Utilizing General Planners to Solve the Train Unit Shunting Problem Extended with Servicing

Bozhidar Andonov [1]

Supervisors: Sebastijan Dumančić[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: Bozhidar Andonov
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Issa Hanou, Rihan Hai

Abstract

When not in use, trains are stored at shunting yards, where they may need servicing. Planning for where the rolling stock should be parked during its stay at the yard is known as the Train Unit Shunting Problem (TUSP) and it is NP-hard. Algorithmic planners can assist with this computationally difficult task by providing a sequence of actions to be executed for the trains in the shunting yard. To output such a sequence, planners should be given a domain and an instance of it as input, both of which can be defined in the Planning Domain Definition Language (PDDL). In this paper, we first formalize the TUSP extended with servicing and implement a domain for it in PDDL. Then, we compare four planners submitted to the International Planning Competition 2018 against 9 instances of the domain. The best of these planners reduces the problem to the Boolean Satisfiability Problem. We improve its performance with 31.5% by removing redundant clauses and an irrelevant computational element of the reduction.

1 Introduction

Trains have to be stored at so-called *shunting yards* when not scheduled for operation. Finding a plan for where the arriving vehicles should be parked, how they should be recomposed, and when they should depart this yard is known as the **Train Unit Shunting Problem**, or **TUSP** [1]. In addition, the rolling stock may need to undergo maintenance (also known as servicing), such as cleaning or repairs, which can only be executed on designated tracks.

Finding plans for shunting yards is mostly done by human planners with little support from computers [2]. This task becomes more difficult and tedious as the size of the yards and the number of trains increases. In fact, the TUSP is an NP-hard problem [1], which is further complicated by the addition of servicing. Its complexity incentivizes the automation of the planning procedure so that it can be done more swiftly and reliably.

Algorithmic planning for the TUSP extended with servicing can be achieved in numerous ways. For instance, J. Mulderij et. al. [3] give a real-world application of multi-agent path-finding algorithms to solve this problem. Another example would be employing local search algorithms and integer linear programming which is what Broek et al. [2] do.

Yet another approach is to translate the problem into a language that general planners can understand. A planner is a program that takes a domain and an instance of it and outputs a sequence of actions that should be performed in order to go from the starting state to the goal state of that instance. Therefore, we can provide a planner with a shunting yard and lists of arriving and departing trains and it will produce a sequence of actions, such that trains arrive in the correct order, have been serviced before departure, and have left the shunting yard at the correct time. All of the described constraints can be encoded into the domains and instances that planners take as input, using a language called the Planning Domain Definition Language (PDDL). If we wanted an even better performance of a specific planner, we could tweak it so that it deals better with the domain.

To our knowledge, not much research has gone into the specialization of general planners to work in the TUSP domain. The acquisition of rolling stock by Dutch Railways (NS) and the increasing demand for transportation requires that planning for these shunting yards gets automated promptly. It could be possible that the optimization of an algorithmic planner beats existing methods, hence why it is crucial that this option is explored and evaluated.

Thus, the research question to be answered can be formulated as follows:

Can general planners be improved for the TUSP extended with servicing?

To answer this question we first give a translation of the domain into PDDL. Afterward, we create 9 instances of increasing complexity of that domain. These are then used to compare 4 planners submitted to the International Planning Competition 2018 based on a metric we define in Section 4.2. The best-performing one reduces the problem of planning to the Boolean Satisfiability Problem (or SAT). We improve it by removing a computation which is unnecessary for our domain and by deleting a set of clauses that are redundant. We then compare the modified planner's performance to its original version and show the tweaked planner is 31.5% better.

The rest of the paper is structured as follows. The next section explores existing literature and provides background information on the Planning Domain Definition Language and its usage by planners. In Section 3, we formalize the train unit shunting problem extended with servicing. The methodology used for conducting the experimental work is described in Section 4. A metric for planner comparison is also devised in that Section. The implementation of the problem's domain and its instances is outlined in Section 5. The planners are compared and the best one is improved in this section, as well. The ethical aspects of the research are part of Section 6. The paper concludes with future work that could be done to enhance the planner further.

2 Background

In this Section, we discuss the background, which is necessary for understanding aspects of the experiments. The first Subsection gives an overview of the existing literature, related to this paper. Next, we provide an extensive explanation of the functionality and purpose of the Planning Domain Definition Language. Finally, we describe how planners use this language and give a few commonly-used planning strategies.

2.1 Related work

To our knowledge, the TUSP (without servicing) is first formalized by Freling et al. [1]. They give an overview of the problem in a constraint-free form, allowing for arbitrary shunting yards, train configurations, and arrival and departure schedules. They also prove it is NP-hard by reducing it to the 0-tram dispatching problem, introduced by Winter [4]. Due to its complexity, large problem instances cannot be solved in

a sensible time, and thus, we have to add a few restrictions to the problem definition.

Hanou et al. [5] study the feasibility of TUSP instances. The constraints they introduce are similar to ours, i.e. they consider only tree-like shunting yard layouts and trains that are of unit length. We have thus depicted the problem in a similar way with the exception of servicing which they do not support. A full description of the problem including all constraints is provided in Section 3.

In order to solve the servicing version of TUSP, a lot of methods that do not rely on general planners, have been devised. In addition to employing multi-agent path finding [3] and local search algorithms [2], previous work has tried using deep reinforcement learning [6]. Xiaoming et. al. [7] employ a multi-commodity network flow model of the problem and solve it using a custom relaxation heuristic.

To our knowledge, solving TUSP with out-of-the-box general planners has not been considered previously. However, it may be the case that existing planners can deal well with this problem, considering its natural translation into the Planning Domain Definition Language. Therefore, in this paper, we deviate from the researched methods and focus on comparing and modifying existing planners, instead.

2.2 The Planning Domain Definition Language

The Planning Domain Definition Language is a standardized planning language used to model domains and instances of these domains. It was introduced in 1998 by Ghalib et al. [8] in an attempt to systemize the domain representations that planners used. It also allowed the first International Planning Competition (IPC) to take place. Since its initial release, PDDL has gone through several major updates that introduce new features. However, we have not employed any of these in our research and have thus left them out. In the remainder of this Subsection, we discuss PDDL version 1.2.

PDDL is based on predicate logic. That is, its foundation are statements that are either true or false at the time of execution. For instance, $At(x, y)$ could be interpreted as x is at y. Logical operators can be used to concatenate different statements. For example, $At(x, y) \wedge Robot(x) \wedge Location(y)$ would mean that robot x is at location y. Quantifiers can be used alongside predicates, as well - $\forall x \exists y(Robot(x) \rightarrow (Location(y) \wedge At(x, y)))$ would mean that every robot is at some location.

