Robust Shunting in a Dynamic Environment

Deriving Proactive Schedules from a Reactive Policy

by

Elwin Duinkerken

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday September 25, 2023 at 15:00 PM.

Student number:	4727797	
Project duration:	September, 2022 – September, 2023	
Thesis committee:	Dr. A. Lukina,	TU Delft, supervisor
	Prof. Dr. M. M. de Weerdt,	TU Delft, co-supervisor
	Prof. Dr. R.M.P. Goverde	TU Delft

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

TU Delft

Summary

When trains are not actively traveling on the main rail network, they to be parked and prepared for their next journey. This is a complex problem, involving several interconnected subproblems. Additionally, there is uncertainty in this environment which can render initial plans infeasible during their execution. To ensure trains are able to depart in time, having finished all their required service tasks, a schedule is created in advance.

The focus of this thesis is to address the challenges associated with generating robust initial shunting plans in an uncertain environment. This thesis focusses on a sequential problem formulation, modeled as a Markov Decision Process (MDP) and uses a policy optimized for this environment. The goal is to design a method capable of deriving robust initial shunting plans from the policy that are likely to remain feasible for a large number of possible plan executions.

The limitation addressed in this thesis, is that conventional policy-rollout techniques generate action sequences that overlook most alternative outcomes, thereby making the overall plan not feasible for a large number of plan realizations. To address this issue, the thesis proposes two distinct solution methods, aimed to consider every possible state that might be encountered, either directly or indirectly.

Through experimentation on realistically generated problem instances, the research concludes that both proposed methods significantly outperform the baseline approach, demonstrating the possibility of extracting robust initial shunting plans from a given policy that was not explicitly designed for this purpose.

Contents

1

Introduction

Throughout the day, most of the trains travel the on main rail network between stations. However, during off-peak hours and at night, when the demand for train services is low, the trains need to be prepared for their next trips. These preparations take place at designated locations, referred to as shunting yards, where the trains are stationed and undergo the necessary preparations to ensure their availability for the next trips.

This thesis focuses on the train operations that take place within a single shunting yard. Trains arrive mostly near the end of the day, where they must remain parked overnight in the yard. However, during their stay on the yard, they must undergo several preparations before they can depart again. These preparations include several service tasks, such as cleaning, washing, and maintenance. Figure 1.1 provides an example of a shunting yard, illustrating its layout and infrastructure.

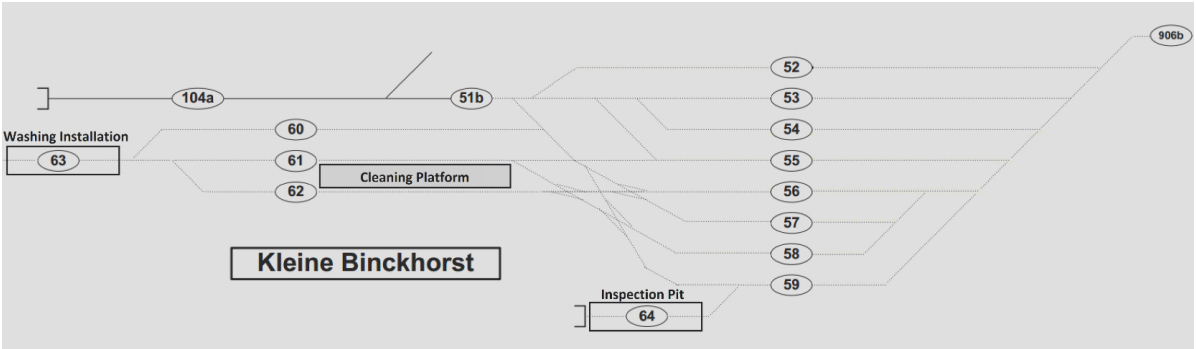

Figure 1.1: Example of shunting yard the "*Kleine Binckhorst*". The yard consists of 8 tracks meant for parking at tracks 52 to 59. Tracks 61 and 62 are positioned alongside a cleaning platform. Track 63 contains a washing installation and track 64 an inspection pit. Trains enter and exit at the gateway track 906b. Tracks 51b and 104a are used to connect the "*Kleine Binckhorst*" to the "*Grote Binckhorst*"

To ensure each train is able to depart in time, having finished all its required tasks, a schedule is created in advance. A shunting plan is typically created for a planning horizon of 24 hours. The plan describes the exact locations of each train during its stay in the yard, along with the necessary track movements required to reach each location. Additionally, the plan specifies the order in which trains will be serviced, ensuring that all service tasks are finished on time. Lastly, an assignment is made to each train, where each incoming train is matched to an outgoing train.

Manually creating a feasible shunting plan is a challenging and time-consuming task. As the number of train units increases, the complexity also increases, making the task of finding a feasible solution more difficult. Existing work has studied methods to automatically generate such shunting plan, with varying levels of success.

Ideally, the initial shunting plan should be designed to be robust against small everyday disruptions. If the initial plan is robust, the plan is more likely to remain feasible without the need to repair the failed plan during execution.

Existing literature has often focused on finding feasible solution in a deterministic environment with normal circumstance This project aims to address the dynamic factors of the shunting problem and its inherent uncertainty. The goal is to develop a solution method that is able to create robust initial shunting plans that are able to deal with disturbances encountered during execution of the plan.

1.1. Problem Context: the Shunting Problem

To ensure every train is able to park on the yard, shunting plans are constructed in advance. The problem of generating a feasible shunting plan is known as the Train Unit Shunting Problem (TUSP).

During their stay on the shunting yard, trains often require various service tasks, such as maintenance and internal and external cleaning. These tasks must be completed within a specific timeframe before the trains are scheduled to depart again. The exact deadline for each train is not known in advance, because it depends on when the train is scheduled to depart. However, the availability of service resources is limited, so there is a limit on the number of trains that can be serviced simultaneously. Furthermore, certain service tasks can only be performed at designated service tracks within the shunting yard, necessitating the relocation of trains to and from these locations. The inclusion of service scheduling in addition to the TUSP is known as the Train Unit Shunting with Servicing (TUSS) problem or the Train Unit Shunting Problem with Service Scheduling (TUSPwSS).

For the remainder of this thesis, we just refer to the TUSS/TUSPwSS as the *Shunting Problem*.

1.1.1. Preliminaries

A shunting yard's layout consists a series of connected tracks. These tracks can be dead-end tracks, which function as Last-In-First-Out (LIFO) queues. In this configuration, the last train to enter the track is the first that is able to exit. Figure 1.2 provides an illustration of a small example layout that contains three LIFO tracks. Other tracks are accessible from both ends, known as "free" tracks. Examples of these tracks can be found in Figure 1.1. Trains enter and exit the shunting yard through designated gateways, which connect the main rail network to the yard. Trains are not allowed to remain parked on gateway tracks, as it would block all other trains from entering and leaving the yard.

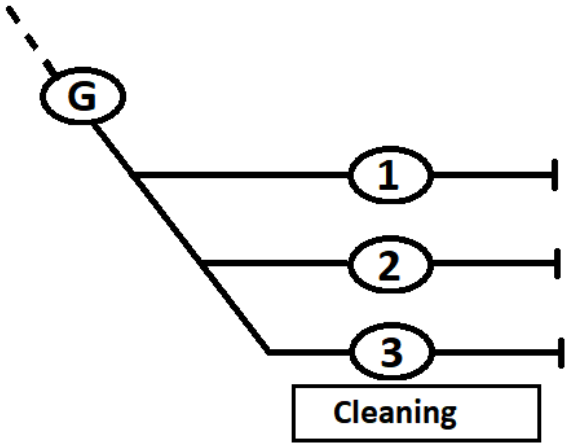

Figure 1.2: Small example layout of a shunting yard. Trains enter the yard on the gateway track G. Tracks 1 and 2 are regular parking tracks. Track 3 contains a cleaning platform, which allows the cleaning tasks to be performed on a train on the track.

All trains have the ability to move in both directions over the tracks. There are several types of train units, each with their own characteristics. One of the characteristic is the train unit's length, which is determined by the number of carriages it contains and the length of the carriages. Figure 1.3 provides an example showing two different train types. Trains can be composed of multiple connected train units,

which move over the tracks as a single train. While on the shunting yard, these trains can be split can combined to form different compositions

Figure 1.3: Example of two train units, the one of top containing three carriages; the one on the bottom containing four carriages. [7]

Most trains have a set of required service tasks that must be completed before their departure. The duration of each service task is dependent on the type of service and the type of train. There are some service tasks that can be performed on any parking track, and only require an available service crew. In contrast, there are also track-specific service tasks that can only be performed at designated service tracks. These tasks are limited to servicing a single train per track.

1.1.2. Subproblems

The Train Unit Shunting Problem with Service Scheduling (TUSPwSS) involves multiple interconnected subproblems: parking, matching, reconfiguration, servicing, and routing. Although some of these problem are difficult on their own, the main challenge comes from the fact that the subproblems are highly interdependent and require a coordinated approach to solve.

- **Parking** When train units arrive at the shunting yard, they must be stored somewhere in the yard until they are scheduled to depart. However, there is limited space on the parking tracks, meaning that only a certain number of train units can be parked on the same track at the same time. Additionally, later arriving trains may block the exit of earlier trains, and a train cannot leave the track if it is blocked by another train.

- **Matching** The input of the shunting problem does not specify which incoming train units should be used to form each outgoing train. As a result, all arriving train units have to be assigned to a departing train such that all departing trains are able to leave in time. Arrivals and departures can be mixed in time, so not every train unit can be used for an outgoing train. Since there are several types of trains, which each have their own specific properties, not every train can be used interchangeably. Matching is considered valid if all outgoing trains have assigned units of the proper subtypes and arrive and complete their service tasks before the departure. The time required for each service task depends on various factors, including the distance of the assigned parking tracks, the routes that can be taken, and how long it must wait before each service task can start.

- **Reconfiguration** Trains sometimes consist of multiple train units, so the trains might need to split and combine to form the correct configurations. Train units can only be combined if they are next to each other, which means the trains may require additional movements to reach the desired configurations. Not all tracks allow for performing splitting and combining actions, particularly the gateway tracks.

- **Servicing** All train units have a set of required service tasks that need to be completed before they can depart from the shunting yard. Service tasks require resources, which are limited, and each resource can only be used by a single task at a time. This means that not all service tasks can be processed immediately upon arrival, but have to be scheduled such that all service tasks are finished before the train unit needs to depart. Some service tasks can only be performed on specific tracks, requiring trains to route to and from these tracks.

- **Routing** A path needs to be found to move the train from its position to the next one. A train can only move over tracks that are not currently obstructed by any other train.

1.1.3. Initial Shunting Plan

The solution to the shunting problem is typically represented as a Partial Order Schedule This schedule consist of a set of activities A, and a set of precedence relations POS. Each activity is performed on a specific train. The order of the activities are defined by the precedence relations.

A service activity requires a corresponding service resource. This resource cannot be used by multiple trains simultaneously. Therefore, a precedence relations imposes an ordering to each related service activity that uses the resource. Similarly, only a single movement can be scheduled simultaneously.

The shunting plan can be visualized by an activity graph, where the nodes correspond to an activity, and the edges to precedence relations. Figure 1.4 shows an example of such an activity graph.

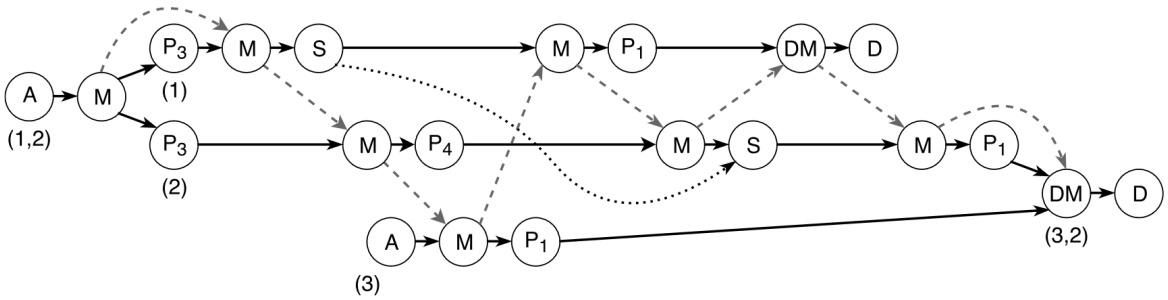

Figure 1.4: An example of a shunting plan, visualized as an activity graph. The activities are denoted as follow: Arrivals (A), Movement (M), Parking (P), Service (S), Departure Movement (DM) and Departure (D). The edges between the activities indicate a precedence relation between them. The corresponding train units are indicated between the parenthesis.

1.1.4. Uncertainty

There is a lot of uncertainty in the real world during the shunting operations. This uncertainty is caused by both minor and major disturbances from different sources and variability in the duration of each activity. According to expert opinion, there is a small disturbance almost every on a shunting location, where there are some days with no disturbances, and others where there are multiple. The most common disturbances include train delays and wrong train configurations, such as trains only containing a single unit instead of the expected two. Larger disturbances include service tasks that take longer than expected, or, in case of a large disturbance on national level, additional arriving trains. These larger disturbances are relatively unlikely.

Any single unconditional shunting plan has the inherent risk of becoming infeasible during execution, as potential changes to the problem instance itself, cause by larger disturbances, could render the plan invalid. However, in this thesis we focus on creating shunting plans that are robust to smaller, more frequent disturbances. Therefore, we only consider two sources of uncertainty: the arrival times of each train and the variable durations of service tasks.

While the expected arrival times of trains and the mean durations of service tasks are specified in a given problem instance, the actual times are subject to variability, sampled from probability distributions. Figure 1.5 provides an example distribution of the duration for a specific activity, where the average duration is 30 minutes, but the actual durations during execution can take on various values with different probabilities.

The goal of a shunting plan, therefore, is to be robust against these smaller kinds of disturbances, where the objective is to find an initial shunting plan that is feasible for as many realizations as possible. This reduces the need for making changes or repairs during execution.

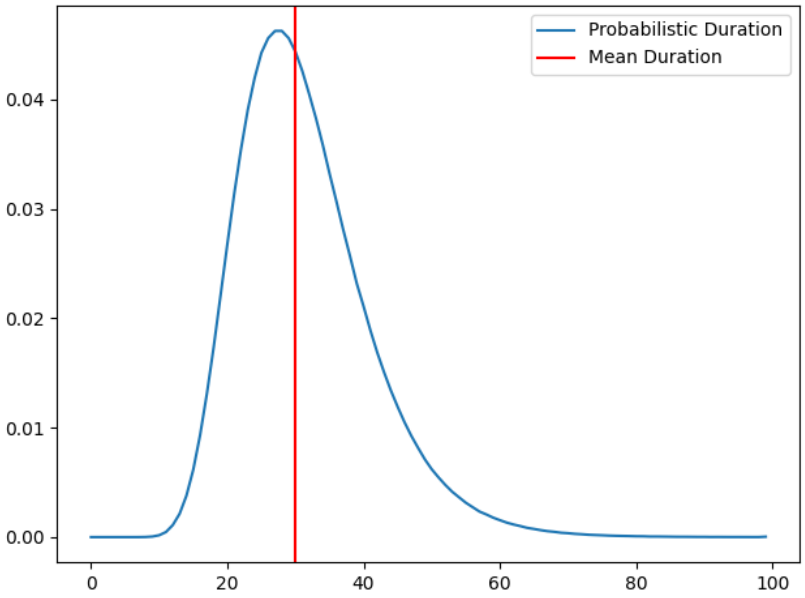

Figure 1.5: Example of the duration of an activity with an average duration of 30 minutes. In this example, the duration is sampled from the probability distribution according to a log-normal distribution using a variance of $\sigma = 0.3$.

1.1.5. Small Example

In this section, we present a small example problem to demonstrate the structure of a problem and corresponding solution. We provide two different solutions to show the contrast in robustness between them. The first solution is feasible but lacks robustness when disturbances are considered. The second solution is not only feasible but is also more robust towards the considered disturbances.

In Figure 1.2, a small example layout of a shunting yard is shown. The yard contains a single gateway track where all the train enter and exit the yard. There are two Last-In-First-Out (LIFO) parking tracks, and one service track designated for cleaning. The problem scenario has a total of three train units with two different type of trains. All train units require a 25-minute cleaning service, as indicated in Table 1.1. The arrival and departure times of the trains are specified in Table 1.2, with the first two train units arriving within the first five minutes, followed by the third train unit arriving after 45 minutes. For now, no disturbances are considered, so the arrival times and service durations are assumed to be entirely deterministic.

Train Unit	Required Services
0	cleaning (25 minutes)
1	cleaning (25 minutes)
2	cleaning (25 minutes)

Table 1.1: Required services

Time	Arriving Train
00:00	0 (VIRM-4)
00:05	1 (SLT-4)
00:45	2 (SLT-4)

Time	Departing Train
01:30	(SLT-4)
02:15	(SLT-4)
02:30	(VIRM-4)

Table 1.2: Example timetables of the arrival and departure events

Table 1.3 presents an example of a feasible solution for the given problem scenario. The table includes the start time and end time of each action. Upon arrival, both trains are parked on separate tracks. Train 0 is cleaned service first, followed by train 2, and train 1 is serviced last. The specific

movement paths of each train unit are specified in Table 1.4. The assignment of the train units to their respective departures is indicated in Table 1.5.

Start	End	Action	Train	Tracks
00:00	00:05	Park	(0)	$G \to 1$
00:05	00:10	Park	(1)	$G \to 2$
00:10	00:15	Park	(0)	$1 \to 3$
00:15	00:40	Service	(0)	3
00:40	00:45	Park	(0)	$3 \to 1$
00:45	00:50	Park	(2)	$G \to 1$
00:50	00:55	Park	(2)	$1 \to 3$
00:55	01:20	Service	(2)	3
01:20	01:25	Park	(2)	$3 \to 1$
01:25	01:30	Depart	(2)	$1 \to G$
01:30	01:35	Park	(1)	$2 \to 3$
01:35	02:00	Service	(1)	3
02:00	02:05	Park	(1)	$3 \to 2$
02:10	02:15	Depart	(1)	$2 \to G$
02:25	02:30	Depart	(0)	$1 \to G$

Table 1.3: Example of feasible plan (not robust)

Train unit	Track locations
0	$G \to 1 \to 3 \to 1 \to G$
1	$G \to 2 \to 3 \to 2 \to G$
2	$G \to 1 \to 3 \to 1 \to G$

Table 1.4: Sequence of each train unit's position on the shunting yard.

Departure	Train unit
0	2
1	1
2	0

Table 1.5: The matching train units to the departure indices.

In the previous example, we assumed a deterministic environment without any disturbances, where the exact arrival times and service durations are known. However, in reality, this is often not the case. Let's modify the scenario so there is now a 20% chance of a train arriving 10 minutes later than schedule time and a 20% chance of a service duration being extended by 10 minutes.

One of the main issues with the previous solution is that train 0 is serviced first, even though it is the last to depart. This choice makes it more difficult for the other trains to complete their service tasks in time. Additionally, after the service tasks are completed, the trains immediately park again and start a new service without buffer time between these actions. While the plan may work under normal circumstances, it lacks robustness. Delays can result in some trains being unable to finish their service tasks on time. Taking these disturbances into account, the plan is only successful approximately 37% of the time.

A more robust plan is presented in Table 1.6. The movement of each train is indicated in Table 1.8 and the matchings in Table 1.8. One of the improvements is that all the trains are serviced in the same order as their departure, ensuring they are ready to depart when required. Another notable aspect is that the trains only move away from the service track after an event is completed. This minimizes the risk of interference with the event, as the even can take place during the service activity. These improvements causes the plan to be more robust, and is feasible about 82% of the time.

Start	End	Action	Train	Tracks
00:00	00:05	Park	(0)	$G \rightarrow 1$
00:05	00:10	Park	(1)	$G \rightarrow 2$
00:10	00:15	Park	(1)	$2 \rightarrow 3$
00:15	00:40	Service	(1)	3
00:45	00:50	Park	(2)	$G \rightarrow 1$
00:50	00:55	Park	(1)	$3 \rightarrow 2$
00:55	01:00	Park	(2)	$1 \rightarrow 3$
01:00	01:25	Service	(2)	3
01:25	01:30	Depart	(1)	$2 \rightarrow G$
01:30	01:35	Park	(2)	$3 \rightarrow 2$
01:35	01:40	Park	(0)	$1 \rightarrow 3$
01:40	02:05	Service	(0)	3
02:10	02:15	Depart	(2)	$2 \rightarrow G$
02:15	02:20	Park	(0)	$3 \rightarrow 1$
02:25	02:30	Depart	(0)	$1 \rightarrow G$

Table 1.6: Example of good plan (robust)

Train unit	Track locations
0	$G \rightarrow 1 \rightarrow 3 \rightarrow 1 \rightarrow G$
1	$G \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow G$
2	$G \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow G$

Table 1.7: Sequence of each train unit's position on the shunting yard.

Departure	Train unit
0	1
1	2
2	0

Table 1.8: The matching train units to the departure indices.

Although the the improved plan is more robust, it still fails 18% of cases. These failures occur when train 0 is delayed while train 1 is not, causing train 1 to arrive first. It is impossible for a single sequential schedule to prevent this issue entirely, as the order of arrival actions is based on the predefined schedule. This issue can only be resolved after the disturbance is observed. One approach to address this is through a repair method that generates a new plan starting from the disrupted state. Another solution method that avoids this issue entirely is by using online planning, which is able to look at the current state and makes decisions in response to the disturbances. If a good policy is used, it becomes possible to solve the problem 100% of the time, ensuring robustness even in the face of uncertainty.

1.2. Motivation

The main goal of this thesis is to develop a method for creating robust initial shunting plans that are likely to be feasible for a large number of potential plan executions. Most existing solutions to the shunting problem have not taken uncertainty into account, and often a fully deterministic environment is used.

We first describe the limitation of the current best solution method. Next, we discuss how previous methods have attempted to address these limitations through the use of a policy. However, this policy-based solution has its own limitation, which leads us to the motivation and purpose of this thesis.

1.2.1. Current Best Solution

Currently, the best solution is based on a local search algorithm, which is able to solve real-world problem instances. The local search method tried to produce robust plans by adding a heuristic value into its objective function, where this chosen heuristic has shown to be associated with the robustness of shunting plans.

While the local search method has shown success in finding feasible shunting plans, it does not have an explicit strategy in the same way that humans construct plans. No apparent strategy is found in the generated shunting plans, which is caused by the random search operations that are used to find the solutions. In contrast, human planners tend to follow specific strategies when creating shunting plans, that are more understandable. For instance, in case of the shunting yard the "*Kleine Binckhorst*" (1.1), a strategy is to store incoming trains at tracks 56 to 59 upon arrival, after which the trains are

scheduled for cleaning and washing on the dedicated tracks. When their service tasks are finished, the trains are parked at tracks 52 to 55 until their scheduled departure.

Another limitation of existing solutions is that a fully static environment is typically assumed, where everything is completely deterministic. In a deterministic environment, the arrival times of incoming trains and the duration of actions are known with certainty. However, in practice, the environment is dynamic and disturbances can occur, such as trains being delayed or actions taking longer than expected. These disturbances that happen during the execution of the plan may cause the remainder of the plan to become infeasible.

1.2.2. Policy Solutions

Various other solutions have been proposed to address the uncertainty in the shunting problem without relying on an initial plan. Instead, these methods can make decisions on-the-fly, reacting to observed outcomes. The output of such methods can be described as a *policy* $\pi(s)$, which essentially functions as a mapping from states to actions. This *reactive* decision-making approach offers a higher level of flexibility compared to methods that create a single initial plan, as it can make more informed decisions based on the current state.

The use of a policy framework can allow the solution to be more interpretable. Previous work has developed policies that follow understandable rules, aligning more closely with the decision-making of human planners. Other works have tried various machine learning techniques with the hope that the policy is able to learn patterns for its decision-making which can be interpreted by humans, and make the solution more predictable.

However, there are several practical advantages to having an initial shunting plan that is made in advance. For instance, it allows for the assignment of crew members and facilitates preparations for upcoming operations. Since reactive planners do not create a single initial plan, they cannot make these preparations. Additionally, if the policy used during execution is suboptimal, it may lead to decisions that result in unsolvable situations later on, without any way to resolve these problems. In contrast, an initial plan can be validated to ensure its feasibility under normal conditions, providing some level of confidence in the solution.

1.2.3. Extracting Shunting Plans from Policies

A policy can be used to generate an initial shunting plan through a process known as a *rollout*. This process repetitively queries the policy in a simulated environment, starting from the initial state s_0, and continuing until the end of the problem instance is reached. During this process, each chosen action by the policy is recorded, forming a sequence of actions that represents the initial shunting plan. This sequence of action can then be repeated during the actual execution of the plan.

$$s_0 \xrightarrow{a_0=\pi(s_0)} s_1 \xrightarrow{a_1=\pi(s_1)} s_2 \xrightarrow{a_2=\pi(s_0)} ...$$

However, preliminary findings have indicated that shunting plans derived from such policies exhibit limited robustness. This means that these plans are prone to becoming infeasible during execution, primarily because the traditional policy-rollout process generates an action sequence without considering the various alternative outcomes for actions that possess variability in their results. This limitation arises from the fact that the rollout typically relies on a single realization, rendering the overall plan vulnerable to other potential plan realizations.

