Applying the Pebble Motion problem: studying the feasibility of the Train Unit Shunting Problem

Issa Kalina Hanou

Master of Science Thesis

TU Delft Delft University of Technology

Applying the Pebble Motion problem: studying the feasibility of the Train Unit Shunting Problem

Master of Science Thesis

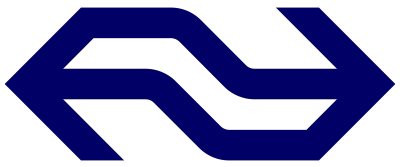
For the degree of Master of Science in Software Technology at Delft University of Technology

Issa Kalina Hanou

June 28, 2022

Thesis Advisor:	Dr. M.M. de Weerdt
Thesis Committee Member:	Dr.Ir. L.J.J. van Iersel
Thesis Daily Co-Supervisor:	Ir. J. Mulderij

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS) · Delft University of Technology



The work in this thesis was supported by the NS. Their cooperation is hereby gratefully acknowledged.

Abstract

Shunting yards are the locations where trains, which are not included in the train schedule at a certain time, are parked until they are required again. Managing the parking of the trains such that all trains can leave at the desired time is a complicated task, and results in the problem formally known as the Train Unit Shunting Problem (TUSP). This problem is an NP-hard problem, and current algorithms cannot always determine whether an instance is feasible. We analyze a simplified variant of the TUSP, leaving out details from the real-world scenario to study the theoretical conditions for basic scenarios to be feasible. To this extent, we identify essential elements of the TUSP and include these in a modification of the Pebble Motion problem. Based on this Pebble Motion variant, we establish new problems that can be studied to analyze the feasibility of the TUSP. For each of these problems, we examine the complexity and look into different solution approaches.

Preface

This thesis is submitted as the final step in obtaining my degree of Master of Science in Software Technology at the Delft University of Technology. I was engaged in researching and writing this thesis from November 2021 till June 2022.

Growing up in Haarlem, we lived close to one of the two oldest train stations in the Netherlands. The train was easily accessible and during my studies in Delft, I would be on the train a lot. In my computer science bachelor's program, my interest was quickly drawn to the Algorithmics section, so it was clear to me early on that I wanted to graduate in this area. When I was acquainted with the algorithmic research at the NS during my master's degree, I pursued this area for my thesis and undertook an internship at NS for the duration of my research.

Many years of research have been dedicated to scheduling problems in traffic, and more specifically to railway scheduling. This work builds upon such research, to assist in better scheduling approaches to relieve work from human planners as their work continues to grow with more and more train passengers every day. A basic level of graph theory is assumed of the readers of this thesis.

I would like to thank the team at the NS for the brainstorming sessions to help out when I was stuck and for their guidance to keep my work relatable to the real-world scenario. In particular, I want to thank Bob Huisman at NS for his ideas and thoughts on my work. Furthermore, I want to thank Jesse Mulderij for his continued support and weekly guidance, as well as numerous discussions about a problem or a proof. Finally, I want to express my gratitude to Mathijs de Weerdt for his excellent supervision and guidance during my thesis, as well as his support to prepare for a future career in research.

Issa Hanou

June 2022

Contents

Lists

List of Theorems and Definitions

List of Figures

List of Problems

List of Linear Programs

List of Methods

List of Tables

List of Algorithms

Chapter 1

Introduction

The first railroad in the Netherlands was built in 1839 and connected the cities of Haarlem and Amsterdam [17]. As the Dutch railroad network expanded, more trains were scheduled and the problem arose of matching the scheduled rolling stock to the passenger demand. Outside peak hours, rolling stock is transported to a shunting yard, where the train units are parked until scheduled for duty again. Besides parking, train units also need to undergo maintenance or cleaning tasks at specified locations within a shunting yard. The Dutch railroad network, operated by the *Nederlandse Spoorwegen*, Dutch Railways (NS), continued to grow and in 2019, 750.000 passengers traveled daily, contributing to 1.3 million train journeys per day [34]. Although this number shrank in the years following due to the Covid pandemic, it is expected to increase again and rise even further in the next couple of years, as the train offers a sustainable and reliable mode of transportation [42]. As passenger numbers rise, more rolling stock is needed to meet the passenger demand, and the problem of scheduling the parking and maintenance of the train units becomes increasingly difficult.

Shunting yards are often located near busy stations, mostly in urban areas in the Netherlands. Therefore, there is only limited space for rolling stock to maneuver, so it is important to use the space efficiently. Within a shunting yard, three components are necessary for operating a shunting yard: matching, parking, and routing. Rolling stock that arrives at the station consists of several train units, which are fixed compositions of train carriages depending on the type of train unit. Matching is the process of pairing incoming train units with scheduled outgoing train units. Train units are scheduled in compositions of one or more train units of the same type. The parking component is concerned with finding a track (part) where the train unit(s) can be parked. Finally, the routing process constructs a collision-free route through the shunting yard. The problem of finding a feasible plan to match, park, and route all train units within a shunting yard is the so-called Train Unit Shunting Problem (TUSP) [14].

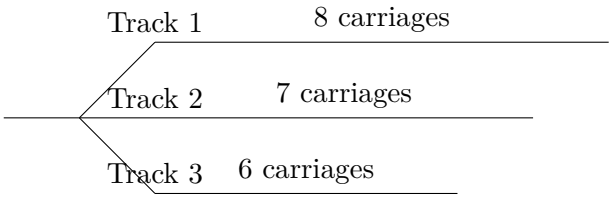
The TUSP can be extended to also include the service tasks that are performed at shunting yards. There are different types of service tasks that rolling stock might need, such as cleaning tasks, minor fixes, inspections, or larger maintenance operations. Within a shunting yard, there are often specific tracks designated for a certain task. Furthermore, not all types of service tasks can be performed at each shunting yard. When the TUSP is extended with the process of assigning train unit service tasks and including these in the shunting plan, it is referred to as the Train Unit Shunting and Servicing problem (TUSS) [28].

The components previously mentioned already shed some light on the complexity of the TUSP. Now, consider all the components that are included in the real-world TUSP. The times at which trains arrive at the shunting yard are determined by the timetables. These timetables are in turn based on the desired demand throughout the day. Once timetables are constructed, rolling stock is assigned, after which a crew can be scheduled to operate the timetables. Based on these timetables, the arriving and departing train unit compositions and times are determined for the shunting yard. Besides the matching, parking,

and routing, the service tasks, too, need to be scheduled in real-world scenarios. Furthermore, crews need to be assigned to operate the trains in the shunting yard as well as perform the service tasks. All these components relate to each other, and changing one component affects all the others, which adds to the complexity of the problem.

1-1 Feasibility and optimality

The feasibility of the TUSP is concerned with determining whether a solution exists in the first place. Once it is known that a certain scenario, referred to as an instance of the problem, is infeasible, then it is no longer necessary to find a solution, because a solution does not exist. We now give several scenarios and show their (in)feasibility. Here, we consider a simple shunting yard and ICM train units. An ICM train is a single-decked type of intercity train (see Figure 1-1b), which can consist of three or four carriages.



(a) Example shunting yard topology

(b) ICM train unit

Figure 1-1: Scenario with four trains

Example 1 Consider the shunting yard shown in Figure 1-1a and the scenario described in Table 1-1. Since the first departure is scheduled after the last arrival, all trains need to be parked at the same time in the shunting yard. However, there is not enough space to park all train units at the same time, so this instance is infeasible.

Train	Train composition	# Train carriages	Arrival time	Departure time
1	ICM-III, ICM-IV	7	12.08	13.52
2	ICM-IV, ICM-IV	8	12.14	13.57
3	ICM-IV	4	12.19	13.45
4	ICM-III	3	12.22	13.49

Table 1-1: Scenario with four trains

Example 2 In the previous example, the infeasibility of the scenario was very clear since there is not enough room to park all train units in the shunting yard. However, a scenario can also be infeasible if there is enough space to park all the trains but the departure times are not met. Consider for example the shunting yard in Figure 1-2 and the same train scenario from Table 1-1.

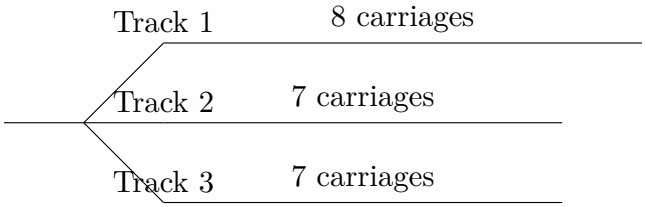


Figure 1-2: Example shunting yard topology

Although we can now park both *Train 3* and *Train 4* on *Track 3*, we know that *Train 4* is the last to arrive. However, *Train 3* has to depart before *Train 4* and the latter cannot move out of the way because *Track 1* and *Track 2* are both still occupied with trains that depart after *Train 3* and *Train 4*. Therefore, *Train 3* is blocked for departure and the instance is infeasible.

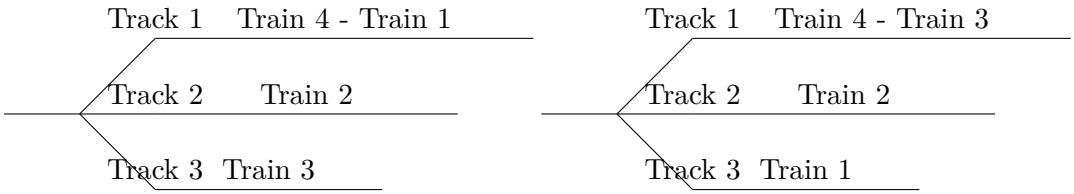
Besides the feasibility of a problem, the optimality of a problem is concerned with finding the best solution. Different measures of optimality exist and may lead to different solutions that are considered optimal. In terms of the TUSP, an optimal measure might be the number of moves that have to be performed in a shunting yard. Another measure could aim for a solution with minimally used space.

Example 3 Taking the same example shunting yard as in Figure 1-1a, consider the scenario described in Table 1-2. Now, it is clear that a solution exists, because the trains can all fit in the shunting yard. However, different solutions are possible.

Train	Train composition	# Train carriages	Arrival time	Departure time
1	ICM-III	3	12.08	13.45
2	ICM-III, ICM-IV	7	12.14	13.42
3	ICM-IV	4	12.19	13.57
4	ICM-IV	4	12.22	13.52

Table 1-2: Scenario with four trains

Consider the solutions presented in Figure 1-3. In Solution 1, *Train 1* is blocked by *Train 4*, but *Train 1* needs to depart earlier. So, *Train 4* would first have to move to *Track 2* after *Train 2* has left so that *Train 1* can depart from the shunting yard. On the other hand, in Solution 2 all trains can depart at the time they are supposed to, without being blocked by another train at that moment. We say that the number of moves is less in Solution 2, and concerning the number of moves, Solution 2 is optimal, because no solution with fewer moves can be found. The number of moves will always consist of moving into the shunting yard to park and moving out for departure. However, we prefer solutions where the number of moves in between is minimal because the maneuvers cost time and money.



(a) Solution 1 **(b)** Solution 2

Figure 1-3: Example shunting yard topology

Reallocation In Example 3, the optimality was considered in terms of the number of moves. The maneuvring of trains within a shunting yard, excluding the train's movement when entering or leaving the shunting yard, is referred to as reallocation. However, reallocation is not just interesting to consider for optimality results. In Example 2, reallocation could result in a feasible solution if *Train 1* or *Train 2* would depart before *Train 3* and *Train 4*. Although reallocation is thus an important aspect that can influence the feasibility, we do not consider reallocation in this thesis. While reallocation is often used in real-world scenarios, it is also very costly because the driver needs to move the train, and usually also turn it around, which requires the driver to walk to the other end of the train. So, although a scenario might be feasible with reallocation, it would also result in a longer solution with more required driving time, for example.

The feasibility is also more complicated when reallocation is considered. The turns that a train needs to make are often very tight, so a train needs to drive by a switch, so it can turn around and go in the

other direction. This was studied by Ahangar et al. [37] for a single train in a rail yard, who found that determining whether a certain angle can be traversed by a train is possible in polynomial time. They looked at the specific layout of a shunting yard, which was a general graph, and defined acute angles as switches where trains cannot make a turn right away but have to pass the switch and turn around afterward, as they are too long to make the turn at once. For example, in Figure 1-4, an acute angle is formed on the path $(a \to b \to c)$, and to go from a to c, a train must first pass b towards d and then turn around to pass over b to c. Therefore, there is an extra aspect to consider in the feasibility of a scenario when reallocation is allowed.

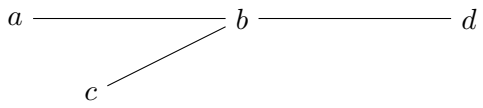


Figure 1-4: An acute angle going from a to b to c

In this thesis, we focus on simple scenarios and their feasibility, and we do not consider reallocation. However, reallocation can be an interesting extension of this work, which we discuss in the Future work.

1-2 Background

Studying the optimality of a problem can be beneficial for minimizing time and money. That is why most of the existing research has studied ways of optimally solving the TUSP. The first approaches considered the different components of the TUSP sequentially and, in doing so, they were able to produce optimal results [13]. However, the TUSP belongs to the complexity class $\mathcal{NP}$*-hard* [16], which is a class of problems that are notoriously difficult to solve; although when given a solution, it can be shown relatively quickly, in polynomial time, that the solution is correct. As solving an $\mathcal{NP}$*-hard* problem to optimality is known to require exponential running times [26], this is not feasible in practice because it takes too much time to find a solution. Therefore, further research has studied ways of finding suboptimal approaches for the TUSP [14].

Currently, the NS uses a local search algorithm to construct plans for their shunting yards [38]. These plans are based on the timetables for passenger trains, to which rolling stock is assigned. Given the arriving train units and the required departing train unit compositions and times, this algorithm matches the train units, assigns parking locations, and creates routes through the shunting yards. This local search method is a great improvement to the manual alternative which was employed before this algorithm and can provide very good, and sometimes even optimal, solutions. However, the major drawback of this approach is that the algorithm does not provide a complete approach to determine the feasibility of the problem. This results in the algorithm not always returning a solution, even when one does exist. Thus, an empty result does not tell us anything about the existence of a solution, because the algorithm may just not have been able to find it.

A problem of which the feasibility is widely studied is the Pebble Motion problem (PM). Given a graph with n nodes, a set of $k \leq n-1$ pebbles, and two configurations of pebbles over the nodes of the graph, the question is whether a plan exists that moves the pebbles from the initial to the goal configuration. An algorithm exists that can determine in linear time whether a solution is possible [10]. This important finding has been used in an approach to studying the feasibility of the Multi-Agent Path Finding (MAPF) problem, which is relatively similar to the PM as it considers a set of agents instead of pebbles, and the question is whether all agents can reach their goal locations. This feasibility approach extended the PM to solve a relaxation of the MAPF problem [20]. The MAPF problem is closely related to the routing component of the TUSP, which was the motivation for a study on whether a relaxation of the TUSS from an extension of the MAPF problem exists [36]. In operations research, a relaxation of a problem can relax either the objective or the constraints of a problem. A constraint relaxation of a model $\mathcal{P}$ is a model $\mathcal{P}'$ with the same objective, such that all feasible solutions for $\mathcal{P}$ are also feasible solutions for $\mathcal{P}'$ [9, Ch 12.2]. Another use of the linear-time feasibility of the PM was proposed for the robot path

planning problem, which is closely related to the MAPF problem, where an abstraction to a class of bi-connected graphs with at least two unoccupied nodes was considered [19]. This study resulted in a polynomial-time algorithm for the feasibility of bi-connected graph class of problems.

Although no feasibility algorithm for the TUSP exists, we could take note of the PM algorithm. However, there are some major differences between these two problems that prevent the direct use of the PM feasibility algorithm for the TUSP. First, the pebbles in the PM are already present on nodes in the graph according to the initial configuration. Yet, the TUSP also includes the routing of trains into and out of the shunting yard. Furthermore, while trains and tracks have a physical length, pebbles are regarded as unit size such that a pebble can fit on a node in the graphs. Other differences include the need for matching train units and the notion of absolute time, both of which are considered in the TUSP, though not in the PM. Finally, although the PM algorithm runs in polynomial time, its solutions are often very large and require many reallocation moves. Therefore, we cannot directly apply this algorithm to a relaxation of the TUSP. Turning back on our discussion about Reallocation, we see again that scenarios where reallocation is the key to a feasible solution result in many extra moves and thus lead to very costly and impracticable solutions.

1-3 Problem statement

This research aims to use insights from the PM to work towards an approach for determining the feasibility of the TUSP. The design of an extension to the PM as a relaxation of the TUSP can be a first step towards designing a feasibility algorithm of the TUSP based on the linear-time feasibility algorithm for the PM. With such an algorithm, a certain scenario can quickly be checked to determine its feasibility, and upon a positive answer, a solution could be generated.

Trees The use of a relaxation allows us to consider the most important aspects of the TUSP, like the length of train units, while also relaxing less limiting factors, like the notion of absolute time. In this thesis, we consider an abstraction of the problem where the underlying graph of a shunting yard can be represented by a tree. There are two main types of shunting yards used by the NS, carousel and shuffleboard layout, of which the latter has a tree-like structure [31] (for more information see Appendix A). Since the layouts differ considerably, we focus only on one of them, namely the tree-like shuffleboard. Furthermore, most research on the PM so far has considered general graphs. However, trees are a simplified case of a general graph structure and form a recursive structure, which can be used to build up small cases into bigger and more general scenarios, while any graph can be transformed into a tree [21]. Moreover, trees do not contain any cycles within the graph, so for any two nodes in the graph, there is only one path possible to move from one to the other node. So, focusing solely on trees provides an approach that can be extended to general graphs later on.

1-3-1 Research question

We now formalize this problem statement into a research question.

> **How can insights into the Pebble Motion problem be used for determining the feasibility of the Train Unit Shunting Problem on trees?**

1-3-2 Research sub-questions

The following sub-questions are aimed to model different components of the TUSP that are necessary extensions of the PM in the design of a TUSP relaxation. These four questions form the base of the four main chapters of this thesis.

1. What are the differences between the PM and the TUSP that are essential for a variant

that is useful for deciding real-world scenarios?

2. What are the implications to the TUSP feasibility when extending the PM with arrival and departure sequences of pebbles?

3. What is the influence on the complexity of the PM variant when including train and track lengths?

4. How can the PM variant be extended to include the TUSP matching subproblem?

1-4 Outline

To answer these questions, we study relevant literature on the TUSP and related problems in Chapter 2. The main differences between the TUSP and the PM are defined in Chapter 3. Then, we look at the implications on the TUSP feasibility of the arrival and departure sequences model in Chapter 4. Afterward, the lengths of trains and tracks are considered, and the PM model is extended with these in Chapter 5. Next, we study the matching component of the TUSP and include this in the model in Chapter 6. Finally, we conclude with the contributions of this research and discuss future research directions in Chapter 7.

Chapter 2

Literature review

In this chapter, we study the literature relevant to this research. First, some background information on algorithmic computation time and problem complexity is discussed. Then, the literature on the Pebble Motion problem (PM) is examined; how it originated, and several algorithms that have been implemented since. Thereafter, we consider the Train Unit Shunting Problem (TUSP) and the different approaches that have been implemented so far. As mentioned in the previous chapter, shunting yards in the Netherlands are often located in urban areas, and the Dutch railroad network is very dense. A lot of existing research on the TUSP has mostly studied the railroad network in the Netherlands, as well as Denmark, since these countries have shunting yards that offer the biggest challenges concerning the available space in the dense network.

2-1 Background

Despite the difference between the feasibility and optimality of a problem, the computation time of an algorithm is important in either case. If an algorithm for solving the TUSP takes a very long time to compute, it may not be feasible for a real-world application. Therefore, it is important to find the bounds of the computation time for any approach. We use the big-Oh notation, $O(n)$, which defines an upper bound on the runtime in terms of the input size n and a possible constant k. This function of n then defines the scale of the computation time, which can be linear $O(n)$, polynomial $O(n^k)$, or exponential $O(k^n)$, for example, although we strive to avoid the latter. The input size n for the TUSP is defined by the number of arriving train units. For example, the NS shunting yard *Kleine Binckhorst* in The Hague might process about 28 train units in a day [31]. Therefore, an exponential computation time with an exponent of 28 would result in a very long computation time, which is infeasible in practice. Besides the big-Oh notation, we also use the big-Omega, $\Omega(n)$, which defines a lower bound, and the big-Theta, $\Theta(n)$, which gives a tight bound, i.e. when the upper and lower bound are the same.