Predicates in PDDL are defined by putting them in the `:predicates` section of the PDDL file. Each predicate needs to be given a name, followed by the variables it takes, alongside their types. In particular, we could define the predicates from the previous examples in the following way:

```
(:predicates
    (robot ?x - object)
    (location ?y - object)
    (at ?x ?y - object)
)
```

If we only wanted to use the At predicate for robots and locations, we could simplify this code by defining the types $robot$ and $location$, as follows:

```
(:types
    robot location - object
)
(:predicates
    (at ?x - robot ?y - location)
)
```

Actions can now be defined based on these predicates. An action has a name and takes the parameters it acts upon as arguments. In order to execute an action, the precondition consisting of predicates is checked. If it holds, the action is performed and the world state is modified (by changing the truth values of other predicates). As an example, take the following code:

```
(:action move-robot
    :parameters
        (?r - robot ?x ?y - location)
    :precondition (at ?r ?x)
    :effect (and
        (not (at ?r ?x))
        (at ?r ?y)
    )
)
```

When carried out, this action takes a robot and two locations and changes the location of the robot to the second one, given that it is at the first one.

While domains define how the world functions, instances of this world should be provided so that planners have something concrete to work with. An instance defines objects that exist in the world, a starting state, and a goal state. The following code defines an instance where a robot starts out in one location and its goal is a second one.

```
(:objects
    r - robot
    loc1 loc2 - location
)
(:init (at r loc1))
(:goal (at r loc2))
```

A plan satisfying the constraints of this instance can be of the following form:

```
1. move-robot r loc1 loc2
```

It would move the robot to the second location, satisfying the goal and thus yielding a valid plan. It is important to note that this is not the only valid plan, however.

```
1. move-robot r loc1 loc2
2. move-robot r loc2 loc1
3. move-robot r loc1 loc2
```

The plan above would first move the robot to the second location, then the first one, then the second one again. The goal state is reached, so the plan is valid. However, the number of steps is higher, rendering this sequence of actions suboptimal.

2.3 Algorithmic planners

General algorithmic planners often utilize PDDL for the definition of problems they solve. As input, they take a domain, consisting of predicates and possible actions, and an instance

of that domain. The instance is converted into an internal representation and is solved using various algorithms and heuristics. The planner then outputs a file containing a sequence of actions whose execution takes the initial state to the goal state.

These planners make use of different strategies when they solve problem instances. A substantial number employ a variation of the heuristic-based fast-downward approach, introduced by M. Helmert [9], as it has proven quite effective when it comes to general planning.

Other planners reduce planning to well-known problems such as the Boolean Satisfiability Problem (SAT, i.e. given sets of boolean literals and clauses, does there exist an assignment of the literals to boolean values such that the conjunction of all clauses is made true [10]). One such planner is the Freelunch planner that team 4 submitted to the International Planning Competition 2018. It first generates literals out of the predicates and actions (also called operators), based on the objects of the instance, and eliminates invariant ones. Then, it produces the initial and goal state out of these literals. Afterward, for each time step, it adds the following clauses.

- If an action is executed at time t, its precondition holds at time t and its effect is made true at time $t + 1$.
- If a predicate is made false (true) at time step $t + 1$, then an operator that made it false (true) should be executed at time step t

Here, timesteps are an abstract term due to the $\exists$-step parallel semantics that the Freelunch planner utilizes [11]. In sequential semantics, each time step would correspond to exactly one action. However, the idea of $\exists$-step parallel semantics is to distribute operators into sets, which are executed one after the other, such that the operators in each set do not affect each other. Timesteps here, therefore, refer to the sequence of the $sets$ of operators and not to the individual actions. Additional $chain$ clauses are also added to guarantee the $\exists$-step semantics of the plan. These are produced based on the strongly connected components of the operator graph, as explained in [12].

After the reduction is complete, it is fed as input to an incremental SAT-solver that starts with one clause and checks how that can be made true. Using this information, it builds up the complete expression one clause at a time to check whether it is satisfiable as a whole. A plan is then generated based on the operators that are required to be true at certain times. This procedure is tried out with 0, 1, 2... timesteps until a plan is found or a timeout is reached.

Newly-developed efficient planners are often submitted to the International Planning Competition, where they are benchmarked against different problems in multiple tracks. Two of these tracks are satisfiability (whether a planner can output **any** plan for the domain instances) and optimality (whether planners output an **optimal** plan for domain instances). We are interested in the former one as optimality can be even more time-consuming to construct for the domain of TUSP extended with servicing. We also only look at planners from the 2018 edition of the IPC as those have been freely provided to us. An overview of the planners in that competition can be seen in the contest's summary, compiled by Torralba et al. [11].

3 Problem Description

The purpose of this Section is to give a thorough description of the constrained version of the TUSP extended with servicing. First, restrictions on the problem are introduced so planners can solve instances in a feasible amount of time. Next, the problem is formalized and an extensive example, illustrating an instance of the domain is given.

3.1 Formalizing the problem

Before formalizing the problem, restrictions on the types of shunting yards and trains should be placed. We do this in order to alleviate the computational complexity of the general TUSP problem. The constraints we consider are as follows:

- Mostly shuffleboard (tree-like) shunting yard layouts are considered. That is, there is a single entry point into each track, with only a single track being the root one. Each track can branch into multiple other ones. This is sensible, as a vast number of real-life shunting yards follow that layout. One of the shunting yards we consider has tracks, that violate this condition. However, these tracks are few and are meant to present a challenge to the planners.
- Trains start their departure after all rolling stock has arrived at the shunting yard. This is also not unusual as shunting yards are mostly utilized during night-time when trains have finished their operations.
- Trains are of unit length, meaning each train takes the same amount of space as all other ones. Train lengths exponentially increase the time required to solve problem instances, hence why they are omitted.
- Tracks consist of a finite number of track parts, where one track part is the size of one train unit. Combined with the restriction above, this implies that a track of size n can be a parking spot for at most n train units at the same time.
- Trains can only leave the shunting yard once they have undergone all services they require.
- Services are provided by tracks, not by track parts. Such tracks also double as parking spots for the rolling stock.
- A general order of the arrival and departure of the trains is provided instead of set times. This abstraction does not strip away too much from the problem as duration costs can be assigned to different actions, yielding a timed plan, as required.

Now that we have imposed restrictions on the problem and it is more feasible for a planner to solve, we can proceed with its formal definition.

Problem: Constrained TUSP with servicing
Input: $I = (T, A, D, S)$, where

- T is a graph (V, E), depicting the shunting yard and its layout. Here V is the set of tracks of the shunting yard and E is the set of tracks that are adjacent to each other. Each track is represented by a 3-tuple, consisting of the name of the track, its length in number of track parts, and a set of services it provides.

- A is a sequence of arriving trains where the first arriving train is the one at the beginning of the sequence;
- D is a sequence of departing trains where the first departing train is the one at the end of the sequence;
- S is a set of tuples, where each tuple links a train with a set of services that the train requires during its stay at the shunting yard.