1.2.4. Purpose of this Thesis

The inability of a policy to create shunting plans that are robust leads us to the purpose of this thesis. Our overall goal is to develop a method capable of creating robust shunting plans. Although the policy itself may not be able to directly produce such robust plans through a conventional rollout, we believe that it still contains valuable information that could be used in a meaningful way. Our hypothesis is that the limited robustness of plans generated via policy-rollout comes from the fact that only a single realization is considered, thereby neglecting all alternative outcomes for each action. To test his hypothesis, we exploring new approaches that do incorporate the alternative outcomes of each action into the planning process.

1.3. Contributions

We present two distinct methods for extracting robust initial shunting plans from a given policy. The first method, referred to as the Probabilistic Action Planner (PAP), aims to take every state that may be encountered into account. Its goal is to maximize the likelihood of successfully achieving the desired outcome for each of these states by selecting actions that are good in most scenarios.

The second approach, referred to as the Adaptive Difficulty Algorithm (ADA), does not explicitly account for every possible state, but tries to account for them indirectly by inserting additional buffer times to each activity.

Figure 1.6 shows how each approach handles the outcomes of actions with probabilistic durations. The baseline policy-rollout bases its decisions solely on the typical or average duration of these actions.

In contrast, the Probabilistic Action Planner (PAP) considers every possible outcome, and the corresponding states it encounters caused by the outcome. Each resulting state influences the chosen action, with each state weighted according to its probability of occurrence. The goal is to maximize the overall likelihood of successfully reaching the goal from the initial state. Since computing the exact probabilities becomes intractable for a larger number of outcomes and decision moments, the approach relies on approximation techniques instead. These approximations are derived from the given policy.

On the other hand, the Adaptive Difficulty Algorithm (ADA) adopts a different strategy. Rather than directly considering every potential state, ADA indirectly addresses them by looking at a more delayed outcome. This approach is expected to be more efficient because it focuses on a single state, although it comes at the cost of overlooking a subset of potential outcomes when the duration is even longer than what is considered. There is a limit to the delay ADA can consider, as excessively long durations will make it impossible to schedule all the required activities within the given timeframe.

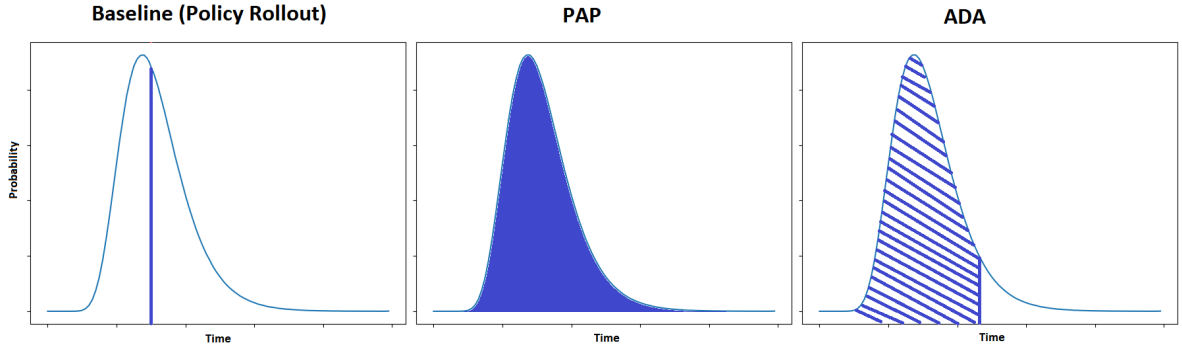

Figure 1.6: Overview of the proposed solution approaches. The baseline is a regular policy rollout, which only looks at a single outcome for each activity. The Probabilitic Action Planner (PAP) tries to take every possible state thay be encountered into account. The Adaptive Difficulty Algorithm (ADA) strategically focuses on a single outcome, recognizing that all other outcomes which take less time can always wait to get to the considered state.

1.4. Research Questions

The overall goal of this thesis is to develop a method that is able to generate robust initial shunting plans. For this thesis, we assume that there is already a policy available that has been optimized for a deterministic environment. However, the issue is that this policy cannot directly produce initial shunting plans that are robust. Traditional policy rollouts result in action sequences that fail to account for the majority of potential outcomes, as they are based on a single realization. Consequently, this renders the overall plan vulnerable to variations in its execution. Nonetheless, there is potential in using the policy to create robust plans by using they policy in a way that considers multiple potential action outcomes during the plan's construction. Therefore, the main research question is:

> **Given a policy optimized for finding feasible solutions, to what extend is the robustness of the initial shunting plan increased when all possible action outcomes are considered during the construction of the plan?**

This thesis focuses on a sequential formulation of the problem, where the output is represented as a sequence of actions, making it a Total Order Schedule. This type of schedule is not the conventional representation of a shunting plan, which is typically represented as a Partial Order Schedule. The issue is that simply repeating an action sequence directly would restrict the plan's flexibility, as each action has to be repeated in the exact same order, which may not always be necessary. Therefore, we ask how we can interpret the resulting sequence as a Partial Order Schedule through a mapping from action sequences to Partial Order Schedule with the following sub-question:

1. **How can action sequences be used as initial shunting plans by mapping them to Partial Order Schedules?**

The main research question is about whether the robustness of initial shunting plans can be improved by considering all possible outcomes during the construction of the plan. A sub-question that we need to answer first, before the main question can be tested, is:

2. **How can alternative outcomes be considered during the planning phase, either directly or indirectly?**

We introduce two distinct methods for extracting robust initial shunting plans from a given policy. The first approach, referred to as the Probabilistic Action Planner (PAP), is designed to be problem formulation-agnostic, with the goal of accounting for every potential state that could be encountered. The second method, referred to as the Adaptive Difficulty Algorithm (ADA), strategically focusses on a single state at each decision moment by exploiting the transition structure inherent in the problem formulation. While ADA is expected to be fast since it only considers one state at a time, it may come at the cost of potentially less robust plans, as it overlooks a subset of alternative states. We test the performance of both methods and analyze the differences and trade-offs between them. The sub-question is:

3. **Is it better to implicitly (ADA) or explicitly (PAP) consider every possible outcome, in terms of the robustness of the plans and the computational efficiency?**

1.5. Outline

- Chapter 2 includes a review of existing literature about the shunting problem.
- Chapter 3 offers the necessary background information and explains the algorithms that are used in subsequent chapters.
- Chapter 4 answers the first research question (1) by explaining how an action sequence is used as an initial shunting plan.
- Chapter 5 answers the second research question (2) by proposing two distinct methods that all possible outcomes into account, either directly or indirectly.
- Chapter 6 describes the problem instances that are used to test the proposed solution methods on.
- Chapter 7 compares the proposed approaches to each other, which is used to answer the third research question (3). It also provides individual analysis of the individual approaches to gain more understanding of how they work.
- Chapter 8 gives a direct answer to each of the research questions, and discusses the limitations of this thesis.
- Chapter 9 concludes the thesis and suggests ideas for future research.

2

Literature Review

2.1. Exact Approaches

The shunting problem was first introduced by Freling et al. [14] They only focus on the matching and parking subproblems. They propose mathematical mixed integer models that solve these two subproblems. First, the arriving trains are matched to the departing train. This is done while keeping the train units together as much as possible, in order to minimize unnecessary shunting operations. After a matching has been created, a parking track is assigned to each of the train units, such that track length is never exceeded and no crossing occur.

Lentink et al. [20] extended the problem by also considering the routing aspect. First the matching is determined using the method from Freling et al. Then before solving the parking, the routing time is estimated. This estimate is then used in the next step to solve the parking subproblem. Finally the actual routes are computed. The authors also mention that a robust solution would only park units of the same subtype on each shunt track.

Instead of decomposing the problem into smaller subproblems, Kroon et al [18] solved the matching and parking problem simultaneously. They introduced the concept of a mixed track, which is a track that has multiple units of a different subtype parked. If all units of a track are the same subtype, the order of the units would be irrelevant. In the objective function they penalize having different types on a track to make the solution more robust.

While these methods may be effective for smaller problem instances, they are not as practical for larger ones due to their slower processing times, which limits their usefulness in practical situations.

2.2. Local Search Heuristic

The current best solution is based on local search.

Jacobsen et al. [16] compared three different meta-heuristics, including guided local search, guided fast search, and simulated annealing, and found that their results were similar to the previous MIP model, but only took seconds to run instead of multiple hours. However, these results were obtained on smaller instances with less than ten trains, only a single service task per train, LIFO tracks, and transportation times set to zero.

Van den Broek [11] later proposed an integrated local search approach using simulated annealing, which outperformed all previous methods and was able to solve the complete problem for real-world instances [10]. It is currently being used by NS and several improvements have been proposed to enhance its performance, including improvements to the initial plan, and using machine learning to predict the feasibility of a search and decide whether to continue or abort the search [22].

In the thesis by Stelmach [26], a proactive-reactive rescheduling strategy was analyzed, where the local search method itself is also used for repairing infeasible plans. Kleine [17] and van der Broek [9] analyzed several robustness measures to predict the feasibility of a plan. Their findings include that measures based on the normal approximation method or minimum slack showed the highest correlation with robustness of a plan. With these findings, they were able to improve the robustness of the created plans by adding the robustness measures into the objective function.

These local search methods are effective at finding feasible solutions relatively quickly for real-world instances. However, due to the random nature of the search operations, no apparent strategy is found in the resulting shunting plans. The correlation between the minimum slack and the robustness of a plan is used as the basis for the the proposed algorithm in section 5.2.3

2.3. Policies for the Shunting Problem

2.3.1. Constructing a Policy

Some research has been performed that looks at greedy heuristics to generate a solution

Van den Akker et al. proposed two solutions [1], a greedy heuristic and a dynamic programming solution. Their approach is based on the ”be ready for departure” principle, meaning that trains should already be combined and waiting in the correct configuration before their departure. To accomplish this goal, the problem is solved backwards. Departed trains are assigned to a shunt track where they came from, and later matched to arriving train units. It is able to create feasible solutions quite fast for smaller problem instances, but as mentioned by the authors, may not be of much practical use without additional improvements.

Haahr et al. [15] compared multiple solution approaches, including constraint programming, column generation, and a randomized construction heuristic. The construction heuristic is a two-stage method that solves matching and parking separately, iteratively evaluating one path very quickly with random seeds until a feasible solution is found or a time limit is reached. The randomized construction heuristic was able to solve almost all instances within a fraction of a second. However, some harder and artificially generated instances were left unsolved. The method also did not consider the servicing aspect of the problem.

Beerthuizen [5] proposed two greedy strategies based on container stacking, including a Type-Based-Strategy which stacked units based on similarity of train type, and an In-Residence-Time-Strategy which assigned a priority to each train unit based on the departing sequence. Donker [24] looked at the influence of resource and temporal constraints. Results indicated that temporal constraints had a larger influence on the problem.

Greedy heuristics are useful because they run in a very short amount of time and can quickly generate plans. However, they may not always provide feasible solutions on their own and would have to be used in combination with additional methods. In Appendix C, we look into some addition search methods that use a greedy heuristic as the based policy An analysis of the policies based on the greedy heuristics is given in Appendix B

2.3.2. Learning a Policy

Recently, there has been some amount of work on using machine learning to solve the shunting problem.

Peer et al. [23] modeled the problem as a Markov Decision Problem and used deep Q-learning to learn a policy for deciding which actions to take. To incorporate uncertainty, the model only knows about the train composition of the first m arrival and departure events, and for events further in the future, it only knows the total number of carriages. In addition to looking at the number of solved instances, the authors also measured the consistency of the generated plans. This work was later extended to include servicing [19].

Another approach was taken by Zhong [27], who used supervised learning with data generated from a local search method. A neural network was used to learn a policy and a tree search was used to find a feasible solution. The authors asked the question what search strategy works best for the problem using the policy network as a heuristic, and found that the Monte Carlo Tree Search method was the best out of the three that were tested.

In his work, Bao [4] presented a solution that uses Decision Trees to train a predictive model for determining the next action in a sequential decision-making process. This approach is able to quickly react in the face of uncertainty. One of their main considerations is about the stability of the generated plan. The robustness is defined as the number of unique solutions generated from the test set, with the goal to solve more problems with fewer unique solutions.

Nieuwelaar [21] proposed a solution that involves two sequential steps: first, a deterministic agent is trained using iterative Bayesian Optimization, and second, when the deterministic agent cannot distinguish between different actions, the Monte Carlo Tree Search method is used. They also analyzed

the impact of problem-related contextual information, and found that it has poor generalization to unencountered problem instances.

The methods discussed provide a policy that can be used in every possible state to determine the next action, making them suitable for handling disturbances. However, it should be noted that the best action provided by the policy does not guarantee a feasible plan for the remainder of the problem. Moreover, all of these methods still assume a deterministic environment.

3
Background

3.1. Markov Decision Process (MDP)

A Markov Decision Process (MDP) is a mathematical framework for modeling decision-making in environments that exhibit the Markov property. The Markov property states that the future state of a system depends only on its current state and not on its past states.

An MDP is formally defined as a tuple $M = (S, A, T, R)$ where

- S is the set of all possible states (the state space)
- A is the set of all possible actions (the action space)
- $T(s'|s, a)$ is the probability that action a in state s will lead to state s'
- $R(s, a)$ is the reward after applying action a in state s

In an MDP, an agent interacts with its environment by taking actions at each decision moment. The agent receives a reward for each action it takes and the environment transitions to a new state as a result of the action. The agent's goal is to maximize its cumulative reward over time by selecting actions that lead to states with high expected rewards.

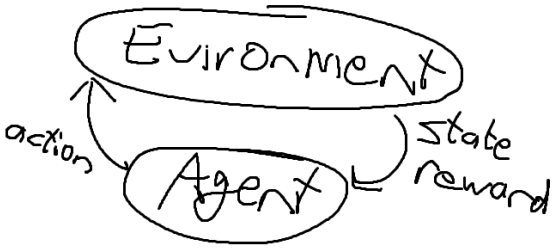

Figure 3.1: The interaction between an agent with its environment in a Markov Decision Process

The solution to an MDP is represented by a policy, denoted as $\pi(s)$. The policy is a function that maps states to actions. It determines which actions that agents chooses. The objective is to find a policy that maximizes the expected reward.

The value of an action a in a state s under a policy π is notated with:

$$Q^{\pi}(s, a) = E_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a]$$

3.2. Partially Observable MDP

A Partially Observable MDP (POMDP) is a framework that extends the MDP. In contrast to an MDP, the current state is not perfectly known. The environment provides observations to the agent, which has to be used to estimate the actual state.

A POMDP is defined as a tuple (S, A, T, R, B, O), where

- S, A, T, R are the same as in MDP
- B is belief vector of belief state, which represents the agent's estimate of the current state
- O is the set of observations that the agent can observe about the current state from the environment

3.2.1. Belief State

The belief state represent the agent's uncertainty about the current state of the environment. For each possible state, the belief state assigns a corresponding probability, indicating the likelihood of the agent being in that particular state.

The belief state **b** is represented as a belief vector:

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

- **b** is the belief vector
- b_i element corresponds to the probability of being in state i
- N total elements, where N is the number of possible states

Since the belief vector **b** is the probability distribution over the possible states, is satisfies the properties of a probability distribution, meaning each element has to be non-negative ($b_i \geq 0$), and the sum of all elements equal 1 ($\sum_{i=1}^{N} b_i = 1$)

The value of a particular action in a belief state can be computed by the weighted sum of the action values for each individual state in the belief vector:

$$Q(\mathbf{b}, a) = \sum_{i=1}^{N} Q(s_i, a) \cdot b_i$$

The value $Q(b, a)$ represents the expected return when taking action a in the belief state **b**.

3.2.2. Particle Filter

A particle filter can be used to approximate a probability distribution. This can be useful when there is large state space, as it might be intractable to keep track of the exact probabilities in the belief state.

The particle filter is a set of particles, where each particle corresponds to a specific state. These particles are sampled from the set of possible states, according to their probabilities. Mathematically, the set of particles $\mathcal{P}$ can be represented as:

$$\mathcal{P}_t = \{(s_1, p_1), (s_2, p_2), ..., (s_K, p_K)\}$$

- K is the total number of unique states in the set
- p_i corresponds to the number of particles corresponding to state s_i

The particle filter is used to estimate the true probabilities in the belief state. The probability of being in a specific state s can be estimated using the following formula:

$$\hat{b}_t(s_i) = \frac{p_i}{\sum_{j=1}^{N} p_j}$$

- $\hat{b}_t(s)$ is the estimated belief in state s at time step t

3.2.3. Non-Observable MDP

A Non-Observable MDP (NOMDP) is a special case of a POMDP, where the agent receives no observations during the entire process. This can be modeled by letting $O = \{o\}$, where the same observation is given for each observation moment, such that is reveals no information about the current state. To solve a NOMDP, the agent has to choose its actions solely on its own predictive model, as it cannot improve its knowledge about its current state.

4

Representation of Initial Shunting Plans

This chapter focuses on answering the first research question (1), which is:

1. **How can action sequences be used as initial shunting plans by mapping them to Partial Order Schedules?**

To answer this question, we first explain the issue of using the output of a sequential problem formulation to create initial shunting plans compared to using the Partial Order Schedule, which is used in practice. Next, we describe how we can map the action-sequence to a Partial Order Schedule. Afterwards, we explain how a Partial Order Schedule is executed in a sequential problem environment.

4.1. Sequential Problem Formulation

In this thesis, we consider a sequential formulation of the shunting problem, modeled as a Markov Decision Process (MDP). The states within this model describe the current positions of all trains on all tracks with the required service tasks that need to be performed on each train and the ongoing activities. The action space consists of the possible movements, service tasks and the possibility to wait for the next decision moment. The transition function is mostly deterministic in its effect, as movement will always transfer the related train to its chosen destination and service actions eventually will be completed. However, the transition function introduces variability in the duration of the actions. The MDP formulation is explained in greater detail in Appendix A.

The output of this problem formulation results in a sequence of actions denoted as $\tau = (a_0, a_1, ..., a_n)$. This sequential nature comes from the fact that each action is chosen one after the other. Since we are interested in creating initial shunting plans, the chosen decisions from the action sequence have to be repeated during the execution of this plan. However, if we simply repeat the action-sequence exactly as a total order schedule, it can become overly restrictive in practice. The conditions during executions of the plan often deviate slightly from each other. This may lead to instances where the next action in the sequence is infeasible, even though the overall decisions stored in the sequence would be feasible under different circumstances, but just not in the exact same order as represented in the sequence.

In practice, an initial shunting plan is represented as a partial order schedule (POS). This POS contains all the scheduled activities and their relative order of execution. This representation allows the possibility of performing activities in parallel, as long as they don't interfere with each other. To utilize the benefits of a POS, we implement a mapping that transforms each action-sequence into a corresponding partial order schedule. This mapping effectively decouples the action-sequence from any irrelevant orderings within the sequence. In summary, even though the initial shunting plan is constructed as a sequence of actions, it is subsequently interpreted and executed as a partial order schedule.

4.2. Create POS from Action Sequence

The partial order schedule is represented with a set of train activities A and a set of precedence relations POS. The activities consists of arrivals, departures, movements, and services, with each activity being specific to a particular train. The precedence relations ensure that activities of the same train or activities using same service resource are performed in the correct order. Each action that involves a movement requires the movement-resource, as only one train can move at the same time. Each action to start a service tasks requires the service-resource, as a crew can only service a single train simultaneously. Actions on the same train need to be performed in their specified order.

During the planning phase within the simulated environment, the ideal next action is not always available yet. There are cases where it is necessary to wait for a specific resource to become available before the activity can be performed. In these cases, the agent may opt to wait until the preferred next action becomes feasible. However, the waiting actions themselves are irrelevant in the creation of the partial order schedule, because waiting for the completion of previous activities is implied by the precedence relations. Therefore, all waiting actions in the action-sequence are disregarded when converting the sequence to a partial order schedule.

To transform the action-sequence into a partial order schedule (POS), we start with an empty activity set A and an empty precedence set POS. We add each activity with their corresponding precedence relations by iterating over the actions in the sequence. Each waiting action is ignored. Every other encountered action is added to the activity set A. For the resource that the action requires, we add a precedence relation with the previous action that required the same resource. For the train that is associated with the action, we add a precedence relation with the previous action of that train.

4.3. Execute POS sequentially

Once an initial shunting plan is formulated as a partial order schedule derived from an action-sequence, the plan can be executed within the sequential problem environment. During each decision-making moment, the agent receives a set of valid actions from the environment, allowing it to select the next course of action. The agent chooses the next action action according to the POS. However, there may be instances where none of the available action should be chosen according to the POS. For example, it may be needed to wait for the completion of a currently ongoing movement or service task, so the corresponding resource can be freed up In such cases, the agent chooses the the ”wait” action, effectively delaying the next decision until the conditions for the next action in the POS are satisfied. If the ”wait” action is also unavailable, the execution is considered to be failed, and the simulation episode is terminated.

4.4. Example

Figure 4.1 shows a simplified example of a Partial Order Schedule. It contains two trains with a movements for each train and a service activity. Table 4.3 shows two different realizations. Although both action sequences are almost completely different, they both correspond to the same plan.

In the first realization, the second train is immediately ready to move after the first movement is finished. The service tasks of the first train is chosen right after the second is instructed to move.

In the second realization, the second train is yet ready to start moving after the first movement, due to some other precedence constraint that is not yet satisfied. The service tasks of the first train can be started without any delay. Before the second is able to move, it first needs to wait for its previous action to be finished.

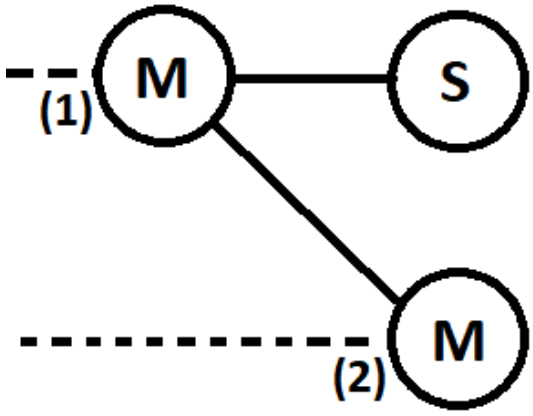

Figure 4.1: Simplified example of a Partial Order Schedule. First, train 1 makes a movement towards another track. After this movement is finished, train 2 is allowed to move. At the same time, train 1 can start its service task.

Action	Train	Tracks	
...	...	...	
Move	(1)	$1 \rightarrow 2$	
Move	(2)	$3 \rightarrow 1$	
Service	(1)	2	$\longleftarrow$

Table 4.1: Action Sequence 1

	Action	Train	Tracks
	...	...	...
	Move	(1)	$1 \rightarrow 2$
$\longrightarrow$	Service	(1)	2
$\longrightarrow$	Wait	-	-
	Move	(2)	$3 \rightarrow 1$

Table 4.2: Action Sequence 2

Table 4.3: Two possible realizations of the same initial shunting plan shown in Figure 4.1. Since both the service action and the second movement can start at the same time, their order in the sequence does not matter.

5

Creating Robust Action-Sequences

This chapter focuses on answering the second research question (2), which is:

2. **How can alternative outcomes be considered during the planning phase, either directly or indirectly?**

To answer this question, we propose two approaches for creating action sequences. Both methods are aimed to address the limitation of using a regular policy-rollout to extract an action sequence from a policy. The issue with a normal rollout is that most of the possible outcomes are not considered, as it is only based on a single realization, making the overall plan not robust to other plan realizations. So instead of doing a normal policy-rollout, we propose do it in a way such that other possible outcomes are taken into account as well.

The first method, referred to as the Probabilistic Action Planner (PAP) aims to account for every possible outcome directly. The second method, referred to as the Adaptive Difficulty Algorithm (ADA) strategically focuses on a single outcome to account for most alternatives indirectly.

5.1. Probabilistic Action Planner (PAP)

During the construction of the initial shunting plan, it is not known with certainty the exact states that are going to be encounter during the execution. This uncertainty arises from the necessity to create the plan in advance, before any observations about the outcomes of any chosen action can be made. This means that the problem has the characteristics of a Non-Observable Markov Decision Process (NOMDP), because no observations can be made during the entire planning process.

By viewing the problem as a NOMDP, we can use existing techniques to solve this version of the problem effectively. We use the notion of a belief state to represent the uncertainty during planning, and approximate its probabilities values using a particle filter. Note that since we cannot make any observations, the belief state is solely used for predicting future states, and can never improve its estimation of the current state over time through observations. To solve the NOMDP, we use Monte-Carlo planning.

First, we explain how to problem is transformed to a NOMDP, and what the differences are compared to the original MDP problem. Next, we describe what the optimal action-value would be if we would already know what the best continuation is after each action at any decision moment. However, since we don't know the best continuation yet during planning, we use Monte-Carlo simulations of the given policy to approximate the action-values. Given the exponential growth in the number of possible states with the increase in decision moments and outcomes, keeping track of the exact belief state is computationally infeasible. Therefore, we use a particle filter-based approach to represent the belief state. Finally, we provide the pseudocode implementation of the overall PAP procedure.

5.1.1. Transform MDP to NOMDP

In this thesis, we consider a sequential formulation of the shunting problem, modeled as a Markov Decision Process (MDP). The states within this model describe the current positions of all trains on all tracks with the required service tasks that need to be performed on each train and the ongoing activities. The action space consists of the possible movements, service tasks and the possibility to wait for the next decision moment. The transition function is mostly deterministic in its effect, as movement will always transfer the related train to its chosen destination and service actions eventually will be completed. However, the transition function introduces variability in the duration of the actions. The MDP formulation is explained in greater detail in Appendix A.