The complexity of different problems is often distinguished using complexity classes, based on the computation time necessary for solving the problem [26]. The main distinction is between problems that can be solved in polynomial time $O(n^k)$, which we consider to be in the class $\mathcal{P}$, and the problems which require an exponential computation time $O(k^n)$, which we consider to be in the class $\mathcal{NP}$-*hard*. Another important class is the class $\mathcal{NP}$. The problems in this class may or may not be solvable in polynomial time. However, given an instance of a problem, it is possible to check whether this instance is a solution in polynomial time. Finally, there is the class $\mathcal{NP}$-*complete*, which is the set of classes that are polynomially verifiable, though also not polynomially solvable. The relation between these complexity classes is illustrated in Figure 2-1, although more classes exist that are not considered in this thesis.

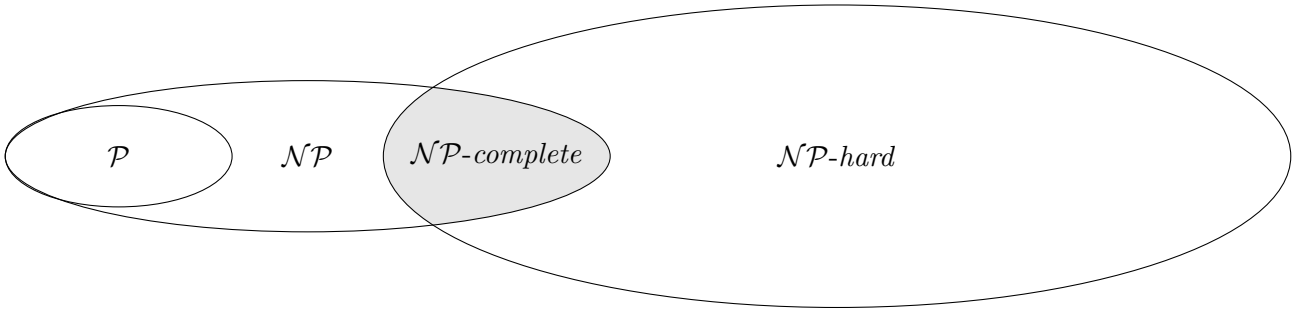


Figure 2-1: The relation between complexity classes

Knowing that a problem is $\mathcal{NP}$-*hard* indicates that trying to solve the problem quickly will be very difficult and time-consuming. Therefore, it is important to know the complexity of a problem when trying to find a good solution approach. Reductions are a method for showing that a problem is in a certain complexity class. If a known $\mathcal{NP}$-*hard* problem can be reduced to a new problem in polynomial time, this is proof that the new problem is also in the $\mathcal{NP}$-*hard* complexity class [26]. However, when we know that a problem is $\mathcal{NP}$-*hard*, this tells us that trying to solve this problem using an exact approach is probably not the most efficient way of tackling the problem. Therefore, looking into special cases of the problem might provide new insights into a solution approach. For example, many $\mathcal{NP}$-*hard* graph problems can be solved efficiently if the underlying graph is a tree [15].

2-2 Problem of Pebble Motion

The PM on a general graph was first described by Kornhauser et al. [4] in 1984, as a generalization of the 15-Puzzle designed by Sam Loyd. This puzzle was formally described for the first time in 1974 by Wilson [1], who looked at the problem from a group theory perspective. Kornhauser et al. looked at a graph as a tree of bi-connected components, which offers a useful structure for moving pebbles. They presented a decision algorithm for instances with fewer pebbles than nodes in the graph, where the initial and goal configurations consist of the same nodes. Their approach divided the problem into transitive subproblems, to determine whether two configurations of pebbles were reachable from each other within that subproblem. This reduced the problem of determining the feasibility of the configurations to determining the feasibility of the transitive subproblems. One advantage of this approach is that concluding one subproblem is infeasible, immediately infers the complete instance to be infeasible. This approach resulted in an efficient algorithm for determining the feasibility of the Pebble Motion problem on a Graph (PMG).

The problem presented by Kornhauser et al. looked at several pebbles in the graph that influenced each other's movements. Papadimitriou et al. [7] studied the problem of Graph Motion Planning with 1 Robot (GMP1R) and referred to the problem by Kornhauser et al. as Graph Motion Planning with k Robots (GMPkR). Papadimitriou et al. showed that deciding whether a solution of length k for the GMP1R problem exists, is $\mathcal{NP}$-*complete*. However, when considering the problem on weighted trees, where a positive weight is associated with each edge, the problem was found to be polynomially solvable. Furthermore, Papadimitriou et al. also gave an approximation algorithm for general graphs. Although this problem only includes one robot, other obstacles are included in the problem. This led to the definition of a hole, a node in the graph on which no obstacle is parked (including the node with the robot). Holes are then used to indicate the nodes to which a pebble can move, to define whether a feasible path exists. Kornhauser et al. showed that their algorithm runs with $\Theta(n^3)$ number of moves and although they did not give the runtime complexity exactly, it is at least $\Omega(n^6)$, which is the number of subproblems to solve.

Auletta et al. [10] improved on the bound of $\Omega(n^6)$ by providing a linear-time algorithm for determining the feasibility of the pebble motion problem on trees. Their algorithm first reduced the PMG to the Pebble Permutation problem on a Tree (PPT), as the algorithm by Kornhauser et al. used the assumption of the initial and goal configuration to consider the same nodes. Then, the permutation is expressed as a sequence of exchanges. Finally, the exchanges are marked to be feasible or not, with the

use of equivalence classes. They found that their algorithm resulted in plans of length $O(k^2(n-k))$ moves, where k is the number of pebbles and n is the number of nodes. This is a fairly high upper bound and has been reviewed in the literature as too high for practical solutions [21]. Although Auletta et al. noted that the problem of finding the shortest plan for $k = n - 1$ pebbles is $\mathcal{NP}$-*complete*, some plan of length $O(n)$ can be computed in $O(n)$ time for any feasible instance of the Pebble Motion problem on a Tree (PMT). So, their algorithm runs in $O(n)$ time but results in plans of $O(n^3)$ (with a lower bound of $\Omega(n^2)$) moves. Such solutions are, however, considered too long in a real-world application, because moving each pebble up to n times costs a lot of time. Although the authors identified the special case where $k = n - 1$ and the plan has $O(n)$ moves, there are many real-world scenarios where this condition of k is not met.

Goraly and Hassin [21] developed an algorithm that continues on the equivalence classes defined by Auletta et al. They give each pebble a color, where the colors match the equivalence classes, and pebbles of the same color are thus said to be equivalent and can be interchanged. Then, the final pebble configuration is based on the color of the pebbles and no longer includes specific pebbles that must reach a certain node. Goraly and Hassin referred to the problem of Pebble Motion with m colors as the m-color Pebble Motion problem (m-PM). Furthermore, they defined transshipment nodes as nodes on which pebbles cannot park, which help in the transformation of a graph into a tree. First, they showed that the m-color Pebble Motion problem on a Tree (m-PMT) can be solved in linear time. Then, they used the work by Auletta et al. on a tree with these transshipment nodes. These nodes are used in the linear-time reduction of a graph to a tree, which is used to determine the feasibility of the m-PM on general graphs. Thus, this problem was also shown to be solvable in $O(n)$ time, although no bounds were given on the number of required moves.

2-2-1 Multi-Agent Path Finding

Surynek [20] used the PM to create a suboptimal approach for solving the Multi-robot Path Planning (MPP) problem. Given a graph with n nodes; k robots; an initial arrangement of robots in the graph; and a goal arrangement of robots in the graph; the MPP problem finds a plan that moves the robots from the initial to their goal arrangement such that no two robots collide. The problems of MPP and PM are thus very similar, and the main distinction is the level of allowed parallel moves [20]. Robots are allowed to move to nodes where another robot is departing in the same time step, while pebbles can only move to nodes that are not occupied in the previous time step. The work by Surynek was the first to relate the MPP to the PM and improved on existing MPP algorithms in terms of runtime and solution length by integrating knowledge from earlier work on the PM.

In a different study, Surynek [19] also studied the MPP problem, though he only considered an abstraction where the underlying graph can be represented by a bi-connected graph with two empty nodes. In a bi-connected graph, each node is connected to two other nodes, often forming a grid-like structure, such that there are always two different paths between any two nodes in the graph. This abstraction allowed Surynek to construct the first polynomial-time algorithm for deciding the feasibility of the MPP problem. However, the approach is limited as it considers only the class of problems where the graph is bi-connected, and there are at least two more nodes than robots.

In comparison to the MPP problem, in which multiple robots are controlled as a single entity, in the Multi-Agent Path Finding (MAPF) problem each agent moves as an autonomous entity through the space [20]. Both problems have many applications in, for example, robotics, traffic control, and video games. Krontiris et al. [25] developed a path planning algorithm for MAPF on general graphs, which was based on Auletta et al.'s linear-time decision algorithm. The planning algorithm used a graph-to-tree conversion as proposed by Goraly and Hassin [21], which resulted in a tree of bi-connected components. This structure is then used to plan the optimal paths for the pebbles. Although this path planning algorithm provided optimal results, Krontiris et al. found that the optimal path planner resulted in solutions of lower quality than existing suboptimal planners, as they were too long for real-world use.

The suboptimal MAPF planner that Krontiris et al. compared their results to, was the Push & Swap algorithm for MPP by Luna and Bekris [22]. The algorithm consists of push operations, which force robots to clear a specified path for another robot to get to its goal, and swap operations, as robots may be required to switch positions along their shortest paths. A swap can affect all agents, and if a swap is impossible, then a solution does not exist. Although this algorithm is not based on the PM, the approach does use a similar equivalence reasoning.

De Wilde et al. [27] showed the algorithm by Luna and Bekris to be incomplete and proposed the MAPF Push & Rotate algorithm as a complete extension to Push & Swap. The rotate operation avoids recursive Swap operations by detecting cycles of agents who all want to move and replacing these Swap operations with a rotation. In their approach, they take note of the theoretical results from Kornhauser et al. [4], accounting for the conditions of a transitive subproblem to detect infeasible swap operations. Their approach is complete, yet it was found not to be able to remain within the $O(n^3)$ bound on the number of moves, as was achieved by Surynek [20]. Furthermore, their approach is not optimal, although heuristics were developed to improve the solution quality and a post-processing step was given to eliminate unnecessary moves.

Later on, Kulich et al. [33] developed an algorithm called Push, Stop & Replan, which further adapts the previously mentioned algorithms for MAPF [22, 27]. The authors considered the non-simultaneous movement restriction to be the main shortcoming of the earlier algorithms, as it resulted in more time steps necessary for the completion of a plan, which was regarded to be ineffective. The Push, Stop & Replan algorithm considers agents of different types and does allow same-type agents to move simultaneously in the graph. There are many applications of MAPF that benefit from the ability to interchange agents of the same type, like the different types of robots within a warehouse [33]. When it does not matter which agent of a certain type reaches a specified goal, more solutions are feasible and it might thus be easier to find such a solution, although the solution space is also larger.

2-2-2 Remarks

The PM has been widely studied and is decidable in linear time. However, it is often not directly applied to real-world practices because the solutions that are produced often consist of many moves which is an undesirable outcome. The problem has been used to study the MAPF problem, which knows many applications to real-world problems, like warehouse automation and traffic control. Still, the plans that are produced result in many moves. Although the TUSP has similarities with the MAPF, these algorithms are not directly applicable due to the costly solutions they produce. The main benefit of the PM perspective is the abstractness of the problem, which allows us to study simple scenarios in detail. Therefore, this thesis does not apply the studies mentioned here directly but rather uses their insights for the TUSP.

2-3 Train Unit Shunting Problem

The study of optimizing solutions to train routing problems has only arisen in the last few decades [8]. Before, most of the planning was done manually. One of the first surveys on optimization models for train routing and scheduling problems was produced by Cordeau et al. [8].

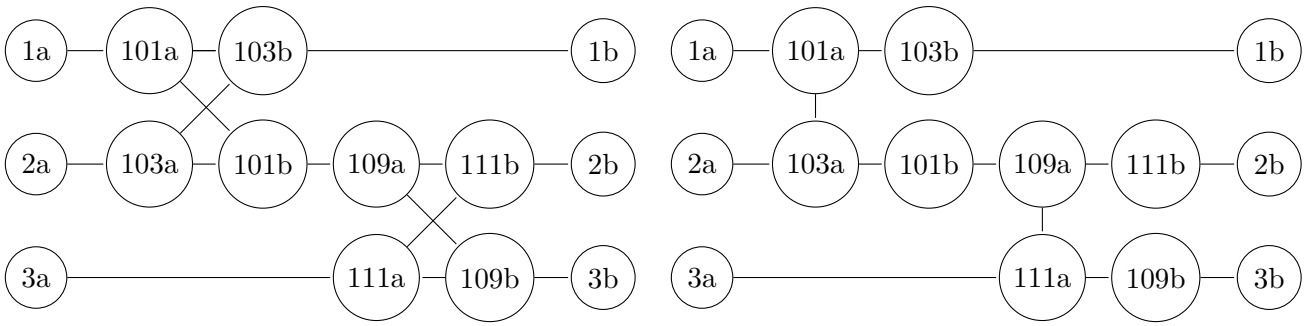
The TUSP was first introduced by Freling et al. [14], who distinguished the subproblems of matching and parking. Their approach was the first to allow tracks to be entered from two sides. They designed a mathematical model using integer linear programming for solving the problem. Their approach first tries to minimize the number of train units that have to be coupled and/or decoupled, and the resulting train units of the matching procedure are assigned a shunt track to park on. By considering the two subproblems sequentially, the possibility of finding a globally optimal solution was lost. However, the goal of this research was to find a method to speed up the manual scheduling approach. They compared their formulated Integer Linear Program (ILP) to its relaxed Linear Program (LP) variant, where the

constraint to prevent train compositions of different types was eliminated. They observed that the gap between the ILP and LP solutions was small enough to suggest their approach was close to optimal, and the gap was found to be acceptable in practice. So, they concluded that their formulated relaxation was an appropriate solution approach.

A model that further extended this approach was proposed by Kroon et al. [18] who solved the subproblems for matching and parking in an integrated manner. After noting that certain scenarios were not solvable using the approach from Freling et al. [14], they studied the possibilities to allow more details from practice. Using this integrated method, they were able to improve the computation time while producing solutions of acceptable quality for shunt planners.

Another study on the difference between an integrated and a sequential approach to solving the different subproblems of the Train Unit Shunting and Servicing problem (TUSS) was done by Kamenga et al. [39]. So, besides matching and parking, routing and service scheduling were also considered. The sequential algorithm they used took a sub-solution from one subproblem as a fixed initial solution for the next subproblem. They found that different sequences of subproblems resulted in different runtimes and that the parking subproblem had the most influence on the runtime of the algorithm. Moreover, their approach to solving the matching subproblem regarded trains as sets of train units. They defined a set of arriving, intermediate, and departing trains to model the coupling and decoupling operations. By assigning non-zero costs to the matching of train units that require either coupling or decoupling, these actions are minimized, as they always require more work than keeping the initial train compositions intact. Finally, they concluded that their solution approach was successful but does depend on the solutions provided by the subproblems tackled first, and thus the characteristics of the instances had a large influence on the performance of their algorithm.

Based on the work by Freling et al., Lentink et al. [13] introduced the first model-based algorithmic approach for the TUSP. Their model considered several physical characteristics of the infrastructure of the shunting yard, so these could be used for solving the problem realistically. One of these characteristics considers the switches, which allow operators to determine which track to route a train to. They noted that to detect all possible conflicts, a switch between two tracks must always have only one connecting edge. For example, in Figure 2-2a a plan that simultaneously moves train A on the path (2a $\rightarrow$ 103a $\rightarrow$ 103b $\rightarrow$ 1b) and train B on the path (1a $\rightarrow$ 101a $\rightarrow$ 101b $\rightarrow$ 2b) is considered feasible, although it results in a conflict. However, the layout in Figure 2-2b does not consider this plan to be feasible, and thus prevents this conflict. The inclusion of nodes 103b and 101b in Figure 2-2b is used to determine the reservation period used to signal whether a train has passed the switch element. As the crossing edges in Figure 2-2a should never be used at the same time to prevent such conflicts, no solutions are lost by eliminating these edges, as any path in the graph is still possible in Figure 2-2b. An important notion of this design is that by avoiding the conflict, it also transforms the graph into a tree, by eliminating cycles.



(a) Graph representation of conflicting switches **(b)** Graph representation of non-conflicting switches

Figure 2-2: Modelling the switches in a shunting yard in a graph

Wolfhagen [30] worked on the routing of trains through a shunting yard with the option of reallocation. In this case, a train that is blocking another train could be reallocated to a different position, so the

blocked train can depart from the shunting yard. An example of this was shown in Figure 1-3a in the Introduction. Wolfhagen tried using an exact method, which proved to be very challenging, and a heuristic was developed to offer planning support. However, the exact method was outperformed by the existing algorithms used by the NS. Specifically, not all instances could be solved within the allowed computation time, and the problem sizes of about 20 or more train units yielded solutions that were too long for use in practice. So, the inclusion of reallocation in an exact approach was found to be difficult to solve, especially for larger instances.

2-3-1 Remarks

Although several elements of the TUSP have been extensively studied, most of the research focuses on how to solve the problem, without looking specifically at if the problem can be solved. Filling this research gap is the main objective of this thesis. Since we do not consider reallocation in this thesis, the routing subproblem is less interesting, and our main focus is the parking subproblem. As we saw that the parking subproblem has the most influence on the runtime, this is an interesting research area that we look into in this thesis.

2-3-2 Train Unit Shunting and Servicing problem

A local search method for finding feasible solutions to the TUSS was proposed by van den Broek [28] and integrated the four subproblems of the TUSS: matching, parking, routing, and service scheduling. The approach first finds a perfect matching between arriving and departing train units, otherwise, it is infeasible and the matching tries to minimize the cost of coupling and decoupling. Given the matching, the service task schedule is created and parking spaces are assigned to all train units, both before and after service tasks. This constitutes an initial solution, which is thereafter improved for the different components. The routing subproblem is not part of the initial solution as the approach assumes non-simultaneous parking because shunting yards are often operated by one driver at a time. Solutions are only said to be feasible if none of the posed constraints are violated. The local search approach was shown to outperform the previously-used mixed integer programming algorithm and was considered to produce useful initial solutions for real-world applications. However, it lacked practical details for generating full solutions.

The approach was extended by van den Broek et al. [38] to include more aspects of the TUSS and offer a more accurate algorithm for the problem. The combining and splitting of train unit compositions were added to the model. For testing, besides using generated instances, a 24-hour real-world scenario at the service station *Kleine Binckhorst* in The Hague was also used. In this scenario, 31 train units are considered which together compose up to 21 trains. The produced plan solutions from the local search algorithm were evaluated by planners from the NS who confirmed that they would work in practice. As such, it was shown to be the first method to solve real-world scenarios with components of the complete problem for shunting and service scheduling. However, some instances remained where the local search approach was unable to find a feasible solution within the time limit, although it could not rule out that such a solution might exist.

An exact approach for the sole planning of service tasks at a shunting yard with a service station was attempted by Huizingh [31], using an ILP. The approach was based on the job-shop and flexible flow-shop problems where each type of service task was modeled as a machine. A fixed sequence for the machines was used, and the tasks that were required for each train unit were set to be jobs for the machines. If a certain train unit did not need a specific task, it would still need to pass the location of this service task, but would not need any processing time, which is undesirable in practice. Although it was found that it is possible to plan the service tasks using an exact approach, it was also observed that for these results to be useful in practice, they would need to be integrated with the other subproblems of the TUSP, which further adds to the complexity of the problem.

Finally, Mulderij et al. [36] studied the TUSS and used a MAPF perspective. They defined a variant of the MAPF problem as a relaxation of the TUSS, by modifying the first to resemble the latter. Although the MAPF problem has similarities with the routing subproblem of the TUSS, it is not a relaxation of the latter by itself. Therefore, the MAPF formulation was extended with a matching component. Furthermore, a restriction on the movement of the trains is implemented in the graph's topology, to extend the MAPF problem. The final main extension to the MAPF problem is the inclusion of the service process, which is an essential component of the TUSS problem. However, as the algorithms for solving the MAPF problem required notions from the PM for completeness [27], the relaxation of the TUSS might still benefit from notions of the PM.

2-4 Literature review conclusions

Concluding this chapter, existing work on the TUSP and TUSS has focused mostly on suboptimal solving algorithms. To the best of our knowledge, no works have been published on the relation between the PM and TUSP, and our approach to modeling the arrival and departure sequences is new. The PM has been widely studied, although most of the work has focused mainly on general graph structures. There exists a linear-time algorithm for determining the feasibility [10], and several algorithms for suboptimally solving this problem [22, 27, 33]. The problem of pebble motion has already been extended for modeling the MAPF problem [20]. The most important remark from this literature study is that no existing work has been published so far on the feasibility of the TUSP. Furthermore, the focus on trees instead of general graphs can provide new insights, as we also discussed in Section 1-3 on our choice of trees. This decision was further strengthened by the result from Lentink et al. [13], who transformed switches in the graph topology such that the resulting structure is more tree-like and infeasible crossings are eliminated. A study of the relaxation of the TUSS from an extension of the MAPF [36] shows a similar approach to what we study in this thesis.