Output: A sequence of moving and servicing actions for shunting yard T, such that trains arrive from the first track part of the root track and depart from that same part based on the arrival and departure orders A and D. Each train also needs to be serviced according to S before departing.

3.2 An example

To give the reader a better understanding of the description of the problem, we provide a small example. We then explain it through text and figures.

Take the following instance of the problem: $I = (T, A, D, S)$

- $T = (V, E)$
  - $V = \{(a, 1, \emptyset), (b, 1, \{cleaning\}), (c, 2, \emptyset)\}$
  - $E = \{(a, b), (a, c)\}$
- $A = D = [t_1, t_2]$
- $S = \{(t_1, cleaning)\}$

This is a 4-tuple of the sets/sequences T, A, D and S. Here, T is the layout of the shunting yard. It consists of 3 tracks (a of length 1, b of length 1, and c of length 2). Tracks a and c are regular tracks, while b is a service track for cleaning. Track a is adjacent to both b and c. From A and D (the arrival and departing train sequences), we learn that there are two trains that will stay at this shunting yard, namely t_1 and t_2. Train t_1 arrives first and leaves last. S contains a single servicing operation and that is the cleaning of t_1. Figure 1 gives a visual representation of this instance.

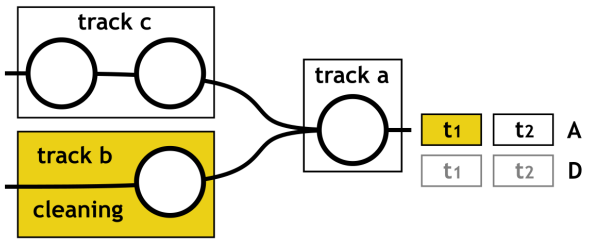

Figure 1: An instance of the formalized problem with tracks a, b and c, where b supports train cleaning. On the right, the arriving train sequence A with t_1 requiring cleaning can be seen. Below A is the departing train sequence D.

4 Methodology

The aim of this Section is to establish the methodology with which the main research question is tackled. First, the main steps of the experimental work are outlined. Then, we devise a metric that is used to compare existing general planners.

4.1 Experimental procedure

The procedure for conducting the experiments can be divided into 4 steps. The following are evaluated in Section 5:

1. **Implement the domain:** The TUSP extended with servicing domain is realized in PDDL based on the formal problem description provided in Section 3.
2. **Define domain instances:** We define 9 instances of the domain based on 3 shunting yards, the biggest one of which is inspired by one of the shunting yards at the Hague Central Station.
3. **Compare existing planners against instances:** We choose 4 planners from the International Planning Competition 2018 to be compared and run them against the defined instances. We compare the results based on the metric described in Section 4.2 and determine the best planner.
4. **Improve best planner:** The planner which scores best reduces the problem to SAT. We modify the algorithm by removing redundant computation and then show that the tweaked version performs better.

4.2 Metric for planner comparison

One of the steps that is executed during the experimental work is comparing the planners. However, in order to objectively quantify each planner's performance, a metric should be devised.

Planners are usually evaluated against 3 criteria [13]. The one that is employed most often is the ratio of solved instances. We define a function R to indicate whether a planner P is successful in solving an instance I in Equation 1.

$$R(P, I) = \begin{cases} 0 & \text{if } P \text{ has no plan for } I \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

The second metric is the time a planner takes to generate a plan. This is applicable only when the planner is able to find a solution to an instance. It is often the case that planners time out on bigger problems. That would be considered an inability to generate a plan, so this metric is not defined in that case. We give a function Q to indicate how a planner P performs for instance I compared to all planners Π based on the time it took to generate a plan. It is given in Equation 2. Here, $t(P, I)$ is the time planner P took for generating a plan for instance I in milliseconds. The metric calculates where a planner stands in time compared to the best planner (which will have a score of 1) and the worst one (which will have a score of 0). To exemplify, if the best planner took $100ms$ and the worst one - $200ms$, a planner with a time of $140ms$ would have a score of $\frac{200-140}{200-100} = 0.6$.

$$Q(P, I, \Pi) = \frac{\max_{P' \in \Pi} t(P', I) - t(P, I)}{\max_{P' \in \Pi} t(P', I) - \min_{P' \in \Pi} t(P', I)} \quad (2)$$

The final criterion that is used to evaluate a planner's performance is the cost of the plan. It is most often expressed in the number of actions the generated plan contains. Similarly to timing, this is also not defined when a solution cannot be

found. Also alike the previous criterion, we provide a function C to indicate how a planner P performs on instance I compared to all planners Π based on the cost of the generated plan, described in Equation 3. Here, $c(P, I)$ is the number of actions the plan for instance I, generated by planner P, contains. Note that this is almost identical to Q. The only difference is the used metric.

$$C(P, I, \Pi) = \frac{\max_{P' \in \Pi} c(P', I) - c(P, I)}{\max_{P' \in \Pi} c(P', I) - \min_{P' \in \Pi} c(P', I)} \quad (3)$$

Based on the provided insights and the defined functions, we propose a generalized function F to compare planners with. It assigns a score between 0 and 1 to each planner P out of Π for instance I. It is defined in Equation 4.

$$F(P, I, \Pi) = R(P, I) \cdot \frac{2 \cdot Q(P, I, \Pi) + C(P, I, \Pi)}{3} \quad (4)$$

Here, R, Q and C are the functions defined in Equations 1, 2 and 3, respectively. Note that a score of 0 is assigned to planners which cannot solve the instance due to R. If only one planner is able to solve a certain instance, it is assigned a score of 1 for that instance. However, the best planner may not always have a score of 1. We also assign a higher weight to the time it took to find a plan, as we are interested in finding *some* plan, which is not necessarily optimal in number of actions. To evaluate a planner against all instances, we average all of these instance scores. The planner with the highest overall score is deemed superior.

5 Experimental Setup and Results

We now set up and conduct the experimental work, as described in Section 4.1. Section 5.1 gives a PDDL implementation of the formalized TUSP with servicing problem. In Section 5.2 we create several instances of the domain, which are used in Section 5.3 to compare planners against the metric we devised. The best planner is then improved and evaluated against the same criterion in Section 5.4.

5.1 Domain implementation

We have been provided with a default implementation of the domain by the supervisors of this research. It has been used in the feasibility study of tree-like shunting yard instances by Hanou et al. [5]. As already mentioned, the constraints that paper follows regarding the problem are similar to ours, hence it is not unreasonable to start with this implementation. We tweak it in order to fully conform to our problem definition. The original domain can be found in Appendix A.1.

As can be seen in Figure 2, there are 4 main types that the modified domain utilizes - `trackpart`, representing a single track part, where only one train unit can be parked; `track` - a track in the shunting yard which consists of at least one track part; `trainunit`, which is a single train; and `service`, which describes services that the tracks provide or trains need. In addition, each `trainunit` can be of one of 4 subtypes - `icm`, `virm`, `sng` and `slt`.

```
trackpart track trainunit service - object
icm virm sng slt - trainunit
```

Figure 2: The types used in the modified version of the domain.

