	n/a	n/a	n/a

Table 3: The results of the second round of the experiment. N/a shows where the 30-minute timeout was reached before the algorithm could finish.

4.4 Setup

We performed our experiment in two rounds. In the first turn, we executed all search algorithms of MetricFF on five problem files, with increasing difficulty. We did this in order to survey the performance of the different search methods. The results of this round are shown in Table 2.

In the second round, we only used the two algorithms that performed the best in the first round, the weighted and the epsilon A* search. Our goal in this round was to determine whether these searches perform similarly on other problems with *MEDIUM* and *HARD* difficulty. The outcomes of the second experiment are shown in Table 3.

5 Discussion

As visible from the results, the EHC algorithm did not perform well on the problems. Arriving and departing action has the same cost (increase *timestep* by one, do not change *cost*), but arrivals accomplish more goals since they park a train by setting the *hasBeenParked* predicate true for the given train. Therefore, the heuristic function prefers them over departures. This way the EHC runs into local maximums where all trains are parked with the least moves, then fails to return to a previous state to continue climbing toward the goal.

The BFS search suffers from a different problem. Even though it is an A* strategy (Hoffmann, 2003), it uses *zero node weights* and only considers the depth of a node when choosing the next, thus possibly selecting non-optimal nodes. This way it can consider nodes that do not lead toward the goal and therefore while traversing the relaxation graph, it runs into more dead-end states.

On the contrary, A* searches with weights perform significantly better. Since the A* epsilon also considers sub-optimal nodes while advancing the goal, for more complex problems, it can find a plan faster than the weighted version of the A*, which only considered optimal nodes. However, the cost of the solutions produced by the epsilon variant is higher. This is also due to the consideration of sub-optimal nodes since they can increase the overall cost unnecessarily.

5.1 Limitations

Since the relaxation technique of the MetricFF system only supports *strongly monotonic tasks*, in a problem where a numeric resource both grows and shrinks throughout time, a real plan might not appear among the relaxed plans (Hoffmann, 2003). In this case, the search can result in a failure even though there exists a valid plan. This did not have a particular impact on our research since all new functions that we introduced are strongly monotonic. However, we realized that this constraint of the MetricFF system highly limits the future extension of our domain. We realized, for example, the capacity restraints of a *track* cannot be modeled due to this limitation, since the available space on the tracks both increases and decreases over time.

Furthermore, due to the flexibility of the numerical values, the relaxation might accomplish all goals, even if in a real plan it is impossible. This way the decision on feasibility is not polynomial (Hoffmann, 2003). The worst case complexity of the A* is bounded by $O(b^d)$, where b is the average number of successor nodes in the graph, and d is the depth (length) of the solution path (Russell J & Norvig, 2003). However, if there is no solution exists (thus the length of the solution is $d = \inf$), the algorithm will have to traverse the entire relaxed graph to conclude the absence of a solution.

Furthermore, since the length of the solution is correlated with the number of *trainunits* in the problem, the execution time of the search expands exponentially with the increase of *trainunits*. This significantly reduces the effectiveness of the planning system for problems with many trains.

5.2 Interpretation of results

Even though the difficulty of a TUSP instance is related to the capacity of the yard, when it is solved by a planning system, the complexity of the search algorithm is more relevant to its performance. Since the A* strategies use heuristics, their execution time varies for each problem, and it depends on the merit of these heuristics. In the worst case, the complexity of the algorithm grows exponentially with respect to the solution length. Furthermore, if there is no solution exists for a problem, the algorithm has to traverse the entire relaxed graph to prove the absence of any solution. This makes the efficiency unreliable since the length of the shortest solution is not known in advance. Therefore, we concluded it is not efficient to use the MetricFF system with our shunting domain to solve real-life instances of the TUSP.

6 Responsible Research

We had to create the problems for the evaluation with great care. Since we wanted to test the general performance of the MetricFF system on our domain, we had to implement solvable, but arbitrary problems. It was necessary to make them arbitrary, to avoid confirmation bias, since it would have been misleading to only feed well-performing problems into the system to gain promising results. To avoid this, we run the algorithm on multiple problem files with the same amount of trains. This revealed that the performance of the search can vary significantly between different problems, and avoided us to derive false conclusions.