Since we cannot make any observations during the entire planning process, the original MDP problem can be viewed as a Non-Observable MDP (NOMDP). Despite this shift, the problem definitions can in principle remain unchanged. The state space, the action space, the transition function and the reward function all remain exactly the same, with the only difference being the addition of a belief state to track the probabilities associated with being in each state. Note that the belief state itself does not require any knowledge about the representation of the individual states. It only needs to know about the corresponding probability of being in each particular state.

Although the action space remains the same as the original MDP, some additional definitions are useful to include to describe how the actions interact with the newly introduced belief state. An important note is that it may be be possible for all actions to be executed in any state. For example, a parking action can only be selected when there is no ongoing movement. From the original MDP formulation, we are given a set of valid actions for every state. But when the current state is uncertain, actions may be feasible only in a subset of possible states in the belief state. The set of valid actions in a belief state is defined to be the union of valid actions across all individual states, as mathematically expressed by

$$\texttt{validActions}(\mathbf{b}) = \bigcup_{i=1}^{K} \texttt{validActions}(s_i)$$

The output to the NOMDP version of the problem is represented as a single sequence of actions. This action sequence will be used as the initial shunting plan, where the sequence describes the relative order in which activities are scheduled. Any waiting actions are therefore not relevant, as they do not correspond to specific activities. Most waiting actions are only needed as prerequisites for other actions, such as waiting for the completion of an ongoing movement before another movement can start. Therefore, waiting actions are filtered out in the Partial Order Representation of the final plan, as described in Chapter 4. What this means during the planning phases, is that any additional insertions of waiting actions can be made, without influencing the resulting shunting plan.

$$POS(\{A, B\}) = POS(\{A, \texttt{wait}, B\}) = POS(\{A, \texttt{wait}, \texttt{wait}, B\}) = ...$$

To use this property, we can allow actions to be chosen in states that require additional waiting actions before the chosen action can be applied. What this means for the NOMDP definition, is that any chosen action that is not valid for a particular state within the belief state can still be attempted by repeatedly applying a wait-action until it becomes feasible. Only in cases where waiting does not lead to a state where the chosen action becomes possible is the action, along with the associated rollout, considered invalid. These cases arise due to the uncertainty of the transitions, making it impossible to know with certainty whether the pre-conditions of an action will be met by waiting first.

5.1.2. Optimal Action Value

The problem contains stochastic transitions that reflect the real-world, such as probabilistic durations for the service tasks and probabilistic arrival times around their scheduled time for each train. The optimal plan is a single sequence of actions that maximizes the probability of reaching the goal state:

$$\tau^* = \texttt{max}_\tau(E[R(\tau)])$$

The action-value function of a trajectory gives the expected return if you start in state s, take an arbitrary action a, and then act according to the given trajectory τ in the environment

$$Q^\tau(s,a) = E[R(\tau)|s_0 = s, a_0 = a]$$

For a belief state, the action-value is the weighted average of all the action-values from the belief state.

$$Q^\tau(\mathbf{b},a) = \sum_{i=1}^{N} Q^\tau(s_i,a) \cdot b_i$$

The optimal action-value of the belief state uses the best future action sequence, that maximizes the weighted value across all states. Note that we have use the same future action sequence for each individual state.

$$Q^{\tau^*}(\mathbf{b},a) = \texttt{max}_\tau Q^\tau(\mathbf{b},a)$$

$$= \texttt{max}_\tau \sum_{i=1}^{N} Q^\tau(s_i,a) \cdot b_i = \texttt{max}_\tau \sum_{i=1}^{N} E[R(\tau)|s_0 = s, a_0 = a] \cdot b_i$$

5.1.3. Approximate Action Value

Calculating the optimal action-values leads to two significant issues.

The first issue is the number of possible states increases exponentially with the number of decision moments and outcomes, so keeping track of the exact belief state is computationally infeasible. Therefore, we use an method known as a particle filter, which provides an approximation of the belief state, which we denoted as $\mathbf{b} \approx \hat{b}$. The details about the belief state and its approximation are elaborated in the next subsection.

The second challenge arises from the recursive relation in the optimal action-value calculation. Determining the value of an action requires knowledge of the optimal sequence of future actions. However, this sequence is determined by the action value function. This makes calculating exact values an exponentially complex problem. To address this, we opt for a simpler solution, where we replace the unknown optimal future action sequence with rollouts of the given policy, expressed as $\tau^* \approx \tau \sim \pi$. Since it is assumed that this policy is optimized for reaching the goal state, it might provide a reasonable estimate of the value that the best overall plan for the current belief state would give. In other words, we sample rollouts from the optimized policy starting from each individual current state as replacement of the overall optimal sequence for every current state.

Combining these two changes, we get the following formula as an approximation to the optimal action value:

$$Q^\pi(\hat{b},a) \approx \texttt{max}_\tau Q^\tau(\mathbf{b},a)$$

$$= \sum_{i=1}^{N} Q^\pi(s_i,a) \cdot \hat{b}_i = \sum_{i=1}^{N} E_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \cdot b_i$$

This formula allows us to make decisions by selecting the action with the highest estimated value at each decision moment, optimizing the expected reward:

$$a_t = \texttt{argmax}_a(Q^\pi(\hat{b}_t,a))$$

5.1.4. Approximate Belief State

The shunting problem has a very large state space, as there are many possible ways to assign the trains to the tracks, and a wide range of potential value for the current time. Moreover, the number of state in the belief vector increases exponentially with each action taken. This exponential growth makes it impractical to calculate the exact probabilities of the belief state. To address this issue, a particle filter is used as an approximation technique.

After selecting an action a, the belief state is updated according to the action and the possible transitions of the environment. To obtain the updated belief state, new particles are sampled. First, a state s is randomly sampled from the existing belief state. If the chosen action a is not feasible in the sampled state s, the a_{wait} action is first repeatedly applied until either the action becomes available, or the a_{wait} action becomes unavailable

When the chosen action a is feasible, a new state is obtained from the simulation environment by applying the action to the state s. Since the transitions are non-deterministic, there are several potential outcomes. The new state s' is then added to the set of new particles. This process continues until K new particles are added. The pseudocode of the procedure is provided in Algorithm 1.

Algorithm 1 Update Belief State

```
1:  function updateBeliefState(𝒫, a, M)
2:      𝒫' ← ∅
3:      while |𝒫'| < M do
4:          s ← sampleState(𝒫)
5:          while a ∉ validAction(s) and a_wait ∈ validAction(s) do
6:              s ← apply(s, a_wait)
7:          if a ∈ validAction(s) then
8:              s' ← apply(s, a)
9:              𝒫' ← 𝒫' + s'
10:     return 𝒫'
```

The initial state s_0 is always known with certainty at the beginning of the planning phase. Therefore, the initial belief vector can be initialized by assigning zeros to all possible states except for the initial state s_0, which is assigned a value of 1. In case of the particle filter, only a single particle corresponding to the initial state needs to be included.

$$\mathcal{P}_0 = \{(s_0, 1)\}$$

5.1.5. Optimize Selection

To optimize the selection of actions at each iteration, we apply the Upper Confidence Bound (UCB). In each iteration, an actions is chosen based on a trade-off between exploration and exploitation. The UCB selects actions based on their estimated value and exploration potential. The more rollouts are performed after selecting a certain action, the more accurate the estimate of that action becomes. This allows the algorithm to focus on actions with higher values, as they are more likely to be chosen as the final action. Actions that are unlikely to be chosen as the final action do not require as much exploration. Instead, the algorithm prioritizes exploring actions with higher values to improve their accuracy and enable a more reliable comparison between them. The action selected in the n-th iteration is given by:

$$a_n = \texttt{argmax}_a(Q^\pi(\hat{b}_t, a) + C\sqrt{\frac{\ln(n)}{N(a)}})$$

- a_i the action selected in the n-th iteration
- C is the exploration constant
- $N(a)$ is the number times a was selected

5.1.6. Implementation

We apply Monte Carlo rollouts to evaluate the value of each possible action. The value of each is the expected reward that will be received after taking the action, which in this case is the probability of reaching the goal state without any constraint violations. The pseudocode of the algorithm is provided in Algorithm 2

Algorithm 2 Probabilistic Action Planner

function ProbabilisticActionPlanner(I, s_0, N, M)
    sequence $\leftarrow \emptyset$
    $\mathcal{P} = \{(s_0, 1)\}$ ▷ Initialize the particle filter
    while $\mathcal{P}$ is **not** terminal **do**
        for $a \in \texttt{validActions}(\mathcal{P})$ **do**
            $V(a) \leftarrow 0$
            $N(a) \leftarrow 0$
        for $i \leftarrow 1$ to N **do**
            $a \leftarrow \texttt{max}_a(\frac{V(a)}{N(a)} + C\sqrt{\frac{\ln(i)}{N(a)}})$ ▷ Use UCB to select actions
            $s \leftarrow \texttt{sampleState}(\mathcal{P})$
            $s' \leftarrow \texttt{apply}(s, a)$
            $R \leftarrow \texttt{rollout}(s', \pi)$
            $V(a) \leftarrow V(a) + R$
            $N(a) \leftarrow N(a) + 1$
        $a \leftarrow \text{max}_a(\frac{V(a)}{N(a)})$
        sequence $\leftarrow$ sequence $+ a$
        $\mathcal{P} \leftarrow \texttt{updateBeliefState}(\mathcal{P}, a, K)$
    return sequence

5.2. Adaptive Difficulty Algorithm (ADA)

In this section, we introduce the Adaptive Difficulty Algorithm (ADA). This approach is designed to increase the efficiency of the decision-making process compared to the Probabilistic Action Planner (PAP), which uses every possible state when choosing its actions, which can be computationally intensive. At the same time, ADA also aims to address the limitation of a regular policy rollout for extracting robust sequences. Instead of just using the expected outcomes of each action during the planning process, ADA tries to strategically focus on a single outcome, such that most alternative outcomes are also indirectly taken into account.

The main idea behind ADA idea relies on the fact that the transitions are entirely deterministic, except for the variability in the durations each activity takes, and the variability in the arrival times. As a result, it is always possible to transition to a state which is more *"delayed"*, simply by waiting. By only considering the states resulting from delayed outcomes, we indirectly account for all alternative outcomes that are less delayed, as the state resulting from these outcomes are able to reach the considered state by waiting for some time. However, if the amount of delay we consider is too large, the problem might become unsolvable, because there is not enough available time to finish each required task. Therefore, we propose a simple solution, which is to iteratively increase the delay we consider during the planning, until we can no longer solve the problem.

5.2.1. Transition Structure

Within the formulated environment, the effect of each action is mostly deterministic in how the state changes. Any chosen action will always be executed as specified, without impacting unrelated components of the state. The only uncertainty about the resulting state arises from the variability in durations of the activities and the uncertainty about the delays in arrival times. Durations are sampled from the corresponding probability distribution, leading to some actions completing earlier than expected while others experience delays. This variability in durations influences the time of the states at each decision moment, where the longer an action takes, the more delayed the resulting decision time in the subsequent state becomes.

It is important to note that there is the option to choose the ”wait” action. By choosing this action, the current time of the state is increased without making any changes to other components of the state. This allows us to always transition from a ”less delayed” state to a ”more delayed” state simply by waiting for some time, without choosing any other actions.

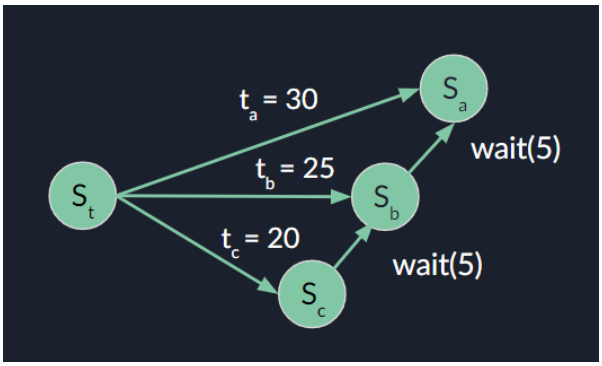

Figure 5.1: After performing an activity in state s_t, there are several possible resulting states, depending on the duration of the activity. The states with a shorter duration (s_c and s_b) can always transition to a state with a larger duration (s_b and s_a respectively) by waiting for some time.

As a result, for every path between a certain state s_i to the goal state, there exists a path to the goal state from all other states s_j that are identical in all aspects except for their current time t. Specifically, the current time of these states (t_j) is lower than that of s_i (t_i). This is possible because these states can always wait for a duration of $t_i - t_j$ to transition to state s_i.

The ability to transition to other states by waiting is exploited by ADA. Rather than explicitly considering every reachable state, only the states that are reached after a larger duration are looked at. By doing so, all alternative states with lower durations are implicitly taken into account. This is because these alternative states can always reach the more delayed state by waiting for the time difference between them.

An important consequence of this approach, is that the environment becomes deterministic again, as the same outcome of each action is used during planning. This deterministic property allow for the possibility of applying additional search techniques that assume a deterministic environment. These additional search techniques can be used to find feasible plans when a single iteration does not immediately lead to a valid shunting plan.

5.2.2. Which Transitions to use?

One question that arises is which transitions should be used during the planning phase. Ideally, the transitions with the largest durations should be selected. By using these transitions, all other possible transitions are implicitly covered. This approach ensures that if there exists a feasible path to the goal state from this most delayed state, all other states also have the ability to follow this path by first waiting some time to reach the most delayed state, after which the path the goal can be followed. In this case, the plan is guaranteed to remain feasible with respect to the considered disturbance model, since there are no transitions that reach a state that cannot reach the goal state.

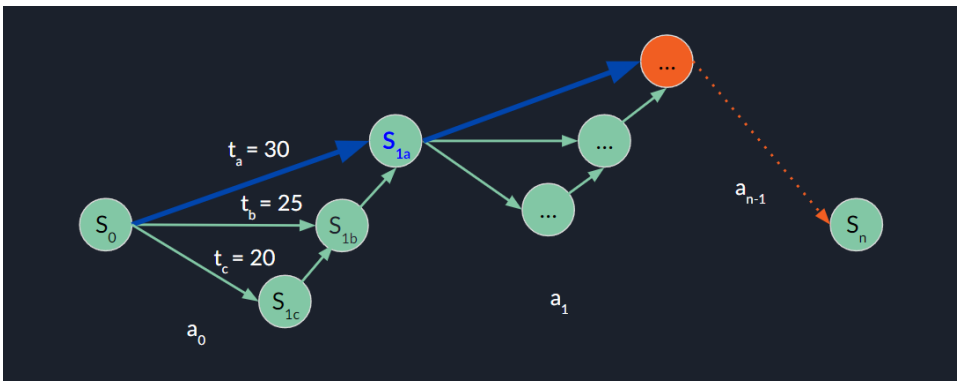

Figure 5.2: The possible transitions to new states after choosing the sequence of actions $\{a_0, a_1, ..., a_n\}$. Each action has several possible durations $t_i \in \{20, 25, 30\}$, which result in a different state S_i.
If the most delayed transition is used to determine the decisions, a feasible path to the goal state is guaranteed for every alternative outcome. However, there might not always be such a path, indicated by the red dotted line to the goal state S_n.

An issue arises when using the worst-case transition at every decision moment during the planning phase. In such an environment, it may be impossible to reach the final goal state since there may not be enough time to allow for waiting after each action. For instance, if the maximum possible duration of every service task is used, there might not be enough total available time to finish all the required service tasks before the given deadlines. Consequently, it may not be feasible to achieve robustness against every possible realization of the durations. However, the objective is not to guarantee 100% feasibly of the plans, but to be as robust as possible for the majority of scenarios.

The solution needs to use durations that cover a significant portion of the possible transitions while still ensuring feasibility. The aim is to select the durations that cover most of the potential variations in transitions without sacrificing the ability to generate a viable shunting plan. By striking this balance, the resulting plan is not only valid but also likely to remain feasible for the majority of possible transitions.

5.2.3. Find Maximum Threshold

The Adaptive Difficulty Algorithm (ADA) aims to find the optimal durations to use for the transitions that cover the maximum number of possible outcomes while still ensuring the ability to construct a feasible plan using these transitions. By increasing the considered durations, the algorithm effectively increases the slack for each activity. This additional time acts as a buffer, allowing each activity to be delayed without effecting the rest of the solution. However, if this buffer time is too high, it won't be possible to create a feasible plan, since the buffer time is not used to progress toward the goal state. On the other hand, setting the buffer time too low compromises robustness against certain outcomes. When an activity exceeds the buffer time, it affects subsequent actions by reducing their available time, potentially risking the ability to meet a deadline.

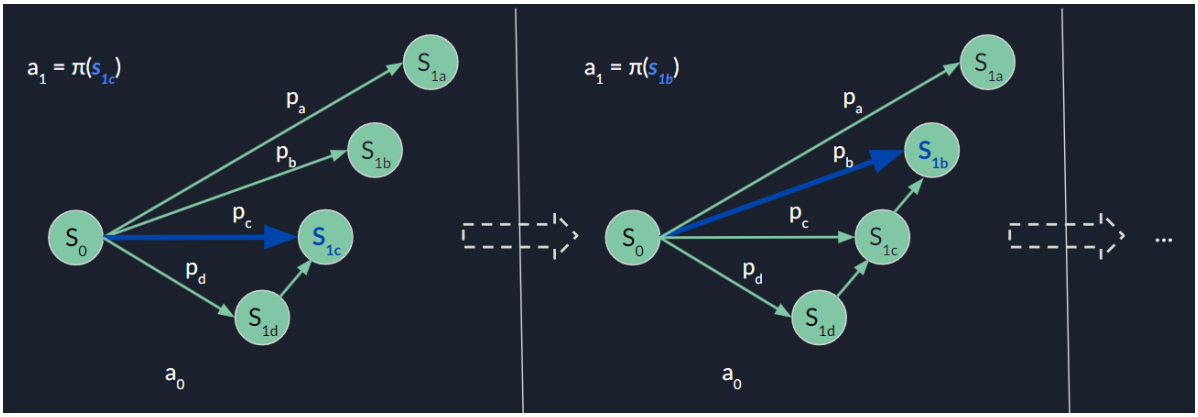

Figure 5.3: The Adaptive Difficulty Algorithm iteratively finds a feasible solution in a deterministic environment. The durations that used for the transitions are increased each iteration, providing more buffer time to each activity, and covering a larger number of possible outcomes.

A shunting plan is considered valid, if it reaches the goal state under the expected circumstances. The Adaptive Difficulty Algorithm (ADA) begins by finding an initial valid plan under these conditions. It then proceeds to iteratively increase the durations used for the transitions. By doing so, ADA continuously pushes the boundaries of the worst possible states that the plan can handle. This process continues until ADA is no longer able to generate a feasible plan. The last valid plan obtained during this process is returned as the final solution. The underlying assumption is that each iteration's plans increase the robustness since they provide each activity with a larger buffer time, thereby covering a larger number of possible outcomes.

Algorithm 3 Adaptive Difficulty Algorithm

```
1:  function AdaptiveDifficultyAlgorithm(I, f_augmentDifficulty)   ▷ I is the problem instance level
2:      bestSolution ← null
3:      while true do
4:          solution ← solve(I)                                   ▷ Solve instance at current difficulty level
5:          if solution ≠ null then
6:              bestSolution ← solution
7:              I ← f_augmentDifficulty(I)                        ▷ Increase difficulty level
8:          else
9:              break
10:     return bestSolution
```

5.3. Summary

In this chapter, we explained two distinct solution methods for creating robust initial shunting plans, the Probabilistic Action Planner (PAP) and the Adaptive Difficulty Algorithm (ADA).

PAP is designed to account for every possible outcome in the shunting process directly. This approach is based in the theoretical optimal strategy, which seeks to maximize the probability of successfully reaching the goal under probabilistic transitions. It does so by predicting the future state probabilities with the belief state, and determining the optimal future action sequence. However, since calculating these values exactly is computationally infeasible, PAP approximates the optimal future action sequence through individual policy rollouts and uses a particle filter to approximate the belief state.

On the other hand, ADA takes a different approach by not directly addressing every possible outcome. Instead, ADA leverages the transition structure to indirectly account for a majority of potential outcomes. It does this by augmenting the policy with additional buffer time, or slack, after each activity. Actions are then selected based on this augmented policy. The main concept is that introducing more slack leads to more robust plans. However, inserting too much slack may render the problem infeasible, as there is insufficient time remaining for all required tasks. Therefore, ADA uses an iterative approach to progressively increase the amount of slack and generate new plans until it can no longer do so. The final plan is chosen from the last valid iteration, since it contains the maximum amount of slack.

Method	Action Selection	Description
baseline	$a_t = \pi(s_t)$	normal policy rollout. use deterministic transitions
theoretical optimum	$a_t = \texttt{argmax}_a(\texttt{max}_\tau Q^\tau(\mathbf{b}_t, a))$	maximize probability of reaching the goal with optimal future action sequence
PAP	$a_t = \texttt{argmax}_a(Q^\pi(\hat{b}_t, a))$	approximate optimal future action sequence with rollouts from the policy
ADA	$a_t = \texttt{max}_x(\pi'_x(s_t))$	augment the policy by waiting additional time x after each activity. use deterministic transitions

The resulting initial shunting plan is constructed from the sequence of actions, denoted as τ, which can be extracted by iteratively selecting actions starting from the initial state s_0.

However, for both the baseline and ADA methods, actions are determined based on a single state, where deterministic transitions to subsequent states can be used. This enables additional search techniques to find valid sequences, in case the direct rollout of the policy does not produce a valid action sequence. In such cases, the policy is used as a guide for the search algorithm to discover a valid sequence:

6

Experimental Setup

To evaluate the performance of the proposed algorithms, we require a set of problem instances for testing. The problem instances are generate artificial generation based on realistic data, where we use the same values used in prior research to maintain consistency whenever available. The first section outlines the procedure for generating the artificial problem instances.

Afterwards, the details of how the experiments are conducted are explained. This includes a disturbance model responsible for introducing uncertainty, and the settings for each algorithm with their hyperparameters, the base-policy that is used, and the additional search method. We conclude with a series of experiments designed to address our research questions and define the relevant metrics for evaluating the outcomes.

6.1. Artificial Instances

We design a problem generator to create artificial problem instance that are similar to real-world scenarios. It uses a modified layout based on a real shunting yard, and samples each problem component according to realistic probabilities. Whenever available, we use the same values used in prior research to maintain consistency.

Layout The layout used as a reference for the experiments is the "*Kleine Binckhorst*", as shown in Figure 1.1. This shunting yard is often used for evaluating solution methods for the shunting problem. The layout in the experiments is not exactly the same, but it does share some important similarities. It includes two designated tracks for cleaning tasks and one track for washing. The parking tracks range from 52 to 59 and can also be used for maintenance inspection tasks. The remaining tracks are not considered as they are not intended for parking trains. In the experimental setup, each track is modeled as a First-In-First-Out (FIFO) track.

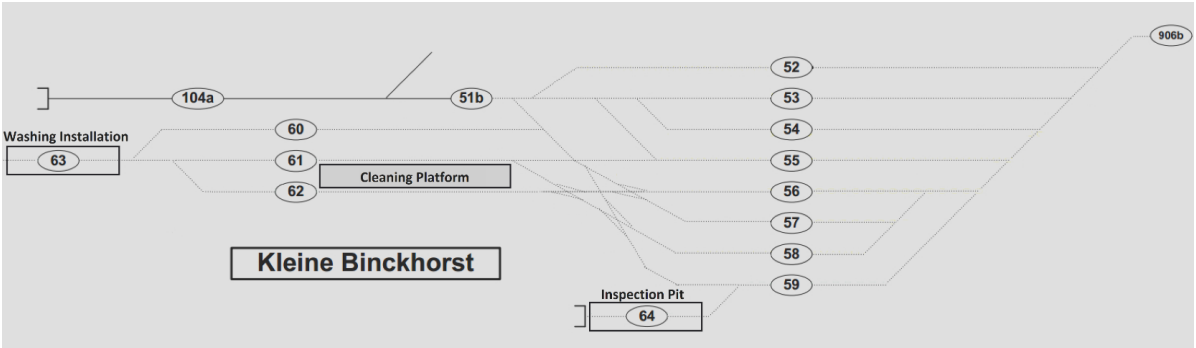

Figure 6.1: Caption

6.1.1. Instance Generator

To simulate realistic scenarios, the problem instances are generated to resemble real-world situations. The timetable of incoming and departing trains, the probability of each train type arriving and their respective set of required service tasks are all sampled based on realistic distributions.

Table 6.1 contains the probabilities that are used to determine the arrival probability with their corresponding service tasks. The table displays the probability of arrival, as well as the probabilities that a specific service tasks is required for the train type. All incoming trains require internal cleaning at one of the service tracks. The majority of trains also require a maintenance inspection. A smaller set of trains also requires an additional service task of external washing.

Sub-type	Arrival	Cleaning	Washing	Maintenance
SLT-4	0.28	1.00	0.16	1.0
SLT-6	0.17	1.00	0.16	1.0
VIRM-4	0.41	1.00	0.16	0.58
VIRM-6	0.10	1.00	0.16	0.58
DDZ-6	0.04	1.00	0.16	0.58

Table 6.1: The train type distribution and the probability that a service task has to be performed on a train type [8].

Table 6.2 provides an overview of the properties of each train, including the length and average service time for each service type. The washing durations are very similar between the different train types. However, the duration for internal cleaning varies a lot, depending on the length of the train. For all experiments, it is assumed that a single service crew is present on the shunting yard to perform maintenance checks throughout the entire planning period, since the service crew can be scheduled after the creation of the initial shunting plan. The walking times of the maintenance crew between the trains is not directly taken into account. The walking times are relatively short and are not expected to have a significant impact on the result, and including these time would increase the complexity of the model by making it dependent on the physical layout of the tracks. It can however be viewed as being part of the uncertainty about the duration.