The predicates that are needed to establish the constraints of the problem are defined in Figure 3. Some of these have already been hinted at when describing the types. The `nextTo` predicate determines whether two track parts are next to each other. These need not necessarily be on the same track. The `onTrack` predicate signifies that a track part is on a certain track. To decide whether a track part is free we can use the eponymous predicate. If it is occupied, the `at` predicate will return true for *some* train unit for that track part. A train has three more properties - `hasBeenParked`, which is false only when the train has not entered the shunting yard yet; `hasDeparted`, which is true if the train has already departed the yard; and `needsService` which determines whether a train unit needs a certain service. Whether a track provides any services can be learned by querying the `isServiceTrack` predicate.

```
(nextTo ?x ?y - trackpart)
(onTrack ?x - trackpart ?y - track)
(free ?x - trackpart)
(at ?x - trainunit ?y - trackpart)
(hasBeenParked ?x - trainunit)
(hasDeparted ?x - trainunit)
(needsService ?x - trainunit ?y - service)
(isServiceTrack ?x - track ?y - service)
(onPath ?x - trackpart)
```

Figure 3: The predicates used in the modified version of the domain.

Since Hanou et al. [5] want to preserve the arrival and departure order of the trains, as they enter and leave the shunting yard, they introduce a *path* track of length $|A|$, which we reuse. In the beginning, trains are positioned there according to their arrival sequence (A). After all vehicles have departed, they will be on the path once again, this time ordered according to their departure sequence (D). The *path* track is connected to the root track of the shunting yard and is treated as a regular track with two exceptions. Firstly, trains cannot be parked there. Secondly, if a train has been parked on a regular track and moves to the *path* track, it is considered departed. However, just like regular tracks, the *path* consists of track parts. Thus, the `onPath` predicate determines whether such a track part lies on the *path*.

Additionally, two predicates from the initial domain have been omitted. The first one is `parkedOn`. After the modification of the original domain, this predicate was never checked in action preconditions and was not used as a goal in the domain instances. Therefore, it was sensible to delete it. The second one is the `switch` predicate. As regular track parts can be next to each other even if they are part of different tracks, switches are redundant and have, therefore, also been excluded from the domain.

Now, we define the actions that are included in the

tweaked version of the domain. The `move-on-arrival` and `move-on-departure` ones have been preserved from the original domain definition and still move trains between track parts of the *path* track. These actions are needed due to the *path* not being a normal track. Since switches have been deprecated, the `move-to-track` and `move-from-track` actions have been erased, as well. Thus, moving from, along, and to a track has been reduced to a single `move` operation. This action takes a train and 2 track parts and checks whether the train unit is currently on the first part track, the second track part is free, and the track parts are next to each other. If so, the first track part is freed, the second one is occupied and the train is parked. Since the original domain definition does not support servicing, an action to service a train has also been defined. It takes a train unit, a track part, a track, and a service. In order to execute the action, the train has to be at the track part and it should need that service. Moreover, the track part should be part of a track that provides this service. If these preconditions hold, the train can be serviced. The full implementation of the domain can be found in Appendix A.2.

5.2 Domain instances

The 9 domain instances that we devise are based on 3 shunting yards. The number of tracks, number of track parts, and the services each yard provides are listed in Figure 4. The big yard is inspired by one of the shunting yards at the Central Hague Station [1]. Note that the latter is not tree-like, i.e. there are tracks with more than 1 entry point, thus we include it as a challenge for the planners to solve. We give a visual representation of the small, medium and big shunting yards in Figure 5, Figure 6 and Figure 7, respectively.

Shunting yard	# Tracks	# Tr. parts	Services
Small	5	10	1 × cleaning
Medium	12	25	2 × cleaning 1 × inspection
Big	37	57	2 × cleaning 1 × inspection 1 × washing

Figure 4: The three shunting yards used in the planner comparison.

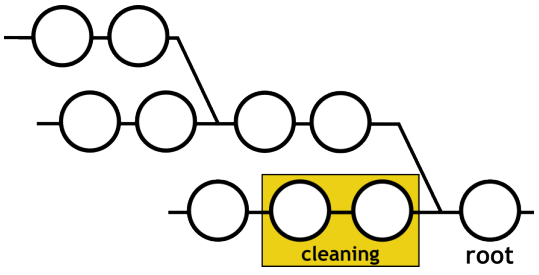

Figure 5: The small shunting yard.

[1] www.sporenplan.nl/html_nl/sporenplan/ns/ns_nummer/gvc-bkh.html

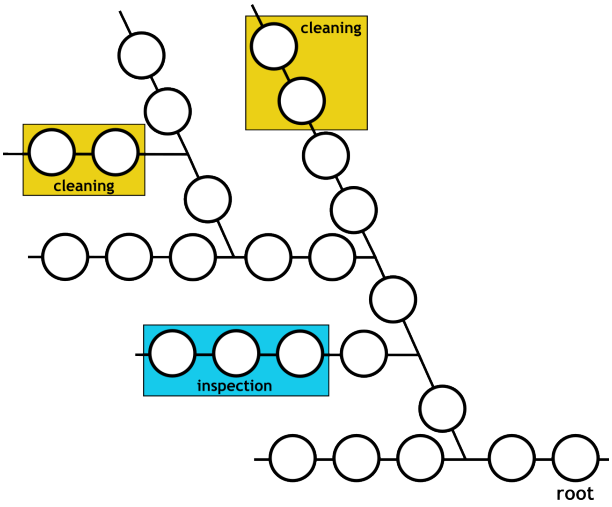

Figure 6: The medium shunting yard.

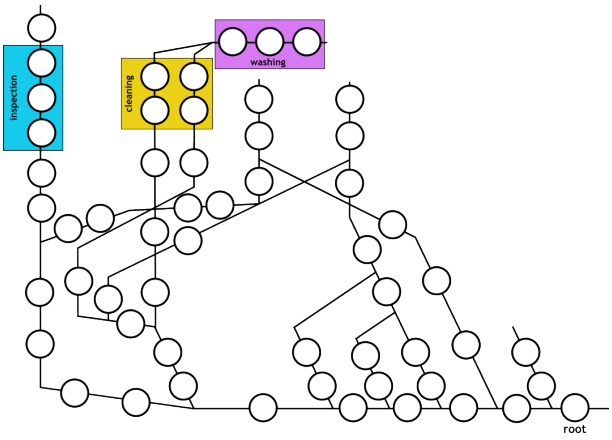

Figure 7: The big (Hague-Central-inspired) shunting yard.

We create 3 increasingly difficult instances from each of the created yards, all of which are shown in Figure 8. In each instance, roughly $\frac{1}{3}$ of the trains require servicing. The distribution of the needed services follows the ratio of track parts dedicated to each service and the total number of service track parts. A train may require more than 1 service. The arriving and departing train sequences are generated randomly. The goal of each instance is to service each of the trains, make sure they have been parked, have departed (they are on the *path* track) and have done so in the correct order.