6.1 Reproducibility

In order for one to reproduce our experiments we provide all of our source code in our online repository[3]. This repository contains the full PDDL file of our domain, all problem files that were used for the experiments, and the outputs. To execute the MetricFF planning system we used the `mapfw.ewi.tudelft.nl` server, which uses an AMD EPYC 7702P 64-Core Processor, from which one 2GHz core was dedicated to run an instance of the MetricFF system. If someone runs the MetricFF planner on the `mapfw` server (or any computer with equivalent computational power) with the same problems as we did, one should get very similar results for their experiments.

7 Conclusions and Future Work

In this paper, we presented our novel approach to support mixed-direction train shunting in the planning domain. We extended an example domain, that was provided to us by the supervisory team of this research. To achieve mixed-direction traffic we introduced a new, more abstract model, that uses a *set* to express the main railway network instead of an ordered list. Furthermore, to enforce the schedule, we used the *numeric fluents* from PDDL 2.1. From this, we concluded that it is possible to support mixed-direction traffic in the domain.

To evaluate the efficiency of our domain we performed an experiment with the MetricFF planning system. We evaluated the performance of the planner on several problem files, that were defined on our new domain. The results showed that the A* search strategies perform reasonably well on small-scale problem instances. However, due to the exponential worst-case time complexity of the algorithms, their efficiency gets significantly worse on problems with more trains. From this, we concluded that the *MetricFF system cannot support efficiently our domain* for large-scale instances of the TUSP, thus *it is not practical to use it in real-life applications*.

This year, the IPC has a numeric track again. Hopefully, the competition will result in the introduction of new, more efficient numerical planning systems. In the future, our experiment can be repeated with the latest most efficient numerical planner, to possibly, obtain better results.

References

Bonet, B., Loerincs, G., & Geffner, H. (1997). Hsp: Heuristic search planner. https://bonetblai.github.io/reports/aips98-competition.pdf

Fox, M., & Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, *20*, 61–124. https://doi.org/10.1613/jair.1129

Freling, R., Lentink, R., Kroon, L., & Huisman, D. (2005). Shunting of passenger train units in a railway station. *TRANSPORTATION SCIENCE*, *39*(2), 261–272. https://doi.org/10.1287/trsc.1030.0076

Haahr, J. T., Lusby, R. M., & Wagenaar, J. C. (2017). Optimization methods for the train unit shunting problem. *European Journal of Operational Research*, *262*(3), 981–995. https://doi.org/10.1016/j.ejor.2017.03.068

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246. https://doi.org/10.1613/jair.1705

Hoffmann, J. (2003). The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, *20*, 291–341. https://doi.org/10.1613/jair.1144

Hoffmann, J. (2001). Ff: The fast-forward planning system. *AI Magazine*, *22*(3), 57. https://doi.org/10.1609/aimag.v22i3.1572

Kroon, L. G., Lentink, R. M., & Schrijver, A. (2008). Shunting of passenger train units: An integrated approach. *Transportation Science*, *42*(4), 436–449. https://doi.org/10.1287/trsc.1080.0243

Lentink, R. M., Fioole, P.-J., Kroon, L. G., & van't Woudt, C. (2006). Applying operations research techniques to planning of train shunting. *Planning in Intelligent Systems: Aspects, Motivations, and Methods*, 415–436.

McDermott, D., & the AIPS-98 Planning Competition Committee. (1998). Pddl— the planning domain definition language. *Technical Report, Tech. Rep.* https://www.cs.cmu.edu/~mmv/planning/readings/98aips-PDDL.pdf

Mulderij, J., Huisman, B., Tönissen, D., van der Linden, K., & de Weerdt, M. (2020). Train unit shunting and servicing: A real-life application of multi-agent path finding. *arXiv preprint arXiv:2006.10422*. https://arxiv.org/abs/2006.10422

Russell J, S., & Norvig, P. (2003). Artificial intelligence: A modern approach (2nd). *Artificial Intelligence and Machine Learning Book*, 97–104.