	(# carriages)	average duration (min)		
Train Subtype	length	Cleaning	Washing	Maintenance
SLT-4	3	15	23	23
SLT-6	4	20	24	27
VIRM-4	4	37	24	10
VIRM-6	6	56	26	14
DDZ-6	6	56	26	18

Table 6.2: Properties of the different train subtypes. The number of carriages and the average duration in minutes for each train subtype [8].

Event Distribution The arrival and departure times in the problem instances are sampled from the probability density functions displayed in Figure 6.2. The planning period always starts at 18:00, and either ends the next day also at 18:00 in case a full day is considered, or at 06:00 in case only the night shift is considered. The departure times are slightly biases towards later times, as they are resampled whenever a departure would need to occur before a corresponding train has arrived.

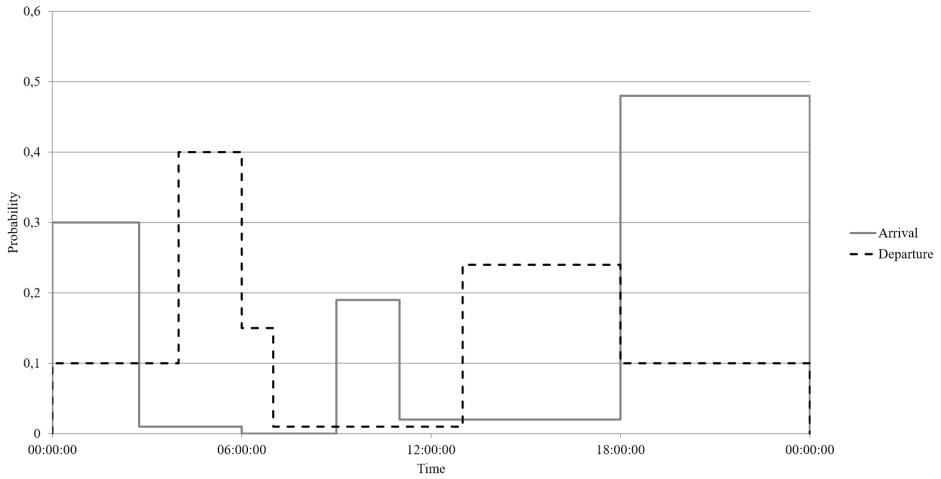

Figure 6.2: The probability density functions of the arrival and departure times [7]

6.2. Experimental Setup

6.2.1. Data Generation

In our experimental setup, we generate artificial problem instances with increasingly more train units in the range from 2 to 40. These values cover a wide range of different problem sizes, so the performance of the algorithms can be compared across problem size, along with the scalability towards larger problems. The instances are generated based on realistic distributions, as described in Section 6.1, and cover a full 24-hour day planning horizon. The main goal of thesis thesis is to investigate the robustness of shunting plans created by different methods. To ensure the generated instances are solvable, each instance is first solved by the deterministic method. If no solution is found in the deterministic setting, the instance is rejected and another one is generated. This guarantees that all instances used in the experiments are solvable, and allow the approaches to create plans that can be evaluated on robustness.

When there are more trains, it becomes more challenging to park all the trains on the yard. To get a sense of the difficulty, we show the maximum number of carriages that have to be parked simultaneously within the shunting yard for each problem size in Figure 6.3. The graph also displays the combined capacity of the parking tracks and the total capacity of the shunting yard that includes the service tracks. Although the service tracks are not intended for parking, they are used temporarily for trains to perform specific service tasks. Some problem instances with 30 train units or more start to exceed the total capacity of the parking tracks. This means that some trains will have to remain on the service tracks for some duration to deal with the limited space. Or course, none of the generated instances exceed the total capacity of the shunting yard, as such instances would be unsolvable and therefore rejected during the generation process.

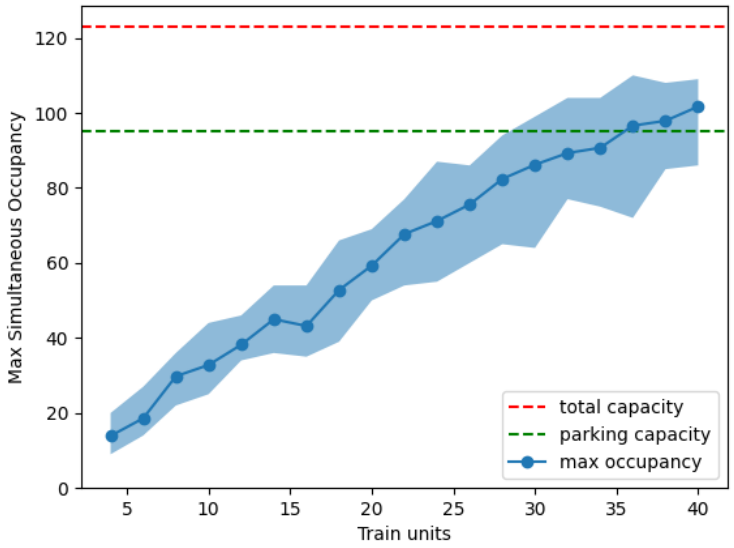

Figure 6.3: The maximum number of train carriages that need to be stored on the shunting yard simultaneously for the generated problem instances. The *parking capacity* is the sum of all lengths of the parking tracks. The *total capacity* also includes the lengths of the service tracks and is a hard limit on the number of carriages that can be stored.

6.2.2. Disturbance Model

In this thesis we focus on creating shunting plans that are robust to smaller, more frequent disturbances. Therefore, we only consider two sources of uncertainty: the arrival times of each train and the variable durations of service tasks. To account for the variability in train arrivals, we sample arrival times from a uniform distribution with a span of 10 minutes around the scheduled times. The duration of service activities is modeled with positive values proportional to the predicted duration, drawing from a log-normal distribution centered around the mean duration with a variance of $\sigma = 0.3$. Both these distributions are chosen to be similar to those used in [6], which is the current state-of-the-art method for creating robust initial shunting plans.

Any single unconditional shunting plan has the inherent risk of becoming infeasible during execution if there are significant changes to the input, caused by some disturbance. Since a shunting plan describes the relative sequence in which trains are intended to move, a change in arrival order would also require the movement order of the related trains to change. However, since our focus is on addressing the smaller, stochastic variations in the environment, it does not provide much additional value to incorporate these scenarios in the experiments. Therefore, to avoid cases where the arrival order changes, we re-sample the arrival times whenever it would change the arrival order for two consecutive trains. This ensures that all trains arrive in their prescribed order.

6.2.3. Hyperparameters

The base-policy provided as input for each approach is the one chosen in Appendix B. This policy is a greedy heuristic rule-based method, which tries to keep trains of the same type on the same tracks. It also tries to predict the matching of the trains based on the arrival order, which is used to determine the order in which the trains are serviced.

The environment used in our simulations is described in Appendix A, which models the shunting problem as a sequential decision problem, using the Markov Decision Process (MDP) framework. Although some simplifications are made to the simulated environment, it still contains the core components of the shunting problem. Therefore, we expect that the environment will be sufficient in providing meaningful and relevant results throughout our experiments.

Whenever a deterministic environment is used in the solution algorithms, either for the baseline or ADA, we have the ability to apply additional search methods. This ability arises from the fact that all transitions within a deterministic environment can be undone and retried without influencing the outcomes. For the experiments in this thesis, the search method chosen is based on result described in Appendix C. The chosen search method is called Monte Carlo Beam Search [2].

Probabilistic Action Planner

We determine the optimal number of particles to use for the Probabilistic Action Planner through experimentation. We test a range of value for the number of particles, and select the lower value that still demonstrates the highest level of robustness across the considered problem instances. This optimal value is used in the final comparisons.

Adaptive Difficulty Algorithm

For this experiment, the slack is increased by 30 seconds after each iteration. A maximum computation time of 30 seconds is allocated for each iteration to solve the problem instance within the deterministic environment. The algorithm is terminated after three consecutive iterations without finding a feasible plan, after which the results of the last valid iteration is returned.

6.3. Experiments

We conclude with a set of experiments that follow from the research questions, and define the relevant metrics.

Robustness The main purpose of the experiments is to assess the extent to which the robustness of initial shunting plans increases when more possible action outcomes are considered during the construction of a plan. The robustness is defined as the percentage of successful plan executions. A plan execution is successful when it manages to depart all trains on their scheduled departure time, having finishing all their required service tasks, without violating any of the constraints or choosing invalid actions. The robustness metric is calculated using the following formula:

$$robustness = 100 \cdot \frac{1}{N} \sum_{n=1}^{N} R_{binary}(s_0, \tau, n)$$

- R_{binary} is the binary reward $\{0, 1\}$ obtained when starting in the initial state s_0 using an initial shunting plan τ in the n-th execution, where a 1 is returned in case the plan successfully reaches the goal state.
- N is the total number of plan executions that are used in the related experiment.

In all experiments where the robustness is measured, each plan is simulated $N = 10000$ times, using randomly drawn samples for each duration and arrival time according to their corresponding distribution.

Required Computation Time Since we want to determine whether it is better to implicitly (ADA) or explicitly (PAP) consider alternative outcomes, we also need to consider the required computation time as an important aspect. The ADA algorithm is expected to be more efficient because it focuses on a single state. To compare the performance between the solution methods, we measure the time it takes in seconds for each approach to create a valid initial shunting plan. The final time is averaged out over 10 different problem instance with the same number of train units.

Although no time limit is used for the available computation time of each approach, ADA is given a maximum computation time of 30 seconds for each individual iteration to solve the augmented problem instance within the deterministic environment. This is because the solver is unable to prove infeasibility for a given problem instance, which would cause it to run indefinitely if not given a limit.

Both the simulation environment and solution algorithms are implemented in Python[1] using the PyPy[2] compiler on an AMD Ryzen 5 3600 processor, using a single core.

[1] `https://www.python.org/`
[2] `https://www.pypy.org/`

Scheduled Service Times To investigate the behaviour of the considered solution methods, we look at the time service-related actions are scheduled. Service-related actions include starting service activities and actions that move trains to and from service platforms. Other action, such as parking incoming trains and departing outgoing trains, are not directly related to any service tasks. Service-related actions provide more useful information to analyze compared to parking and departing actions, because their scheduled times are determined by the decisions of the planner. In contrast, the schedule times of parking and departing actions are determined by the given input timetable that describes the incoming and outgoing trains. This makes the scheduled times of these actions fully outside the control of the planner, and only a part of the problem instance itself. Since we want to analyze differences between the planning approaches, we ignore the non service-related actions from the shunting plan in this experiment.

The starting times of each service-related action are determined by applying the shunting plan in a deterministic environment, and remembering the current time of the system whenever a service-related action is chosen. The deterministic environment operates under expected conditions where all trains arrive on schedule, and where all activities take their average duration.

7

Results

This chapter presents the results of our conducted experiments. The main purpose of the result is to be able to asses the extent to which the robustness of initial shunting plans increases when more possible action outcomes are considered during the construction of a plan. Additionally, we want to determine whether it is better to implicitly (ADA) or explicitly (PAP) consider alternative outcomes.

We first look at the robustness of the shunting plans, and the time required for generating these plans. Afterwards, we investigate the scheduled time of the service activities between the different methods to gain more understanding of their scheduling behaviour.

After showing the main results, we present additional results aimed to gain more understanding of how both proposed method behave, with a separate analysis of both PAP and ADA with experiments that are specific to each method.

7.1. Final Results

7.1.1. Robustness

The goal of the proposed solution approaches is to generate robust initial shunting plans that likely remain feasible during execution, despite uncertainty about the exact durations of the activities and arrival times. The durations are unknown in advance and are sampled from a probability distribution for each execution. The robustness of a shunting plan is estimated by tracking the number of times the goal state is reached, where all trains departed on time, having completed all their required service tasks without violating any constraints. Each shunting plan is simulated 10000 times, using randomly drawn samples for each duration and arrival time according to their corresponding distribution.

The results in Figure 7.1 show a significant difference between the robustness between the deterministic baseline method and both proposed solution approaches. The deterministic baseline method performs poorly in the probabilistic environment. A possible reason for this, which is further investigated in 7.1.3, is that it tends to schedule as many service tasks as early as possible. In contrast, both proposed methods (PAP and ADA) create shunting plans that are significantly better than the deterministic baseline. For smaller problem instances, they even achieve close to 100% robustness. This makes sense, as there are less activities to schedule over the same time window, meaning each train can afford a relatively larger buffer time for each activity. However, as the problem instances increase in size, the robustness of both proposed methods decreases, even getting close to 0% for the largest size. This outcome is reasonable, as these larger instances are often difficult to solve or may be unsolvable, even without considering the uncertainties in durations.

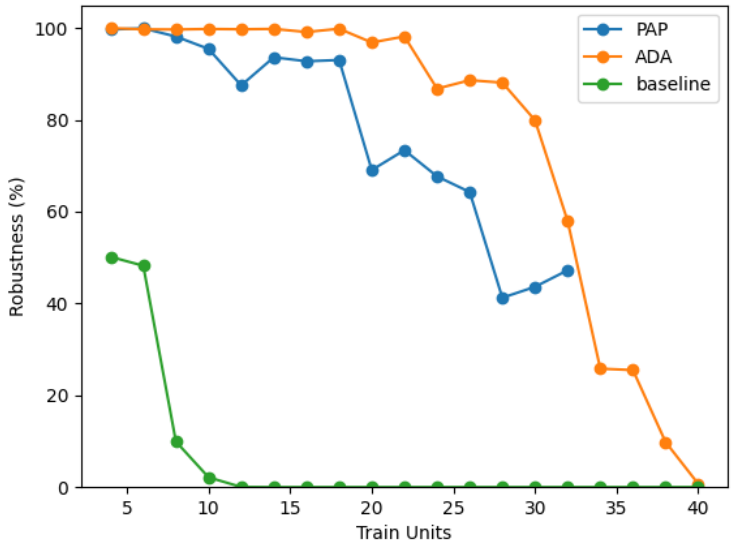

Figure 7.1: The percentage of feasible shunting plan realizations, denoted as the robustness, for increasingly challenging problem instances. The performance of the deterministic baseline method performs poorly in the realistic environment. Both proposed approaches, the Probabilistic Action Planner (PAP) and the Adaptive Difficulty Algorithm (ADA), show significantly higher levels of robustness, even reaching robustness measurements close to 100% for smaller instances. As the problem instances get larger, the robustness levels decrease towards 0% for both methods, although PAP is cut-off early, since it was unable to produce valid solutions within the time limit. ADA consistently shows a higher level of robustness compared to PAP.

In all instances, the Adaptive Difficulty Algorithm (ADA) consistently outperformed the Probabilistic Action Planner (PAP), indicating that indirectly considering most states leads to more robust shunting plans compared to explicitly considering every possible transition outcome. An explanation for this result is that PAP heavily relies on approximation techniques to make each decision. When the approximated values are inaccurate, the decision may be not optimal. In contrast, ADA does not need to make these approximations. The accuracy of the approximations in PAP and its effect on the resulting shunting plans is further investigated in section 7.2.

7.1.2. Required Computation Time

Figure 7.2 shows the average elapsed time for the proposed solution approaches. The deterministic baseline method is often nearly instantaneous, with most instances taking less than a second to compute. The computational time does not significantly increase with larger problem instances. In contrast, the Probabilistic Action Planner (PAP) requires substantially more time, taking over 10 minutes to create shunting plans for instances with 30 or more trains units. The required computation time for PAP steadily increases with the problem size. This is expected, as the required time is primarily determined by the total number of rollouts that are performed during its planning. Since PAP faces more decisions with larger instances while performing the same number of rollouts for each decision, the computational time proportionally increases.

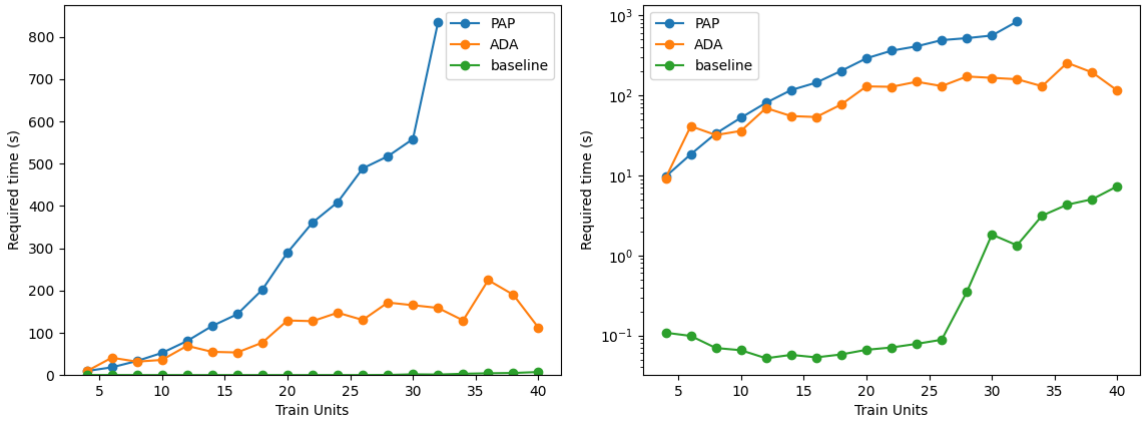

Figure 7.2: The required computation time for each of the considered methods on increasingly larger problem instances. The deterministic baseline method is often nearly instantaneous, with most instances taking less than a second. The Probabilistic Action Planner (PAP) requires the most amount of time, which increases steadily with the problem size. The Adaptive Difficulty Algorithm (ADA) requires less time than PAP, and does not have the same level of increase in required time for larger problem sizes.

Figure 7.2 also shows the computational performance of the Adaptive Difficulty Algorithm (ADA) in comparison to the baseline and PAP. ADA also requires significantly more computation time than the baseline method but is notably faster than PAP. The computational time for ADA heavily relies on the duration it takes to solve a single iteration in a deterministic environment, as the instance needs to be solved multiple times in this environment. Given that an iteration in a deterministic environment is very fast, as shown by the baseline method, the total required time of ADA is also relatively short.

ADA's required time does not increase as significantly as PAP's when dealing with larger problem instances. This is because the deterministic solver used in ADA maintains a consistent speed, regardless of problem size. Furthermore, an important aspect to consider is that ADA performs a lot fewer iterations for larger problem instances, as there are less opportunities to add buffer times. The maximum amount of buffer time that is possible for each problem size and its effect on the resulting shunting plans is further investigated in section 7.3.

A important note is that each iteration in ADA is designed to seek a feasible solution rather than prove infeasibility. ADA continues to search for a feasible plan until the predefined time limit is reached. As a result, this time limit predominantly determines the final computation time for ADA. However, it is worth noting that the plan from the previous iteration is already available before the next iteration is started. In case there is a strict time limit, ADA could be used as an any-time algorithm which simply returns the last valid plan found during the previous iterations.

7.1.3. Scheduled Service Times

To gain some insights into the behaviour of the different approaches, we look at the times when decisions related to service tasks are scheduled. These time are determined by applying the shunting plan in a deterministic environment, operating under expected conditions where all trains arrive on schedule, and all activities take their average duration.

Figure 7.3 presents the cumulative decision index of all the actions related to service tasks over time. Several general observations can be made from the graph. Firstly, most service-related actions are scheduled in the first part of the plan. This is because the majority of trains arrive either at the start or after 15 hours from the start, as described in Figure 6.2. Little to no actions are scheduled, towards the end of the plan. This makes sense, as all service tasks need to be completed before the departures. The horizontal sections in the graph indicate intervals when no new activities are planned. This is desirable from a robustness perspective, as it reduces the risk of the plan failing during this interval. However, it also means that no progress is made towards finishing the required services, which potentially impacts the robustness of future segments.

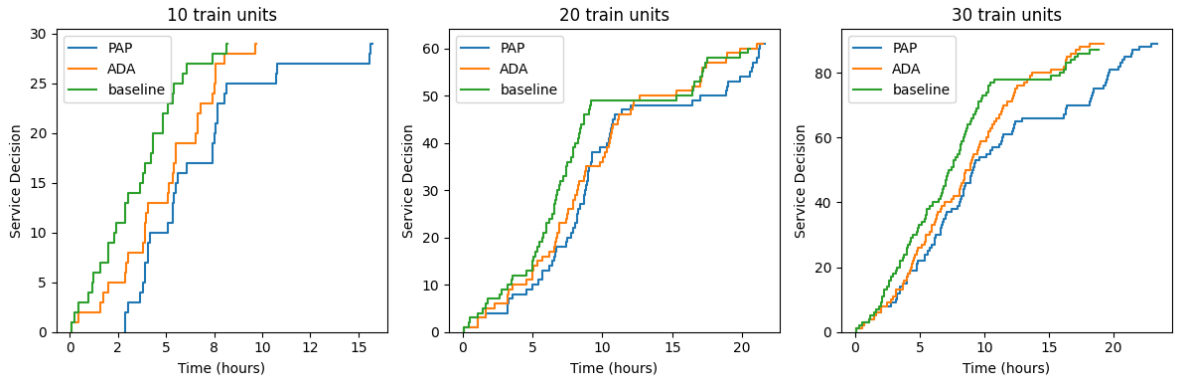

Figure 7.3: The cumulative decision index of all the actions related to service tasks over time. Results are shown off three problem instances with a different number of train units. The deterministic baseline method generally schedules all tasks as soon as possible. The Probabilistic Action Planner (PAP) tries to wait as long as possible before scheduling new tasks to decrease the chances of failing the action. The Adaptive Difficulty Algorithm (ADA) uses the same strategy as the baseline, but is forced to wait additional time before being able to choose new actions.

There are clear differences in scheduling strategies among the solution approaches. The baseline method schedules most tasks as soon as possible, as it cannot assess the risks associated with having no extra buffer time. However, it is able to recognize that no progress is made when no tasks are chosen at a given decision moment. Since it is optimized to find valid plans, it almost always selects one of the service-related actions whenever possible.

On the other hand, the Probabilistic Action Planner (PAP) is a lot slower with scheduling the service related tasks. It causes the actions to be more spread-out over the entire planning horizon. This strategy provides more flexibility to the shunting plan, as the actions can be delayed without causing feasibility issues. The result of this flexibility is that the shunting plans are more robust, as reflected in the results of Figure 7.1.

The Adaptive Difficulty Algorithm (ADA) takes a measured approach that also spreads out the tasks more evenly across the planning horizon. It often lies between the baseline and PAP in terms of schedule times. Since ADA solves the problem instances within a deterministic environment, it also has a similar strategy to the baseline, where it tends to start the services as soon as possible. But because the durations are longer, ADA is often forced to wait additional time before being able to choose new service actions, causing the actions in the plan to be more spread-out.

7.2. PAP Analysis

The performance of the Probabilistic Action Planner (PAP) heavily relies on the number of particles and rollouts that are used. In this experiment, the aim is to analyze the impact of varying this parameter on the algorithm's performance. We will specifically focus on three aspects: the algorithm's ability to find valid plans, the robustness of the generated plans, and the required computation time to create these plans. By analyzing the results, we can determine the optimal number of particles.

We focus on three instances with different problem sizes: a smaller instance with 10 train units, a medium instance with 20 train units, and a larger instance with 30 train units. The algorithm is applied several times with an increasing number of particles. For simplicity, and to only have a single variable to change, we set the number of rollouts at each decision moment equal to the number of particles. To test a wide range of values for the number of particles, we double the number of particles at each measurement, starting with 20, and ending with 1280 ($\{20, 40, 80, 160, 320, 640, 1280\}$).

7.2.1. Validity

A shunting is considered to be valid, if it doesn't violate any of the constraints. For example, trains cannot overtake each other on the same track. Additionally, the plan should be feasible under the expected circumstances where all transitions align with expected arrival times and service durations. The primary objective of the solution is to generate a shunting plan that is valid and remains feasible for a majority of possible realizations. Before evaluating the robustness of the plan, its validity must be ensured, meaning there exists at least some feasible paths that reach the goal state.

While a solution approach in a deterministic environment may use additional search techniques to find valid shunting plans, the PAP algorithm operates within a non-deterministic environment, preventing the use of such techniques. Therefore, there may be instances where the algorithm is unable to produce a valid plan, especially as the problem becomes more complex.

If the PAP algorithm fails to generate a valid shunting plan, it is restarted until a valid plan is found, with a maximum of 10 attempts allowed before termination. Figure 7.4 shows the number of attempts required to find a valid plan. The results show that the algorithm does not always succeed in finding a valid shunting plan on the first attempt, particularly when a small number of particles is used. For larger problem instances, the smallest particle values are unable to produce a single valid plan within the allowed attempts. Although a larger number of particles reduces the number of invalid attempts, there are still instances where multiple iterations are needed before a valid plan is found.

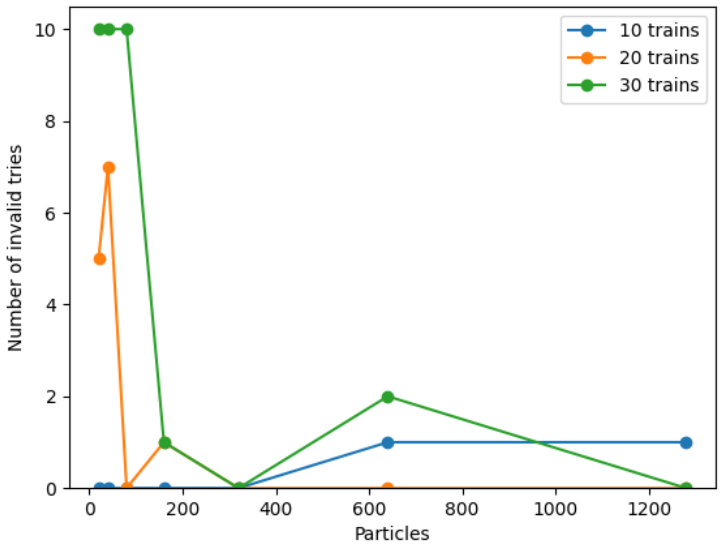

Figure 7.4: The number of invalid solutions that the PAP algorithm returns before finding a valid shunting plan.