Shunting yard	Number of trains		
	Instance 1	Instance 2	Instance 3
Small	2	5	8
Medium	5	10	15
Big	10	20	30

Figure 8: Number of trains used in each of the instances of the shunting yards. The more trains an instance has, the more complex it is.

5.3 Planner comparison

The course supervisors provide us with planners from the IPC2018, hence why we use those for solving the constrained TUSP with servicing. Out of the 23 available algorithmic planners, 6 are able to find a solution to at least one instance and only 4 manage to find a plan for at least 2 instances. Since we want the planner to be somewhat usable, we restrict our attention to the aforementioned 4 ones. Namely, the planners that we compare in this section are the baseline one given by the contest and the planners of teams 2, 4, and 7. Team 4's algorithm reduces the problem to SAT and uses an incremental SAT-solver to find a solution, which is then translated into a plan for the instance. The other three planners solve the problems by using variations of the fast-downward approach, instead. Note that team 2's and team 7's planners have been constructed from the baseline planner.

The planners were executed in Singularity containers on TU Delft's MAPFW server. The machine utilizes 12 AMD EPYC 7702P 64-Core Processors and 8 gigabytes of RAM. It runs the Ubuntu 22.04.2 LTS Linux distribution, while the Singularity images run on the Xenial 16.04 operating system.

To determine how long each planner takes to generate a sequence of actions, we employ Unix's `time` command and look at the number of seconds spent in user mode. This is an adequate measure of time since it is the one spent executing the planner's code. In order to avoid outliers both in time and in number of actions, we run the planners 5 times against each instance. The metric for a planner and an instance is then computed using the average time and number of actions for that planner and instance.

We present the averaged results of running the planners in Figure 9 and Figure 10. We observe that the baseline and team 2's planner are able to deal with only 2 out of the 9 instances, whereas team 7's can handle 3. It is surprising to find that the baseline planner performs better on time than team 2's and team 7's planners on the instances that all planners are able to solve. Since the latter 2 are derived from the baseline one, we expected the opposite to be the case. We attribute this to team 2's and team 7's additional computation, which does not work well for this domain. In terms of number of actions, the three planners perform quite similarly. In contrast, team 4's planner finds a solution to all 9 instances and consistently performs best in terms of time and plan size. The only exception is the small shunting yard with 2 trains, where it finds a plan with 13 actions, while the baseline's and team 2's action sequences consist of 11 actions. The raw data, containing all executions of the planners can be found in Appendix B.1.

In Figure 11 we observe how each of these planners performs against the metric we devised in Subsection 4.2. It is evident that team 4's planner greatly outperforms the other competitors. We attribute this to its ability to find solutions for all instances and doing so swiftly. Since these two criteria form the main part of our metric, it is sensible that team 4 scores so high and the others have a score below 0.12. Therefore, we conclude that the fast-downward approach is not suitable for the constrained TUSP with servicing, whereas reducing instances to SAT leads to an efficient method of finding plans for this domain.

Instance	Baseline	Team 2	Team 4	Team 7
S2	0.2362	0.2584	0.1394	0.2612
S5	N/A	N/A	0.2328	18.192
S8	N/A	N/A	16.0856	N/A
M5	285.88	1251.573	0.4978	1063.124
M10	N/A	N/A	1.602	N/A
M15	N/A	N/A	14.0436	N/A
B10	N/A	N/A	4.769	N/A
B20	N/A	N/A	25.3628	N/A
B30	N/A	N/A	1660.835	N/A

Figure 9: Average time over 5 runs for running the baseline, team 2's, team 4's, and team7's planners against 9 instances of the domain (in seconds). The first letter of the instance encodes the shunting yard (S - small, M - medium, B - big); the following digit(s) specify the number of trains for that instance. N/A is placed if the planner times out or does not produce a solution for that instance.

Instance	Baseline	Team 2	Team 4	Team 7
S2	11	11	13	13
S5	N/A	N/A	48	48
S8	N/A	N/A	135	N/A
M5	135	137	105	131
M10	N/A	N/A	240	N/A
M15	N/A	N/A	500	N/A
B10	N/A	N/A	394	N/A
B20	N/A	N/A	1103	N/A
B30	N/A	N/A	2060	N/A

Figure 10: Number of actions in the generated plans when running the baseline, team 2's, team 4's, and team7's planners against 9 instances of the domain.

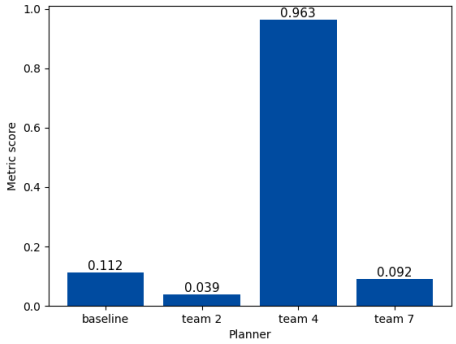

Figure 11: Metric scores of the four compared planners.

5.4 Improving the best planner

Team 4's planner consists of 3 components. First, the reduction to an instance of SAT, adhering to $\exists$-step semantics, is done. The result is fed into an incremental SAT-solver, which gradually builds the solution by querying a third component - the Lingeling SAT-solver [14]. In this paper, we focus on tweaking the reduction itself, as improving state-of-the-art SAT-solvers is unachievable in the time constraints that we are imposed with.

We modify the reduction in two ways. Firstly, we remove

one type of clauses, namely that changing the boolean value of a predicate implies executing an operator that does so. We can scrap these, as they are guaranteed to hold by another set of clauses, already in the reduction - executing an operator means its precondition holds at time t and its effect - at time $t + 1$. The removed clauses' purpose is to narrow down the search space of the SAT-solver by introducing more constraints it has to adhere to. However, this may have an adverse effect on bigger instances, as it introduces yet another set of clauses that have to be satisfied. Thus, in order to make the reduction more compact and see how that impacts big instances, we omit it. Secondly, we remove the computation of the operator graph as there are no strongly connected components of more than 1 node, meaning no operators mutually affect each other for any of the instances. Thus, it does not add any clauses to the reduction, making its deletion reasonable.

The planner is now reevaluated using the same setup as before. The modified algorithm is still able to solve all 9 instances. The changes in time and number of actions are presented in Figure 12 and Figure 13, respectively. In terms of time, the enhanced planner performs almost identically to the original one for small instances of the problem. The insignificant fluctuations ($< 5\%$) imply that the changes did not influence these instances much. As expected, the modifications had a notable impact on the most complex problems of each yard. A great improvement is observed for the most difficult ones of the small and big shunting yards, 52.1% and 366.4%, respectively. However, the runtime of the most complex medium instance became worse by 18.6%. We believe both results are due to the impact of redundant clauses, as explained earlier. Regarding the number of actions, the changes are minor and can be simply explained with the change of the reduction. The raw data, containing all executions of the modified planner is part of Appendix B.2.