Chapter 3

From Pebble Motion to the Train Unit Shunting Problem

The Train Unit Shunting Problem (TUSP) is a complex problem for which currently no feasibility approach exists. In this thesis, we take note of the Pebble Motion problem (PM), for which such an approach does exist. As we studied in Chapter 2, several variants of the original linear-time feasibility algorithm for Pebble Motion on Trees by Auletta et al. [10] have been proposed. In this chapter, we state the formal problem definitions and discuss the differences between the PM and the TUSP. These differences are the necessary elements that must be included in the PM variant that we study in this thesis to model the TUSP.

3-1 Problem definition

We start with a formal definition of the Pebble Motion problem on a Tree (PMT) [10].

Pebble Motion on Trees problem (PMT)

Input: $I = (T, P, C, G)$: given is a tree $T = (V, E)$; a set P of $n < |V|$ pebbles; an initial configuration C of these pebbles; and a goal configuration G.

Question: Is there a sequence of moves, which each transfers a pebble from its current position $v \in V$ to an adjacent unoccupied vertex $v' \in V \ni \{v, v'\} \in E$, to move the pebbles from C to G?

Next, we give the formal definition of the Train Unit Shunting Problem on Trees (TUSP). This is the decision variant of the TUSP that is mostly studied in the literature. It is based on the optimization variant as formulated by Freling et al. [14] but without the costs associated with the shunting tasks that were used in the original formulation. We omit the original optimization variant since we are only interested in the feasibility study of the TUSP. Besides the omission of the costs, in this thesis, we only look at the on-site problem, between the arrival and departure of trains at the shunting yard, and leave out any information about tracks at the railway stations. Finally, we already discussed our choice to focus only on trees in Section 1-3. In the rest of this thesis, we refer to the problem below simply as the TUSP.

Train Unit Shunting Problem on Trees (TUSP)

Input: $I = (T, U, a, d, \kappa)$: given is a tree $T = (V, E)$ which is the topological representation of a shunting yard; a set of n trains U with the train unit composition of each train; the arrival time $a(t)$ of a train $t \in U$ at the shunting yard; the departure time $d(t)$ of a train $t \in U$ at the shunting yard; and the capacity $\kappa(v)$ of a track $v \in V$.

Question: Is there a matching between the arriving and departing train units and a parking plan at the shunting tracks, such that neither i) crossings nor ii) type mismatches in the train unit matching occur, and iii) the capacity of a shunt track is never exceeded?

3-2 Problem differences

Comparing the definitions of the PMT and the TUSP, we note the following differences.

1. Pebbles exist in the tree, while trains arrive and depart at the shunting yard according to the timetable.
2. Nodes provide space to fit exactly one pebble, while shunting tracks have a capacity that cannot be exceeded.
3. Pebbles are unique, while trains consist of train unit compositions that belong to a certain type of train.

For each of these differences, we explain the necessity of including the element in a variant of the PM, because it is a crucial element of the TUSP. First, we discuss the arrival and departure of trains at the shunting yard. The relaxation of the TUSP must include this arrival and departure behavior, otherwise, solutions to the relaxation could be considered feasible while in reality the arrival or departure of a train could implicate a feasible solution. The timetable determines the arrival and departure actions in hours, minutes, and maybe even seconds. However, there is no notion of time considered in the PM. The most important aspect of the arriving and departing trains is the order in which they do so, such that an arriving train is not crossing paths with a departing train (condition i of the TUSP). Therefore, we use arriving and departing sequences, where trains have a relative order, but no absolute time is associated with the arrival and departure actions.

Furthermore, the length of trains and shunting tracks is an important aspect that should be included in the relaxation of the TUSP. The length of a train differs per train type and is usually expressed in the number of train carriages that are included in the train unit, while shunting tracks have a fixed length that can only fit a predefined number of train carriages. As the capacity of the shunting tracks may not be exceeded, this is a necessary component for the relaxation to ensure feasible solutions to the TUSP (condition iii).

Finally, we consider the matching of train units of the arriving train compositions to the departing train compositions. Because the NS uses different types of trains, a solution to the TUSP must provide a train unit of the correct type on departure. We propose to use the m-color Pebble Motion problem on a Tree (m-PMT) model proposed by Goraly and Hassin [21], which is defined for general graphs. Consider a scenario where there are m different types of train units. The m-PMT model uses a color configuration instead of a pebble configuration, and thus answers whether a plan exists that moves the pebbles such that the color arrangement is satisfied. Using an extension of the PM to the m-PMT in the relaxation of the TUSP will thus satisfy the matching constraint (condition ii of TUSP).

In the rest of this thesis, we study these three differences mentioned here. In each chapter, we introduce a new variant of the PMT that adds another property to increase the resemblance to the TUSP. However, the three components that we discussed here are added in turn, and we start with the initial PMT.

Chapter 4

Influence of the arrival and departure sequences

In this chapter, we add the arrival and departure sequences to the tree topology of the Pebble Motion problem on a Tree (PMT). Based on this variant of the Train Unit Shunting Problem on Trees (TUSP), we study the implications of the feasibility of the problem. As mentioned in the previous chapter, we do not yet consider the length of train units or shunting tracks, which are discussed in Chapter 5, nor the matching of train units, discussed in Chapter 6. So, in this chapter, we assume all trains to consist of only one train unit with a length of one, such that nodes in a tree, which represent a parking space on a shunting track, can fit exactly one train unit. Furthermore, we assume every train to be of a unique type, so there is no matching required because the incoming and outgoing trains need to match exactly.

Arriving and departing sequences The trains that enter a shunting yard arrive in a sequence and are modeled by pebbles. We say there are n pebbles, that arrive in a sequence $S = (s_1, .., s_n)$ and depart in a sequence $D = (d_1, .., d_n)$. The arriving sequence is defined as the first to the last arrival, while the departing sequence is defined as the last departure to the first departure. More formally, for any arriving or departing action a, if $t(a)$ returns the timestamp related to this action, then S and D are defined by $t(s_1) < t(s_2) < .. < t(s_n)$ and $t(d_1) > t(d_2) > .. > t(d_n)$. This ensures that when $S = D$, the order will be completely Last In, First Out (LIFO). These sequences define the order in which trains arrive and depart. So, for any arriving (respectively, departing) sequence position, s_i (respectively, d_i) returns the pebble that arrives (respectively, departs) at position i in the sequence. For any pebble p, the arriving (respectively, departing) sequence position can be retrieved by $S(p)$ (respectively, $D(p)$).

The new model We can represent a shunting yard with a tree T because we consider only shunting yards that can be represented by trees. To model the arrival and departure sequences, we add a simple path $L = (v_1, .., v_n)$ to T. This path is connected to the root node t_0 of T with the edge $e_T = \{t_0, v_1\}$ such that removing this edge e_T would disconnect T from L. Finally, L should be of length n so all n trains can fit on this path, which models a track. From a train perspective, we now have the graph $G = (T \cup L \cup e_T)$ where trains can arrive in the shunting yard represented by T by using path L. Once they have parked in a node of T, they can be rerouted again to L to compose the departure sequence. The PMT is defined over a tree, so from a pebble perspective, the initial configuration is formed on path L and the goal configuration is also formed on path L, so we say the goal configuration is a permutation of the initial configuration. The tree T can then be used by the pebbles to produce this permutation. We summarize the construction of this graph in Definition 4-0.1 and provide a visualization in Figure 4-1.

Definition 4-0.1 (Shunting tree) A **shunting tree** $T_L = \{T \cup L \cup e_T\}$ consists of the tree T, rooted by node t_0, which is connected to the path $L = (v_1, .., v_n)$ through the edge $e_T = \{t_0, v_1\}$, such that removing this edge e_T would disconnect T from L.

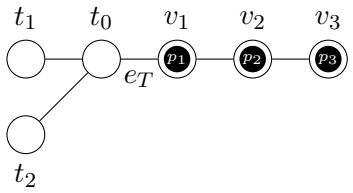


Figure 4-1: Example of a shunting yard tree T and a path L

As the tree T represents the shunting yard topology, we add one restriction to this model. The nodes of L cannot be used by the pebbles except when forming the initial and goal configurations. This way, we ensure that the permutation that must be reached is possible using only the nodes of the actual shunting yard. This constraint is formalized in Property 4-0.1.

Property 4-0.1 (Parking) $\forall v \in L$: v cannot be used as a parking space.

We start with several assumptions that are used throughout this chapter. The first two assumptions are based on how the path of L can be used in the feasibility study. Assumption 4-0.3 restricts the theory in this chapter to consider only the parking of trains upon their arrival, without routing trains to different sections of the shunting yard. Reallocation is a term commonly used in Train Unit Shunting Problem (TUSP) literature, but we do not consider it in this thesis, as mentioned in the introduction. The last two assumptions regarding the other problem differences, mentioned in Section 3-2, are considered in the next chapters: the lengths of trains and tracks are discussed in Chapter 5 and the matching problem is addressed in Chapter 6.

Assumption 4-0.1 (One track for arriving and departing) There is only one track that leads into the shunting yard where trains can arrive or depart. This means there is only one path L in the tree G.

Assumption 4-0.2 (Non-mixed traffic) The traffic is assumed to be non-mixed. This means that first, all trains arrive in the shunting yard before the first departure is scheduled again. This implies that the shunting yard must have enough parking space for all n trains.

Assumption 4-0.3 (No reallocation) The trains arrive in the shunting yard, where they park on their assigned track, and afterward leave again straight from their parking location, so there is no reallocation.

Assumption 4-0.4 (Unit lengths) All trains consist of one train unit, with a length of one train carriage. So, all trains have the same length. The nodes in the tree T used for parking these trains all have a size of one, such that each node fits one train (unit).

Assumption 4-0.5 (No matching) All trains consist of one train unit and each train has a unique train type, so the matching is trivial.

We now formally define, under the aforementioned assumptions, the problem that we study in this chapter as the Pebble Motion problem on a Tree with Arrival and Departure (PMTAD).

Pebble Motion problem on a Tree with Arrival and Departure (PMTAD)

Input: $I = (T_L, P, S, D)$: given is a shunting tree T_L; a set P of n pebbles; an arriving sequence of pebbles S which positions the pebbles on L; and a departing sequence of pebbles D which positions the pebbles on L.

Question: Is there a sequence of moves, which each transfers a pebble from its current position on $v \in T$ to an adjacent unoccupied node $v' \in T \ni \{v, v'\} \in T$, to move the pebbles from S to D?

Since we assumed no reallocation is possible, a feasible scenario of PMTAD requires the tree T to be able to park all pebbles in such a way that they can leave in the correct departure order. We study different approaches to determine whether finding such a parking plan is possible. Clearly, this problem is relevant to the TUSP literature, however, it might also find applications in other areas. Consider, for example, distribution control, like warehouse automation or distribution center scheduling, where the goods arrive in sequential order and need to depart in different orders. This problem is also seen in container ports and is often referred to as marshaling [24].

4-1 Basic cases

Given a Shunting tree, we can derive statements about the structure and size of this tree that determine the feasibility of an instance of the PMTAD. An instance of the PMTAD is feasible if the tree T allows the arriving sequence of the pebbles to be permuted to the departing sequence, both positioned on path L.

We discuss two basic cases, the first defines the minimally required size of the tree (Proposition 4-1.1), and the second gives a condition for all sequence permutations to be feasible (Proposition 4-1.2). The first case considers an identical arrival and departure sequence. Then, each pebble can find a parking node in the tree T to park and depart again once it is his turn in the sequence.

Proposition 4-1.1 (Minimal tree size) Let an instance $I = (T_L, P, S, D)$ be given such that $S = D$. Now, I is feasible if and only if $|T| \geq n$.

Proof. case ($\Longrightarrow$): Because we assumed that all n pebbles have to park in T before they depart (Property 4-0.1), the size of T must be at least n. Because S and D are identical, there are no requirements for the tree structure, even if the tree is just a simple path, the pebbles can move into T and out again in the same order. So, if I is feasible, then all pebbles are able to find a parking space, and thus $|T| \geq n$.

case ($\Longleftarrow$): Since there are no cycles in a tree, each pebble q that is parked higher in the tree than pebble p will have arrived after p, i.e. $S(q) > S(p)$. Since the arriving sequence equals the departing sequence $D = S$, we have $D(q) > D(p)$, so q will leave the tree before p and p then has a free path to L. Therefore, we conclude that the instance I is feasible. □

Besides the identical arriving and departing sequences, we also consider a case concerning the tree structure, where each possible permutation of S is feasible.

Proposition 4-1.2 (Tree with n children) Let $I = (T_L, P, S, D)$ be given such that the root node t_0 of T has n children. The instance I is now feasible for any combination of S and D.

Proof. The tree T is rooted by node t_0 and if t_0 has n children, then all pebbles can park in a different child node, and since there is no pebble parked in t_0, there are no blocked pebbles, so every combination of S and D can be realized. □

4-2 Partitions of the arriving sequence

We now consider more complex scenarios, and we say the departing sequence is a permutation of the arriving sequence. Formally, a **permutation** $\sigma(S, D) = \binom{S=(s_1,..,s_n)}{D=(d_1,..,d_n)}$ defines the mapping of one sequence to the other, where $\sigma(s_1)$ defines the position of the pebble at position s_1 in the departing sequence.

Humans can quickly determine simple patterns in a sequence, for example, we can clearly see the pebbles that are in the same arrival and departure order. Say there are three pebbles (p_1, p_2, p_3) that arrive in that order, and the departure sequence is the same; meaning that first p_3 departs, then p_2, and finally p_1; the pebbles can park in a tree of sufficient length and depart when necessary (Proposition 4-1.1). The same principle can be applied to pebbles that do not arrive exactly after each other but are still in the same relative order. For example, if the arriving sequence is (p_1, p_3, p_2, p_4) and they depart as (p_1, p_2, p_3, p_4), then p_1 and p_2 are still in the same relative order, and so are p_3 and p_4, so both pairs could be parked together such that they can depart in their correct order.

Now, we use this idea to determine the ordered sets of pebbles that are in the correct relative order, which we call a partition $\Pi(S)$ of the arriving sequence. Here, we assume that the departing sequence D is a chronologically ordered sequence, i.e. $(p_1, p_2, \ldots, p_n)$. Since we consider the order of the pebbles, we refer to each set of the partition as a **totally ordered set (toset)**, because between each pair of pebbles within such a set their order in S and D is respected. The partition is defined in Definition 4-2.1, and

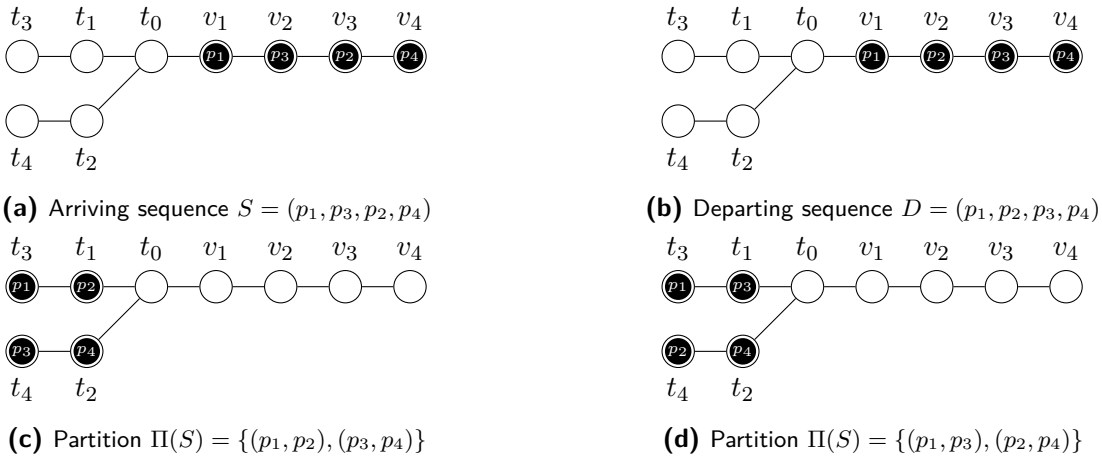
there can be different partitions for a sequence S. Consider for example the sequence (p_1, p_3, p_2, p_4) again. One way of defining the partition is $\{(p_1, p_2), (p_3, p_4)\}$, but $\{(p_1, p_3)(p_2, p_4)\}$ is also a valid partition. This example is shown in Figure 4-2.

Definition 4-2.1 (Partition) A **partition** $\Pi(S)$ of an arriving sequence S is the set $\Pi(S) = \{\pi_i\}$, where each item π_i is a **toset** such that:

i) all tosets are disjoint: $\pi_i \cap \pi_j = \emptyset$, $\forall \pi_j \in \Pi(S) : \pi_i \neq \pi_j$,

ii) the union of the tosets includes all pebbles: $\bigcup \pi_i \in \Pi(S) = P$, thus $\sum_{\pi_i \in \Pi(S)} |\pi_i| = n$, and

iii) the tosets respect the orders of S and D: $S(p_k) < S(p_l) \wedge D(p_k) < D(p_l)$, $1 \leq k \neq l \leq |\pi_i|$, $\forall \pi_i \in \Pi(S)$.



(a) Arriving sequence $S = (p_1, p_3, p_2, p_4)$

(b) Departing sequence $D = (p_1, p_2, p_3, p_4)$

(c) Partition $\Pi(S) = \{(p_1, p_2), (p_3, p_4)\}$

(d) Partition $\Pi(S) = \{(p_1, p_3), (p_2, p_4)\}$

Figure 4-2: Different partitions for $S = (p_1, p_3, p_2, p_4)$ and $D = (p_1, p_2, p_3, p_4)$

Number of partitions First, we strive to get an idea of the number of partitions that can be defined for a given sequence. Here, we do not make a distinction between two partitions that are of equal size and have equally sized tosets, though the pebbles might be distributed differently over the tosets. For example, the partitions $\{(p_1, p_2), (p_3, p_4)\}$ and $\{(p_1, p_3), (p_2, p_4)\}$ can be considered equivalent partitions for the sequence (p_1, p_3, p_2, p_4). Since the partition is a set, the order of the tosets within the partition does not matter, and the partitions $\{(p_1, p_2), (p_3, p_4)\}$ and $\{(p_3, p_4), (p_1, p_2)\}$ are the same. So, to determine the number of different partition sizes, we look at Table 4-1 which gives the different sizes of the tosets in a partition. The pattern we see is known as the **partition number** of the length of the sequence, which can be expressed by Euler's generating function $\sum_{n=0}^{\infty} p(n)x^n$ [12]. This can also be represented by an alternating sum of the pentagonal numbers: $p(n) = p(n-1) + p(n-2) - p(n-5) + p(n-7) - \ldots$, and we can determine that the number of partitions grows sub-exponentially.

n	$p(n)$	Possible different partitions
1	1	$\{[1]\}$
2	2	$\{[2], [1,1]\}$
3	3	$\{[3], [2,1], [1,1,1]\}$
4	5	$\{[4], [3,1], [2,2], [2,1,1], [1,1,1,1]\}$
5	7	$\{[5], [4,1], [3,2], [3,1,1], [2,2,1], [2,1,1,1], [1,1,1,1,1]\}$
6	11	$\{[6], [5,1], [4,2], [4,1,1], [3,3], [3,2,1], [3,1,1,1], [2,2,2], [2,2,1,1], [2,1,1,1,1], [1,1,1,1,1,1]\}$

Table 4-1: Possible size distributions of tosets in a partition

Determining the feasibility The partition of a sequence can be used to determine which pebbles should be parked together. Since we established that the shunting yard is represented by a tree T, we can express the tree in a set of branches $B(T)$, see Definition 4-2.2.

Definition 4-2.2 (Branch set) A **branch set** $B(T)$ of the tree $T = (V, E)$ is the set $B(T) = \{b_i\}$, where each branch $b_i \subset V$ is a set of nodes such that:

i) all branches are disjoint: $b_i \cap b_j = \emptyset$, $\forall b_j \in B(T) : b_i \neq b_j$,

ii) a branch starts from the child of a branching node $v \in V$,

iii) a branch is a subtree of T, and

iv) the size of a branch $|b_i|$ is given by the number of nodes it includes.

Each branch b can be used to park the pebbles associated with a single toset π if the branch is large enough to hold $|\pi|$ pebbles. However, the branch set is not unique to a tree, unless each branch must be a simple path. Otherwise, we can define the tree as one very large branch which includes subbranches, or define each branch individually. In Subsection 4-4-1, we elaborate more on possible branch sets and how to construct them.

Pairwise comparison To determine the feasibility of a given instance of PMTAD with a sequence S and a tree T, we use a pairwise comparison of the partition $\Pi(S)$ and branch set $B(T)$. We use a subscript $X_{\geq}$ to indicate the set X is represented as an ordered list according to element sizes from largest to smallest; and subscript $X_{\leq}$ to indicate the set X is returned as an ordered list from smallest to largest item size. Then, we can define when a partition and branch set are Pairwise comparable.