7.2.2. Robustness

Figure 7.5 shows the relation between the number of particles and the robustness of the valid shunting plans generated by the PAP algorithm. The graph shows that the robustness of the plans generally improves with a larger number of particles. However, the robustness is not guaranteed to increase with more particles, and there may be diminishing returns beyond a certain threshold. Interestingly, for the problem instance with 20 trains, the shunting plan created using only 20 particles exhibits significantly higher robustness compared to plans generated using a greater number of particles within the range of 40 to 320. This finding emphasizes the potential unreliability of the algorithm's performance. The result may be attributed to the approximations used in the method, which introduces uncertainty about what the best decisions are at a given moment.

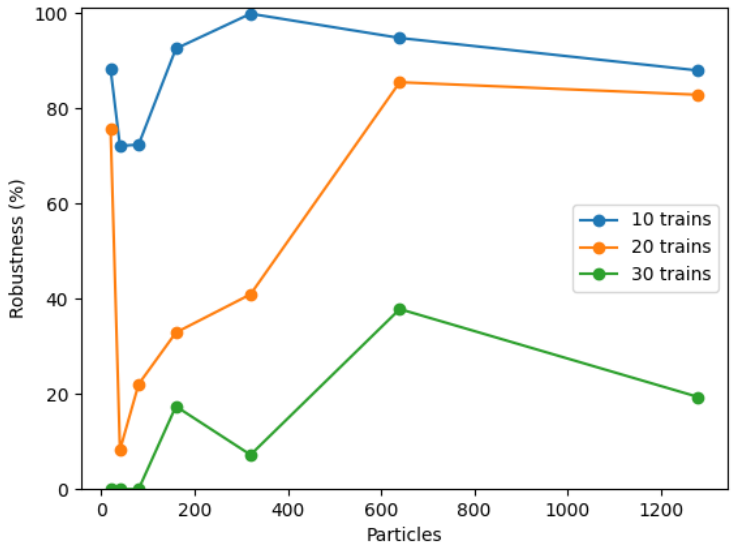

Figure 7.5: The number of feasible executions of the shunting plans produces by the PAP algorithm with an increasing number of particles. As the number particles increases, the robustness also increases.

7.2.3. Value Approximation

The PAP algorithm greatly relies on approximate values for its decision making. These approximations include determining the Q-value of each action and the probabilities of the belief state. During the planning phase, the action with the highest value is selected. Within the context of the considered problem, the value of an action represents the likelihood of reaching the goal state after executing that action. However, since the actual value is unknown, it is estimated through a number of rollouts. Accurately estimating these values can be challenging, especially in larger problem instances that have a larger planning horizon, and thus a larger rollout. The results of the rollouts also depends on the policy's effectiveness and the number of iterations that is used.

Figure 7.6 displays the value associated with each selected action at the moment of decision-making. For each decision moment, we look at the percentage of rollouts that successfully reached the goal state from the current belief state. In the case of larger problem instance, only a few rollouts manage to reach the goal state early in the process. However, as the agent progresses and moves closer to the goal state, the value of the rollouts increases steadily, eventually reaching 100% feasibility near the end.

Conversely, for smaller problem instances, the value of the rollouts begins and ends with 100% feasible outcomes. However, there is a dip in the middle, indicating a decrease in the percentage of successful rollouts. This decline occurs because, initially, the agent has maximum flexibility to adapt to uncertainties during the rollouts. But as more actions are chosen, this flexibility diminishes. The previously chosen actions replace the first part of the rollouts, resulting in a fixed current partial plan that cannot be changed. As a result, the agent must continue from its current belief state, only being able to react to future outcomes.

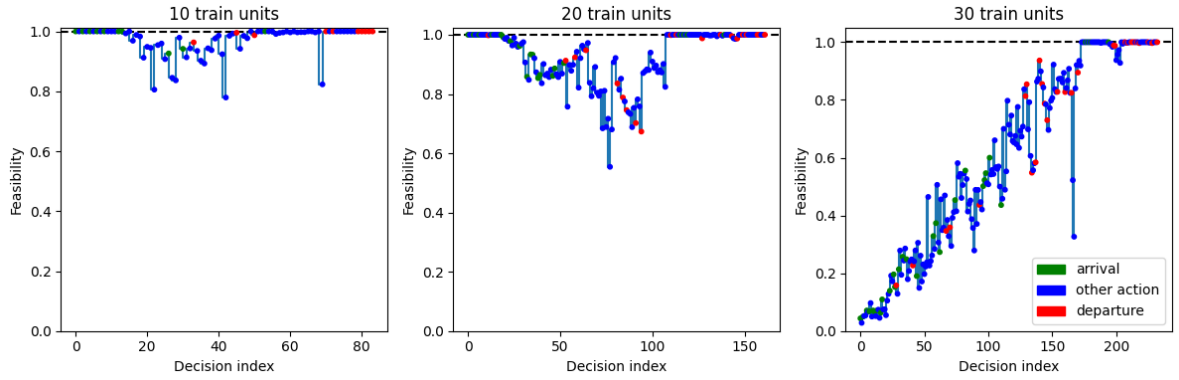

Figure 7.6: The ratio of feasible rollouts at each decision moment for three problem instances of difference sizes. Decisions about arrivals and departures are indicated with green and red circles respectively.

7.3. ADA Analysis

The Adaptive Difficulty Algorithm (ADA) follows an iterative approach to increase the durations of activities in the problem instance. This additional duration can be interpreted as buffer time or slack, which allows for some delay in the completion of a task without impacting other parts of the plan. The algorithm continually increases the slack until it reaches a point where it is unable to find a feasible plan with the current amount of slack. We expect the performance of ADA to be largely determined by the achieved level of slack.

In our evaluation, we initially examine the maximum slack attained for progressively larger problem instances. Afterwards, we assess the assumption about the correlation between slack and plan robustness, investigating whether increased slack leads to more robust shunting plans.

7.3.1. Maximum Possible Slack

Figure 7.7 shows the maximum slack reached by ADA for each problem size. As expected, the slack value decreases as the problem instances become larger. This observation makes sense since the total available time is the same for all problems. So when there are fewer tasks to complete, the available time can be distributed more freely, allowing for a larger amount of slack. However, the number of tasks increases in larger instances, resulting in less available time for each activity, and a decrease in the maximum achievable slack.

The maximum slack that ADA is able to achieve contains a lot of variation across the problem instances of the same size. Since the search algorithm used in each iteration is unable to prove the infeasibility of a given instance, we cannot infer from these results whether the obtained maximum slack reached the theoretical limit or is just a result of the search algorithm's inability to find a feasible plan within the designated time.

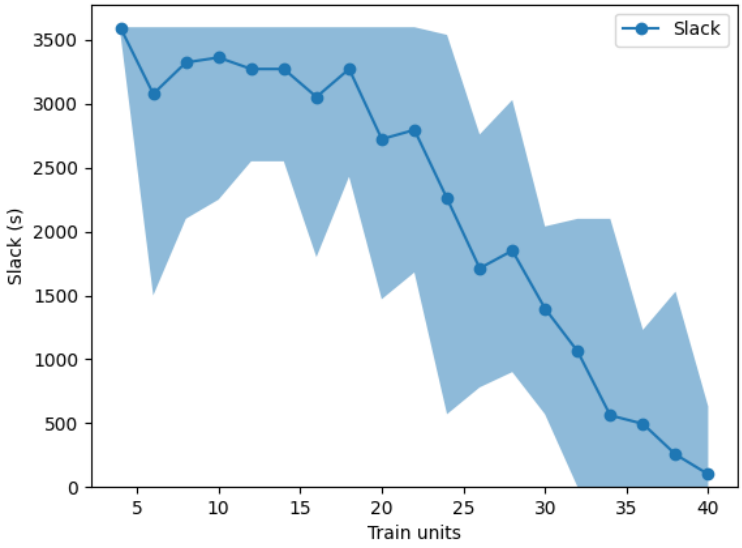

Figure 7.7: The maximum achievable slack found by the Adaptive Difficulty Algorithm (ADA) for a range of problem sizes with increasingly more train units. The area is bounded by the minimum and maximum slack from the 10 results per problem size. As there are more train units, the slack tends to decrease.

7.3.2. Correlation with Robustness

The Adaptive Difficulty Algorithm uses and iterative approach to generate the shunting plans, progressively solving more challenging versions of the problem instance. Each iteration all arrivals are delayed a bit more, and the durations for the service activities take more time. The algorithm continues until it can no longer find a feasible plan for these conditions, after which the last valid plan obtained during the process is returned as the final solution. It is based on the assumption that the robustness of the plans increases with more iterations, since there is more slack for each activity. To verify this assumption, we examine the robustness of plans created at each iteration.

The results, displayed in Figure 7.8, show a clear upward trend. This indicates that the robustness generally increases with more iterations, although a strict improvement in not guaranteed for consecutive iterations. The larger problem instance requires more slack to achieve the same level of improvement as the smaller instances. This can be explained by the fact that there are more trains in the larger problem instance, resulting in more possible points of failure due to delays. So when each individual decision is optimized using the same amount of slack as the smaller instance, there are more decisions in total, which causes the overall robustness to be lower comparatively. It is also worth noting that there is a limit on the maximal robustness that can be achieved, resulting in diminishing returns after a certain amount.

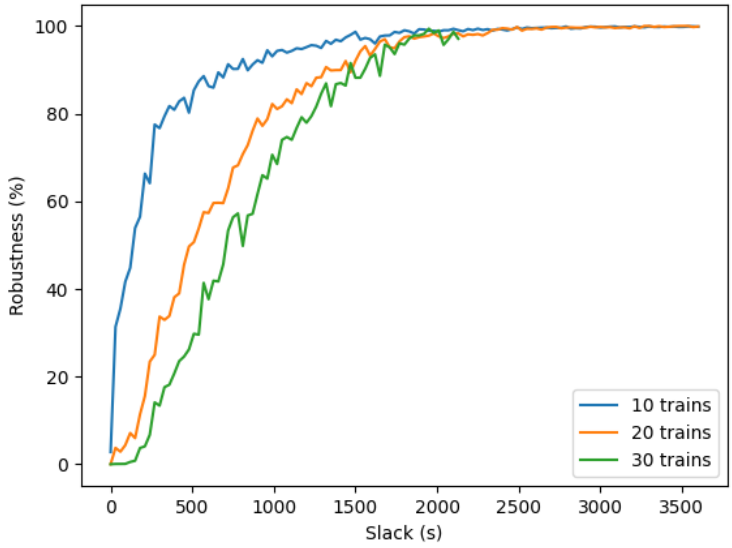

Figure 7.8: The graph shows the effect the amount of slack has on the robustness of the created solutions in three scenarios with a different number of train units. The Adaptive Difficulty Algorithm (ADA) increases this slack after every iteration under the assumption that the new plans get robustness. This assumption aligns with the results, that indicate a general increase in robustness with more slack.

8

Discussion

This chapter provides an answer to the research questions. Afterwards, we discuss some notable findings from the experiments, and address the limitations of this thesis

8.1. Answer to Research Question

Sub-Question 1 An initial shunting plan is typically represented as a Partial Order Schedule, whereas the output to a sequential decision formulation is a sequence of actions. This action sequence can be viewed as a Total Order Schedule, where each action follows from the previous one. This difference in schedule representation caused us to ask the first sub-question:

1. **How can action sequences be used as initial shunting plans by mapping them to Partial Order Schedules?**

To answer this question, we described a procedure for converting an action sequence into a Partial Order Schedule (POS). This procedure goes through all the actions in the sequence, and adds all non-waiting actions to the activity set A. Waiting actions are ignored, because their usage is implied by the precedence constraints within the POS. Precedence relations are added by keeping track of the previous action for each resource and for each train, and adding a precedence relation towards the next action that uses the same resource or train.

Sub-Question 2 This thesis is motivated by the inability of an optimized policy to produce robust initial shunting plans from a regular rollout. This limitation is hypothesized to be caused by the fact that a rollout only considers the expected outcome during the construction of the plan. Therefore the second sub-question is:

2. **How can alternative outcomes be considered during the planning phase, either directly or indirectly?**

In this thesis, we proposed two distinct methods that consider multiple outcomes during planning. First, we proposed the Probabilistic Action Planner (PAP), which aims to account for every possible outcome directly. Since we cannot make any observations during the entire planning process, the original MDP problem can be viewed as a Non-Observable MDP. By viewing the problem as a NOMDP, we can use existing techniques to solve this version of the problem effectively. We use the notion of a belief state to represent the uncertainty during planning, and approximate its probabilities values using a particle filter. To solve the NOMDP, we use Monte-Carlo planning.

Secondly, we proposed the Adaptive Difficulty Algorithm (ADA), which strategically focuses on a single outcome, and accounts for alternative outcomes indirectly. The solution idea is based on the fact that the transitions are deterministic, except for variability in the durations, making it possible for less delayed state to transition to a more delayed state simply by waiting. To determine the maximum amount of delay we can afford to consider, we iteratively increases the delay until the problem becomes unsolvable. The latest valid plan is returned as the final solution, as it considers the most delayed states.

Sub-Question 3 Between both proposed solution approach, we want to know which one is better. Given that ADA, focuses solely on a single state at each decision moment, we expect it to be faster. However, since it also overlooks a subset of alternative states, it could result in less robust plans. Therefore, the last sub-question is:

3. **Is it better to implicitly (ADA) or explicitly (PAP) consider every possible outcome, in terms of the robustness of the plans and the computational efficiency?**

From the experimental results we confirm that ADA indeed requires less time to compute, and has better scalability for larger problem sizes. However, the results also show that ADA consistently outperforms PAP in terms of robustness across all problem sizes. Further analysis revealed a potential limitation of PAP, as it occasionally struggles to produce valid plans for larger problem instances. In contrast, ADA consistently succeeds in finding valid plans, thanks to the use of additional search techniques. Based on these results, we conclude ADA is the better approach, as it is more reliably able to find valid plans, which are also more robust, and found is less time.

Main Research Question We proposed two method for creating robust initial shunting plans that take all possible action outcomes into consideration, either directly (PAP) or indirectly (ADA). We compared their performance to the baseline method, which performs a regular policy-rollout under normal conditions. This allows us to answer the main research question:

> **Given a policy optimized for finding feasible solutions, to what extend is the robustness of the initial shunting plan increased when all possible action outcomes are considered during the construction of the plan?**

The experimental results on realistically generated problem instances showed that both the PAP and ADA approaches significantly outperformed the baseline. The baseline only remains feasible for only approximately 50% of realizations with six trains or fewer and almost always becomes infeasible during execution with ten trains or more. In contrast, both PAP and ADA consistently maintained a feasibility rate of above 90% for problem sizes up to 20 trains, only dropping in performance for larger instances. ADA is even able to maintain a feasibility rate of over 95% for up to 22 trains.

8.2. Findings

The difference in robustness between the baseline solution and the examined methods can be attributed to a fundamental conflict in strategy between the scheduling strategies. In a deterministic environment, the strategy of *"starting each activity as soon as possible"* seems to be most effective, as there is no risk associated with executing the action. Moreover, starting an activities generally does progress the solution towards the goal state, which causes the planner to prefer choosing the activities over waiting. However, this strategy is not robust in the probabilistic setting, as it does not account for uncertainties in activity durations, as there is very little buffer time between the activities. In contrast, the strategy of *"distributing the available time between activities by waiting"* is more successful in the probabilistic setting. This approach provides each activity with buffer time, allowing delays without disrupting the remainder of the plan.

Interestingly, despite starting almost always service tasks earlier than the Probabilistic Action Planner (PAP), ADA still achieves better robustness. Since the Adaptive Difficulty Algorithm (ADA) creates solution in a deterministic environment, is uses the strategy of *"starting each activity as soon as possible"*, which explains why it tends to start the tasks relatively early. However, because ADA is forced to add additional buffer time to each activity, the resulting shunting plans remain robust.

Because of the scheduling strategy of ADA, there often seems to be large portions of unused time towards the later parts of the day, where little to no activities are scheduled. This observation suggests that there may still be potential to improve the robustness of the plans further. One possibility is to strategically distribute the unused time towards the end of the plan as extra slack to the activities, thereby increasing the robustness of the overall plan. Alternatively, another option could be to deliberately leave this free time in the later parts unused. This could offer an advantage for repair strategies, as the additional free time in later parts of the day can be used to reschedule activities in the event of failures or unexpected disruptions.

The performance of the Probabilistic Action Planner (PAP) algorithm depends on the accuracy of the approximations it uses. PAP tracks the probabilities of being in a particular state using a particle filter, which approximates the belief state. Additionally, the value of each action is determined through a number of policy rollouts, where the number of rollouts equals the number of particles in the particle filter. The greater the number of particles that are used, and more accurate the approximations for the belief state and the action-values become. However, the results indicate that this estimate can be highly unreliable (Figure 7.6), leading to suboptimal decisions.

For challenging problems, the value of the rollouts often starts near 0%, which is lower than the actual value, as evidenced by the resulting robustness of these plans, which is above 80%. This inaccuracy is caused by the given policy that is used, which is not optimal, making the planner overly pessimistic about its outcome. This could cause it to prefer starting actions early to progress towards to goal, at the cost of potentially risking early failures for some states that are unable to start these actions.

On the other hand, for easier problems the value starts near 100%. This is an overestimate of what the actual value should be, because the rollouts still have maximum flexibility to react to encountered disturbances when performing them. However, once the planner commits to a single sequence, it loses this flexibility. As a result, the planner is too optimistic about its decisions, causing it to prefer robust actions that are unlikely to fail, at the cost of not actually making any progress towards the goal state. If the progress is delayed to often, it may be forced to eventually resort to suboptimal actions that are not robust. This may explain the slight decrease in performance when more particles are used in PAP, as can be observed in Figures 7.4 and 7.5. When more particles are used, the probability of sampling more delayed states is also increased. This causes most reasonable actions to receive scores that are slightly less than 100%, whereas the other actions that are safer but less productive maintain a perfect score.

8.3. Limitations

Probabilistic Action Planner The Probabilistic Action Planner (PAP) is compared to the theoretical optimal solution. The theoretical optimal strategy chooses actions that maximize the probability of reaching the goal by continuing with a single sequence of actions regardless of the encountered states. Since we don't know the optimal continuation of the future action sequence, PAP approximates this single sequence with multiple individual rollouts. The idea is that these rollouts resemble the optimal continuation, given that the policy is optimized to find feasible solutions. However, the difference of using policy rollouts compared to using a single action sequence, is that the rollout is able to react to the states it encounters. This allows each individual rollout to be different from each other. As we can observe from the results in Figure 7.6, the usage of multiple reactive rollouts instead of a static sequence leads to issues in accurately estimating the value of each action.

It should be mentioned that this thesis only used a single policy for all the tests. This choice of policy might influence the results. For example, a policy that is very consistent in its choices and that tends to make the same decisions regardless of differences in the exact state, might show different results. This is because the individual rollouts more closely resemble the same sequence of action. Whether this kind of policy would lead to better results remains to be investigated.

Adaptive Difficulty Algorithm Although the Adaptive Difficulty Algorithm (ADA) generally outperforms the alternatives, there can be cases where it does not perform well. Since ADA cannot know during the planning phase which specific delays are going to occur during execution, it aims to optimize against all possible scenarios. To accomplish this, it optimizes every decision equally. However, this equal optimization strategy may cause problem when there is a difference in the extend of which certain decision can be optimized. For instance, consider a scenario where one of the train needs to perform its service tasks immediate upon arrival after which it immediately needs to depart again and cannot afford any additional buffer time for any of these actions. The algorithm will terminate after the first iteration, as it is unable to add any buffer time to this train. The resulting plan will not be robust, as the other trains in the problem also do not include any buffer time. In such cases, the algorithm should ideally prioritize increasing buffer times for the remaining trains to increase the overall plan's robustness, even if there is a single train that runs the risk of being delayed.

Simulation Environment There are several assumption and simplification made in the simulation environment that cause the model to slightly deviate from the real-world.

The problem formulation relied on static routing times for the duration of each movement. This simplification was made to prevent cases where where movements are ongoing when a train needs to depart, causing the departing to be delayed. However, in reality the actual duration of each movement is uncertain and contains some variance. This variability was not taken into account in the conducted experiments. Although the movement durations is relatively short compared to the duration of the service tasks, the inclusion of probabilistic movement times would increase the accuracy of evaluating and improving the robustness of the shunting plans.

Another limitation of the model is its restriction to allow only a single train to move at a time. In practice, multiple trains are allowed to move simultaneously as long as their paths do not interfere with each other. Allowing simultaneous moves has the potential to increase the feasibility and robustness of the generated plans. Particularly, when movement to and from the service tracks can occur during arrival and departure events. This would mean that even if a train is expected to arrive or depart soon, the services within the yard can still be chosen, reducing the dependency on exact arrival times, thus increasing the flexibility.

This thesis focused only on scenarios with trains composed of a single train unit. However, in reality, trains often consist of multiple units, which can be of different types. Including the reconfiguration aspect increases the complexity of the problem, and makes it more difficult to find feasible plans. Splitting and combining trains takes a significant amount of time to perform, and also includes some variability in the duration. Incorporating the reconfiguration aspects would provide a more realistic representation of the problem.

9

Conclusions

This thesis aims to address the challenges of generating robust initial shunting plans in an uncertain environment. We focus on a sequential problem formulation, modeled as a Markov Decision Process (MDP), and using a policy framework optimized for this environment. The goal of this thesis is develop a method that is capability of creating robust initial shunting plans that are likely to remain feasible for a large number of possible plan executions.

An initial shunting plan can be extracted from a policy through a rollout in a simulated environment under normal conditions. However, this conventional rollout technique leads to action sequences that fail to account for most the alternative outcomes for actions that have variability in their outcome, causing the overall plan to not be feasible for a large number of possible realizations.

To address this limitation, we introduced two distinct solution methods, one which explicitly considers all possible outcomes, whereas the other aims to cover most outcomes indirectly. We asked to what extend the influence of multiple possible outcomes results in increased robustness of the generated plans, and whether it is better to implicitly or explicitly consider every possible outcome.

The first method we proposed is the Probabilistic Action Planner (PAP), which predicts the probabilities of all the possible future states the agent can encounter during execution, and makes its decisions based on the weighted average of each action-value for each state.

The second method we proposed is the Adaptive Difficulty Algorithm (ADA). Rather than explicitly considering every possible reachable state, ADA capitalizes on the structure of the transitions. It is always possible to transition to a more delayed version of the state, simply by not starting a new activity and waiting instead. By focusing on a single delayed transition at each decision point, ADA indirectly accounts for all states that are less delayed. Since only a single transition is used for each action, the environment effectively becomes fully deterministic, which enables ADA to utilize additional search techniques to find feasible shunting plans.

Experiments on realistically generated problem instances showed that creating initial shunting plans based on all possible states significantly increases the robustness towards probabilistic variations in plan executions, as both proposed method significantly outperform the deterministic baseline method. Furthermore, ADA performed better than PAP in every aspect, as it is more reliably able to find valid plans, which are consistently more robust, and found in less time.

We noticed a difference the scheduling strategy between the solutions created in a deterministic environment compared to PAP, which uses a non-deterministic environment. Both the baseline policy-rollout and ADA follow the *"starting each activity as soon as possible"* strategy, whereas PAP distributes the available time more between the activities. The main difference between the baseline and ADA is that ADA inserts additional buffer times to its activities, causing it to be slower in scheduling the activities.

Although ADA showed promising results, there may still be cases where it cannot produce robust solutions. Additionally, certain assumptions and simplifications were made to facilitate the experiments, which may impact some of the result. Nevertheless, this thesis showed how robust initial shunting plans can be created using a policy optimized for feasibility in a deterministic environment.

9.1. Future Work

An important area of future work is to address the current limitations of the model, as highlighted in Chapter 8. Most notably, the inclusion of combining and splitting, and incorporating the variability in movement durations. By addressing these aspects, a more realistic evaluation of the proposed solution methods can be done.

An interesting area of future research could be to expand on the concept of augmenting problem instances to generate more robust plans. This thesis only focuses on a single augmentation technique, which is to increasing the considered duration of each activity. However, there may be other potential techniques that result in more robust solutions. One approach could be to impose additional restrictions on the allowed plans. For instance, by disallowing two consecutive trains to park on the same track, and including a simple repair mechanism in the final plan that switches the order of parking actions when necessary ensures that both trains are always able to park on their designated tracks, regardless of their arrival sequence.

By applying such restrictions, it may allow additional repair mechanisms to resolve a number of disturbances in a predictable and consistent way. This approach could improve the overall robustness to beyond what is achievable by only considering the slack. The reason for this, is because the additional slack is only used to delay the start time of the next activities. This so called *right-hand-shift* approach is incapable to handle certain types of disturbances, such as the earlier example where the order of operations needs to be modified to deal with different order of train arrivals.

An aspect that was not considered in this work is the ability to repair a plan after it fails. Repairing a plan introduces an additional dimension of preference for the repaired plan to be similar to the original plan.