Instance	Before	After	Change
S2	0.1394	0.1404	-0.7%
S5	0.2328	0.2238	4.0%
S8	16.0856	10.5742	52.1%
M5	0.4978	0.5170	-3.7%
M10	1.6020	1.4230	-3.7%
M15	14.0436	17.0402	-18.6%
B10	4.7690	4.7764	-0.2%
B20	25.3628	24.6526	2.9%
B30	1660.8350	356.0948	366.4%

Figure 12: Average time over 5 runs after running the planner before and after the improvement (in seconds). The last column indicates the change - a positive percentage implies an improvement.

The metric is recomputed based on the change in time/actions over the original value. For instance, if the original planner's time metric is 1, a 50% improvement would yield 1.5. This is done in order to reuse the already computed metrics of team 4's planner. After recalculation, the score of the modified planner comes to 1.2659, which is a 31.5% positive change from the initial planner's 0.9629. Thus, we can conclude that we have improved it.

Instance	Before	After	Change
S2	13	13	=
S5	48	48	=
S8	135	133	1.5%
M5	105	113	-7.1%
M10	240	250	-4.0%
M15	500	494	1.2%
B10	394	392	0.5%
B20	1103	1131	-2.5%
B30	2060	2131	-3.3%

Figure 13: Number of actions in the generated plans after running the planner before and after the improvement.

6 Responsible Research

We deem it crucial that research is conducted responsibly. Therefore, in the rest of this section, we describe the measures we have taken to make the experimental procedure transparent and reproducible. Furthermore, we discuss the integrity of this research and how it may affect stakeholders, closely related to this problem, such as Dutch Railways (NS) and its employees.

6.1 Research reproducibility

We believe we have been as explicit as possible when it comes to reproducibility. While we cannot guarantee that the implemented domain is bug-free, we have manually tested it on small instances in order to verify that it behaves in a correct manner. In that regard, researchers can find both the original and the modified versions of the domain in Appendices A.1 and A.2. We also provide the full codebase, including several instance generators and the modified planner. These can be found in TU Delft's research repository.

We have fully depicted the instances the planners were compared against, along with the metric that we employed for their comparison. We have also described the computer configuration we used during this performance evaluation in detail. Although the time results could be skewed due to differing system loads at the time of executing the planners, we believe that the user time is an adequate measure for such comparisons. This is mainly attributed to the metric's minimal deviation from the actual time the planner has run on the processor. To further minimize the risk of outliers, we ran the planners on each instance a total of 5 times and averaged the resulting execution times, as well.

6.2 Research integrity

No funding has been received for conducting this research, therefore, there is no conflict of interest to report. The main stakeholder of this problem, NS, might suffer adverse consequences if the resulting planner is blindly trusted and/or unmonitored. For instance, generating a plan that causes two trains to be parked at the same spot, due to an unintended code error, might cause damage to the trains, were the driver not to be cautious during the execution of the plan's actions. Even though we have tried to ensure that the domain and its instances are modeled properly, we advise human planners to

review the automatically generated plans before they commit to them.

There might also be concerns that the complete integration of this tool at Dutch Railways may cause people to become unemployed. However, the planner, at least at its current stage, should only be used to help with the decision-making of human planners. It is by no means fully-fledged software that is ready to replace human workers. Thus, we advise strongly against dismissing people from their workplace for the sake of automation.

7 Conclusions and Future Work

In this paper, we improve a planner for a constrained version of the Train Unit Shunting Problem extended with servicing. For that purpose, we first give a formal description of the problem and introduce constraints to make instances more feasible to solve. These restrictions include considering only tree-like shunting yards, trains of unit length, and tracks consisting of a finite number of track parts. We then model the problem in the Planning Domain Definition Language by modifying an existing domain to suit our needs. Afterward, we evaluate 4 planners from the International Planning Competition 2018. We discover that the baseline planner, provided by the contest, and the planners which derive from it, namely team 2's and team 7's, cannot handle the 9 created domain instances well. In contrast, team 4's planner is able to deal with all instances and does so swiftly, which places it first in a metric for planner comparison that we devise. This planner reduces the problem to the Boolean Satisfiability Problem and then uses state-of-the-art SAT-solvers to find a solution. We enhance it by modifying the reduction not to include redundant clauses, typically used for narrowing down the search space of the SAT-solver, as these overcomplicate big instances. We also remove a computation that increases the time of the reduction, without providing any benefit for our problem. These tweaks result in a 31.5% increase in the performance of the planner, meaning we do improve it.

Future work might want to consider non-tree-like shunting yard layouts, since those are abundant in the Netherlands, as well. While one of the shunting yards we tested against had a few tracks that were not tree-like, we believe more thorough research should be conducted to test the feasibility of planners for such layouts. Since we have found a planner that generates action sequences quickly even for bigger instances, further research should look into trains that are composed of multiple train units. We believe the speed of the planner might be able to compensate for the exponential increase in complexity. We also suspect that the improved planner can be enhanced further by swapping the employed SAT-solver with a faster one, or one that works better for the modified reduction. Due to the non-modularity of team 4's planner, alongside time constraints, we were unable to do so, and therefore leave it for future work.

References

[1] R. Freling, R. M. Lentink, L. G. Kroon, and D. Huisman, "Shunting of passenger train units in a railway station," *Transportation Science*, vol. 39, no. 2, pp. 261–272, 2005.

[2] R. van den Broek, H. Hoogeveen, M. can den Akker, and B. Huisman, "A local search algorithm for train unit shunting with service scheduling," *Transportation Science*, vol. 56, no. 1, pp. 141–161, 2022.

[3] J. Mulderij, B. Huisman, D. Tönissen, K. van der Linden, and M. de Weerdt, "Train unit shunting and servicing: a real-life application of multi-agent path finding," 2020.

[4] T. Winter, "Online and real-time dispatching problems," 2000.

[5] I. K. Hanou, M. M. de Weerdt, and J. Mulderij, "Moving trains like pebbles: A feasibility study on tree yards," *Proceedings of the International Conference on Automated Planning and Scheduling*, 2023. To be published.

[6] E. Peer, V. Menkovski, Y. Zhang, and W.-J. Lee, "Shunting trains with deep reinforcement learning," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 3063–3068, 2018.

[7] X. Xu and M. M. Dessouky, "Train shunting with service scheduling in a high-speed railway depot," *Transportation Research Part C: Emerging Technologies*, vol. 143, p. 103819, 2022.

[8] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld, "Pddl - the planning domain definition language," 1998.

[9] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[10] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*, vol. 185. IOS press, 2009.

[11] A. Torralba and F. Pommerening, "Ipc 2018 - classical tracks," *28th International Conference on Automated planning and Scheduling*, 2018.