Definition 4-2.3 (Pairwise comparable) A branch set $B(T)$ and partition $\Pi(S)$ are **pairwise comparable** if we can create an ordered list of both sets in the same order, either both $\geq$ or both $\leq$, such that:

i) there are at least as many branches as tosets: $|\Pi(S)| \leq |B(T)|$, and

ii) each toset can fit on a branch: $|\pi_i| \leq |b_i|$, $1 \leq i \leq |\Pi(S)|$.

The tosets of a partition can be pairwise comparable with either individual branches $b_i \in B(T)$ or with disjoint subsets of branches $B_i \subset B(T) \ni \forall B_j \neq B_i \subset B(T) : B_i \cap B_j = \emptyset$.

So, both the branch set and the partition are not unique to the tree or the sequence, respectively. We now consider three different approaches, which each have a benefit for the practical application of this work, depending on the information available to the planning team. First, we take a partition and a branch set and establish the conditions that determine whether the scenario is feasible (Section 4-3). Then, we study the scenario of finding a set of branches given a certain partition (Section 4-4). We also give a method for finding a certain partition, specifically the optimal partition with the minimal number of tosets (Section 4-5). In the third approach, we determine whether a partition exists that matches a certain branch set (Section 4-6). Finally, the three approaches are combined to derive a method that either detects the scenario to be infeasible or returns a feasible partition, which can be used to generate the parking plan for the associated trains (Section 4-7)

4-3 Given a sequence and a tree, determine the feasibility

In a real-world scenario, we have a fixed shunting yard topology, which is represented by the tree T, and the planners at the NS might offer a certain plan for the train schedule, resulting in sequences S and D. Then, we need to quickly decide whether this scenario is feasible. In this section, we derive several conditions which can be used to determine in polynomial time whether an instance $I = (T_L, P, S, D)$ of the PMTAD is feasible or infeasible. However, these conditions do not give a complete approach and several scenarios will not be decidable in polynomial time.

4-3-1 Simple conditions

From Section 4-1, we already know two basic cases. Mostly, we can conclude from the proof of the Minimal tree size that we always need to be able to park all the pebbles in the tree T, because the nodes of the incoming path L cannot be used to park pebbles, this leads to Corollary 4-3.1. We also reformulate the proposition Tree with n children in Corollary 4-3.2.

Corollary 4-3.1 (Minimal size of the tree) If there are fewer nodes than pebbles, i.e $|T| \leq n$, then the instance is infeasible.

Corollary 4-3.2 (Trivial size of the tree) If there are at least as many branches as pebbles, i.e. $|B(T)| \geq n$, then for any sequence S, the instance is feasible.

4-3-2 Directed acyclic graph

To get a better sense of the relations between pebbles in a sequence, we can take a sequence S and transform it into a Directed Acyclic Graph (DAG). We put this graph on a grid for visibility and construct it by Method 4-3.1. An example is shown in Figure 4-3. Since we assume the departing sequence is the chronological sequence, the column number indicates the identification number of the pebble. Furthermore, by definition of the Arriving and departing sequences, the pebble at position s_1, which is the left-most position, arrives first; while the pebble at position d_1, which is also the left-most position, departs last. In the example in Figure 4-3, pebble p_1 arrives after p_4 but also departs later, so if they are parked in the same branch, then p_1 blocks p_4 upon departure.

Method 4-3.1 (Construct a directed acyclic graph from sequence)

1. Construct an $n \times n$ grid with the departure positions on the x-axis and the arrival positions on the y-axis.
2. Create a point for each pebble p: the row is the position of p in S and the column is the position of p in D.
3. Create the positive graph $DAG(S)^+$ by connecting the point of pebble p with *green* to every point of a pebble that arrives earlier but departs later (these are lines with a positive coefficient).
4. Create the negative graph $DAG(S)^-$ by connecting the point of pebble p with *blue* to every point of a pebble that arrives earlier and departs earlier (these are lines with negative coefficient).

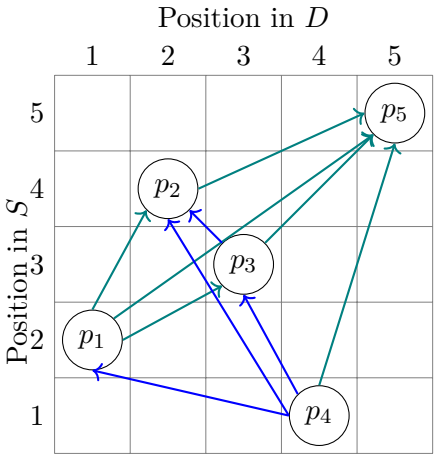


Figure 4-3: Visualization of directed acyclic graph of $S = (p_4, p_1, p_3, p_2, p_5)$

In Lemma 4-3.1, we show that Method 4-3.1 indeed always constructs a graph without cycles.

Lemma 4-3.1 (Sequence to DAG) Method 4-3.1 constructs a graph without cycles in $O(n^2)$ time.

PROOF. Method 4-3.1 traverses the pebbles, and for each pebble, the pebbles that have a higher position in the arriving sequence are traversed. This traversal is done in $O(n^2)$ time. We use a proof by contradiction. Suppose there is a cycle. Then, a cycle (p, q, r) would mean that $S(p) > S(q)$, $S(q) > S(r)$, and $S(r) > S(p)$, which is not possible within one sequence S, which has $S(a) < S(b)$ for every two pebbles (a, b) in the sequence. So, this leads to a contradiction, and we conclude there are no cycles in the DAG constructed by Method 4-3.1. □

KENDALL'S τ-DISTANCE

By Method 4-3.1, we traverse the sequence S to find for every pair of pebbles (p, q) whether $S(p) > S(q)$. If so, we add an edge in the $DAG(S)^+$ or in the $DAG(S)^-$ otherwise. If $S = D$, this means that for every pair, the edge will be added in the $DAG(S)^+$ and there will be no edges in the $DAG(S)^-$. There can be at most $n-1$ edges from the node of the first pebble in S to any other node. Subsequently, there can only be $n-2$ edges from the node of the second pebble in S. This results in $(n-1) + (n-2) + \cdots + 0$ edges, which sums up to $\frac{n(n-1)}{2}$. Similarly, there are at most $\frac{n(n-1)}{2}$ edges in the $DAG(S)^-$, although this happens when S is the exact reverse of D.

We note that this bound on the number of edges is also known as Kendall's τ-distance between two sequences, which gives the number of adjacent inversions that are necessary to transform one sequence into the other [5][Ch 6]. The distance is accurate for the $DAG(S)^-$: when there are no inversions necessary there are zero edges in the $DAG(S)^-$; and reversed for the $DAG(S)^+$, in which case there will be $\frac{n(n-1)}{2}$ edges if the distance is 0. Consider, for example, the sequences $S_1 = (p_1, p_3, p_2, p_4)$ and $D_1 = (p_1, p_2, p_3, p_4)$, for which the τ-distance is 1, since we can do one pairwise inversion of pebbles p_2 and p_3 to transform S_1 into D_1. There will be one edge from p_3 to p_2 in the $DAG(S)^-$, but $\frac{n(n-1)}{2} - 1$ edges in the $DAG(S)^+$.

Now, we can use this DAG to determine certain feasibility rules on the sequence. First, we introduce the known Longest Path problem. This problem is in the class $\mathcal{NP}$-*complete* when it considers general graphs, however, on directed acyclic graphs the problem is in $\mathcal{P}$ [23].

LONGEST PATH PROBLEM

Input: $I = (G)$: given is a graph $G = (V, E)$.
Question: Find the simple path in G including the maximum number of edges in E.

Consider the instance of the Longest Path problem below. On a general graph, we can keep following edges, and thus the path never really ends because there are cycles (Figure 4-4). However, on the DAG in Figure 4-5, we see that the options are more limited, and we can construct the red path $d \to e \to a \to b \to c$ that is of the maximal length.

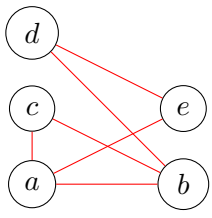


Figure 4-4: Longest path on a general graph

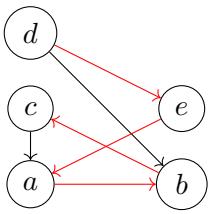


Figure 4-5: Longest path on a directed acyclic graph

Now, we use the Longest Path problem on our constructed negative DAG, to provide a condition for the feasibility of a pebble sequence and a tree (Lemma 4-3.2).

Lemma 4-3.2 (Minimal number of branches) The minimal number of branches $|B(T)|$ for I to be feasible is the number of nodes in the longest path $L(S)^-$ in the negative graph $DAG(S)^-$.

PROOF. Given is the negative graph $DAG(S)^-$ of a sequence S. We take the longest path through this

graph. By construction of the $DAG(S)^-$, for each pair of consecutive pebbles (p_i, p_j) along the path, we know that p_i arrives before p_j and departs earlier, so p_i and p_j cannot be parked in the same branch. Otherwise, p_j arrives later and then blocks p_i for departure. Therefore, each of the pebbles corresponding to the nodes in the longest path must be parked in a different branch, so the number of nodes in this path represents the number of necessary branches in T for I to be feasible. □

This proof results in the following corollary on the infeasibility of a sequence.

Corollary 4-3.3 (Size of the longest path) If there are fewer branches than the size of the longest path $L(S)^-$ in the $DAG(S)^-$, i.e. $|B(T)| < |L(S)^-|$, then I is infeasible.

We can further extend this knowledge to another simple case. Suppose that the branches can be infinitely long, then it is sufficient to know if there are enough branches to park the different pebbles which cannot be on one branch together. We give this case in Lemma 4-3.3.

Lemma 4-3.3 (Infinitely long branches) If there are $m = |L(S)^-|$ branches and each branch is of infinite, or at least $n - m + 1$, length then the instance is feasible.

Proof. Suppose that the branches can be infinitely long. We know from Corollary 4-3.3, that we always need at least $m = |L(S)^-|$ branches, otherwise the instance is definitely infeasible. Then, the feasibility depends on being able to park all pebbles in a way that matches a partition. If the branches are infinitely long, this is always possible. Moreover, since we know m pebbles park in the different branches, there can be at most $n - m + 1$ pebbles parked in a single branch, so this size is sufficient. □

Next, we use the known *NP-complete* Vertex-Disjoint Path Cover problem which finds a set of paths that do not share vertices among each other and together cover all vertices in the graph [29]. However, when the graph underlying this problem is a DAG this problem is known to be polynomially solvable [32].

Vertex-Disjoint Path Cover problem

Input: $I = (G, K)$: given is a graph $G = (V, E)$ and an integer K.

Question: Is there a set of K paths $\mathcal{P}$ in G such that every vertex in V belongs to exactly one path?

The problem is illustrated below, which shows a solution for the DAG in Figure 4-7, in fact, both the red paths and black ones are a solution. However, for general graphs, this is more complicated. This is most clear when we consider the case $K = 1$ (red path in Figure 4-6), then the problem becomes the Hamiltonian Path problem in which the goal is to determine whether a path exists that visits every vertex exactly once. This is one of the most well-known *NP-complete* graph problems [26].

For DAGs, the problem is in $\mathcal{P}$ and can be solved by transforming the problem into a bipartite graph and solving its matching problem. We create two nodes $v^\flat, v^\sharp$ in the bipartite graph for each v in the DAG, and add an undirected edge between $u^\flat$ and $v^\sharp$ for every edge (u, v) in the DAG. Now, we can find a matching in this bipartite graph of size $|V| - K$ which corresponds to a path cover of size K, and such a matching can be found in polynomial time [32].

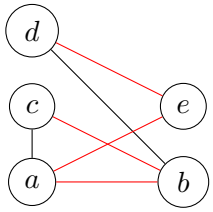


Figure 4-6: Path cover on a general graph with $K = 1$

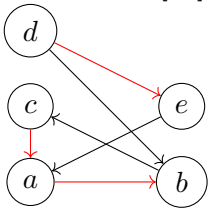


Figure 4-7: Path cover on a directed acyclic graph with $K = 2$

Now, we use the Vertex-Disjoint Path Cover problem on our constructed $DAG(S)^+$ to find a feasible partition for a pebble sequence and a tree in Method 4-3.2 and prove this is a valid partition in Lemma 4-3.4.

Method 4-3.2 (Finding a partition in a DAG)

1. Given is a directed acyclic graph $G = (V, E)$ of a pebble sequence and a path cover $C \subseteq E$
2. For each path in C, create a set of nodes $U \subseteq V$
3. For each set of nodes U, take the pebbles associated with these nodes and add them to a toset π
4. Create a partition of the tosets

Lemma 4-3.4 (Finding a partition in a DAG) Given the positive graph $DAG(S)^+$, we can construct a valid partition $\Pi(S)$ by taking a vertex-disjoint path cover of $DAG(S)^+$.

Proof. Given is the positive graph $DAG(S)^+$ of a sequence S. If we take a path cover C, we construct the partition by adding the pebbles associated with the nodes of a path together in a toset. By definition, the set of paths includes all the nodes in the graph, so the union of the tosets is P (condition ii of Partition). Since we use a vertex-disjoint path cover, the tosets in $\Pi(S)$ are disjoint (condition i). Finally, because the $DAG(S)^+$ respects the order of the pebbles in S by construction, we know that the pebbles in a toset are totally ordered (condition iii). So, we conclude that the partition based on a vertex-disjoint path cover is a valid partition for S. □

Based on Lemma 4-3.4, we derive the following two corollaries on specifying the partition. The latter can be achieved by selecting the specific edge for the path cover, and deleting all other edges connected to the nodes in the $DAG(S)^+$ of these pebbles, so they are no longer considered as other possible edges of the path cover.

Corollary 4-3.4 (Find a vertex-disjoint path cover) Given the number of branches $|B(T)|$, we can find a vertex-disjoint path cover of size $K = |B(T)|$.

Corollary 4-3.5 (Including a specific edge) Given two specific pebbles, which are in the correct relative order, we can ensure these are parked in the same branch, i.e. are included in the same toset of a partition.

4-3-3 The longest possible sequence

When we know that there are fewer branches than there are pebbles, we know that we should use as much space as possible. Moreover, if there are exactly n nodes in the union of the branches of T, then we know that we must use precisely every node. Therefore, we can take the largest branch of length $m = \max_{b \in B(T)} |b|$ and try to find a toset π of at least size m, so we can park a pebble from π on each node in the branch. We show the importance of this case in Lemma 4-3.5.

Lemma 4-3.5 (Largest branch) Let the tree T defined by branches $B(T)$ have $\sum_{b \in B(T)} |b| = n$, if there is no partition $\Pi(S)$ with a toset $\pi \in \Pi(S)$ such that $|\pi| \geq m = \max_{b \in B(T)} |b|$ then the instance I is infeasible.

Proof. Suppose that the branch set $B(T)$ has n nodes in total such that n pebbles can park there. Since we know that there are n nodes available for parking, every node in the tree must be used for parking a pebble for the instance to be feasible. So, the largest branch of size m must thus be filled with m pebbles. If there is no partition $\Pi(S)$ with a $\pi \in \Pi(S)$ such that $|\pi| \geq m$, then the largest branch cannot be filled with m pebbles of the same toset. So, there will be at least one pebble parked in the largest branch which blocks another pebble. Therefore, the instance is infeasible. □

We can find the largest possible toset by taking the Longest path of the $DAG(S)^+$ constructed by Method 4-3.1. This result is used in Corollary 4-3.6.

Corollary 4-3.6 (Parking in largest branch) If $\sum_{b \in B} |b| = n$ and the length of the longest path $L(S)^+$ in the $DAG(S)^+$ is smaller than the size of the largest branch, i.e. $|B(T)| > |L(S)^+|$, then the instance is infeasible.

We can also extend this result with empty nodes in T that can be left empty in the largest branch, while the other nodes must all be used to park a pebble. We take $c \geq 0$ to be the number of empty nodes in the tree (Corollary 4-3.7).

Corollary 4-3.7 (Parking with abundance) If $\sum_{b \in B} |b| = n + c$ and $|B(T)| > |L(S)^+| + c$, then the instance is infeasible.

4-3-4 Polynomial incomplete partition algorithm

We can use the Longest Path approach to design a polynomial-time algorithm that can find a partition to match a given branch set, which is given in Algorithm B-2 in Appendix B. By iterating the branches from largest to smallest, we find the longest path of length $|b|$ in the $DAG(S)^+$ to fill that branch with a toset of at most size $|b|$. There may be more than one path of the same length, so we select a random path of that length. Then, deleting all the edges connected to the nodes associated with any of the pebbles of that toset leaves the next branch to be filled only with pebbles that have not been selected for a toset yet. This way, we can find a partition that greedily fills the branches. Finally, we check if all pebbles of the sequence are included in the created partition. If not, we say the instance was infeasible. However, this does not always mean that the instance is definitely infeasible. Because we take just some path of a certain length, we might rule out the feasible solution, so the algorithm is not complete.

To determine the quality of this approach we tested all possible sequences of lengths three and four, which are $n!$ possibilities. Then, we selected the possible branch set lengths $\{(3), (2,1), (1,1,1)\}$ for $n = 3$ and $\{(4), (3,1), (2,2), (2,1,1), (1,1,1,1)\}$ for $n = 4$. We take all possible branch sets for the respective sequence length. We can easily determine which instances of sequence and branch set combinations are feasible, so we run all instances ten times and keep track of how many instances are solved correctly.

For $n = 3$, all ten runs returned the correct results for all instances. For $n = 4$, there was no perfect run. The incorrect infeasible instances were always found for the branch set $(2,2)$, while it was not always the same sequence which was not solved, see Table 4-2. The inability of the algorithm to find a feasible solution is thus not dependent on the sequence. However, all these instances do have in common that they need to find the match of two pebbles, while the other two are also compatible in a toset, so they should select the correct path of length 2 for filling the first branch.

Run/Sequence	1243	1324	1342	1423	2134	2314	2413	3124	3142	**Total**
1			1	1	1		1	1	1	6
2							1	1		2
3		1		1					1	3
4	1			1		1				3
5							1	1		2
6			1	1		1		1		4
7			1	1		1	1	1		5
8			1	1		1	1	1	1	6
9			1				1			2
10					1			1	1	3
Total	1	1	5	6	2	4	6	7	4	

Table 4-2: Results for tests with $n = 4$

Since we established that in a DAG we can find the longest path in polynomial time, we know that all operations in this algorithm can be done in polynomial time. This algorithm can thus be used to check

if a simple and quick solution can be found, but the absence of such a solution does not in fact mean that the instance is infeasible. So, we continue to study different approaches.

4-4 Match branches to a partition

In this approach, we assume that a partition $\Pi(S)$ is given. We are then given a tree T for which we want to find a branch set $B(T)$ such that we can assign each toset $\pi \in \Pi(S)$ to a subset of branches $B_i \subset B(T)$ such that $|\pi| \leq \sum_{b \in B_i} |b|$ and all subsets of branches are disjoint from each other. Then, every toset can park its pebbles on one branch, or split them over several branches, such that no two tosets have pebbles parked on the same branch. We give the general approach for finding $B(T)$ when a partition is given. First, we give a formal definition of the Branch Set for Partition Problem (BSPP).

Branch Set for Partition Problem (BSPP)

Input: $I = (T, \Pi(S))$: given a tree T and a partition $\Pi(S)$ of n pebbles based on an arriving sequence S of these pebbles.

Question: Is there a branch set $B(T)$ of branches such that i) all branches are disjoint and ii) the subsets of branches are Pairwise comparable with the tosets of $\Pi(S)$?

Now, we show that if an instance of the BSPP is feasible, then the associated instance of the PMTAD is feasible. This relation between the two problems shows that the BSPP is a restriction of the PMTAD, if the former is feasible, then so is the latter. However, since the PMTAD is more general this does not work both ways.

Lemma 4-4.1 (Relation between BSPP and PMTAD) Let an instance $I = (T_L, P, S, D)$ of the PMTAD and an instance $J = (T, \Pi(S))$ of the BSPP be given, such that T is the tree of T_L and $\Pi(S)$ is a partition of S. If J is feasible, then I is feasible.

Proof. Suppose that the instance J is feasible, then we can define a set of branches $B(T)$ such that its disjoint subsets are Pairwise comparable with the tosets of $\Pi(S)$ by conditions i) and ii) and all subsets B_i are disjoint. This will allow each toset $\pi \in \Pi(S)$ to park on one or more branches such that no other toset is assigned to one of those branches. Since $\Pi(S)$ is a valid partition, we know that the pebbles within each toset cannot block each other when parked in the tree, because they are already in the correct relevant order compared to each other. Furthermore, the different tosets get assigned disjoint subsets of branches, so the pebbles of two different tosets cannot block each other. Therefore, the availability of a branch set $B(T)$ meeting these conditions guarantees that the instance I is feasible. □

Next, we give an algorithm for the BSPP in Subsection 4-4-1 to show that $BSPP \in \mathcal{P}$.