Related to this, existing research has focused on improving the robustness of plans and minimizing the need for repairs. However, there are instances where it is impossible for a plan to remain feasible, making repairs unavoidable. Moreover, it may be difficult or even impossible to repair the plan anymore, as part of the plan has already been executed. Therefore, it would be worthwhile to explore the possibility of optimizing the resilience of an initial plan, in addition to the robustness. In other words, how easily can an initial plan can be repaired once it becomes infeasible. A hypothesis is that providing more slack to activities later in the plan increases the ability to repair the plan following a disturbance. An interesting note is that the plans created by the Adaptive Difficulty Algorithm generally results in having more free time towards to end, which could work well for rescheduling purposes.

References

[1] J.M. Akker, van den et al. “Shunting passenger trains : getting ready for departure”. English. In: *Proceedings of the 63rd European Study Group Mathematics with Industry (SWI 2008, Enschede, The Netherlands, January 28-February 1, 2008)*. Ed. by O. Bokhove et al. CWI Syllabus. Centrum voor Wiskunde en Informatica, 2008, pp. 1–19. isbn: 978-90-365-2779-8.

[2] Hendrik Baier and Mark H. M. Winands. “Beam Monte-Carlo Tree Search”. In: *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. 2012, pp. 227–233. doi: 10.1109/CIG.2012.6374160.

[3] Hendrik Baier and Mark HM Winands. “Nested Monte-Carlo Tree Search for Online Planning in Large MDPs.” In: *ECAI*. Vol. 242. 2012, pp. 109–114.

[4] Shiwei Bao. “A Robust Solution to Train Shunting Using Decision Trees”. MA thesis. 2018.

[5] Erik Beerthuizen. “Optimizing train parking and shunting at NS service yards”. MA thesis. 2018.

[6] R.W. van den Broek. “Towards a Robust Planning of Train Shunting and Servicing”. In: (2022). doi: https://doi.org/10.33540/1301.

[7] Roel van den Broek. “Train Shunting and Service Scheduling: an integrated local search approach”. MA thesis. 2016.

[8] Roel van den Broek, Han Hoogeveen, and Marjan van den Akker. “How to Measure the Robustness of Shunting Plans”. In: OpenAccess Series in Informatics (OASIcs) 65 (2018). Ed. by Ralf Borndörfer and Sabine Storandt, 3:1–3:13. issn: 2190-6807. doi: 10.4230/OASIcs.ATMOS.2018.3. url: http://drops.dagstuhl.de/opus/volltexte/2018/9708.

[9] Roel van den Broek, Han Hoogeveen, and Marjan van den Akker. “How to measure the robustness of shunting plans”. In: *18th workshop on algorithmic approaches for transportation modelling, optimization, and systems (atmos 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[10] Roel van den Broek et al. “A local search algorithm for train unit shunting with service scheduling”. In: *Transportation Science* 56.1 (2022), pp. 141–161.

[11] Roel W van den Broek. “Train Shunting and Service Scheduling: an integrated local search approach”. MA thesis. 2016.

[12] Tristan Cazenave and Nicolas Jouandeau. “Parallel nested monte-carlo search”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–6.

[13] Guillaume M JB Chaslot et al. “Progressive strategies for Monte-Carlo tree search”. In: *New Mathematics and Natural Computation* 4.03 (2008), pp. 343–357.

[14] Richard Freling et al. “Shunting of Passenger Train Units in a Railway Station”. In: *ERIM Report Series Reference* ERS-2002-74-LIS (2005). doi: https://ssrn.com/abstract=371017.

[15] Jørgen Haahr, Richard Lusby, and Joris Wagenaar. *A comparison of optimization methods for solving the depot matching and parking problem*. Tech. rep. 2015.

[16] Per Munk Jacobsen and David Pisinger. “Train shunting at a workshop area”. In: *Flexible services and manufacturing journal* 23 (2011), pp. 156–180.

[17] Esmee Kleine. “Generating robust solutions for the Train Unit Shunting Problem under uncertainty: a local search based approach”. MA thesis. 2019.

[18] LG Kroon, RM Lentink, and A Schrijver. “Shunting of Passenger Train Units: An Integrated Approach”. English. In: *Transportation Science* 42.4 (2008), pp. 436–449. issn: 0041-1655. doi: 10.1287/trsc.1080.0243.

[19] Wan-Jui Lee, Helia Jamshidi, and Diederik M Roijers. “Deep reinforcement learning for solving train unit shunting problem with interval timing”. In: *Dependable Computing-EDCC 2020 Workshops: AI4RAILS, DREAMS, DSOGRI, SERENE 2020, Munich, Germany, September 7, 2020, Proceedings 16*. Springer. 2020, pp. 99–110.

[20] Ramon M. Lentink et al. “Applying Operations Research Techniques to Planning of Train Shunting”. In: *Planning in Intelligent Systems*. John Wiley & Sons, Ltd, 2006. Chap. 15, pp. 415–436. isbn: 9780471781264. doi: https://doi.org/10.1002/0471781266.ch15.

[21] Ludo van den Nieuwelaar. “Automatic algorithm configuration with search heuristics for the Train Unit Shunting Problem”. MA thesis. 2021.

[22] Paulo Roberto de Oliveira da Costa et al. “Data-driven policy on feasibility determination for the train shunting problem”. In: *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2019, Würzburg, Germany, September 16–20, 2019, Proceedings, Part III*. Springer. 2020, pp. 719–734.

[23] Evertjan Peer et al. “Shunting trains with deep reinforcement learning”. In: *2018 ieee international conference on systems, man, and cybernetics (smc)*. IEEE. 2018, pp. 3063–3068.

[24] Donker Sanne. “Completing a partial solution of the Train Unit Shunting Problem: investigating the influence of resource and temporal constraints”. MA thesis. 2022.

[25] Maarten P. D. Schadd et al. “Single-Player Monte-Carlo Tree Search”. In: *Computers and Games*. Ed. by H. Jaap van den Herik et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–12. isbn: 978-3-540-87608-3.

[26] Fabian Stelmach. “Proactive-Reactive Approach for Stable Rescheduling of the Train Unit Shunting Problem”. MA thesis. 2020.

[27] Shijian Zhong. “Solving Train Maintenance Scheduling Problem with Neural Networks and Tree Search”. MA thesis. 2018.

A

Simulation Environment

In this chapter, the shunting problem is formulated as a Markov Decision Problem (MDP). The essential components that form an MDP, including the state space, action space, transition function, and reward function, are explained in detail. This sequential formulation of the problem is used to run simulations of problem instances, which the proposed solution methods use to create shunting plans, and in which experiments can be performed to test the quality of the created shunting plans.

First, a set of assumptions specific to the shunting problem is defined. Afterwards, the essential components that form an MDP, including the state space, action space, transition function, and reward function, are explained in detail. Finally, the behaviour of the environment is defined.

A.1. Prerequisites

Simplifications and Assumptions In the real-world, there are a lot of details that influence certain aspect of the problem. However, modeling each of these aspects in detail would not only be challenging but could also risk the completion of this project. To effectively address the scope of the project and develop an appropriate solution strategy, it is necessary to introduce certain assumptions and simplifications.

The first goal behind the simplifications and assumptions is to make it easier to implement the simulation environment within the scope of this thesis. The second goal of these simplifications is to enable the design of a policy that can be used for the purposes of this thesis. Existing policies that have been proposed in previous research struggle to effectively address the complexities of the shunting problem, even in a deterministic environment. The focus of this thesis is on the uncertainty aspects in the problem, rather than designing an improved policy that can solve the most complex formulation of the problem.

It's worth noting that each of these simplifications and assumptions have been made before in previous work, although not every work has used the same set of ones.

- The first simplification made is that trains only consist of a single train unit. This means that the trains under consideration do not require any splitting or combining of multiple units between trains.

  By excluding these operations, we make it significantly easier to design a policy that works well. Determining which trains to split and combine can be especially difficult in this sequential problem formulation, because the decisions regarding splitting and combining have far-reaching consequences that are difficult to predict at the time of decision-making. For example, combining two train units excludes the possibility of departing the units individually without first splitting them again. Moreover, split and combine actions have the potential to considerably extend the planning horizon when chosen repeatedly in succession.

  It's worth noting that it is technically possible to solve real-world instances by assuming incoming trains are split into individual units, and combined right before their departure.

- Different trains cannot move in parallel though the shunting yard. While real-world operations might allow multiple trains to move simultaneously as long as their routes do not intersect, we restrict the movement of trains to ensure no collisions can occur.

  This restriction aligns with the assumption made by most existing solutions, which also do not account for simultaneous train movements. In the past this restriction was primarily implemented due to safety considerations, meaning that the realistic problem instances should remain solvable under this constraint.
- The time taken to move from one track to another takes a fixed amount of time. In reality, the movement time depends on several factors such as the chosen route, the number of tracks and switches traversed and the number of times the train needs to reverse its direction. The main reason for this simplification is to align with the previous assumption of non-parallel movements. By assuming a fixed movement time, it can be ensured that no other train is still in motion when another train needs to start moving to the departing gateway, thereby avoiding simultaneous movements.
- We assume there is always exactly one service crew available at each service track. This implies that a dedicated crew is assigned to each service track to perform necessary servicing tasks for the trains parked on that track. In real-world scenarios, service crew availability may vary over time or be subject to other constraints and limitations, such as having moments of break. The possibility of having multiple groups of service crew at the same service location does not impact our model, as only one train can be serviced at each service location simultaneously. Furthermore, crew members can be dynamically assigned as needed based on the shunting plan.
- Trains of the same type are interchangeable, meaning that if a train of a certain type is required to depart, any train with the correct type configuration can be used as the departing train. This is generally the case, although in practice there are sometimes exceptions. For instance, when a specific train may need to travel to particular shunting yards for specialized maintenance. Most existing policies are designed with interchangeable trains in mind. Therefore, we exclude any pre-matchings in the problem input.
- The location of a parked train on a track remains fixed throughout the shunting process. Once a train is parked on a track, it does not need to relocate within that track. The position of a train is solely determined by its relative position to other trains already parked on the track. This simplification greatly simplifies the problem, without really changing the problem significantly. The action space is reduced, as there are no actions required for repositioning, and the planning horizon is reduced, as these repositioning actions do not have to be chosen. This simplification is not expected to significantly impact the results, because the repositioning can often be done while other actions are ongoing elsewhere in the yard. It should be noted that no previous literature on the shunting problem has taking this aspect into consideration.

Layout

The shunting yard consists of a collection of interconnected tracks where the trains can be parked and serviced on. For the purpose of this research, only layouts are considered where the tracks are fully connected. In other words, it is ensured that every track within the yard is reachable from every other track. Additionally, this thesis focuses on two types of tracks: Last In First Out (LIFO) and First In First Out (FIFO). These track configurations determine the order in which trains enter and exit the tracks. A LIFO track stores the most recent trains at the front, blocking the exit of all existing trains currently parked on the track, where it is the only possible train on the track to exit the track again. In contrast, FIFO tracks park the most recent train after the existing trains on the track, making it the last train to exit the track.

The layout of the shunting yard is assumed to remain static during execution, as scheduled changes to the yard are known about in advance, and large disturbances that influence the availability of the yard are unlikely to occur. This means that the initial configuration of tracks, switches, and other elements within the yard remains unchanged throughout the solution.

In the input scenario, the following properties are described:

- The number of tracks in the yard
- The maximum length of each track

- The type of each track (LIFO or FIFO)
- Which service tasks can be performed on which tracks
- Which train subtypes are allowed to park on which tracks

A.2. MDP formulation

In this section, a formulation is given of the shunting problem using the framework of a Markov Decision Process (MDP). The MDP formulation provides a formal framework for modeling and analyzing sequential decision-making problems. An MDP is formally defined as a tuple $M = (S, A, T, R)$, where S represents the set of all possible states, A denotes the set of all possible actions, $T(s'|s, a)$ is the probability of transitioning from state s to state s' after taking action a, and $R(s, a)$ signifies the reward received after applying action a in state s.

Each of these components is described in detail in the upcoming sections. The state space is discussed in Section A.2.1, followed by an explanation of the action space in Section A.2.2. The transition function will be detailed in Section A.2.3, and lastly, the reward function is described in Section A.2.4.

A.2.1. State Space

The state space is the set of all possible states that the system can be in. A state encompasses all the relevant information about the system at a given time. To satisfy the Markov property, the state should not be dependent on the history of previous states and actions.

In the shunting problem, a state should encapsulate the current position and status of all the trains within the yard. It provides a snapshot of the shunting yard's configuration, including the locations of trains on different tracks and all associated service requirements.

The following list are all the properties that together form a state:

- The current time
- A timetable of the expected events
  - The expected arrival times of future trains
  - The departure times of the sequence of required train subtypes
- The location of all train units on the shunting yard
  - The track it is currently on
  - The position on the track relative to the other trains on the track
- The service tasks that need to be performed on each train unit
- The start time of all ongoing actions

The timetable of events is derived from the specific problem instance and is provided at the beginning of the planning time. Since the exact arrival time of trains is not known in advance, the expected time is stored instead, which corresponds to the scheduled arrival time under normal circumstances.

Each train unit within the shunting yard must be assigned to a track. This can be a parking track, a service track, the gateway track, or a connecting movement track. However, some cases may arise where the available space in the shunting yard is insufficient to assign all the train units. This can occur either due to an infeasible problem instance where too many trains are present on the yard simultaneously, or due to an inefficient usage of the parking tracks in the solution.

To prevent early termination of simulation episodes, an additional track is included in the possible locations of a train in the state. This track has an infinite capacity and can store any train unit that does not have a valid track to move to. Because this track does not actually exist, any usage of the track violates the space constraint, and a penalty may be applied as a result. The penalty that is given depends on the reward function, which is explained in section A.2.4.

The ongoing actions in the shunting problem refer to the movements and service actions. These actions are durative in nature and can take place simultaneously. However, the precise duration of these actions may not be known in advance. Therefore, only the start time of the actions is stored, since this value is known.

A.2.2. Action Space

The action space is the set of all possible actions that an agent can take in a given state. An action in an MDP is a decision that the agent makes that affects the state of the system and transitions it from one state to another. An overview of the action space is provided in table A.1.

Symbol	Description	Prerequisites
$A_{startpark}(r_n)$	start moving train r_n	there will be a valid Park action available
$A_{park}(r_n, t_i, t_j)$	move train r_n from track t_i to t_j	the exit of the track is not being blocked; there is enough space on the next track; there is no other ongoing movement; the move will finish before the next departure
$A_{service}(s_a, r_n, t_i)$	start service s_a on train r_n at track t_i	the task can be performed on the track; the train requires the service task; there is no ongoing service on the selected train; the service track has enough space
$A_{depart}(r_n)$	depart train r_n	the train has the correct type configuration
A_{wait}	wait for the next decision moment	there is no train on the gateway; there is no departing train to be chosen; the *startpark* is not the previous action

Table A.1: Overview of the action space, with a brief description of each action symbol and the prerequisites required to execute the action

StartPark One of the actions in the action space is *startpark*. This action indicates that a train needs to be parked at a different location within the shunting yard from its current location. Although the *startpark* action itself does not involve any immediate movement, it forces the agent to make a subsequent decision about the train's next destination.

While strictly speaking, this action may not be necessary since the *park* action could be chosen directly, it is included based on the concept of move-groups. By introducing this additional decision layer, the initial branching factor is reduced, providing a more manageable set of choices for the agent. For example, consider a scenario where there are 2 trains with 10 available destination tracks for each train. Instead of considering all 20 possible parking actions at once, the introduction of the *startpark* action allows for 2 initial actions followed by the 10 corresponding parking actions, thereby reducing the total number of actions to consider to 12. This simplifies the decision-making process without altering the total number of outcomes.

The only prerequisite for selecting the *startpark* action is that a valid *park* action will be available for the agent to choose from after the *startpark* action is taken. This means that all the prerequisites for the parking action also apply, although not for any particular destination track.

Park The *park* action describes the movement of a train from its current track to another. Several constraints must be satisfied for the *park* action to be possible. Firstly, the exit of the current track must not be obstructed by another train, allowing the train to move away from the current track. Secondly, there must be sufficient space available on the destination track to park the relevant train. Additionally, no other ongoing movement should be taking place in the shunting yard currently, since this is not allowed. Moreover, the duration of the move must be such that it will be completed before the next departure event. This requirement is critical as the departing train must reach the exiting gateway within the specified time, and simultaneous moves are not permitted within the model.

Service The effect of the *service* action depends on whether the train can be serviced on its current track or not. If the train cannot be serviced on its current track, it must first be moved to a the service track. In that case, the same constraints as the *park* action also apply, meaning the train must be able to park on the service track.

Note that the *service* action is still possible if the service track is currently occupied with another train. Once the train arrives on the service track and the service crew becomes available, the service will start automatically, eliminating the need to explicitly reapply the *service* action.

The prerequisites that always apply are that the train requires the service task, and that there is no ongoing service activity on the train.

Depart The *depart* action causes the chosen train to start moving towards the exiting gateway. Once the train reaches the gateway, it is automatically departed from the shunting yard by the environment. This action allows any train to be chosen on the yard that meets the correct type configuration for the respective departure.

An important note is that the *depart* action can be chosen with trains even if the train is currently not able to depart. The reason why a train would not be able to depart is because its exit might be blocked, or because it still has unfinished service tasks. In such cases, a penalty may be applied to account for the infeasibility of the move, as explained in section A.2.4. By allowing these infeasible actions at the cost of penalties, the problem becomes more tractable as it ensures the possibility of reaching the end state.

Only considering the actually valid actions would make the problem significantly more difficult, as a lot of additional actions may be required to make the departure valid. Moreover, a snowball effect could arise where one delayed departure causes subsequent trains to also experience delays. Allowing infeasible departures allows the simulation episode to continue normally, independent of the previous departures.

Wait The *wait* action is the absence of an action. Choosing to wait results in no immediate movement or change. Instead, it allows the environment to fast-forward to the next decision moment, providing more flexibility in the order of operations. Waiting can be useful when better decisions only become possible at later decision moment. For instance, a train may want to wait for another train to finish their service task so it can leave their track and free up space.

As sometimes it is necessary to make an immediate decision, it is not always possible to wait. Waiting is not possible if there is a train occupying the gateway since it is not permitted to remain parked at the gateway but must be moved to another location. Furthermore, waiting is not possible when a train needs to be selected for the current departure event. When a *startpark* action has already been chosen, a parking action for the train must be selected first before any waiting can occur.

A.2.3. Transition function

This section aims to provide a detailed explanation of the transition function, describing how the state changes as different actions are executed. The transition function $T(s'|s,a)$ describes the effect that each action has on the current state. In the shunting problem, many actions are durative, meaning they require a certain amount of time to complete. For example, actions such as train movements and services take time to perform. At the same time, the model allows for the selection of new actions while others are still in progress. This means that most actions in the shunting problem only modify the state by adding the action to the set of ongoing activities, without completing them immediately. An overview of how each action influences the state can be found in Table A.2.

Action	Effect on the state
$A_{startpark}(r_n)$	the $T_{parktrain}(r_n)$ trigger is activated
$A_{park}(r_n, t_i, t_j)$	the train is remove from its current track the movement is added to the ongoing activities of the state with a timestamp of the current time
$A_{service}(s_a, r_n, t_i)$	if the train is already on the right track $\rightarrow$ the service is added to the ongoing activities with a timestamp of the current time else $\rightarrow$ a movement to the service track is added to the ongoing activities with a timestamp of the current time and the train is remove from its current track
$A_{depart}(r_n)$	a movement to the gateway track is added to the ongoing activities with a timestamp of the current time
A_{wait}	$-$

Table A.2: The immediate effect each action has on a given state.

Whenever an activity is added to the set of ongoing activities within a state, a timestamp of the current time is included. This inclusion is necessary because the exact duration of an action may not be known at the time the activity starts. The duration of each ongoing activity is determined by the disturbance model, which handles the variability in the completion times. The disturbance model is described in section A.3 in the next part of this section. By incorporating timestamps and accounting for uncertain durations, the transition function captures the dynamic nature of the shunting problem, allowing for more accurate modeling of state transitions over time.

A.2.4. Reward function

In this section, the design of the reward function is discussed. The reward $R(s, a)$ is the value provided by the environment after applying action a in state s. Ultimately, there is little intrinsic value of individual actions. What matters is whether the agent can reach a goal state where all trains have arrived and departed on time, and all service tasks have been completed, while avoiding any constraint violation.

As mentioned in Section A.2.2, certain invalid actions are allowed to be performed at the cost of potential penalties. The following actions describe all the situations where a constraint is violated:

- **Crossing** Moving a train from a track when the exit is blocked by another train is referred to as a *crossing*. This constraint is relaxed only for departing moves, to allow them to meet the scheduled deadline.
- **Unfinished service tasks** Departing a train with unfinished service tasks is technically possible, but is not desirable.
- **Space shortage** In the case that there is no available parking track on the shunting yard, the train is kept on an imaginary track with infinite space that does not physically exist.

In principle, a solution is deemed feasible if it successfully reaches the goal state while satisfying all the specified constraints. Therefore, a binary reward could be used, with a value of 1 for a feasible plan and 0 for infeasible plans.

$$R_{binary}(plan) = \begin{cases} 1 & \text{if the plan is feasible} \\ 0 & \text{otherwise} \end{cases}$$

In practice, finding a feasible plan can be challenging. In such cases, a binary reward is unable to compare the quality of two infeasible plans. This limits the ability to utilize previous results to guide a solution method to find better plans.

To address this issue, another approach is a weighted reward function. This function assigns a weight to each type of constraint violation. By summing the weighted violations, a single reward value is calculated. This approach allows for a better evaluation of plan quality, enabling solution approaches to use the reward signal to produce better solutions.

$$R_{weighted}(plan) = -((W_c \cdot c) + (W_u \cdot |u|) + (W_p \cdot p))$$

- c: the number of crossings in the plan
- u: the set of unfinished service tasks in the plan
- p: the total number of carriages that were not able to be parked in the plan
- W_c: the weight of the crossing penalty
- W_u: the weight of the unfinished service tasks penalty
- W_p: the weight of the space shortage penalty

A.3. Environment

The environment is what the agent interacts with. In each state, the environment provides a set of possible actions for the agent to choose from. The agent then selects an action, after which the environment updates the state accordingly based on the chosen action. However, due to the real-time nature of the problem, some consideration has to go regarding the timing of the decision moments.

One option for decision-making is to use discrete decision points. These decision points are defined at specific time intervals, and the agent only makes decisions at these predefined points. At each decision point, the agent considers the current state chooses from the set of available action at that time. However, if the sampling rate is too low, the agent may not have enough decision points to make all the necessary decisions within the given time frame. On the other hand, if the sampling rate of decision points is too high, most of the decision moments contain no possible actions other than waiting. For instance, when all service crews are already in use or when a train is already in motion.

A more efficient strategy is reactive planning. In reactive planning, triggers are activated every time the state changes in a specific way, indicating that the agent needs to make a decision. The simulation environment keeps track of all the changes made to the state and detects when a triggers needs to be activated. This allows the agent to respond immediately once the state has change and make the next decision based on the updated state.

A.3.1. Triggers

In the context of this project, the shunting problem is simulated rather than observed directly from the real world. The simulated environment needs to fast-forward the state until a change occurs that allow the agent to make a new decision. At that point, the simulation pauses, and waits for the the agent's decision. Once the agent selects an action, the simulation resumes, and the process repeats.

A trigger is activated in response to specific events or actions taking place within the state. They serve as the indicator for the agent to make decisions or respond to changes in the environment. An overview of all the triggers is provided in Table A.3.

Trigger	Caused by	Decision to make
$T_{parkstarted}(r_n)$	$A_{startpark}(r_n)$ chosen	a parking track for train r_n needs to be chosen
$T_{servicedone}$	service action finished	which activity to do next
$T_{movedone}$	move action finished	which activity to do next
$T_{matchdeparture}(d_i)$	departure event	which train to use for departure d_i

Table A.3: Overview or all the triggers, with the change in state that caused the trigger to activate, and the next decision to make

The *servicedone* and *movedone* are activated when the corresponding action is completed. These triggers signify the availability of resources and allow the agent to select new activities that might make use of the freed-up resources.

The *matchdeparture* is triggered by when a train is required for departure. At this decision moment, the agent needs to choose which train to use for the departure event.

One notable trigger is the $T_{parkstarted}(r_n)$ trigger, which is unique as it is not caused by an external event but is chosen directly by the agent through the $A_{startpark}(r_n)$ action. The purpose of this trigger is to ask the agent to decide on a parking location for the given train. Note that the $A_{startpark}(r_n)$ action is the only available action if there is a train on the gateway track, since trains are not allowed to stay at the gateway track. This effectively means that the $T_{parkstarted}(r_n)$ trigger is also indirectly activated by arrival events.

A.3.2. Fast Forward

Fast-forwarding allows the environment to progress to the next relevant trigger. If a trigger has already been activated, there is no need to fast-forward further until the agent makes a decision the chosen action is applied. Once the action is chosen, the trigger is removed and the environment can proceed to fast-forward again until the next decision moment and activate the new trigger.

The determine to which point the environment should fast-forward, it first needs to determine the earliest time of the next event and the earliest finish time of the activities. Regarding the timing of the events, departure times are known and static, while arrival times are based on the disturbance model that is used. The determination of the timing of the next activity is similar, move times are known exactly, and service times are based on the disturbance model. The environment then advances to whichever event occurs first by updating the current time, activating the corresponding trigger, and updating the state accordingly. An overview of all state changes that occur along with trigger activation is provided in Table A.4.