[12] J. Rintanen, K. Heljanko, and I. Niemelä, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12, pp. 1031–1080, 2006.

[13] A. Howe and E. Dahlman, "A critical assessment of benchmark comparison in planning," 2002.

[14] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," 2013.

Appendix

A PDDL Domains

The purpose of this Section is to give the complete definitions of the PDDL domains, used throughout this research. The domain that was provided by the supervisory team is presented in Section A.1. The one that is extended with servicing actions is given in Section A.2.

A.1 Original domain

The domain we have been given is presented in Figure 14. It does not contain anything related to services but has predicates and actions we employ in the modified domain.

```
(
    define (domain original-domain)

    (:requirements :adl)

    (:types
        trackpart track trainunit - object
        icm virm sng slt - trainunit ; these are the different types of train units
    )

    (:predicates
        (nextTo ?x ?y - trackpart) ;track part x next to other track part y
        (onTrack ?x - trackPart ?y - track) ;track part x on track y
        (at ?x - trainunit ?y - trackpart) ;train unit x on track part y
        (hasBeenParked ?x - trainunit) ;true if x is parked on some track
        (free ?x - trackpart) ;trackpart x has nothing parked there
        (parkedOn ?x - trainunit ?y - track) ; indicates x parked on track y
        (onPath ?x) ;trackpart x is on the arrival/departure path L
        (switch ?x) ;trackpart x is a switch
    )

    ; action to move a trainunit to a neighbouring trackpart on a track, to park it
    (: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 to move a trainunit to out of a track, and reset the parkedOn predicate
    (: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 to move a trainunit along a track
(: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))
    )
)

; Can only move back to departure if all trains have been parked.
(: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 to move train unit over the arrival path towards the shunting yard
(: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))
        )
    )
)
```

Figure 14: The domain we start with, which was used for the feasibility study of tree-like shunting yards. Predicates and actions are commented, so that the reader can gain a better understanding of them.

A.2 Modified domain

The modified domain, which includes support for servicing can be found in Figure 15. In contrast to the original one, this domain includes service objects, which represent different types of services, e.g. cleaning. In addition, two predicates, specific to the servicing of trains are introduced - `isServiceTrack`, which signifies that a train track is able to provide a service, and `needsService`, showing whether a train requires a certain service. There is a new `service-train` action that is executable when a train is on a service track and needs the exact service the track is for.

```
(
    define (domain servicing-simple)

    (:requirements :adl)
    (:types
        trackpart track trainunit service - object
        icm virm sng slt - trainunit ; these are the different types of train units
    )

    (:predicates
        (nextTo ?x ?y - trackpart) ;track part x next to other track part y
        (onTrack ?x - trackPart ?y - track) ;track part x on track y
        (onPath ?x) ;trackpart x is on the arrival/departure path L
        (at ?x - trainunit ?y - trackpart) ;train unit x on track part y
        (free ?x - trackpart) ;trackpart x has nothing parked there
        (hasBeenParked ?x - trainunit) ;true if x is parked on some track
        (isServiceTrack ?x - track ?y - service) ; track x is a track for service y
        (needsService ?x - trainunit ?y - service); train unit x needs service y
        (hasDeparted ?x - trainunit) ; train unit x has departed
    )

    ; action to service a train for a service
    (:action service-train
        :parameters (?train - trainunit ?at - trackpart ?track - track ?service - service)
        :precondition (and
            (at ?train ?at)
            (isServiceTrack ?track ?service)
            (onTrack ?at ?track)
            (needsService ?train ?service)
            (not (hasDeparted ?train))
        )
        :effect (and
            (not (needsService ?train ?service))
        )
    )

    ; action to move a trainunit to a neighbouring trackpart and park it
    (:action move
        :parameters (?train - trainunit ?from ?to - trackpart)
```

```
        :precondition (and
                        (at ?train ?from)
                        (free ?to)
                        (nextTo ?from ?to)
                        (not (hasDeparted ?train))
        )
        :effect (and
                    (at ?train ?to)
                    (not (at ?train ?from))
                    (free ?from)
                    (not (free ?to))
                    (hasBeenParked ?train)
        )
    )

    ; Can only move back to departure if all trains have been parked.
    (: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))
                    (hasDeparted ?train)
        )
    )

    ; Action to move train unit over the arrival path towards the shunting yard
    (:action move-on-arrival
        :parameters (?train - trainunit ?from ?to - trackpart)
        :precondition (and
                        (at ?train ?from)
                        (free ?to)
                        (nextTo ?from ?to)
                        (not (hasDeparted ?train))
                        (not (hasBeenParked ?train))
                        (onPath ?to)
        )
        :effect (and
                    (at ?train ?to)
                    (not (at ?train ?from))
                    (free ?from)
                    (not (free ?to))
        )
    )
)
```

Figure 15: The modified domain that also supports servicing. Predicates and actions are still commented for readability.

B Raw Experimental Data

The aim of this section is to present the raw results of running the planners against the 9 domain instances we have devised. As mentioned in the main part of the paper, each planner was run 5 times against each instance in order to exclude any outliers in the execution time and number of steps. We have also done 2 sets of experiments - one to compare the baseline, team 2's, team 4's and team7's planner, and another one to evaluate the improved version of the best (team 4's) planner. Therefore, we dedicate a subsection to the results of each set of experiments.

B.1 Raw planner comparison data

We observe the performance of each of the planners against the 9 created instances of the constrained TUSP with servicing domain in Figure 16. Each instance is encoded as a combination of a letter, signifying the shunting yard (S - small, M - medium, B - big) and digits, which symbolize the number of trains in that instance.