Van Den Broek, R., Hoogeveen, H., Van Den Akker, M., & Huisman, B. (2022). A local search algorithm for train unit shunting with service scheduling. *Transportation Science*, *56*(1), 141–161. https://doi.org/10.1287/trsc.2021.1090

[3] https://github.com/cakasa/rail-planning

A The example domain

```
(define (domain example)
(:requirements :adl)
(:types
    trackpart track trainunit - object
    icm virm sng slt - trainunit
)
(:predicates
    (parkedOn ?x - trainunit ?y - track)
    (onTrack ?x - trackPart ?y - track)
    (at ?x - trainunit ?y - trackpart)
    (hasBeenParked ?x - trainunit)
    (nextTo ?x ?y – trackpart)
    (free ?x - trackpart)
    (onPath ?x)
    (switch ?x)
)
(:action move-to-track
    :parameters (?train - trainunit ?from ?to - trackpart ?t - track)
    :precondition (and (at ?train ?from) (free ?to)
                    (nextTo ?from ?to) (onTrack ?to ?t)
                    (switch ?from))
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (free ?from) (not (free ?to))
                    (hasBeenParked ?train) (parkedOn ?train ?t))
)
(:action move-from-track
    :parameters (?train - trainunit ?from ?to - trackpart ?t - track)
    :precondition (and (at ?train ?from) (free ?to)
                    (nextTo ?from ?to) (onTrack ?from ?t)
                    (switch ?to))
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (free ?from) (not (free ?to))
                    (not (parkedOn ?train ?t)))
)
(:action move-along-track
    :parameters (?train - trainunit ?from ?to - trackpart ?t - track)
    :precondition (and (at ?train ?from) (free ?to)
                    (nextTo ?from ?to) (onTrack ?from ?t)
                    (onTrack ?to ?t))
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (free ?from) (not (free ?to)))
)
(:action move-to-departure
    :parameters (?train - trainunit ?from ?to - trackpart)
    :precondition (and (at ?train ?from) (free ?to)
                    (nextTo ?from ?to) (onPath ?to)
                    (forall (?unit - trainunit) (hasBeenParked ?unit)))
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (free ?from) (not (free ?to)))
)
(:action move-on-arrival
    :parameters (?train - trainunit ?from ?to - trackpart)
    :precondition (and (at ?train ?from) (free ?to)
                    (nextTo ?from ?to) (not (hasBeenParked ?train))
                    (onPath ?from))
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (free ?from) (not (free ?to)))
)
)
```

B Our domain

```
(define (domain tusd)
(:requirements :equality :strips :typing :quantified-preconditions
:conditional-effects :fluents)
(:types
    trackpart track trainunit - object
    icm virm sng slt - trainunit
)
(:predicates
    (onTrack ?x - trackPart ?y - track)
    (at ?x - trainunit ?y - trackpart)
    (nextTo ?x ?y – trackpart)
    (entrance ?x - trackpart)
    (free ?x - trackpart)
    (hasBeenParked ?x)
)
(:functions
    (cost)
    (timestep)
    (arrive ?x - trainunit)
)
(:action move
    :parameters (?train - trainunit ?from ?to - trackpart ?t - track)
    :precondition (and (at ?train ?from) (free ?to) (nextTo ?from ?to)
                    (onTrack ?from ?t)(onTrack ?to ?t))
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (free ?from) (not (free ?to))
                    (increase (cost) 1))
)
(:action move-between-tracks
    :parameters (?train - trainunit ?from ?to - trackpart ?t1 ?t2 - track)
    :precondition (and (at ?train ?from) (free ?to) (nextTo ?from ?to)
                    (onTrack ?from ?t1)(onTrack ?to ?t2))
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (free ?from) (not (free ?to))
                    (increase (cost) 2))
)
(:action move-to-departure
    :parameters (?train - trainunit ?from ?to - trackpart)
    :precondition (and (at ?train ?from) (entrance ?to)
                    (nextTo ?from ?to))
    :effect (and (at ?train ?to)
                    (not (at ?train ?from))
                    (free ?from)
                    (increase (departed ?train) (timestep))
                    (increase (timestep) 1))
)
(:action move-on-arrival
    :parameters (?train - trainunit ?from ?to - trackpart)
    :precondition (and
                    (= (timestep) (arrive ?train))
                    (at ?train ?from) (free ?to)
                    (nextTo ?from ?to) (entrance ?from)
                )
    :effect (and (at ?train ?to) (not (at ?train ?from))
                    (not (free ?to)) (hasBeenParked ?train)
                    (increase (timestep) 1))
)
)
```