Trigger	Change in state
$T_{parktrain}(r_n)$	add train r_n to the gateway track
$T_{servicedone}$	remove the corresponding service action from the ongoing activities remove the task from the set of required services of the train
$T_{movedone}$	remove the corresponding park action from the ongoing activities add the train to the new track
$T_{matchdeparture}(d_i)$	–

Table A.4: The changes that are made to the state in addition to activating a trigger

It's important to note that each fast-forward iteration may sample a different value from the disturbance model, potentially resulting in a sampled finish time that is lower than the current time of the state. However, this is not possible since the previous fast-forward already established that this did not occur. To resolve this issue, rejection sampling is used. This ensures that every sample drawn from the disturbance model is consistent with the current state, avoiding inconsistencies in the simulation.

An example of this is shown in figure A.1. Consider two probability density functions A and B, which describe the probability of something happening at a certain time. The first decision moment is determined by sampling from both A and B, resulting in s_1 and s_0 respectively. Since s_0 occurs before s_1, time in the environment is fast-forwarded to $t = s_0$. After the agent has made a decision, and time needs to progress again, it rejects all samples from A that are lower than the current time ($\leq s_0$). The possible times that will be accepted are indicated by the blue area. The disturbance model does not need to sample from B again, since it already happened.

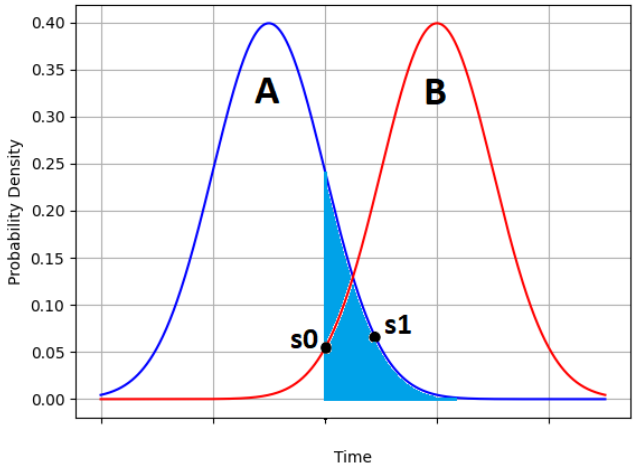

Figure A.1: Example of rejection sampling in the fast-forward method.

B

Policy Analysis

In this chapter, we perform an analysis of various policies to solve the shunting problem and evaluate their performance. A series of experiments are conducted to assess the performance of each policy under different scenarios and conditions. Based on the results of these experiments, we can conclude the most effective policy that will be used throughout the thesis.

B.1. Policy Structure

The solution of MDP is a policy. A policy is a function that takes as input a state and set of possible actions, and outputs the best action according to the policy.

The main difficulty of the shunting problem is due to the interconnectedness of the various subproblems. However, the MDP formulation as decribed in Chapter A provides an opportunity to divide the overall policy into smaller subpolicies. Although this division is not necessary, since each subpolicy responds to different triggers, the modular structure makes it easier to understand the overall policy, and analyze the effectiveness of each individual policy component. Each subpolicy is responsible for a specific set of actions and triggers, addressing different aspects of the problem. An overview of the different subpolicies, along with the corresponding actions and triggers, is provided in Table B.1. Each subpolicy is explained in detail the following subsections.

sub-policy	actions it is responsible for	triggers it responds to
ParkingPolicy	$\{A_{park}\}$	$\{T_{parkstarted}\}$
ServicePolicy	$\{A_{service}, A_{startpark}, A_{wait}\}$	$\{T_{movedone}, T_{servicedone}\}$
DepartingPolicy	$\{A_{depart}\}$	$\{t_{matchdeparture}\}$

Table B.1: Overview of the different subpolicies and the triggers it responds to with the available actions it can take

The decisions about a single train within the shunting yard is depicted in Figure B.1. Upon arrival at the shunting yard, the train is automatically placed on the gateway track by the environment. When the train is on the gateway, the agent is forced to determine a parking track since trains are not permitted to remain on the gateway track. The selection of the parking track is decided by the parking policy, which chooses the parking track for each train.

While a train still has pending service tasks, it should not be chosen as the departing train. The service policy is used for activities related to actions within the yard, such as deciding when to start servicing a train. Once the service task is completed, the train must leave the service track and relocate to a parking track again. The precise moment for this transition is determined by the service policy. However, the parking track is chosen by the parking policy again. This process continues until the train has finished all of its required services.

Once a train has successfully completed all of its required service tasks, it can be chosen as the next train for departure, without getting any penalty for having unfinished tasks left. After the train is chosen to depart, the train will automatically exit the shunting yard as soon as it arrives at the departing gateway.

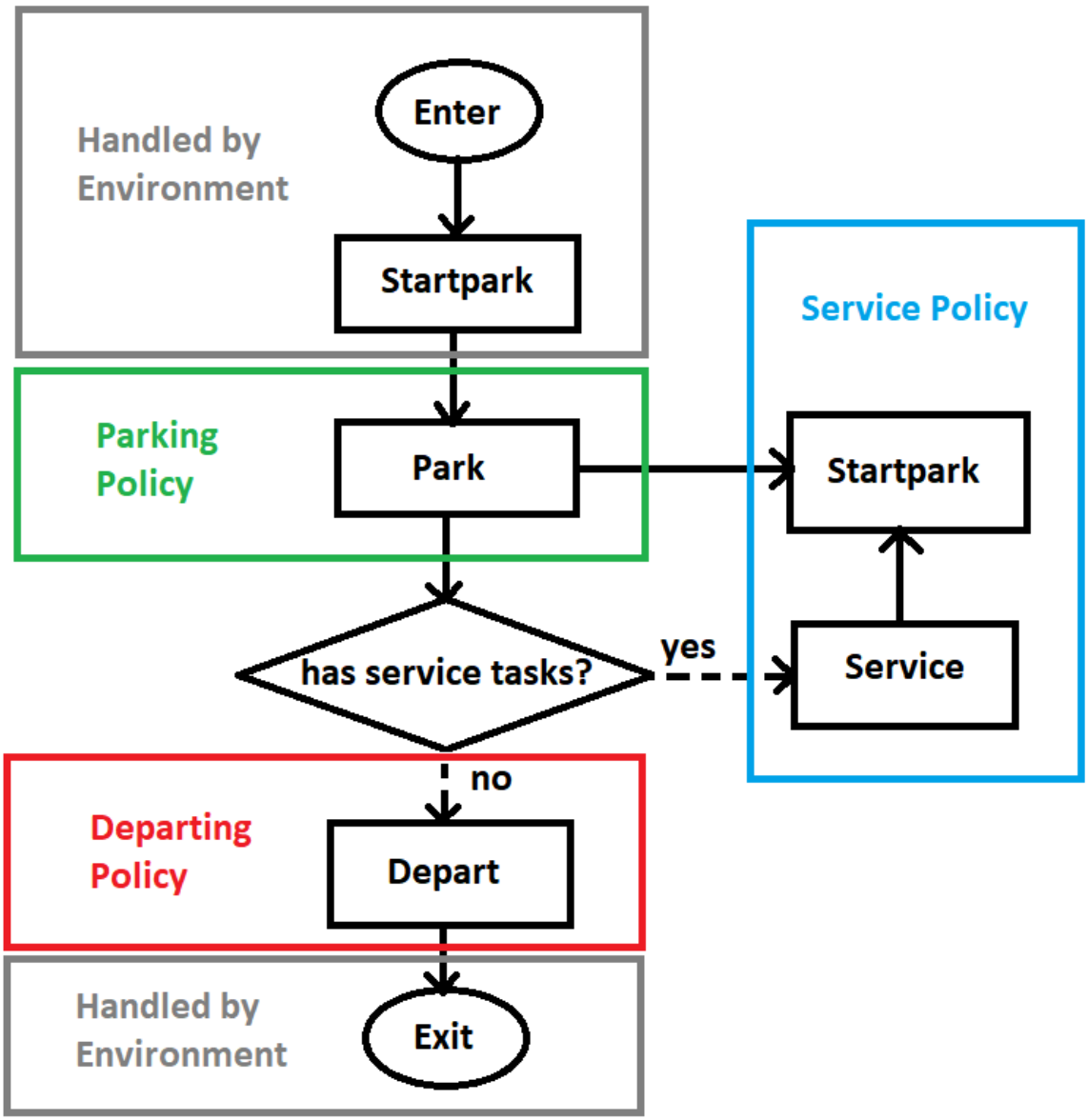

Figure B.1: The flow of an individual train, with the actions it takes while on the shunting yard. The

B.2. Parking Policy

The parking policy is responsible for determining the parking spots for each train. It is important that the trains are placed, such that no trains are blocked when they need to leave their parking spot. Another relevant aspect of the parking policy is the efficiency of which it can divide the total track capacity. In case there is a large number of trains on the yard simultaneously, there needs to be an available parking spot for each train to avoid space shortage problems.

In this section, three rule-based policies are considered. First, a priority-based strategy, which seeks to predict the priority of each train and assigns parking spots accordingly. Second, a type-based strategy, which aims to group trains of the same type together on the same tracks. Finally, a combination of the two policies is proposed.

Priority-based Parking The priority-based parking strategy assigns a priority value to each incoming train based on its predicted position in the departing sequence. Although the exact matching of each departure is unknown at the start, trains that arrive earlier generally have more time to complete their

service tasks and prepare for departure. Therefore, it is likely possible to match incoming trains to the first departing train of the same type that is not matched yet. As a result, each arriving train is given the priority value corresponding to the index of the first non-matched departure of the same type.

This priority value is used throughout the decision making process. For this policy it is assumed the higher priority train will leave the track before the lower priority train. However, the rules for parking differ slightly depending on the type of track involved. In the case of LIFO (Last-In, First-Out) tracks, the last train to arrive is the first one to leave, therefore trains are parked adjacent to trains with lower priority values. Conversely, in the case of FIFO (First-In, First-Out) tracks, trains are parked behind trains with higher priority values, since the parking train will only be able to leave after the already parked train.

Type-based Parking The type-based parking strategy focuses on grouping trains of the same type together on the same tracks. The primary objective of this policy is to minimize the number of crossings, which occur when a train leaves its track while the exit of the track is currently blocked by another train. By keeping trains of the same type next to each other, crossings can be easily avoided. If a train is blocked from exiting the track due to another train of the same type that is blocking the exit, it is possible to substitute the obstructed train and use the other train instead. This substitution is only possible when both trains are exactly the same. However, if there are differences between the two trains, such as one requiring a service task while the other does not, they are no longer considered identical and cannot be used interchangeably.

Combined Parking Strategy The type-based parking policy has several instances where there may be a tie between the possible actions. For instance, if there are multiple tracks available that all contain the same train type, or when the train is parked on one of the empty tracks. The combined policy is fundamentally the same as the type-based policy, but uses the priority-based policy to resolve these ties.

B.3. Service Policy

In the shunting problem, several trains enter and exit the shunting yard. During their time in the yard, it is necessary to perform several service tasks, such as cleaning and maintenance, to ensure the trains are prepared for their next departure. The service policy is responsible for determining the next activity on the shunting yard. The primary objective of the service policy is to ensure that all trains have finished their required service tasks before its departure.

This subsection explains the implementation of three rule-based policies. The no-waiting policy, which prioritizes avoiding waiting whenever possible, the priority-based strategy that predicts train priorities for sequential service allocation, and finally the type-based strategy that assigns priority based on train types.

No-waiting A random policy for selecting actions randomly chooses from all available actions at any given decision point. Choosing the *wait* action is often possible at most decision moments, so it has a relatively high chance to be picked by the random policy during execution. However, waiting is generally not as useful as other actions since it does not actively contribute to completing the required services. Waiting is mostly used in providing flexibility in the possible order in which service tasks are performed. For a random policy, the specific order of service tasks is insignificant, because it does not make informed decisions about the order of the service tasks. Therefore, the *no-waiting* policy modifies the random policy by disallowing the wait action to be chosen when alternative options are available. It still chooses randomly between the remaining actions.

Priority-based Service The priority-based service policy use the priority of trains to determine the next train to service. Similar to the priority-based parking policy, the same rule is utilized for assigning priorities. The priority of each incoming train is determined based on the index of the first non-matched departure of the same type in the departing sequence. Trains with higher priorities are assumed to depart before trains with lower priorities.

There are two main actions for the service policy to consider. The *service* action $A_{service}$ initiates a service by starting the necessary sequence of action on the train. The *startpark* action $A_{startpark}$

concludes the service by re-parking the train on the service track after completing the service tasks. The priority-based service policy always selects the train with the highest priority for servicing, without distinguishing between the two actions. The emphasis is solely placed on the priority of the trains to ensure they are finished in time between their departure

B.4. Departing Policy

The departing policy is responsible for deciding which train to select for a given departure event. Generally, the chosen train should meet two essential criteria: it should not be obstructed by another train on its route, and it should have completed all its designated service tasks. The policy should always prioritize selecting a train that satisfies these conditions whenever possible. However, there are cases where there are multiple trains that satisfy these conditions. Additionally, there are other cases where no valid train can depart without getting a penalty. In both these situations, the departing policy has to choose which train to depart.

Reward-based Departing The reward-based strategy uses a simple principle: it selects the action that results in the least immediate penalty while randomly picking between actions of equal value.

For each train in our system, the penalty is calculated that would be given if that train were chosen for departure. It then chooses the train with the lowest penalty.

As a result, this approach always selects a valid departure whenever possible, as no penalty will be given. In cases where a valid departure is not possible, the train is chosen that minimizes the number of crossings and already has completed the most service tasks.

Priority-based Departing The priority-based strategy is an extension of the reward-based strategy. It uses the same method to determine the best trains for departure. However, in case there is a tie between multiple trains, the priority of the trains is used. Specifically, it prioritizes the departure of trains with higher priorities.

Similar to the priority-based parking and service policies, the same rule is used for assigning the priorities. The priority of each incoming train is determined based on the index of the first non-matched departure of the same type in the departing sequence

B.5. Evaluation

In this section we perform a number of experiments with several combinations of the subpolicies. We test which overall policy performs the best, and analyze what the most likely cause of failures are for the policies.

B.5.1. Experimental Setup

The problem instances are generated in the same way as described in Section 6.1. The instances are modeled after realistic scenarios, using realistic distributions for the train types, their required service and the timetable of arrival and departure times.

Layout The experiments conducted in this study are based on the layout of the ”*Kleine Binckhorst*” shunting yard. The layout of the yard includes eight parking tracks, with available space ranging from 7 to 19 carriages. Additionally, there are three dedicated service tracks within the yard. One track is designated for external washing, while the remaining two tracks are used for internal cleaning

To analyze the impact of different shunting yard layouts, each experiment is performed twice. In the first instance, all the tracks are modeled as First-In First-Out (FIFO) tracks, while in the second instance, all the tracks are modeled as Last-In-First-Out (LIFO). This is to analyze the different between the two types of shunting yard layouts, and how well each policy performs on them.

B.5.2. Analyze Parking Policies

The first experiment analyzes the performance of the different parking policies. For this experiment we only look the night shift, which is from 8PM to 6AM. Furthermore, we assume all arrivals occur before the first departure. Since we want to determine the best parking policy, no service tasks included.

Feasibility Both the type-based and the priority-based strategies perform significantly better than random parking. There is however an interesting difference between the performance of the policies on FIFO and LIFO tracks.

For the FIFO tracks, the priority-based parking strategy is slightly better than the type-based strategy. The priority-based strategy naturally work well for FIFO track, because the priority is also determined using a FIFO strategy, where the first arriving trains are matched to the first possible departure.

For the LIFO tracks, the type-based parking strategy starts to outperform the priority-based solution. In general, it seems to be more challenging to find a feasible solution for LIFO tracks. A possible reason why the type-based strategy works well for LIFO tracks, is due to the interchangeability of similar trains that are of the same type. If two trains are parked on the same track, the train in front blocks the exit of the other trains. However, when both trains are the same type of train, the train in front can simply be selected when required for departure instead of the one in the back.

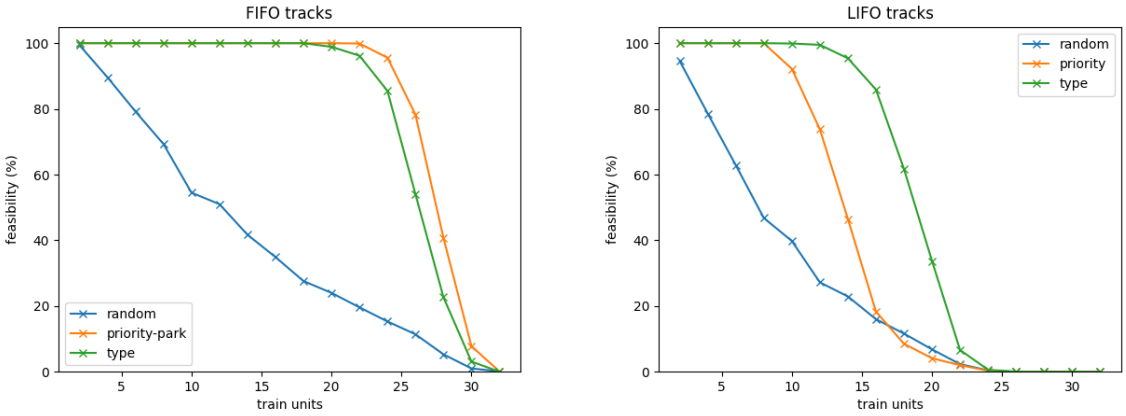

Figure B.2: Feasibility of each policy with increasing number of train units

Crossings Figure B.3 shows the average number of crossings each policy encountered.

We see that for the LIFO tracks the type-based strategy is able to avoid the most amount of crossings. This aligns with the aim of the policy, which is designed to avoid crossings by parking interchangeable trains next to each other.

Interestingly, the priority-based strategy has a higher number of crossings than the random parking strategy on LIFO tracks. An possible reason for this, is that the priorities are assigned based on a FIFO ordering, which is the opposite of the track ordering.

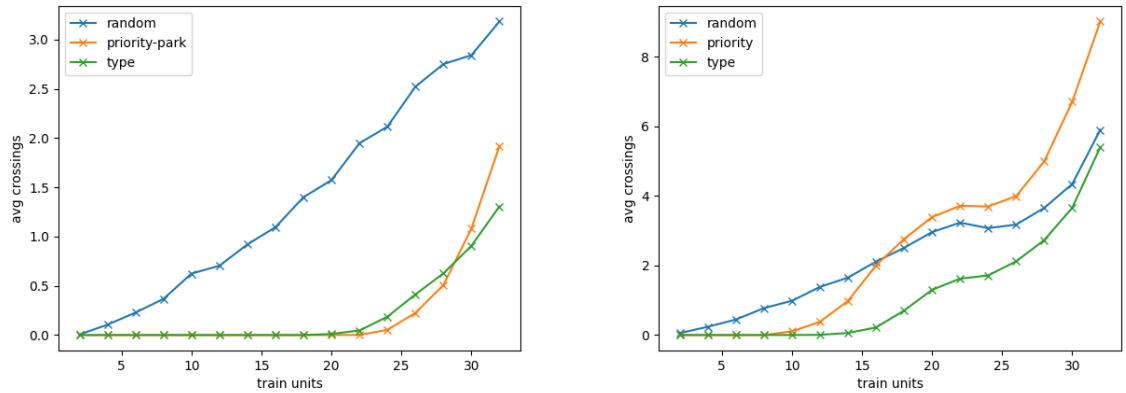

Figure B.3: Average number of crossings of each policy with increasing number of train units

Space Shortage Figure B.4 shows the average space shortage each policy encountered.

Predictably, as the number of trains reaches its limit, it becomes more likely there is not enough total available space on the shunting yard to park all the trains. The type-based policy performs slightly worse compared to both the priority-based and random strategy. The type-based strategy always keeps trains of the same type on the same track, which does not lead to the most efficient use of track space if the length of the track cannot be perfectly divided into the length of the train type.

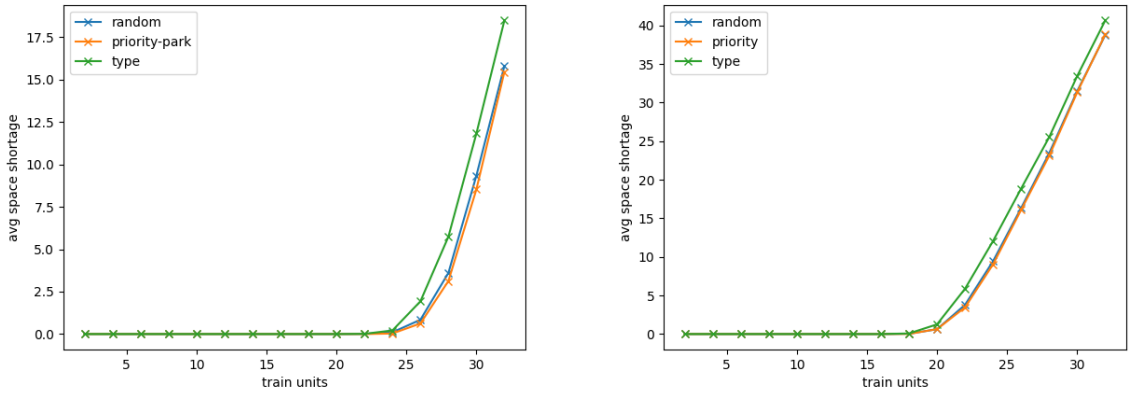

Figure B.4: Average number of space shortage of each policy with increasing number of train units

B.5.3. Analyze Combined Policies

In this experiment, we analyze the performance of the service policies in combination with parking. We sample arrivals and departures according to the realistic distribution. We compare several combinations of parking and service sub-policies:

name	parking sub-policy	service sub-policy	departing sub-policy
random	random	random	reward-based
no-wait	random	no-waiting	reward-based
priority	priority-based	priority-based	priority-based
type	type-based	no-waiting	reward-based
combined	combined	priority-based	priority-based

Night Shift We first only consider the night shift from 8PM to 6AM, where all arrivals occur before the first departure. Figure B.5 shows the percentage of feasible simulations for each policy.

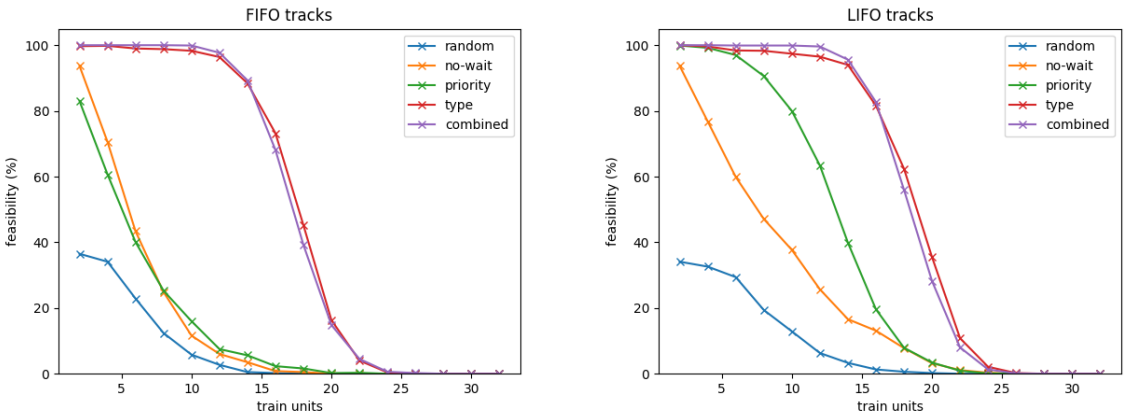

Figure B.5: Feasibility of each policy with increasing number of train units

In contrast to the parking experiment, the priority-based strategy does not perform well when services are included into the problem. Both combinations that use the type-based strategy for parking perform significantly better than the rest.

The no-wait policy significantly outperforms the random policy, even though the strategies are very similar to each other. This suggest that waiting is in general not a good action to choose, which makes sense because waiting does not progress the solution towards the desired goal.

There is no notable improvement by using the priority-based service strategy compared to no-waiting. Since all trains arrive before departure, the service order is less important, because most trains can be finished before the first departure. This means that servicing the trains in a random order is good enough.

Realistic Problem Instances Next, we look at a full planning horizon of 24 hours. This means that arrivals and departures are mixed in time. Figure B.6 shows the percentage of feasible simulations for each policy.

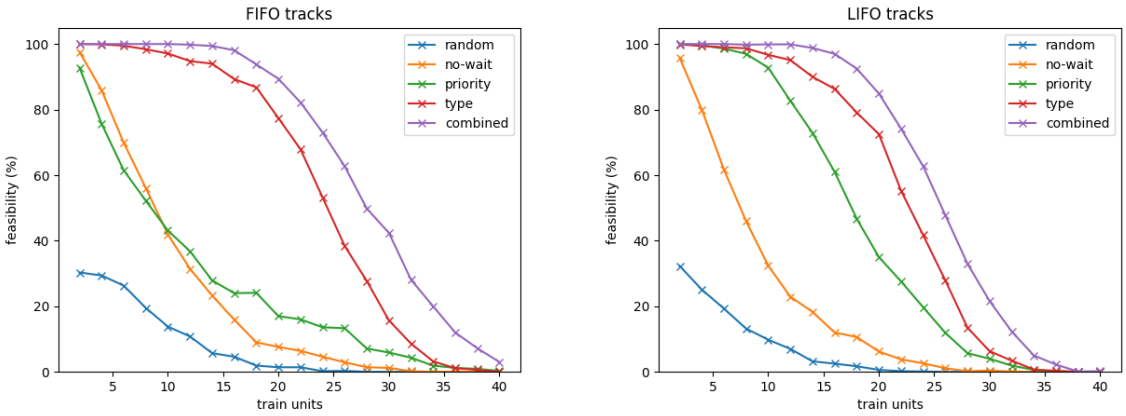

Figure B.6: Feasibility of each policy with increasing number of train units

The results are similar to the previous experiment which considered the night shift. The policies which use a priority-based strategy are better compared to before. An explanation for this is that the arrivals and departures are now mixed in time. This means that, in contrast to the previous experiment, not every train can be finished before the first departure. This makes it more important to have a service strategy that allows the necessary trains to be finished in time of departure.