Instance	Run #	Planners							
		Baseline		Team 2		Team 4		Team 7	
		# Actions	Time (in s)	# Actions	Time (in s)	# Actions	Time (in s)	# Actions	Time (in s)
S2	1	11	0.258	11	0.243	13	0.139	13	0.237
	2	11	0.244	11	0.280	13	0.137	13	0.286
	3	11	0.199	11	0.283	13	0.139	13	0.226
	4	11	0.218	11	0.243	13	0.141	13	0.291
	5	11	0.262	11	0.243	13	0.141	13	0.266
	Average	11	0.2362	11	0.2584	13	0.1394	13	0.2612
	Std. Dev.	0	0.0270	0	0.0211	0	0.0017	0	0.0289
S5	1	N/A	N/A	N/A	N/A	48	0.197	48	17.440
	2	N/A	N/A	N/A	N/A	48	0.230	48	21.320
	3	N/A	N/A	N/A	N/A	48	0.249	48	17.350
	4	N/A	N/A	N/A	N/A	48	0.247	48	17.014
	5	N/A	N/A	N/A	N/A	48	0.241	48	17.836
	Average	N/A	N/A	N/A	N/A	48	0.2328	48	18.192
	Std. Dev.	N/A	N/A	N/A	N/A	0	0.0213	0	1.7729
S8	1	N/A	N/A	N/A	N/A	135	15.772	N/A	N/A
	2	N/A	N/A	N/A	N/A	135	17.009	N/A	N/A
	3	N/A	N/A	N/A	N/A	135	15.959	N/A	N/A
	4	N/A	N/A	N/A	N/A	135	15.762	N/A	N/A
	5	N/A	N/A	N/A	N/A	135	15.926	N/A	N/A
	Average	N/A	N/A	N/A	N/A	135	16.0856	N/A	N/A
	Std. Dev.	N/A	N/A	N/A	N/A	0	0.5237	N/A	N/A
M5	1	135	278.870	137	1610.970	105	0.499	131	1059.600
	2	135	283.093	137	1463.843	105	0.518	131	1269.711
	3	135	300.479	137	1012.200	105	0.500	131	924.251
	4	135	285.024	137	1107.994	105	0.506	131	1153.288
	5	135	281.934	137	1062.858	105	0.466	131	908.772
	Average	135	285.88	137	1251.573	105	0.4978	131	1063.124
	Std. Dev.	0	8.4608	0	268.2128	0	0.0193	0	153.2403
M10	1	N/A	N/A	N/A	N/A	240	1.588	N/A	N/A
	2	N/A	N/A	N/A	N/A	240	1.621	N/A	N/A
	3	N/A	N/A	N/A	N/A	240	1.609	N/A	N/A
	4	N/A	N/A	N/A	N/A	240	1.581	N/A	N/A
	5	N/A	N/A	N/A	N/A	240	1.611	N/A	N/A
	Average	N/A	N/A	N/A	N/A	240	1.6020	N/A	N/A
	Std. Dev.	N/A	N/A	N/A	N/A	0	0.0168	N/A	N/A

Instance	Run #	Planners							
		Baseline		Team 2		Team 4		Team 7	
		# Actions	Time (in s)	# Actions	Time (in s)	# Actions	Time (in s)	# Actions	Time (in s)
M15	1	N/A	N/A	N/A	N/A	500	14.432	N/A	N/A
	2	N/A	N/A	N/A	N/A	500	13.779	N/A	N/A
	3	N/A	N/A	N/A	N/A	500	14.406	N/A	N/A
	4	N/A	N/A	N/A	N/A	500	14.215	N/A	N/A
	5	N/A	N/A	N/A	N/A	500	13.386	N/A	N/A
	Average	N/A	N/A	N/A	N/A	500	14.0436	N/A	N/A
	Std. Dev.	N/A	N/A	N/A	N/A	0	0.4512	N/A	N/A
B10	1	N/A	N/A	N/A	N/A	394	4.427	N/A	N/A
	2	N/A	N/A	N/A	N/A	394	4.624	N/A	N/A
	3	N/A	N/A	N/A	N/A	394	5.488	N/A	N/A
	4	N/A	N/A	N/A	N/A	394	4.947	N/A	N/A
	5	N/A	N/A	N/A	N/A	394	4.359	N/A	N/A
	Average	N/A	N/A	N/A	N/A	394	4.7690	N/A	N/A
	Std. Dev.	N/A	N/A	N/A	N/A	0	0.4623	N/A	N/A
B20	1	N/A	N/A	N/A	N/A	1103	24.282	N/A	N/A
	2	N/A	N/A	N/A	N/A	1103	28.052	N/A	N/A
	3	N/A	N/A	N/A	N/A	1103	25.172	N/A	N/A
	4	N/A	N/A	N/A	N/A	1103	25.118	N/A	N/A
	5	N/A	N/A	N/A	N/A	1103	24.190	N/A	N/A
	Average	N/A	N/A	N/A	N/A	1103	25.3628	N/A	N/A
	Std. Dev.	N/A	N/A	N/A	N/A	0	1.5710	N/A	N/A
B30	1	N/A	N/A	N/A	N/A	2060	1558.626	N/A	N/A
	2	N/A	N/A	N/A	N/A	2060	1714.331	N/A	N/A
	3	N/A	N/A	N/A	N/A	2060	1670.570	N/A	N/A
	4	N/A	N/A	N/A	N/A	2060	1694.363	N/A	N/A
	5	N/A	N/A	N/A	N/A	2060	1666.287	N/A	N/A
	Average	N/A	N/A	N/A	N/A	2060	1660.8350	N/A	N/A
	Std. Dev.	N/A	N/A	N/A	N/A	0	60.3265	N/A	N/A

Figure 16: Performance of the 4 compared planners against 9 instances over 5 runs in terms of time and number of actions. N/A is given when a planner is not successful in finding a solution to that instance. The average and standard deviation of each planner-instance combination across the 5 runs is also presented.

B.2 Raw improved planner evaluation data

We present the evaluation of the improved planner against the same 9 instances using the same metrics as before. The results can be found in Figure 17.

Instance	Run #	# Actions	Time (in s)	Instance	Run #	# Actions	Time (in s)
S2	1	13	0.118	M15	1	494	16.224
	2	13	0.155		2	494	17.120
	3	13	0.147		3	494	18.700
	4	13	0.153		4	494	16.730
	5	13	0.129		5	494	16.427
	Average	13	0.1404		Average	494	17.0402
	Std. Dev.	0	0.0169		Std. Dev.	0	0.9874
S5	1	48	0.231	B10	1	392	5.051
	2	48	0.215		2	392	5.351
	3	48	0.224		3	392	4.335
	4	48	0.226		4	392	4.635
	5	48	0.223		5	392	4.510
	Average	48	0.2238		Average	392	4.7764
	Std. Dev.	0	0.0058		Std. Dev.	0	0.4157

Instance	Run #	# Actions	Time (in s)	Instance	Run #	# Actions	Time (in s)
S8	1	133	12.061	B20	1	1131	25.810
	2	113	10.220		2	1131	24.787
	3	113	9.883		3	1131	25.587
	4	113	10.286		4	1131	23.446
	5	113	10.421		5	1131	23.633
	Average	113	10.5742		Average	1131	24.6526
	Std. Dev.	0	0.8544		Std. Dev.	0	1.0870
M5	1	113	0.476	B30	1	2131	365.719
	2	113	0.563		2	2131	360.603
	3	113	0.513		3	2131	361.705
	4	113	0.519		4	2131	329.950
	5	113	0.514		5	2131	362.497
	Average	113	0.517		Average	2131	356.0948
	Std. Dev.	0	0.0309		Std. Dev.	0	14.7391
M10	1	250	1.441				
	2	250	1.413				
	3	250	1.381				
	4	250	1.449				
	5	250	1.431				
	Average	250	1.4230				
	Std. Dev.	0	0.0271				

Figure 17: Performance of the improved team 4's planner over 5 runs for the 9 instances of the domain the original 4 planners are tested against. Both the number of actions in the final plan and the time to generate one are reported. The average and standard deviation across the 5 runs is also shown.