4-4-1 Branch set matching

A branch does not necessarily have to be a simple path, so there could be branching nodes within a branch that form subbranches. These subbranches can also be seen as distinct branches on their own, which would imply that the set $B(T)$ is in fact larger.

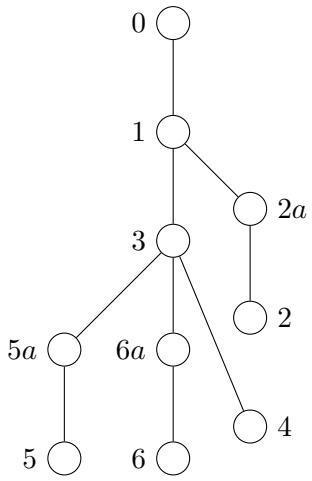
First, we take a look at an example shunting yard topology as used by the NS. We only discuss the tree-like shuffleboard type of shunting yard that the NS uses (see Appendix A). The NS instances are formatted in two files, a graph file, and a scenario file. Here, we only consider the graph file, which is structured as follows (see Listing 4.1). The first line tells us this is a file of the type graph, the second line gives the number of nodes in the graph, and the third line says the nodes are represented as a map. Then, a line for each node follows, which first gives the name of that node and then its neighbors, which are separated by spaces. Since shuffleboard graphs have a tree structure, we know that the first neighbor of the node is the parent, and afterward the children, if any, are listed. The resulting tree is shown in Figure 4-8b. The tree is rooted by node 0, and we could say there is a branch of 9 nodes. However, when we regard

nodes 1 and 3 as branching nodes we could say there are four branches: $\{\{2a, 2\}, \{5a, 5\}, \{6a, 6\}, \{4\}\}$ or two branches if only node 1 is a branching node: $\{\{2a, 2\}, \{3, 4, 5a, 5, 6a, 6\}\}$.

```
type graph
nodes 10
map
0 1
1 0 2a 3
2a 1 2
2 2a
3 1 4 5a 6a
4 3
5a 3 5
5 5a
6a 3 6
6 6a
```

Listing (4.1) Shuffleboard instance

(a) Input graph file



(b) Tree generated from the shuffleboard instance

Figure 4-8: Example of the graph file

We want to be able to answer the BSPP question. When we know the sizes of the partition and its tosets, we can create the branch set $B(T)$ such that $|B(T)| = |\Pi(S)|$. We now describe how to find this set of branches for a given tree T to match the partition of S. The pseudocode is given in Algorithm B-3 in Appendix B.

Using a depth-first search, we establish the order of the nodes in the tree from the root down. This order is then converted to branches that are matched to the partition. In general, this matching already finds the best feasible instance correctly. However, sometimes we need to merge branches to fit all the pebbles. For example, take a look at Figure 4-9 and consider the sequence $S = (p_6, p_7, p_5, p_1, p_2, p_3, p_4)$. This could be assigned the partition $\{(p_1, p_2, p_3, p_4), (p_6, p_7), (p_5)\}$. Therefore, a condition is added to solve this example by merging the $\{5a, 5\}$ and $\{6a, 6\}$ branches to park the toset of (p_1, p_2, p_3, p_4). However, not all cases will be solved by this algorithm because multiple partitions exist for a sequence and not all partitions can be matched to a certain branch set. For example, given the sequence $S' = (p_5, p_1, p_2, p_3, p_6, p_7, p_4)$, a straight-forward partition is $\{(p_1, p_2, p_3), (p_5, p_6, p_7), (p_4)\}$. Then, the first toset (p_1, p_2, p_3) might be assigned to the branch subset $\{\{2a, 2\}, \{4\}\}$. The next toset (p_5, p_6, p_7) must be assigned to the branch subset $\{\{5a, 5\}, \{6a, 6\}\}$ to fit all three pebbles. However, there will then be no branch left which is disjoint with both subsets to park the last toset with pebble p_4. So, the algorithm returns an infeasible result, although the partition $\{(p_1, p_2, p_3, p_4), (p_6, p_7), (p_5)\}$ is also valid for S' and would have been feasible as we can see in Figure 4-9.

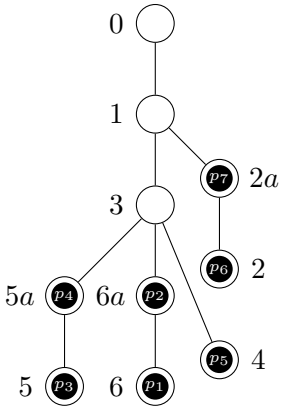


Figure 4-9: Scenario where branches need to be merged

Depth-first search on a tree is an operation that is possible in $O(n + m)$ where n is the number of nodes

and m is the number of edges [15, p.92-94]. We perform this search for all nodes in the graph and there are no other costly procedures, so the whole algorithm can be done in polynomial time. Therefore, we conclude that $BSPP \in \mathcal{P}$.

4-4-2 Remarks

If we can find a branch set of T that matches the given partition of S, then the instance J of the BSPP is feasible, and thus also the associated instance I of the PMTAD. However, if we cannot find such a branch set this does not necessarily imply that I is infeasible. For example, consider the example in Figure 4-10 and the sequence $S = (p_1, p_3, p_2, p_4)$. The current configuration clearly shows J and thus I is feasible because the partition $\Pi_1(S) = \{(p_1, p_3), (p_2, p_4)\}$ matches the branch set. However, the partition $\Pi_2(S) = \{(p_1, p_3, p_4), (p_2)\}$ does not match the branch set in this tree since we stated that each toset must be assigned a subset of branches that is disjoint with the other branches. So, if $\Pi_2(S)$ were given, then J would be infeasible, but as we already established I is feasible because it does not depend on $\Pi(S)$.

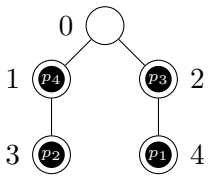


Figure 4-10: Example of branch and partition definition

So, although the BSPP is solvable in polynomial time, it is not a complete approach to determine whether an instance of the PMTAD is feasible. The representation of $\Pi(S)$ can have a big influence on determining the feasibility of the latter problem, and this approach is thus not useful for real-world scenarios. Furthermore, because we restricted each toset to park in a different branch, the current approach is not complete, because sometimes several partitions can share a branch, like in Figure 4-10. Therefore, in Section 4-6, we discuss a different approach that does provide a complete approach to determine the feasibility of the PMTAD. Though first, we look into a way of finding an optimal partition for a sequence. This approach can be used to generate a partition as input for Algorithm B-3.

4-5 Optimal partition

To find an optimal partition, we use a max-flow min-cost approach by translating the sequence to a graph. In a max-flow min-cost approach, the directed graph has a source and a sink and the goal is to find a flow that minimizes the cost of the edges used, respects their capacities, and maximizes the flow over the edges between the source and the sink. Method 4-5.1 transforms the arriving sequence into a partition.

Method 4-5.1 (Transforming an arriving sequence into a Max-Flow problem)

1. Create a source s
2. Create a node p_{iin} for every pebble p_i
3. Connect the source to every p_{iin} with cost 1, lower bound 0, and upper bound 1
4. Create a node p_{iout} for every pebble p_i
5. Connect every p_{iin} to its p_{iout} with a cost of 0, and a lower and upper bound of 1
6. Create a sink t
7. Connect every p_{iout} to the sink with cost 0, lower bound 0, and upper bound 1

8. For every pebble p_i connect its node $p_{i_{out}}$ to the node $p_{j_{in}}$ of pebble p_j if $S(p_i) < S(p_j)$ with cost 0, lower bound 0, and upper bound 1

An example is illustrated in Figure 4-11 for the arriving sequence $(p_5, p_2, p_3, p_1, p_4)$. The partitions can be found in this model as described in Method 4-5.2. There exist two different partitions of minimal length for this sequence. The first partition $\{(p_5), (p_2, p_3)(p_1, p_4)\}$ is shown with dashed lines, and the partition $\{(p_5), (p_2, p_3, p_4), (p_1)\}$ is shown with red lines (some edges are used for both). Both have a max-flow of 3, and thus require three different branches for a solution, although the sizes of the branches differ for the two partitions. We now show that the solution of the max-flow is indeed the minimal number of tosets of a sequence (Lemma 4-5.1).

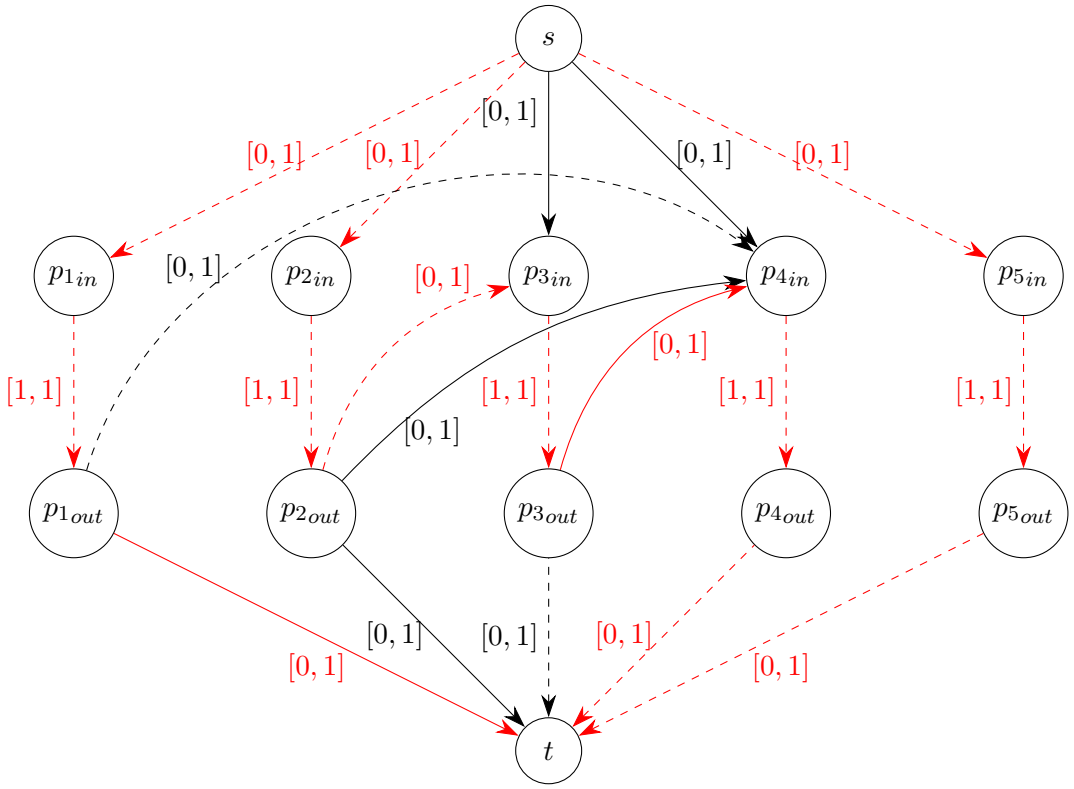


Figure 4-11: Max-Flow representation of a sequence

Method 4-5.2 (Recovering the partition from the Max-Flow solution)

1. Take the outgoing edges of s that have $f(s, v) > 0$
2. Follow each edge to t and keep track of the nodes it visits in between, for each in-out pair only keep the out-nodes
3. Construct a toset for each path from s to t by including the pebbles that correspond to the out-nodes

Lemma 4-5.1 (Max-Flow gives the minimal partition size) If the max-flow for a sequence S by the construction of Method 4-5.1 results in a solution, then this solution represents the minimal size of a partition of S.

Proof. First, we show that the solution of the max-flow algorithm is indeed a valid Partition. Then, we show that this is also optimal. However, there may be more than one optimal solution.

From Definition 4-2.1, we know that we need three conditions for a valid partition. By setting a lower bound on the edges between the in- and out-node for each pebble, we ensure that each pebble is included in a toset, thus we meet condition ii) $\bigcup \pi_i \in \Pi(S) = P$. The capacities on the edges connecting to the source and sink further ensure that each pebble is included in exactly one toset, so we have condition i) $\pi_i \cap \pi_j = \emptyset, \forall \pi_i \neq pi_j \in \Pi(S)$. Finally, by following the paths of the flow from the source to the sink,

the pebbles are always considered in order, as there are only edges between a pebble p_i and pebbles that arrive later than p_i, so we meet condition iii) $p_k < p_l, \forall 1 \leq k \neq l \leq |\pi_i|, \forall \pi_i \in \Pi(S)$.

We need to ensure that all pebbles are used, so we want to maximize the flow through the graph, while also minimizing the cost of the used edges, and thus the number of tosets. In other words, since there is only a cost associated with the outgoing edges from the source, we want to minimize the number of these edges used. Since there is a lower and upper bound of 1 on the edges from the in- to the out-node of a pebble, we know that the maximum flow can be at most the number of pebbles, while this will also be the minimum flow. Therefore, the only question is how many of the outgoing edges from the source are used, and since we use a min-cost objective, this number of edges will be minimal. This results in a minimal number of tosets in the partition since we trace the outgoing edges from the source to sink to find the different tosets (see Method 4-5.2). □

We can solve this max-flow problem with the Linear Program (LP) given in Equation 4-1. Here, we use a decision variable $f(u, v)$ to indicate whether the edge (u, v) is used, and this decision variable is a continuous variable that must be between 0 and 1. We want to maximize the flow out of the source s, but we want to minimize the number of outgoing edges of s that are used. So, we use the objective of minimizing the flow over the edges from the source to the incoming pebble nodes (Equation 4-1a), while we constrain all edges between the in- and outgoing nodes of a pebble to be used by setting the sum of their flows to equal the length of the sequence, which is the number of pebbles n (Equation 4-1b). Then, we constrain all the pebble nodes to have equal incoming and outgoing flow for a correct Max-Flow approach (Equation 4-1c). Finally, we define that the flow over an edge cannot exceed the edge capacity, which is one for all edges (Equation 4-1d) and must always be non-negative (Equation 4-1e).

$$\text{minimize} \sum_{v:(s,v)\in E} f(s,v) \tag{4-1a}$$

$$\text{subject to} \sum_{p\in S} f(p_{in}, p_{out}) = n \tag{4-1b}$$

$$\sum_{u:(u,v)\in E} f(u,v) = \sum_{w:(v,w)\in E} f(v,w) \quad \forall v \in V \setminus \{s,t\} \tag{4-1c}$$

$$f(u,v) \leq 1 \quad \forall (u,v) \in E \tag{4-1d}$$

$$f(u,v) \geq 0 \quad \forall (u,v) \in E \tag{4-1e}$$

LP 4-1: Finding the optimal partition

Then, we can use an LP solver to return an optimal solution. An LP can be solved in polynomial time. Furthermore, the construction of this Max-Flow formulation can be done in polynomial time as all constraints are applied to only the edges and nodes in the Max-Flow graph. Since the in- and out-nodes of the pebbles are based on the sequence of pebbles S, and this is a transitive relation between the pebbles, there can be at most $\frac{n(n-1)}{2}$ edges (Box: Kendall's τ-distance). So, there are order $O(n)$ nodes and up to $O(n^2)$ edges. The problem of finding the Max-Flow is also often connected to finding the minimum path cover through this graph, and this problem is known to be in $\mathcal{P}$ for a DAG (Box: Vertex-Disjoint Path Cover problem)

4-6 Find a partition on the available branches

In this approach, we take a branch set $B(T)$ of a tree, for which we then try to find a partition of sequence S that fits on the branches. First, we give a formal definition of the Partition for a Pebble Sequence on a Tree (PPST) problem.

Partition for a Pebble Sequence on a Tree (PPST)

Input: $I = (B, P, S, D)$: given is a set of branches $B(T)$ based on a shunting tree T_L; a set P of n pebbles; an arriving sequence of pebbles S; and a departing sequence D.

Question: Is there a valid Partition $\Pi(S)$ into tosets $\Pi_1, .., \Pi_m$ such that the tosets of $\Pi(S)_{\geq}$ and the individual branches of $B(T)_{\geq}$ are Pairwise comparable?

Now, we prove that the PPST problem is closely related to the PMTAD if the branches of $B(T)$ are all simple paths that branch from the root of the tree T. In contrast to Lemma 4-4.1, where a partition was given and we tried to find a branch set, we are now given a branch set and try to find a partition to match it. The lemma on the Relation between PPST and PMTAD is however not the converse of the lemma on the Relation between BSPP and PMTAD.

Lemma 4-6.1 (Relation between PPST and PMTAD) Let an instance $I = (T_L, P, S, D)$ of the PMTAD and an instance $J = (B(T), P, S, D)$ of the PPST problem be given, such that $B(T)$ is a branch set of the tree T of T_L. The instance I is feasible if and only if the instance J is feasible.

Proof. Proof of ($\Longrightarrow$): Suppose that I is feasible. We show that this implies that there is a $\Pi(S)$ that must be Pairwise comparable with $B(T)$ by conditions i) and ii).

First, we need to be able to park each toset in a different branch, which requires $|\Pi(S)| \leq |B(T)|$. In the general case, we have that no two tosets, say π and ρ, can be parked on the same branch. Only if for each pebble p_k in π we have that $p_k < q_1 \,, q_1 = \min_{q \in \rho}$, we know we can park all pebbles of π in a branch, and park the pebbles of ρ above these pebbles in the branch. This also implies that we could merge the two tosets into $R = (\pi, \rho)$. However, in general, we do not assume this to be true, so parking two arbitrary tosets π and ρ on the same branch would result in a conflict. From the definition of a partition, there would be a pebble $p \in \pi$ which arrives later than the pebbles in ρ, while p also has to depart later than all the pebbles in ρ. This means p will be blocking the pebbles from ρ, so a conflict arises, and we can conclude that each toset needs its own branch for parking its pebbles and thus require condition i).

Because we have ordered both the partition and branch set by the non-ascending size of the tosets and branches respectively, we can do a pairwise comparison. We start with the largest toset π_1 which we park in the largest branch, which can only fit if $|\pi_i| \leq |b_i|$, where b_i is the set of nodes associated with the largest branch. Because we know that $|\Pi(S)| \leq |B(T)|$, we know that for each toset there is a branch, and we require that this branch is large enough to fit all pebbles of the toset condition ii).

So, we have shown that both conditions i) and ii) must hold if I is feasible, so J is also feasible.

Proof of ($\Longleftarrow$): Suppose J is feasible, so there is a partition that matches the set of branches $B(T)$. There are $|B(T)|$ branches in the tree T and each branch b_i provides enough space to park $|b_i|$ pebbles. So, a toset π_j of size $|\pi_j| \leq |b_i|$ can park in the branch b_i. We can park up to $|B(T)|$ different tosets, in other words, if $|\Pi(S)| \leq |B(T)|$, then each toset can be parked on a separate branch (condition i). Now, we order the set $B(T)$ non-ascendingly, and we take a partition $\Pi(S)$ which is ordered non-ascendingly according to the toset size, then we can pairwise match a toset to a branch, and ensure condition ii). So, if there exists a partition $\Pi(S)$ that meets conditions i) and ii), then the I is feasible. □

From the proof of Lemma 4-6.1, we can derive the following Corollary 4-6.1 on merging tosets.

Corollary 4-6.1 (Merging tosets) Two tosets π and ρ can be merged into toset $R = (\pi, \rho)$ if and only if $p_k < q_1 \,, \forall p_k \in \pi \,, q_1 = \min_{q \in \rho}$.

With Lemma 4-6.1 we showed that when a partition can be found that matches the tree, the PMTAD is feasible. When each branch is a simple path, which is a direct child from the root of the tree, then this is both a necessary and sufficient condition. However, if the branches are not simple paths, then it is only a sufficient condition since the nested branching nodes could provide space for more feasible partitions to exist.

If we have a partition $\Pi(S)$ that is smaller than the set of branches $B(T)$ but there is no matching such that each toset is Pairwise comparable with a branch, then it might still be possible to feasibly park the pebbles. We explore this in Lemma 4-6.2.

Lemma 4-6.2 (Combining branches for a partition) Let an instance $I = (T_L, P, S, D)$ of the PMTAD and a branch set $B(T)$ be given. If there exists a strictly smaller partition $\Pi(S)$ ($|\Pi(S)| < |B(T)|$) such that its tosets are Pairwise comparable with the disjoint subsets of branches $B_i \subset B(T)$, then the instance I is feasible.