B.6. Conclusions

Based on the results, the type-based parking strategy is the most impactful sub-policy to use, regardless of the other sub-policies that are used. Even if the services are chosen randomly, the type-based parking strategy still leads to relatively good performance.

The priority-based service policy does seem to improve the performance over randomly selecting services in case the arrivals and departures are mixed in time. In case all trains arrive before the first departure, there is no notable difference between priority-based strategy and random without waiting.

The biggest improvement for the service-strategy is to exclude the waiting actions, suggesting that waiting is in general not a good action to choose.

The best overall policy is the combined policy, which consists of combined parking (type-based with priority-based for tie-breakers), priority-based service and priority-based departures.

C

Search Methods for a Deterministic Environment

In this chapter, we analysis various search methods that can be used to find feasible solutions to the shunting problem. Although existing policies cannot reliably solve the problem, they may be able to guide a search method towards the most promising direction. The goal of this chapter is to determine whether our problem formulation allows for Monte Carlo-based search techniques to work well in finding feasible plans, and which search method is the best one to use.

Table C.1 provides an overview of all the search methods that are considered in this chapter. It should be noted that the search methods we consider are designed for a deterministic environments, so they cannot be used in non-deterministic environments. Each method is explained in more detail in the next sections.

Search Method	Description Summary	Search space
Randomized Iterative Greedy Search	iteratively perform policy rollouts and keep track of the best rollout so far	policy solution space
Monte Carlo Tree Search	iteratively perform policy rollouts to estimate the action-value of a state and build a search tree from the initial state	full solution space in tree. policy solution space outside tree
Nested Monte Carlo Search	improve the rollout quality of higher levels by performing rollouts of a lower level	full solution space (bias towards policy)
Nested Monte Carlo Beam Search	same as Nested MC Search; but don't commit to a single move, instead, keep a "beam" of the best states	full solution space (bias towards policy)

Table C.1: A brief description summary of the considered search methods with their respective solution space that is reachable from the search method

C.1. Randomized Iterative Greedy Search

Although a given policy may fail to reliably generate a feasible plan, there may still be chance that it does find a feasible solution eventually. The randomized iterative search method exploits this property in a very simple way, by iteratively using the policy to construct solutions and keeping track of the best solution found so far. Since the search process itself requires minimal additional time, it is able to quickly perform a relatively large number of iterations. Additionally, the implementation and understanding of this method is very straightforward.

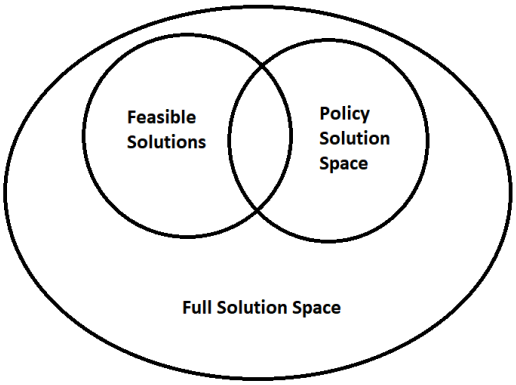

Figure C.1: Diagram of the policy's solution space and the space of feasible solutions.

An interesting property that arises from this method is that the final solution always remains confined within the search that is reachable from the given policy. For example, if the type-based parking policy is used, the resulting plan will always avoid parking trains next to other trains of different types whenever possible. This means that the final plan is always restricted by the rules of the the policy. A diagram in Figure C.1 shows the solution space that is reachable using the policy, and the feasible solution space. The randomized iterative search is restricted to the to the set of outcomes that is reachable from policy. The set of feasible solution this search method is able to find is described by the intersection of the policy's solution space and the set of feasible solutions. If there exists little to no overlap between these sets, the randomized greedy search method may never find a feasible plan, or take a lot of time to compute.

Each iteration runs completely independently from the others. A benefit of this method is that it does not introduce any additional bias into the algorithm. Every iteration is equally (un-)likely to produce a feasible solution A downside, however, is that it is very computationally inefficient, since it does not use the results from previous iterations to improve the consequent ones.

C.2. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that can be used for sequential decision making problems. It has proven to be a successful method for game-playing problems. The algorithm can handle large search spaces by iteratively building a search tree in an asymmetric way towards the most promising region. Each iteration of the MCTS algorithm consists of the following four steps:

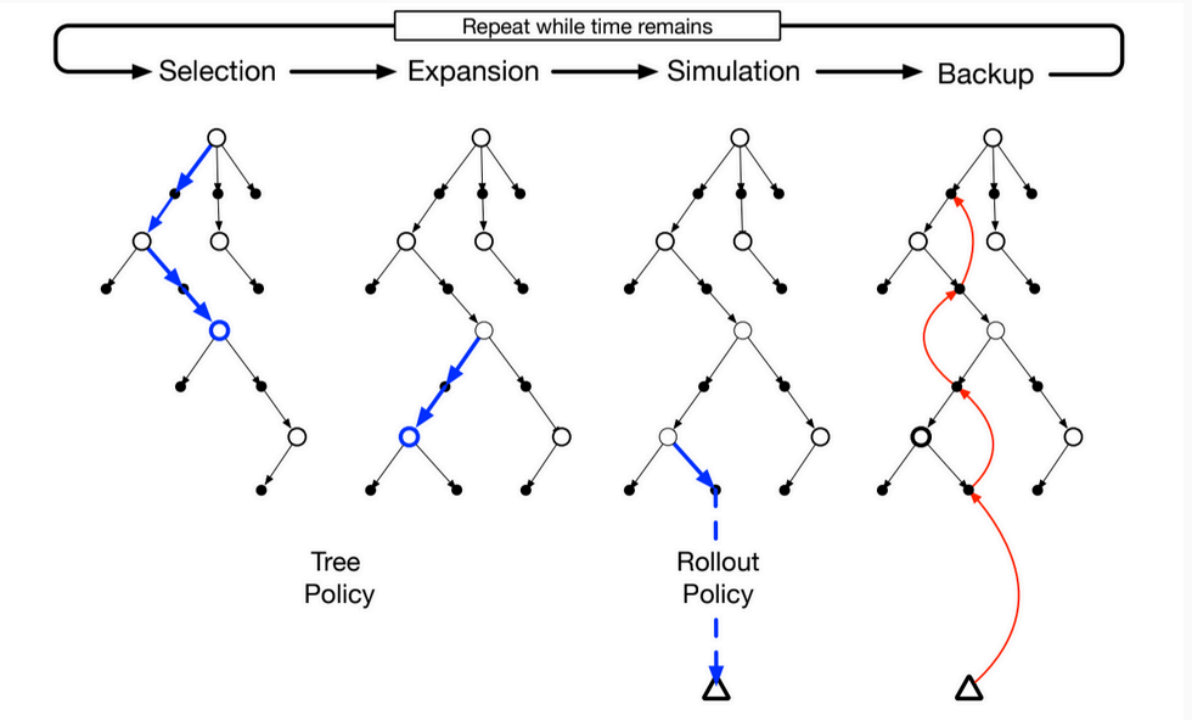

Figure C.2: TODO: source https://lcalem.github.io/blog/2018/12/01/sutton-chap08

1. **Selection**: Starting from the initial state, actions are chosen based on the values in the current version of the search tree. This step continues to pick actions from the tree, until a terminal state or a leaf node is reached.
2. **Expansion**: Once a leaf node has been reached, the tree is expanded with at least one new node corresponding to the new state.
3. **Simulation**: Actions are chosen based on a simulation policy until a terminal state is reached. Often a random policy is used, but other policies can be used as well.
4. **Backpropagation**: The final result of the simulation is added to all the nodes that were chosen during the selection phase. Each visited node also increments its counter that tracks how often the state has been visited. These statistics are used to estimate the value of the visited state and guide the selection of actions for future iterations.

Selection The aim of the selection step is to pick the actions from the search tree that seem to be the best so far based on the results of previous iterations. The value of each action in a state is estimated by the average reward of the simulations:

$$Q(s,a) = \frac{v_i}{n_i}$$

- v_i: total reward of node i
- n_i: the number of visits of node i

The selection procedure however, needs to maintain a balance between exploration and exploitation. This is typically done by treating every node in the search tree as a multi-armed banded problem, and minimizing the regret. The formula is called Upper Confidence Bounds applied for Trees (UCT):

$$\texttt{UCT} = Q(s,a) + C\sqrt{\frac{\ln N_i}{n_i}}$$

- N_i: the number of visits of the parent of node i

- C: the exploration constant, theoretically equal to $\sqrt{2}$, but often chosen empirically

MCTS is often used for two-player games, where there is some uncertainty on the opponent's play and where the rewards are indicated by either a loss, draw of win. To allow for a wider range of possible values, and when there is no opponent, an extension can be made by adding a "possible deviation" term to the UCT formula [25]:

$$\texttt{UCT'} = \texttt{UCT} + \sqrt{\sigma^2 + \frac{D}{n_i}}$$

- σ^2: the variance of the node's simulation results
- D: uncertainty constant, so infrequently visited nodes will be considered less certain

An accurate evaluation of a node may require a lot of simulations, as the statistics are not reliable when a node is visited few times. To address this issue, the node can be initialized with a heuristic value that estimates the quality of the action, without requiring any simulations. A common method to do this is by adding a progressive bias to the value of the node [13]:

$$Q'(s,a) = Q(s,a) + \frac{b_i}{n_i}$$

- b_i: the heuristic score of node i

Expansion Once a leaf node has been reached, the tree is expanded with at least one new node corresponding to the new state. Normally each unvisited action of a node is chosen at least once before adding deeper layers to the tree. However, this causes less exploitation to occur with nodes deeper in the tree, especially when the branching factor is large. One technique that aims to reduce this effect is progressive unpruning [13], which uses heuristic knowledge to immediately prune some of the possible actions, but eventually unprunes them after some time. This temporarily eliminates obviously poor choices allowing the search to focus more time on better options.

Simulation Nested approaches [12] and [3] have been successful in solving single-player puzzles and similar optimization tasks. These approaches are designed to optimize moves at all stages of the search process, rather than just near the root of the tree where most of the search time is typically spent. The idea behind nested approaches is to recursively use the search method itself as the rollout policy, which leads to the higher level search producing better results.

Backpropagation The final result of the simulation is added to all the nodes that were chosen during the selection phase. Simulations later on are generally more important, since both the tree policy and the simulation policy improve with more iterations. Therefore the results can be weighted according the current average performance of the simulations.

$$n_i \leftarrow n_i + p_t$$

$$v_i \leftarrow v_i + r_t \cdot p_t$$

- n_i: the number of visits of node i
- v_i: the total value of node i
- p_t: the average performance of the simulations at iteration t
- r_t: the return value obtained by simulation t

C.3. Nested Search

Nested approaches [12] and [3] have been successful in solving single-player puzzles and similar optimization tasks. These approaches are designed to optimize moves at all stages of the search process, rather than just near the root of the tree where most of the search time is typically spent. The idea behind nested approaches is to recursively use the search method itself as the rollout policy, which leads to the higher level search producing better results

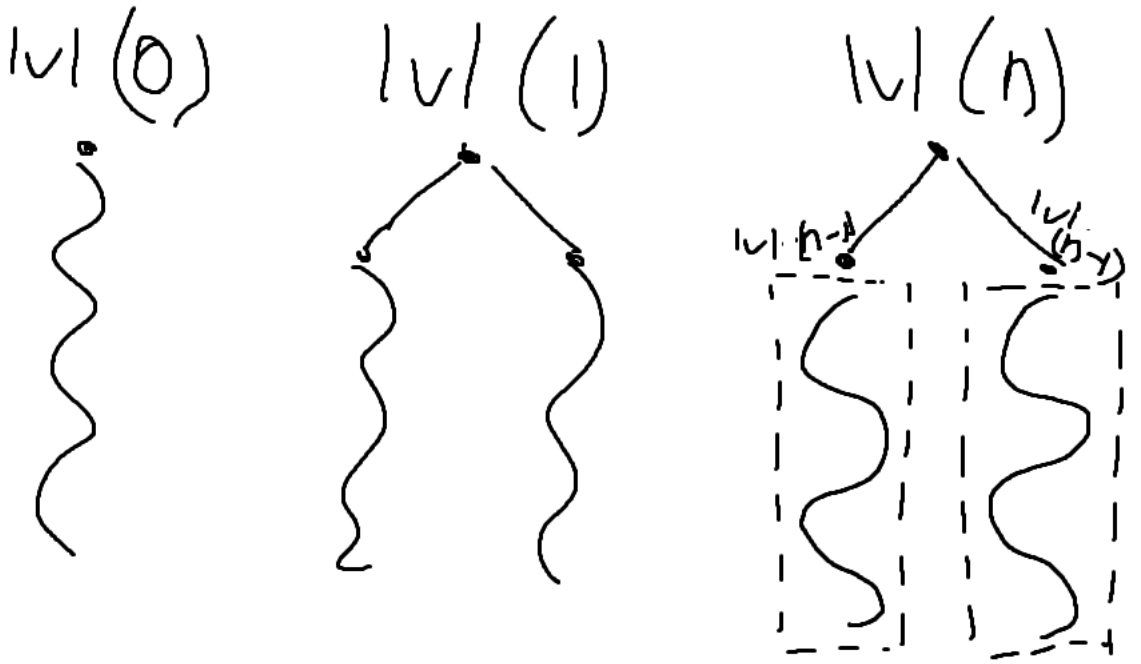

Figure C.3: Overview of nested search with increasingly higher levels. On the left, a regular rollout is shown, represented as a wavy line. A level 1 nested search performs a rollout for each current possible action to estimate its value. This process is generalized to higher level nested searches, where a level-l search performs a level-$(l-1)$ nested search for each action before choosing the action and repeating the process in the next state.

A level-0 nested search is defined as the normal rollout. A level-l search performs a full simulation, where at each step the highest scoring action is chosen based on a level-$(l-1)$ nested search for each possible action. The computational complexity of the algorithm is $O(a^n h^{n+1})$, where a is the branching factor, h the planning horizon, and n the initial nested search level.

The required runtime is not known at the start of the search. For the purpose of this thesis, the algorithm is slighty modified to allow for more flexibility in the designated runtime, without requiring explicit parameter tuning. It is possible to make the algorithm anytime, by wrapping the search in an iterative loop. Each iteration, the search parameters are gradually increased. We start with a level-0 search, and increment the level with each new iteration. This means the algorithm keep restarting in the hope to eventually find a feasible solution. Since the runtime complexity is exponential with respect to the level that is used, a previous lower level search has a relative low impact on the runtime compared to the new higher level search.

C.4. Beam Search

Monte Carlo Beam Search [2] is an extension of Nested Monte Carlo Search. It combines Nested Monte Carlo Search with Beam Search. Nested Monte Carlo Search only keeps a single state at the time, whereas Monte Carlo Beam Search remembers the best b states in a 'beam'.

The time complexity of the algorithm is $O(w^n a^n h^{n+1})$, where a is the branching factor, h is the planning horizon, n is the nested search level and w the size of the beam.

C.5. Evaluation

To evaluate the performance of the search methods, we conduct experiments in four different scenarios. First, a short night shift without any service tasks is considered. Afterwards, the same experiment is repeated, but with the inclusion of service tasks. The third experiment analyzes a full day without any service tasks. Again, the same experiment is repeated, but with the inclusion of service tasks. These experiments are chosen to compare with the results of the current state-of-the-art method proposed by van den Broek [6] based on Simulated Annealing (SA). It should be noted, however, that a direct comparison is not possible, as the problem instances are not exactly the same. The SA method also considers trains that consist of multiple train units, which significantly increases the problem's difficulty, since trains need to split and combine to form the correct configurations. Nonetheless, the SA method is included as a point of reference due to the similarity in the experimental setup

Experimental Setup In each experiment, 50 problem instances are generated for every even number of total train units. The generation method described in Section 6.1 is used for this. A maximum time limit is set to only 10 seconds of computation time to find the solutions. Which is very short, as the duration for which the plans are created is much longer. But it proved to be sufficient to find most plans.

Night shift without service tasks

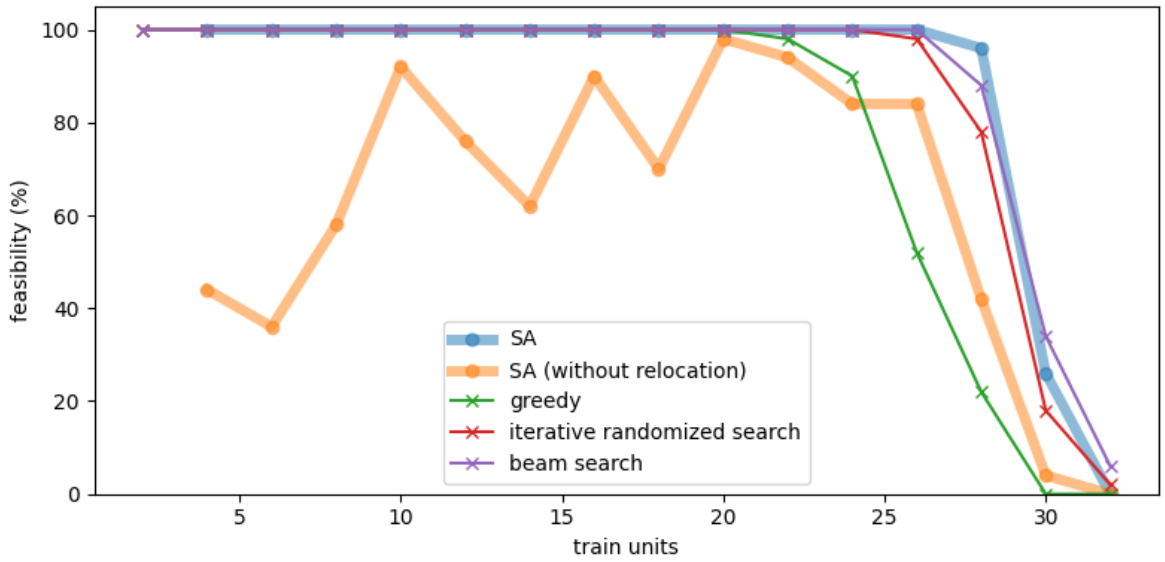

Figure C.4: Night shift without services

Night shift, including service tasks

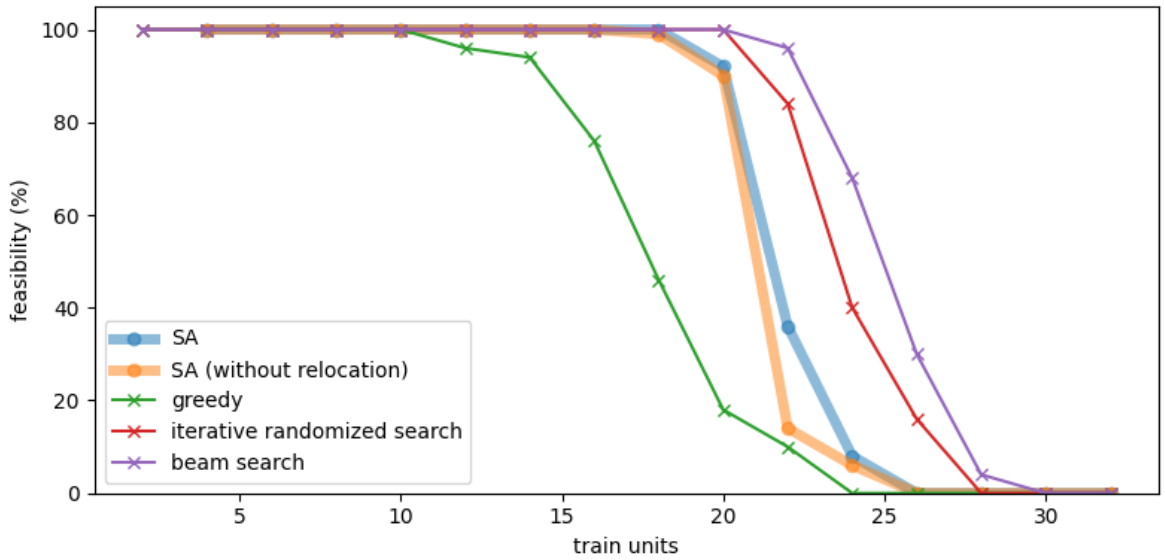

Figure C.5: Night shift including services

Full day, without service tasks

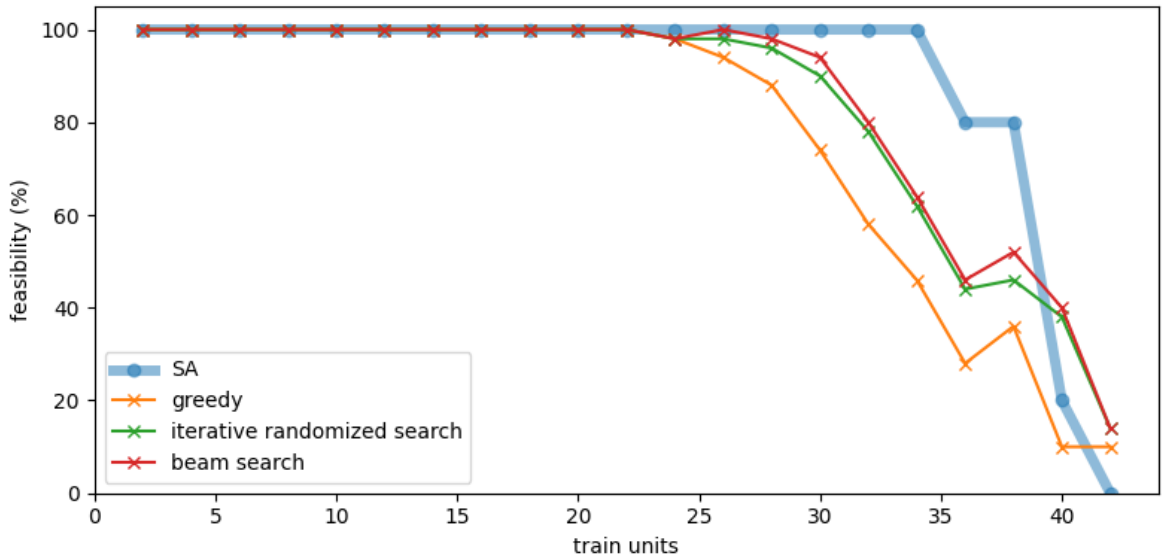

Figure C.6: Full 24 day planning, without services

Full day, including service tasks

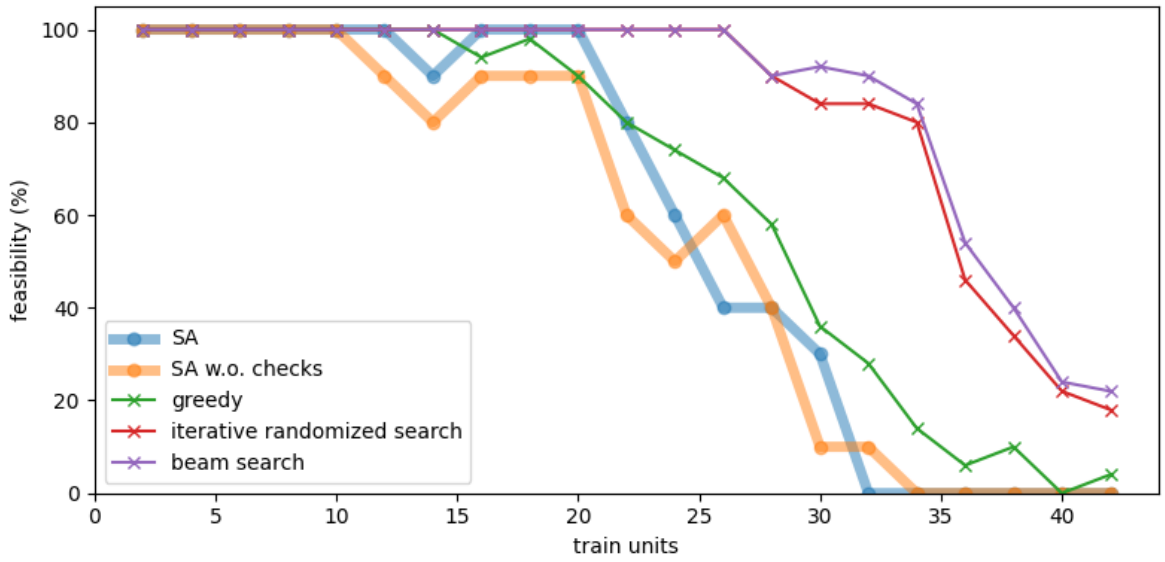

Figure C.7: Full 24 day planning, including services

C.6. Conclusions

The results indicate that that the search techniques are successful in finding feasible plans for the shunting problem. The Randomized Iterative Greedy Search (RIGS) already seems sufficient for most problem instances, although it does perform consistently slightly worse compared to the beam search method. It also has to be noted that very little computation time was used to find the solutions, meaning that the algorithms may find more solution if given more time. This is especially true for the beam search method, as it should be able to search more effectively compared to RIGS when given more time.