Proof. Suppose we have a set of branches $B(T)$ and a sequence of pebbles S expressed by $\Pi(S)$ such that $|\Pi(S)| < |B(T)|$. We order both sets in non-descending order of item size to be able to do a pairwise comparison. Now, we start with the smallest toset π_1 and park it on the smallest possible branch b which is large enough, i.e. $|\pi_1| \leq |b|$, if such a branch exists. Otherwise, we combine the first m branches in the ordered set $B(T)_{\leq}$ to form $B_1 \subset B(T), B_1 = \{b_1, ..b_m\}$ such that $|\pi| \leq \sum_{b \in B_1} |b|$. Not all the pebbles of one toset have to park on the same branch as long as there are no pebbles from two different tosets parked on the same branch because pebbles of the same toset cannot block each other by definition. Thus, the subsets of branches each have to be disjoint with the other subsets. We continue assigning tosets to a branch or a subset of branches that can hold all the pebbles of the toset. If this is possible for every toset, then the instance is feasible. □

Now, we turn back to the original problem, the PPST. We explore a method to find this partition in Subsection 4-6-1.

4-6-1 Finding a specific partition

Due to the complexity of the problem and the number of possible partitions, we suspect that the PPST problem is $\mathcal{NP}$-hard. We have tried to prove this by constructing a reduction from a known $\mathcal{NP}$-*hard* problem. Considered problems included SubSetSum [SP13], 3D Perfect Matching [SP1], Job Scheduling with Precedence Constraints [SS9] and Graph K-Colorability [GT4] [2, NP-complete problems]. However, none of the attempted reductions were complete as the construction of S is a complicated process.

We can, however, distinguish a special case of the PPST problem when there are m branches, each of size 2; and $n = 2m$ pebbles. We show this case can be solved in polynomial time in Theorem 4-6.1.

Theorem 4-6.1 (2-PPST is in P) The Partition for a Pebble Sequence on a Tree problem is in $\mathcal{P}$ if there are $m = |B(T)|$ branches of size 2 and $n \leq 2m$ pebbles.

Proof. Given is an instance $I = (B, P, S, D)$ of PPST, such that $m = |B(T)|$ and $n = 2m = |P|$. We can construct a $DAG(S)^+$ of the sequence S in polynomial time by Method 4-3.1. We can convert the $DAG(S)^+$ into a bipartite matching problem by creating a node $v^\flat$ and $v^\sharp$ for every $v \in DAG(S)^+$. Then, we add an undirected edge $\{u^\flat, v^\sharp\}$ for every edge $(u, v) \in DAG(S)^+$. If we can find a matching in the bipartite graph of size m, then we create a partition $\Pi(S)$ with m tosets of size 2. We now show I is feasible if ($\Longrightarrow$) and only if ($\Longleftarrow$) a matching of size m exists.

Proof of ($\Longrightarrow$): Suppose a matching of size m exists. By construction of the $DAG(S)^+$, we know that an edge (u, v) can only exist if the pebbles associated with u and v can be in the same toset, so the tosets respect the order (condition iii of Partition). Because no two edges in the bipartite matching may share an endpoint, all tosets are disjoint (condition i of Partition). Furthermore, there are $2m$ nodes on each side of the bipartite graph, so a matching of size m means that exactly all nodes are covered (condition ii of Partition). Because there are m tosets and m branches which are all of size 2, the branch set and partition are also Pairwise comparable. So, I is feasible.

Proof of ($\Longleftarrow$): Suppose I is feasible. Then, we have a partition $\Pi(S)$ with m tosets of size 2, which fit on m branches of size 2. So, we can create a bipartite matching between the pebbles, which will make m pairs of two pebbles. □

Now, we demonstrate a way to find a partition to match a given set of branches $B(T)$. We take the Max-Flow approach that was given in Section 4-5 and extend it to include the knowledge of the branch set. Afterward, we discuss what this means for solving the problems of both PPST and PMTAD.

4-6-2 Max-Flow extension

We can extend Method 4-5.1 by adding an intermediate source node s', which is connected to s with a lower and upper bound of $|B(T)|$. The source s is only connected to s', while s' is connected to the in-nodes for all pebbles with a lower bound 0 and an upper bound 1. The new method is described in Method 4-6.1.

Method 4-6.1 (Using the number of branch nodes in the Max-Flow model)

1. Create a source s
2. Create an intermediate source s'
3. Connect s to s' with a lower bound $|B(T)|$ and an upper bound $|B(T)|$
4. Create a node $p_{i_{in}}$ for every pebble p_i
5. Connect s' to every $p_{i_{in}}$ with a lower bound 0 and an upper bound 1
6. Create a node $p_{i_{out}}$ for every pebble p_i
7. Connect every $p_{i_{in}}$ to its $p_{i_{out}}$ with a lower bound 1 and an upper bound 1
8. Create a sink t
9. Connect every $p_{i_{out}}$ to the sink t with a lower bound 0 and an upper bound 1
10. For every pebble p_i connect its node $p_{i_{out}}$ to the node $p_{j_{in}}$ of pebble p_j if $S(p_i) < S(p_j)$ with a lower bound 0 and an upper bound 1

In Equation 4-2, we present the LP formulation for finding a partition that meets the size of the set $B(T)$. Since we know the size of the solution, i.e. the number of branches $|B(T)|$ that we want to find, we include this as a constraint. We no longer need a minimization objective, instead, we set the sum of the branches into the in-nodes to be equal to the number of branches (Equation 4-2a). Furthermore, we constrain the edge (s, s') to also have a lower and upper bound of exactly the number of branches (Equation 4-2d), while all other edges have a lower bound 0 (Equation 4-2e) and an upper bound 1 (Equation 4-2f). We implement the lower bound on the edge between in- and out-nodes by setting their sum to be equal to n (Equation 4-2b). Finally, we ensure that all flow is equal on the incoming edges and outgoing edges of all nodes except $\{s, s', t\}$ (Equation 4-2c). Since we only allow for an integer flow because this represents the number of pebbles in a toset, we implement this problem as an Integer Linear Program (ILP), which can be solved in exponential runtime.

An ILP solver can find a solution that creates $k = |\Pi(S)|$ tosets in a partition, and the partition can then be found using Method 4-5.2. However, there is usually more than one partition of that size for a sequence. As we established in the paragraph Number of partitions, there are sub-exponentially many partitions for a sequence of length n. Even when we already know the number of tosets in a partition, we still have many different ways of distributing n items over k tosets. In general, this division would result in $\binom{n}{k}$ possibilities, but since a partition is a set of tosets, we can eliminate a lot of sets due to symmetry. Let us look at the possibilities for some smaller numbers of n in Table 4-3. We consider the number of different possibilities $p_k(n)$ to distribute n pebbles over k non-empty tosets, without symmetry. The number $p_k(n)$ is recursively defined in Equation 4-3 and grows sub-exponentially [3].

$$\begin{aligned}
\text{subject to} \quad & \sum_{v:(s',v)\in E} f(s',v) && = |B(T)| && && \text{(4-2a)}\\
& \sum_{p\in S} f(p_{in}, p_{out}) && = n && && \text{(4-2b)}\\
& \sum_{u:(u,v)\in E} f(u,v) && = \sum_{w:(v,w)\in E} f(v,w) && \forall v \in V \setminus \{s, s', t\} && \text{(4-2c)}\\
& f(s,s') && = |B(T)| && && \text{(4-2d)}\\
& f(u,v) && \geq 0 && \forall (u,v) \in E \setminus \{(s,s')\} && \text{(4-2e)}\\
& f(u,v) && \leq 1 && \forall (u,v) \in E \setminus \{(s,s')\} && \text{(4-2f)}
\end{aligned}$$

LP 4-2: Finding a branch-set-sized partition

n	k	$\binom{n}{k}$	$p_k(n)$	Possible different partitions for $p_k(n)$
4	2	6	2	$\{[3,1],[2,2]\}$
5	2	10	2	$\{[4,1],[3,2]\}$
5	3	10	2	$\{[5],[4,1],[3,2],[3,1,1],[2,2,1],[2,1,1,1],[1,1,1,1,1]\}$
6	2	15	3	$\{[5,1],[4,2],[3,3]\}$
6	3	20	3	$\{[4,1,1],[3,2,1],[2,2,2]\}$
6	4	15	2	$\{[3,1,1,1],[2,2,1,1]\}$
7	2	21	3	$\{[6,1],[5,2],[4,3]\}$
7	3	35	4	$\{[5,1,1],[4,2,1],[3,2,2],[3,3,1]\}$
7	4	35	3	$\{[4,1,1,1],[3,2,1,1],[2,2,2,1]\}$

Table 4-3: Possible size distributions of tosets in a partition of size k

$$p_k(n) \begin{cases} 0 & \text{if } k > n, \\ 0 & \text{if } n \geq 1, k = 0, \\ 1 & \text{if } n = k = 0, \\ p_k(n-k) + p_{k-1}(n-1) & \text{if } 1 \leq k \leq n. \end{cases} \tag{4-3}$$

We expect that there will be no polynomial-time algorithm to determine whether a partition exists given a branch set. However, since we only need to find one partition that matches the branch set, there is a good chance that this can be found relatively quickly in most cases. We present a method of finding a partition to fit a set of branches B. The pseudocode is given in Algorithm B-4 in Appendix B. First, we can check whether a partition exists using the LP we derived in Equation 4-1. Here, we assume that $|\Pi(S)| = |B(T)|$, so we want the partition to exactly match the branch set. We construct the ILP as in Equation 4-2 and verify if we have found a fitting partition with the correct toset sizes. Then, we keep adding the previous solution as a constraint in each iteration if we have not found the solution. Once the ILP returns an infeasible solution, we terminate and conclude that sequence S is infeasible.

The major drawback of this approach is that it simply adds the previously found solution as a constraint to make sure a new solution is returned on each iteration. However, this will still require the ILP to iterate over all possible solutions as well as all symmetric solutions, and there are sub-exponentially many solutions (Equation 4-3). In theory, we want to find all lattice points in the polyhedra generated by the constraints in Equation 4-2. A method exists for finding how many such lattice points there are [11], however, ideally, we want to include the knowledge of the branch sizes in the ILP formulation.

4-6-3 Remarks

We suspect that the PPST problem is $\mathcal{NP}$-*hard*, so it is not solvable in polynomial time. Nonetheless, we did find a special case (Theorem 4-6.1) when there are $2m$ trains and m tracks that can each park two trains, and this problem can be solved in polynomial time using a bipartite matching approach. We also found the variant where branches have infinite length to be easily solvable by simply checking the condition of having enough branches (Lemma 4-3.3).

We have seen several approaches to determine the feasibility of instances of the PPST. In Subsection 4-4-1, we established that some partition can be found in polynomial time using the DAG of a sequence (Lemma 4-3.4), but this does not necessarily match a given branch set so it is not necessarily a solution to the PPST. Including the knowledge of the branches in Equation 4-2 to answer the PPST question would result in an exponential number of constraints which is not solvable in polynomial time anymore. The ILP in Equation 4-2 itself is also not an efficient way to find the partition solution of PPST due to its manner of traversing the solution space.

So, the general problem of finding a partition to match a given branch set $B(T)$ is not possible to determine in polynomial time. In the next section, we combine the knowledge of this section and Section 4-3 into an algorithm that can determine the feasibility of an instance of PPST, which has an upper bound of $O(2^n)$ on the runtime.

4-7 Feasibility approach

We use the insights from the previous sections to create an approach for determining the feasibility of a sequence S on a given branch set $B(T)$ (the PPST question). First, we use Corollary 4-3.2 which provides a clause that can immediately determine if a certain feasible solution exists. With the combination of Corollary 4-3.1 and Corollary 4-3.3, we can also rule out some infeasible cases in a very short amount of time. When we already know that a sequence is infeasible, we no longer have to look for a partition. However, these two corollaries do not rule out all infeasible cases. For example, the sequence $S = (p_4, p_2, p_7, p_5, p_6, p_1, p_3)$, which can be solved on the branch set $B(T) = \{2, 2, 3\}$, but not on the branch set $B'(T) = \{4, 2, 1\}$. So, we can further expand our approach using Corollary 4-3.6 to rule out this case, because we know there is no toset possible of size four. Yet, this still leaves infeasible cases that are undetected.

If an instance was not marked as infeasible, then we could use Corollary 4-3.4 to find a partition with the correct number of branches. The problem of finding a path cover is often translated to a Max-Flow problem, similar to what we did in Subsection 4-6-2, which, in general, can also be solved in polynomial time [35]. However, there we ran into the problem that we could not include the size of the branches in the Max-Flow formulation. To do this, an exponential number of constraints would have to be added in a devious way, and the problem would thus no longer be polynomially solvable. So now, we use a backtracking algorithm to find a path cover with path sizes that match the branch sizes, because we can then easily specify that we need to find tosets of a certain size that are disjoint to each other. Here, we determine a path cover based on possible paths through the DAG, so we first discuss the possible number of paths.

4-7-1 Number of paths

We now consider the number of paths that can be created in our $DAG(S)^+$. When $S = D$, we have the maximum number of paths because every possible partition is then possible, which results in $\frac{n(n-1)}{2}$ edges in the $DAG(S)^+$ (Lemma 4-3.1). There are at most $\frac{n(n-1)}{2}$ different paths of length two, which are the triangular numbers. For paths of length three, the maximum number is $\frac{n(n-1)(n-2)}{6}$, which are the tetrahedral numbers. This will continue, and these sequences can be found on Pascal's triangle, which is the triangular array of binomial coefficients. The first seven rows are shown in Figure 4-12.

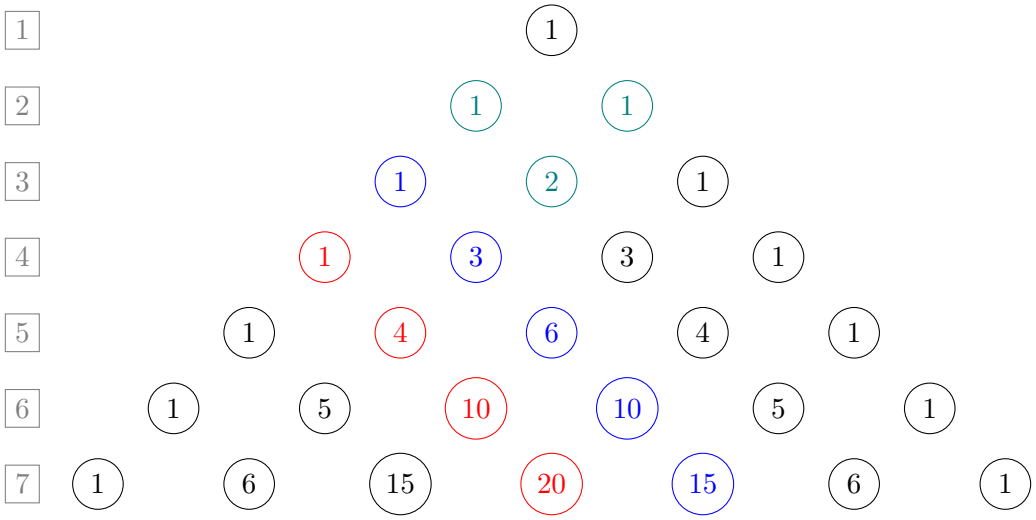


Figure 4-12: Pascal's triangle

This triangle can be constructed, by adding two parent nodes together to find the number of their child node, for example, adding the green numbers results in $1 + 1 = 2$, which is their parent. In every diagonal, we can find a sequence. The first diagonal consists of only ones, and the second diagonal is the chronological number sequence. The third diagonal, in blue, shows the triangular numbers, next are the tetrahedral numbers (shown in red), and these sequences continue with every diagonal. We can use this triangle to find the maximum number of paths in the $DAG(S)^+$ for a sequence S of length n. We can count the number of paths for $n = 2$, $n = 3$ and $n = 4$ when $S = D$, which are 1, 4 and 11 respectively. We find these numbers as sums of the first $n-1$ numbers from left to right of the $n+1$'th row. In other words, sum the sequences up and till the triangular numbers (blue diagonal). So, for $n = 4$, we look at row 5 and we sum the first three: $1 + 4 + 6 = 11$.

In Pascal's triangle, the sum of a horizontal row is that power of 2, and the last two rows that we do not add to our sum are the sequences of n and ones. We conclude that for a sequence of length n, there will find be at most $2^n - n - 1$ paths. However, we also consider the n paths of length 1 to allow for tosets with one pebble. So, the total number of paths to consider for a sequence of length n is $O(2^n)$ paths.

4-7-2 The algorithm

Algorithm 4-1 gives the backtracking algorithm, and we analyze its runtime in terms of n pebbles (nodes in the DAG). As shown in Lemma 4-3.1, there are at most $\frac{n(n-1)}{2}$ edges in the constructed DAG. We generate all possible paths $\mathcal{P}$ in the AllPaths method, which uses a depth-first search to generate all paths from a given node. DFS considers every edge for every node, which is $O(n \cdot n^2) = O(n^3)$, and since this is done for every node, $\mathcal{P}$ is created in $O(n^4)$. The feasibility checks can be done in $O(n)$ given that the paths were already created in a previous step. The PathCover method does not run in polynomial time, since it will backtrack upon a conflict and could theoretically consider every path combination of $|\mathcal{P}| = O(2^n)$ paths.

4-7-3 Results

Algorithm 4-1 is mainly a proof-of-concept of an approach to solving instances of the PPST. We performed some basic testing of the functionality and determined the extent of the abilities of the algorithm to solve different instances. For $n = 3$ and $n = 4$, we created instances of all the possible sequences of those lengths and the different branch sets that can be formed for trees with exactly n nodes. We set up these tests and averaged the runtime for each one over ten runs. Each instance was solved within less than one millisecond. Furthermore, we set up testing with longer sequences, where the sequence

was generated randomly with a shuffling approach over a list of all n numbers. The branch set was then generated by taking a random partition number of n. For each value of n, five random instances were created. The tests were then executed for $n = 10, 15, 20, 25, 30, 35, 40, 45, 50$ and the timeout was set to five minutes.

These values were chosen to get a grasp of what sizes can be solved within this time limit. Further testing could indicate more precise values of n that are solvable within the available time. Furthermore, these values represent realistic scenarios, for example, the *Kleine Binckhorst* shunting yard processed roughly 28 trains every day in 2018 [31]. Furthermore, the larger values of n represent possible future scenarios when NS might operate more rolling stock. This test setup was repeated five times in total. Table 4-4 shows the number of random instances per run that were correctly solved within the timeout and the average runtime for a single instance is given for every value of n. The results were checked for correctness to be valid partitions for the given sequence.

Test run	$n = 10$	$n = 15$	$n = 20$	$n = 25$	$n = 30$	$n = 35$	$n = 40$	$n = 45$	$n = 50$
1	5/5	5/5	5/5	5/5	0/5	0/5	0/5	0/5	0/5
2	5/5	5/5	5/5	5/5	0/5	0/5	0/5	0/5	0/5
3	5/5	5/5	5/5	5/5	0/5	0/5	0/5	0/5	0/5
4	5/5	5/5	5/5	5/5	0/5	0/5	0/5	0/5	0/5
5	5/5	5/5	5/5	5/5	0/5	0/5	0/5	0/5	0/5
Average time	20	180	5575	490000	T/O	T/O	T/O	T/O	T/O

Table 4-4: Test results for Algorithm 4-1

The results show an exponential runtime, which was expected as the PathCoverUtil method will eventually traverse all possible paths if no solution is found, and we established that the number of paths is 2^n. Furthermore, a single test for $n = 30$ resulted in an average runtime of 22 minutes, so the exponential trend continues as n grows even larger and instances of $n > 30$ cannot complete with a runtime that is still useful in practice. However, a runtime of half an hour can still be acceptable when constructing a daily plan, and instances with $n = 30$ are actually representative of real-world scenarios [38].

Algorithm 4-1 Approach for finding a partition based on a path cover

```
procedure SOLVE(S, B)
    if |B| = |S| then return TRUE
    E⁺, E⁻ ← CONSTRUCTDAG(S)
    if ISINFEASIBLE(S, B, E⁺, E⁻) then return FALSE
    𝒫 ← ALLPATHS(E⁺, S)
    Sort B in descending order
    P ← ∅
    if ¬PATHCOVER(𝒫, S, B, P) then return FALSE
    return P
procedure CONSTRUCTDAG(S)
    E⁺ ← ∅, E⁻ ← ∅
    for i ∈ [0, n] do
        for j ∈ [i + 1, n] do
            if S[i] > S[j] then
                Add edge (S[i], S[j]) to E⁺
            else
                Add edge (S[i], S[j]) to E⁻
    return E⁺, E⁻
procedure ISINFEASIBLE(S, B, E⁺, E⁻)
    if |S| > ∑_{b∈B} |b| then          ▷ Infeasible if there are fewer parking nodes than pebbles
        return TRUE
    if ALLPATHS(E⁻, S) > |B| then      ▷ Infeasible if there are fewer branches than tosets
        return TRUE
    if ALLPATHS(E⁺, S) < max_{b∈B} |b| then   ▷ Infeasible if largest branch cannot be filled
        return TRUE
    return FALSE
procedure ALLPATHS(E, S)
    𝒫 ← ⋃_{p∈S} DFS(p, ∅, [p])         ▷ Get all paths starting from any node in the DAG(S)
    return 𝒫
procedure DFS(node, seen, p)           ▷ Depth-first search to find paths down from node
    Add node to seen
    paths ← ∅
    for t ∈ neighbors(node) do
        if t ∉ seen then
            p_t ← [p, t]
            Add p_t to paths
            paths ← paths ∪ DFS(t, seen, p_t)
    return paths
procedure PATHCOVER(𝒫, S, B, P)
    Create dictionary 𝒫_L of paths per length
    Add paths of length 1 for every p ∈ S   ▷ Then a pebble can be assigned to a toset of size 1
    return ¬PATHCOVERUTIL(P, S, 0, B, 𝒫_L)
procedure PATHCOVERUTIL(P, S, i, B, 𝒫_L)
    if ⋃_{π∈P} π = S then              ▷ If whole sequence in partition, partition found
        return TRUE
    if i > |B| then                    ▷ If all branches have been considered, partition does not exist
        return FALSE
    for p ∈ 𝒫_L[B[i]] do
        if P ∩ p = ∅ then
            P ← P ∪ p                  ▷ Add found toset to partition
            if PATHCOVERUTIL(P, S, i + 1, B, 𝒫_L) then
                return TRUE
            P ← P \ p                  ▷ Remove found toset to partition
    return FALSE
```

4-8 Conclusion

In this chapter, we studied the Pebble Motion problem on a Tree with Arrival and Departure. This variant of the Pebble Motion problem (PM) includes the arrival and departure of pebbles using pebble sequences that are positioned on the designated track in our Shunting tree. We tried to find an approach to quickly solve the PMTAD and looked into several methods. We identified several conditions which can quickly determine the (in)feasibility of problem instances. One of the tools we used was to convert a sequence to a Directed Acyclic Graph, which shows the relations between pebbles that arrive earlier but depart later, and vice versa. Based on this DAG, graph-theoretical algorithms can be used to establish some of the feasibility conditions. However, there are still many scenarios that cannot be decided by these conditions. Then, we considered the BSPP problem, where the partition of the sequence is already given and we want to find a matching branch set from the given tree. Although the BSPP $\in \mathcal{P}$, it is not a complete approach because it does not always find the correct result, so an infeasible result cannot guarantee that the scenario is infeasible. The other way around, the PPST problem assumed the branch set was already given, and the goal is to find a partition that fits in the branches. There exist sub-exponentially many partitions for a given sequence, and we were not able to find an algorithm that can determine the correct one in polynomial time. In the worst case, all partitions have to be considered, so we expect the problem to be $\mathcal{NP}$-hard, although we were unable to prove this. Finally, we combined all the knowledge of this chapter to create an algorithm that can find a partition of the sequence, given a branch set. The algorithm starts to check the established feasibility conditions and then uses the DAG to find a path cover to represent a partition. However, Algorithm 4-1 has a worst-case exponential runtime.

Chapter 5

Modeling the train and track lengths

In this chapter, we discuss the implications of including train and track lengths in an extension to the Pebble Motion problem (PM), and more specifically in the model described by the Pebble Motion problem on a Tree with Arrival and Departure (PMTAD). Because we ultimately want to combine the insights of both the length extensions and the arrival and departure modeling, we immediately extend the arrival and departure model.

First, we start with some background on the train and track lengths. Trains consist of different train units and trains must always be composed of the same type of train unit. For each type, there are two different lengths and each train unit consists of a predefined number of train carriages, referred to in Dutch as *bakken*. The train unit type always specifies the number of carriages in one unit, by use of Roman numerals, and the unit cannot be decomposed into individual carriages. One carriage of a train unit is always roughly the same length independent of the type. So, in the rest of this chapter, we use the lengths defined by the number of carriages in a train unit, and for tracks, we specify the number of carriages that can fit on that track. Figure 5-1 gives an overview of the main types of rolling stock used by the NS in 2022 with the different lengths of train units for that type [42]. In the rest of this chapter we consider only these train units, and thus only consider the three unit lengths of three, four, and six.

(a) Sprinter Lighttrain: SLT-IV, SLT-VI **(b)** Sprinter Nieuwe Generatie: SNG-III, SNG-IV **(c)** Verlengd Interregio Materieel: VIRM-IV, VIRM-VI **(d)** Intercity Materieel: ICM-III, ICM-IV

Figure 5-1: Four main train unit compositions used by the NS in 2022 [41]

Model criteria In our model, we strive to meet two main criteria. First, we extend from the PMTAD, so we want the extension for the length definition to be compatible with the approaches discussed in Chapter 4. Those theorems are defined on n pebbles, so this parameter should ideally also be used in the extension we propose in this chapter. In other words, we want our model to be an extension of the PM. Second, we want our model to be relevant for practical application. Since we look for the properties of the tree and sequences to realize a feasible scenario, solutions that require much more space than necessary are not very useful in practice. In other words, we want our model to be as close to the real-world scenario as possible.

Model conditions There are several aspects to consider when including the train and track lengths. First, trains should always be able to fit on their assigned parking track, however, we also want to use the space as efficiently as possible. Therefore, we want to use accurate lengths, so we could park either a long train or several short trains on the same track. Second, the length of trains might also affect their routing through a shunting yard. Some angles between different tracks might be very sharp, so the train must be able to make that turn.

Such sharp angles most likely occur quite often in shunting yards that are represented by trees, since most trees will form an acute angle between the two children of a branching node, as this requires the least amount of space for creating these tracks. For example, in the layout of the NS *Kleine Binckhorst* shunting yard located in The Hague [28], shown in Figure 5-2, the nodes labeled **957** and **960** form acute angles (as do most nodes in this layout). Therefore, this will influence the possible routes that a pebble can take in the tree. However, the problem of making such sharp turns only applies when we consider reallocation. In this thesis, we do not consider reallocation, so we can ignore this condition.

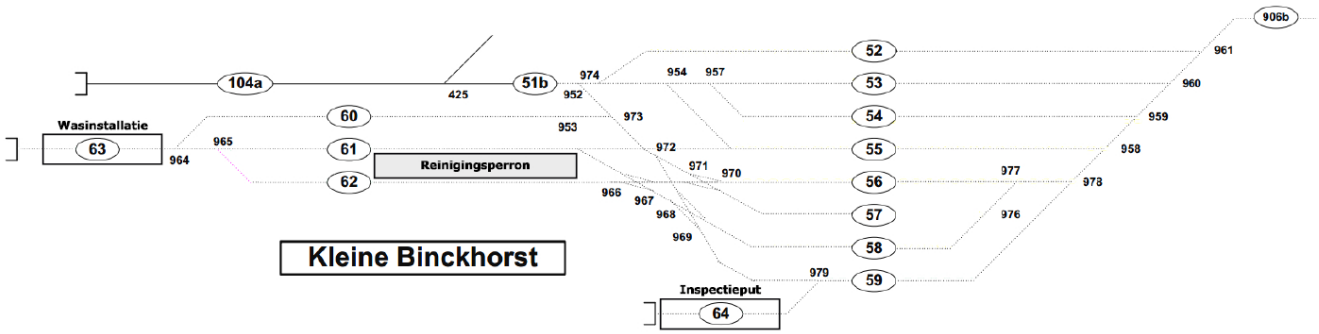


Figure 5-2: Shunting yard and service site '*Kleine Binckhorst*'

5-1 How to fit pebbles on the track

We now consider five different ways of modeling the lengths to make sure that the train units fit on a track. As we are extending the PMTAD model, we want to find a way to ideally encode these lengths in terms of graph theory, using the existing nodes and edges in the tree. We discuss the length of train units, expressed by the number of train carriages, and the size of pebbles.

5-1-1 Node sizes

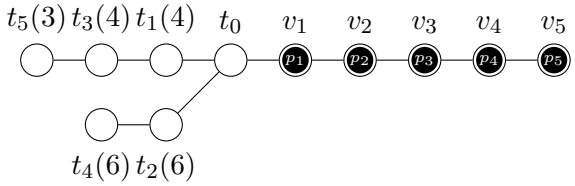
A first instinct would be to define a size $s(t)$ for each node t in T. Then, we can still consider n pebbles such that each pebble represents a train unit, and we can define a function $\ell(p)$ to return the size of a pebble. Consequently, for a pebble to park at a node, it must fit on that node, so we would require $s(t) \geq \ell(p)$. However, this would result in the matching of train units to parking tracks where they fit, while we also have to find a feasible parking track to begin with. Furthermore, this approach requires an assignment of sizes to nodes, which can be complicated, for example, when there is a long track in the shunting yard, which could hold several train units. Then, the question arises whether to create one node with a large size and allow multiple pebbles to park on this node, or define various nodes which each hold one pebble.

In the latter case, a new question arises about how to define which node is assigned a certain size, because the order of differently sized nodes on a track can influence the feasibility of an instance. This effect is illustrated in Figure 5-3, which shows the node sizes in between brackets. In this scenario, pebble p_1 can park on node t_5 and p_2 in t_3. However, pebble p_3 cannot park on node t_1 because it does not fit there, so it would park on node t_4, and node p_4 follows to park on node t_2. Finally, pebble p_5 can park on node t_1. Although all pebbles have found a parking space where they fit, pebble p_4 is now blocking

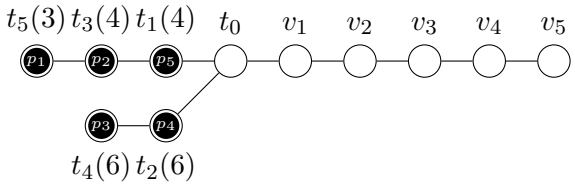
pebble p_3 which has to depart earlier, so the scenario is infeasible. If node sizes were not considered, then p_3 could park on node t_1 and the scenario would be feasible.

Pebble	Type
p_1	ICM-III
p_2	ICM-IV
p_3	VIRM-VI
p_4	SLT-VI
p_5	SLT-IV

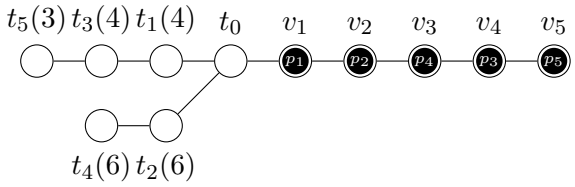
(a) Train unit compositions



(b) Arriving sequence



(c) Intermediate configuration



(d) Departing sequence

Figure 5-3: Example of influence of node size

Although a naive solution could just assign the maximum size, in our case six train carriages, to each node this would result in solutions with a lot of unused space, which is not a practical solution in the real-world scenario. Most shunting yards are used at a close-to-maximum capacity.

5-1-2 Pebbles for train carriages

A second approach could indicate that every node in the tree T can only fit one train carriage. We could then create a pebble for each carriage of a train unit, and allow each pebble to only park on one node. However, train units cannot be decomposed, so we would still need to define the composition of a train unit, such that the different carriages are not somehow split over different branches in a tree. Otherwise, the solution no longer represents a feasible solution in the real-world scenario since a train unit cannot be split.

Given the number of train carriages of every train unit, we could use the parameter n for the number of train units, and the parameter l for the number of train carriages. Then, we require $l \geq 3n$ since the smallest train unit that we consider consists of three train carriages. However, this still leads to complications because existing algorithms for the PM move pebbles through the graph, and we need to make sure that pebbles belonging to the same train unit are not split over different tree branches. Moreover, the partition that we defined in the previous chapter would also be affected by the existence of many more pebbles because it would always have to assign the pebbles of the same train unit to the same toset.

5-1-3 Pebble sizes

Another way of tackling the problem is to consider n pebbles and use a function $\ell(p)$ to return the size of that pebble. If we allow only one pebble of size one to park on a single node, we would need a function to assign multiple nodes on which one pebble is parked, because each train unit has more than one carriage. This approach would require a way to define that a pebble can park on different nodes at the same time, for which it must hold that all nodes are unoccupied, together they provide enough space to park the pebble, and they must be adjacent nodes on a simple path. The last condition is important because we prefer trains not to park on switches in the shunting yards, which are the branching nodes in our trees. Thus, we want pebbles to park in a branch so the branching node can still be used to navigate trains to other branches. Therefore, this approach is also not without complications.

5-1-4 Edge lengths

Besides nodes and pebbles, we can also study the length in terms of edges in the tree. We could give edges a length, or cost, which then corresponds to the available space at the connected node. We consider the edge that connects a node t in the tree to its parent node and we refer to this edge as the **ancestor edge** e_t to which we assign a length $\ell(e_t)$. Furthermore, we create a cost function γ for pebbles, such that $\lambda(p)$ returns the size of the pebble p, which represents the number of carriages in a train unit.

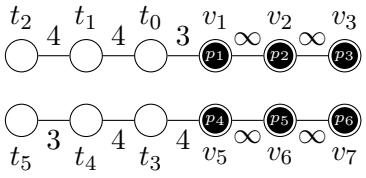
This approach provides the benefit that edge costs are a very common constraint in graph theory: existing algorithms for graph problems already use an edge cost to determine whether a certain problem scenario is feasible. For example, a well-known case of such use of edge costs is adopted in network flow, where an edge has a flow and a capacity (as we also used in Section 4-5). The objective of this problem is to find a flow through the network from source to sink such that the capacity of all edges is never exceeded [15]. In Multi-Agent Path Finding (MAPF) algorithms, edge costs are mostly used to determine the cost of a solution, where minimal costs represent the shortest paths and thus faster solutions [22].

The length assignment to the edges is a complicated process similar to the assignment of node sizes which was proposed in Subsection 5-1-1. The example given in Figure 5-3 can also be applied to edge lengths if the node sizes in this example were represented as lengths of the ancestor edges, the effect is the same.

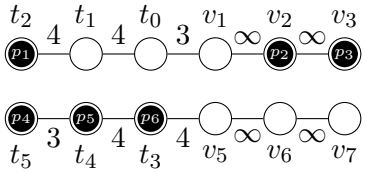
Finally, there are two variants of cost constraints that we can consider: pebbles can either never cross an edge if their cost exceeds the edge length, or they can only not park on the nodes associated with such edges. This effect is demonstrated in Figure 5-4. If there is a long track in the shunting yard, this can be modeled as a single branch of the tree which is a simple path of k nodes. In practice, the length of the track is expressed in meters, but assigning the edge lengths in terms of train carriages as $(3, 4, 4)$ or $(4, 4, 3)$ can make a big difference for the scenarios which are feasible on this track.

Pebble	p_1/p_4	p_2/p_5	p_3/p_6
Type	ICM-III	ICM-IV	ICM-IV

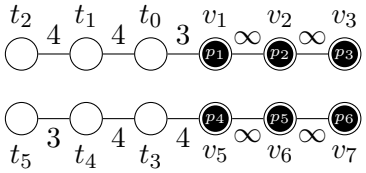
(a) Train unit compositions



(b) Arriving sequence



(c) Intermediate configuration



(d) Departing sequence

Figure 5-4: Example of complicated edge length assignment

In the upper assignment of Figure 5-4, the pebbles p_2 and p_3 cannot pass over the edge $\{t_0, v_1\}$ if the first, harder, constraint is imposed such that pebbles cannot traverse *any* edge which is shorter than its length. On the other hand, if the edge length is only considered for parking, as in the latter, softer, constraint, then all pebbles can park because the edge lengths correspond to the exact same order as the arriving pebbles. In the lower assignment, all pebbles can move onto the track and they fit exactly on each node. However, this assignment was now fitted to the arriving sequence, which we cannot know in advance in most cases. We could define the general case that defines the lengths of a track such that the edges which are lowest in the tree, the closest to the leaf nodes, have the lowest edge lengths. In the example of Figure 5-3, the lowest node sizes were also the furthest down tree, which resulted in an infeasible scenario.

5-1-5 Track nodes

Finally, we consider a model which creates one node for each track, and a track is represented as a simple path, which is also referred to as a corridor in the literature on the Train Unit Shunting Problem (TUSP). We define a cost function $\ell(t)$ for all nodes $t \in T$, which returns the size of the track. The size can be expressed in the number of train carriages, although it is also possible to express this directly in the exact measurement of the train unit in meters.

The advantage of this approach is that there is no concern for defining the cost of specific nodes of a track, so no added complexity is created for determining which train unit fits on which part of the track. However, this approach also loses the ability to define which pebble enters the node first and to keep the order of the arriving and departing pebbles at the node.

5-1-6 Conclusion

Considering the options that we discussed, we try to meet both Model criteria as closely as possible. So, we want the model to be a compatible extension of the PMTAD and we want it to simulate the real-world scenario as accurately as possible. We summarize the results in Table 5-1.

Model	PM extension	Real-world compatibility
Node size	Yes	No
Train carriage pebbles	No	Yes
Pebble size	No	No
Edge length	Yes	No
Track nodes	No	Yes

Table 5-1: Summary of different modeling approaches

We already stated that the edge length and node size approaches are very similar, where the same constraint is enforced either on the node or its ancestor edge. Although the edge length also provides the option to constrain a pebble to only traverse the edges which are smaller than its own size, this would only add further constraints and is thus not a necessary advantage of this model. Both approaches have the main disadvantage that the assignment of sizes to either nodes or edges is a complicated process. This problem is solved by the one node per track approach, as here only the total size of the pebbles parked on this track is relevant. However, this approach loses the possibility of keeping the order of the pebbles on the track. The models based on the pebbles, where each node can still hold only a train carriage of size one, carry the most disadvantages because the adaptation of a Partition would be particularly difficult.

Clearly, none of the proposed models is the obvious choice because each has its own advantages and disadvantages. However, we can combine the most promising approaches of the edge length and track nodes, which can counter each other's weaknesses. We do so by assigning a cost length only to the edge that connects a branch to its branching node. We give this edge the capacity of the total track length, expressed in the number of train carriages that fit. This model is more formally defined in the next section.

5-1-7 The proposed model

We formulate an extension of the PMTAD which includes the train and track lengths. Pebble Motion problem on a Tree with Arrival, Departure, and Length inclusion (PMTADL) defines this extension and assigns a capacity to the ancestor edges of branching nodes.

Pebble Motion on a Tree with Arrival, Departure, and Length inclusion (PMTADL)

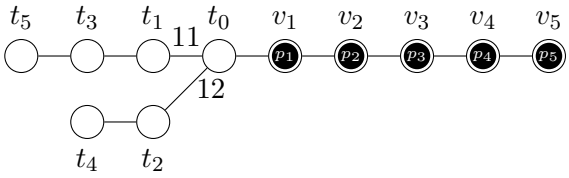
Input: $I = (T_L, P, S, D, E_B, \ell, \lambda)$: given is a shunting tree T_L; a set P of n pebbles; an arriving sequence S; a departing sequence D; a set of branch edges E_B which connect a branch to its branching node; a function $\ell(e)$ which returns the length of an edge e; and a function $\lambda(p)$ which returns the size of a pebble p.

Question: Is there a sequence of moves, which each transfers a pebble p from its current position on $v \in T$ to an adjacent unoccupied node $v' \in T$, to transform S to D such that the edge capacity $\ell(e)$ for $e \in E_B$ is not exceeded by the sum of the sizes of the pebbles parked in the branch of this branching edge e?

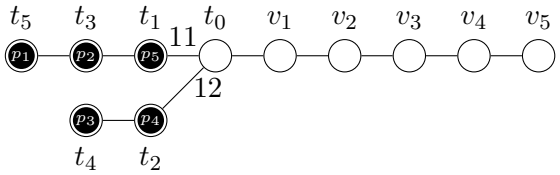
A visualization of this model is given in Figure 5-5. This shows a scenario similar to Figure 5-3, however, the node sizes are now combined in one track length. We consider a similar set of pebbles with given lengths, although in this case, the sizes of pebbles p_1 and p_2 are interchanged. In Figure 5-5, this does not influence the feasibility, because it does not matter whether the larger pebble parks on node t_5 or node t_3 since these nodes are on the same track. In Figure 5-3, this scenario would have been infeasible. Nevertheless, the main issue that was also shown in Figure 5-3 still exists in Figure 5-5. Pebble p_3 and p_4 both have a size of six, and neither of them can thus fit on the branch (t_1, t_3, t_5) with two other pebbles, so they both have to park in the branch (t_2, t_4). However, they cannot be in the same toset of the partition because they are switched in the departure sequence compared to the arriving sequence, so the scenario is infeasible. Since the track lengths represent the actual sizes of the shunting yard, this example shows a real-world scenario where the track length influences the feasibility and should thus also be reflected by the model.

Pebble	Type
p_1	ICM-IV
p_2	ICM-III
p_3	VIRM-VI
p_4	SLT-VI
p_5	SLT-IV

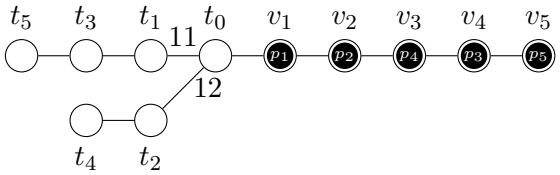
(a) Train unit compositions



(b) Arriving sequence



(c) Intermediate configuration



(d) Departing sequence

Figure 5-5: Example of influence of edge length

In the rest of this chapter, we discuss the track lengths of an edge e_b where $b \in B(T)$ is a branch of the tree T. So, in Figure 5-5, we have two branch edges $e_{b_1} = \{t_1, t_0\}$ and $e_{b_2} = \{t_2, t_0\}$, such that $E_B = \{e_{b_1}, e_{b_2}\}$ where $\ell(e_{b_1}) = 11$ and $\ell(e_{b_2}) = 12$.

5-2 Extending the partition approach

As discussed in Subsection 5-1-7, the PMTADL is an extension of the PMTAD. This means that the former is a more general version of the latter because the PMTAD is a special case of PMTADL where $\lambda(p) = 1, \forall p \in P$ and $\ell(e_t) = 1, \forall t \in T$ where the branches are just single nodes, so we define the lengths on single edges. Thus, the length of a branch $b \subset B(T)$ was the number of nodes in that branch. With this knowledge, we can extend the theory from Chapter 4 to the new problem because everything that applies to the theory on the PMTAD also applies to the PMTADL.

As the PMTADL is an extension of the PMTAD, we formulate the Partition for a Pebble Sequence

on a Tree with Length inclusion (PPSTL) as an extension of the Partition for a Pebble Sequence on a Tree (PPST) using an updated definition of Pairwise comparable as formulated in Definition 5-2.1.

Definition 5-2.1 (Pairwise comparable in length) A branch set $B(T)$ and partition $\Pi(S)$ are **pairwise comparable in length** if we can create an ordered list of both sets in the same order, either both $\geq$ or both $\leq$, such that:

i) there are at least as many branches as tosets: $|\Pi(S)| \leq |B(T)|$,

ii) there are enough nodes in each branch for the number of pebbles: $|\pi_i| \leq |b_i|, 1 \leq i \leq |\Pi(S)|$, and

iii) the branch capacity is not exceeded: $\sum_{p \in \pi_i} \lambda(p) \leq \ell(e_{b_i}), 1 \leq i \leq |\Pi(S)|$.

The tosets of a partition can be pairwise comparable with either individual branches $b_i \in B(T)$ or with disjoint subsets of branches $B_i \subset B(T) \ni \forall B_j \neq B_i \subset B(T) : B_i \cap B_j = \emptyset$.

We construct a reduction from the known $\mathcal{NP}$-*hard* Classic Partition Problem (PP) [2][Problem SP12] to the PPSTL to show that the latter is $\mathcal{NP}$-*complete* (Theorem 5-2.1). Just like in the original Pairwise comparable definition, we can compare the tosets to either individual branches $b \in B(T)$ or disjoint subsets of branches $B_i \subset B(T)$.

Partition for a Pebble Sequence on a Tree with Length inclusion (PPSTL)

Input: $I = (B, P, S, D, \ell, \lambda)$: given is a set of branches $B(T)$ based on a shunting tree T_L; a set P of n pebbles; an arriving sequence of pebbles S; a departing sequence D; an edge length function ℓ; and a pebble size function λ.

Question: Is there a valid Partition $\Pi(S)$ into tosets $\Pi_1, .., \Pi_m$ such that the tosets of $\Pi(S)_{\geq}$ and the individual branches of $B(T)_{\geq}$ are Pairwise comparable in length?

Classic Partition problem (PP)

Input: $I = (X)$: given is the multiset X of positive integers.

Question: Is there a partition of X into two disjoint subsets X_1 and X_2 such that the sum of the numbers in X_1 equals the sum of the numbers in X_2?

We note that by this definition of the PP, there can only be a solution to the PP if the $\sum(X)$ is even, otherwise, no two subsets can be the same size.

Theorem 5-2.1 (PPSTL is NP-complete) The Partition for a Pebble Sequence on a Tree with Length inclusion problem is $\mathcal{NP}$-*complete*.

Proof. PPSTL $\in \mathcal{NP}$: Given a solution $\Pi(S)$ to an instance $I' = (B, P, S, D, \ell, \lambda)$ of the PPSTL, we can establish whether the solution is a valid solution of the PPSTL by checking whether all tosets are disjoint, their union equals P, the tosets respect a total order, there are at least as many branches as tosets, there are at least as many nodes in a branch as pebbles in its pairwise compared toset, and the sum of the pebble sizes never exceeds the branch edge's length. Since all these checks can be verified in at most n steps, this process can be done in $O(n)$ time, so we conclude the PPSTL $\in \mathcal{NP}$.

PPSTL $\in \mathcal{NP}$-*hard*: Given an instance $I = (X)$ of the Classic Partition Problem, we construct an instance $I' = (B, P, S, D, \ell, \lambda)$ of the PPSTL by creating 2 branches in B with each $n-1$ nodes and a size $\ell(b) = \frac{\sum X}{2}$. Then, we create a pebble $p_x \in P$ with size $\lambda(p_x) = x$ for every element $x \in X$ and add all pebbles to the sequences S and D in non-descending order: $S(p_x) < S(p_y), D(p_x) < D(p_y) : \forall x, y \in X \ni x < y$.

All these steps can be done in $O(n)$ time, so the reduction is polynomial. We now prove that we can answer the Classic Partition Problem question with *yes* if ($\Longrightarrow$) and only if ($\Longleftarrow$) we can answer the PPSTL question with *yes*.

Proof of ($\Longrightarrow$): Suppose we can answer the Classic Partition Problem question with *yes*, then there is a partition of the items in X over the subsets X_1 and X_2 such that $|X_1| = |X_2|$. Then, we take the items that are assigned to X_1 and put all the pebbles associated with these items in one toset. We do the same for the set X_2. Because the Classic Partition Problem finds a partition over disjoint subsets, we know that the tosets are disjoint (condition i of Partition). Because the partition must assign each item in X to either X_1 or X_2, we know that all pebbles are in a toset (condition ii of Partition). Because the pebbles were added to S in sequential order of item size, this order is still maintained in the tosets (condition iii of Partition). We have created two tosets and there are two branches in the created instance I', so $|\Pi(S)| \leq |B|$ (condition i of Pairwise comparable in length). Because there are $n-1$ nodes in each branch, we know that there are always enough nodes in the branch to fit the number of assigned pebbles, because there must always be at least one node in the other branch to ensure that the sums of X_1 and X_2 are still equal (condition ii of Pairwise comparable in length). Since the sums of the items in X_1 and X_2 are equal and there are two branches with both a capacity of half the sum of all numbers of X, we know that the capacity of neither of the branches is exceeded by the sum of its elements and thus the sum of the pebbles in that toset (condition iii of Pairwise comparable in length). Therefore, the created partition is valid and pairwise comparable in length with B, so we can answer the PPSTL question with *yes*.

Proof of ($\Longleftarrow$): Suppose we can answer the PPSTL question with *yes*, then there is a partition of pebbles into tosets $\Pi_1, .., \Pi_{|B|}$ which is Pairwise comparable in length with the individual branches of B. By construction, we know there are $|B| = 2$ branches, with each a size of $\frac{\sum X}{2}$. So, both tosets contain pebbles whose sizes sum up to exactly $\frac{\sum X}{2}$. By construction, we know that each pebble with a size λ was created from the element λ. Given that there are $|\pi_1|(\text{resp}, |\pi_2|)$ pebbles assigned to a toset $\pi_1(\text{resp}, \pi_2)$, we can take the $|\pi_1|(\text{resp}, |\pi_2|)$ elements from X that correspond to these pebble sizes and add them to the subset of $X_1(\text{resp}, X_2)$. Because the edge capacity may not be exceeded by the pebble sizes we know that there are no more elements than those that sum up to $\frac{\sum X}{2}$, although there may be empty nodes within at least one of the branches of B. Because the tosets of a partition $\Pi(S)$ are disjoint, each element of X can be in X_1 or X_2 but not both. So, we conclude that we can create a solution to the PP from $\Pi(S)$ and can thus answer the PP question with *yes*. □

We have shown that PPSTL is an $\mathcal{NP}$-*complete* problem, and thus we cannot determine whether a scenario is feasible in polynomial time (unless $P = NP$). We can alter Algorithm 4-1, which solves the PPST problem, to use the updated definition of Pairwise comparable in length to solve the PPSTL problem, though this will not be an algorithm with polynomial runtime.

Chapter 6

Including the matching subproblem

As we discussed in Chapter 3, the matching subproblem is one of the differences between the Pebble Motion problem (PM) and the Train Unit Shunting Problem (TUSP). The NS has different train types as we saw in Figure 5-1. For the train schedule, however, it does not matter whether a specific train of type ICM-III or a different train of the same type and length is used to operate a route in the schedule. Therefore, we consider pebbles to be of a certain type such that only the types need to be matched together, not specific pebbles. In this chapter, we extend the Pebble Motion problem on a Tree with Arrival and Departure (PMTAD) with the addition of matching pebbles. We return to the assumption of No matching and now consider a set of types which the pebbles can be. As such, we no longer match exact pebbles in the arriving and departing sequence but instead match a pebble of a certain type to a pebble of that same type in the departing sequence. However, in this chapter, we do not consider train and track lengths, and thus assume that each pebble has a size of one and each branch has a size equal to the number of nodes it contains. Figure 6-1 shows an example of this idea, where we consider three different types of pebbles: blue, green, and red.

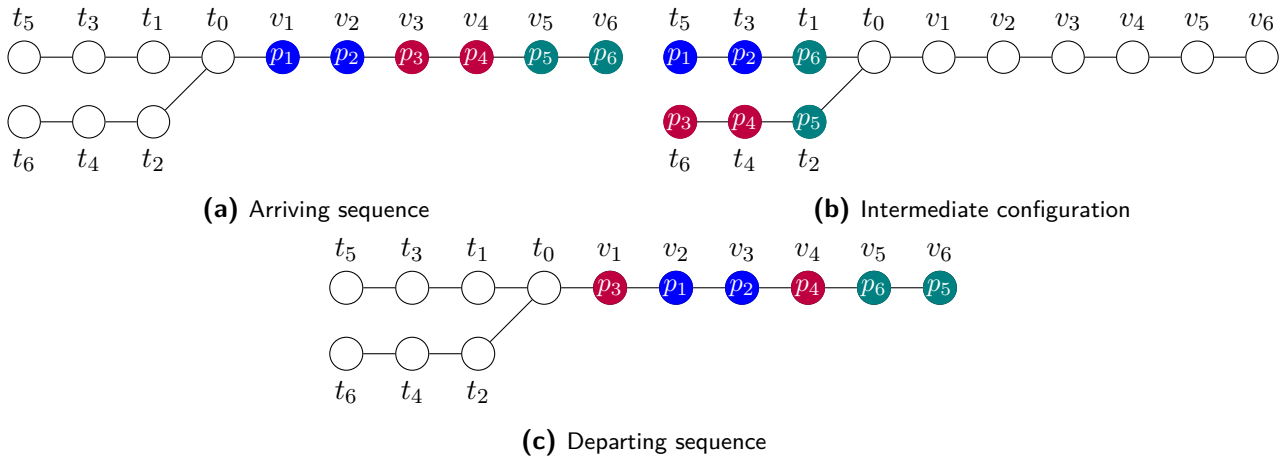


Figure 6-1: Example of matching three different train unit types

In this example, it does not matter whether the green pebble p_5 or p_6 takes the position on v_6 in the departing seqeunce as long as it is a green pebble. In fact, we can realize different departing sequences that consider the individual pebbles which all respect the type of the pebble, in this case $D_1 = (p_3, p_1, p_2, p_4, p_5, p_6)$ and $D_2 = (p_3, p_1, p_2, p_4, p_6, p_5)$ are both possible.

6-1 The new model

We now give a formal definition of the extended problem as the Pebble Motion problem on a Tree with Arrival, Departure, and Color matching (PMTADC).

Pebble Motion on a Tree with Arrival, Departure, and Color matching (PMTADC)

Input: $I = (T_L, P, S, D, \gamma)$: given is a shunting tree T_L; a set P of n pebbles; an arriving sequence S of colors; a departing sequence D of colors; and a function $\gamma(p)$ that returns the color of pebble p.

Question: Is there a sequence of moves, which each transfers a pebble p from its current position on $v \in T$ to an adjacent unoccupied node $v' \in T$, to transform S to D such that the color $\gamma(p)$ of a pebble matches the colors of the sequences?

This is an extension of the PMTAD and instances of the former problem are instances of the latter problem where each pebble $p \in P$ has a unique color. We introduce a new Color partition.

Definition 6-1.1 (Color partition) A **color partition** $\Pi_C(S, D)$ of sequences S and D is the set $\Pi_C(S, D) = \{\pi_i\}$ such that

i) all tosets are disjoint: $\pi_i \cap \pi_j = \emptyset$, $\forall \pi_j \in \Pi(S) : \pi_i \neq \pi_j$,

ii) the union of the tosets is the arriving sequence: $\bigcup \pi_i \in \Pi(S) = P$, thus $\sum_{\pi_i \in \Pi(S)} |\pi_i| = n$, and

iii) the tosets respect the order of the colors: $S(\gamma(p_k)) < S(\gamma(p_l))$, $1 \leq k \neq l \leq |\pi_i|$, $\forall \pi_i \in \Pi(S)$.

Tosets still contain specific pebbles, so the notion of disjoint sets and the union is still the same as in the original partition. However, more partitions are symmetric because we can create different partitions that each have different pebbles in a certain toset though their colors are the same. If we take the example from Figure 6-1, we have the partition $\Pi_C(S, D) = \{(p_1, p_2, p_6), (p_3, p_4, p_5)\}$, however, the partition $\Pi_C(S, D) = \{(p_1, p_2, p_5), (p_3, p_4, p_6)\}$ corresponds to the same assignment of colors to the tosets. We say these two partitions are equivalent.

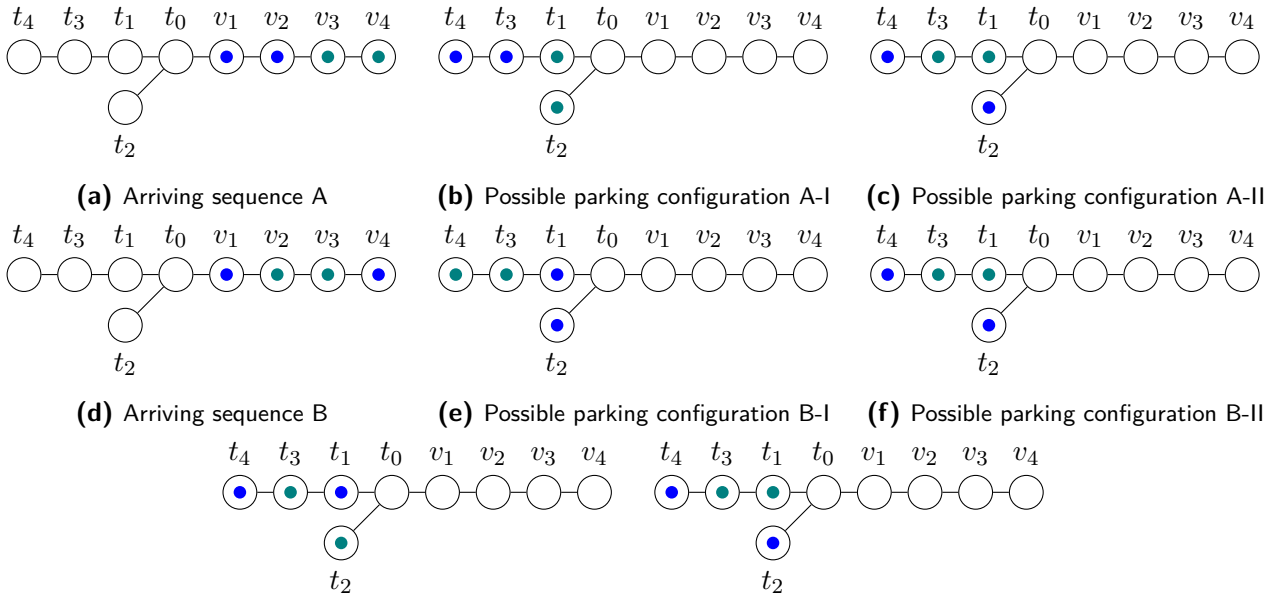
We now consider the number of different color partitions for a given arriving and departing sequence of a certain length. Consider again the number of different possibilities $p_k(n)$ to distribute n pebbles over k non-empty tosets without symmetry, as we showed in Table 4-3. As an example, we take $n = 4$ and $k = 2$, use two colors blue (b) and green (g), and show the different color distributions in Table 6-1. Here, we only look at the distribution of the colors and for now take the sequences to be $S = D = (p_1, p_2, p_3, p_4)$, so all partitions are possible. Of course, the color distribution of $\{b, b, b, g\}$ is similar to $\{g, g, g, b\}$ but reversed, so we leave these symmetries out. Just like in Table 4-3, the partitions of $\{(g, b), (b, g)\}$ and $\{(b, g), (g, b)\}$ are considered equivalent.

$p_{kC}(n)$	|Tosets|	C	Π_C possibilities
6	$\{3, 1\}$	$\{b, b, g, g\}$	$\{[(b, b, g), (g)], [(b, g, b), (g)], [(g, b, b), (g)], [(b, g, g), (b)],$ $[(g, b, g), (b)], [(g, g, b), (b)]\}$
4	$\{3, 1\}$	$\{b, g, g, g\}$	$\{[(b, g, g), (g)], [(g, b, g), (g)], [(g, g, b), (g)], [(g, g, g), (b)]\}$
4	$\{2, 2\}$	$\{b, b, g, g\}$	$\{[(b, b), (g, g)], [(b, g), (b, g)], [(g, b), (b, g)], [(g, b), (g, b)]\}$
1	$\{2, 2\}$	$\{b, g, g, g\}$	$\{[(b, g), (g, g)], [(g, b), (g, g)]\}$

Table 6-1: Possible color distributions of tosets in a partition of size k

However, knowing the general number of possible partitions does not tell us everything. Because there is a lot of symmetry involved in this problem it would be more interesting to know the possible departure sequences given an arriving sequence and branch set. Consider for example Figure 6-2. There are two different possibilities of parking the pebbles, leaving out symmetric cases. In Figure 6-2b, there are three different departure sequences possible, which correspond to the three positions at which the pebble parked

on t_2 can insert itself: $(g, b, b, g), (g, g, b, b), (g, b, g, b)$. For Figure 6-2c, the same options are possible but with green and blue inverted. However, with a different arrival sequence, like in Figure 6-2d, there are four different ways of parking the four pebbles: every pebble can park on node t_2 to result in a different configuration.



(a) Arriving sequence A **(b)** Possible parking configuration A-I **(c)** Possible parking configuration A-II

(d) Arriving sequence B **(e)** Possible parking configuration B-I **(f)** Possible parking configuration B-II

(g) Possible parking configuration B-III **(h)** Possible parking configuration B-IV

Figure 6-2: Possible partitions for two arriving sequences

This example showed the numerous different partitions that are possible now that pebbles are no longer matched uniquely to their departure order. Next, we consider the effect this has on the complexity of the PMTADC.

6-2 Extending the partition approach

As we established, there are many ways of parking the colored pebbles. We are mostly interested in finding a partition that provides a feasible solution. We now extend the Partition for a Pebble Sequence on a Tree (PPST) problem to the Partition for a Pebble Sequence on a Tree with Color matching (PPSTC) using again the Color partition and Pairwise comparable definitions.

Partition for a Pebble Sequence on a Tree with Color matching (PPSTC)

Input: $I = (B, P, S, D, \gamma)$: given is a set of branches $B(T)$ based on a shunting tree T_L; a set P of n pebbles; an arriving sequence of pebble colors S; a departing sequence of pebble colors D; and a function $\gamma(p)$ that returns the color of pebble p.

Question: Is there a valid Color partition $\Pi_C(S, D)$ into tosets $\Pi_1, .., \Pi_m$ such that the tosets of $\Pi_C(S, D)_{\geq}$ and the individual branches of $B(T)_{\geq}$ are Pairwise comparable?

First, we consider a special case of the PPSTC, the k-PPSTC, which has k colors and k branches. If we can park all pebbles of the same color in one branch and have a branch for each color, then the instance is always feasible. We show this variant is in $\mathcal{P}$ in Theorem 6-2.1. This special case provides an upper bound to the number of required branches for an instance of PPSTC.

Theorem 6-2.1 (k-PPSTC is in P) The Partition for a Pebble Sequence on a Tree with Color Matching problem is in $\mathcal{P}$ if there are k different colors of pebbles and at least $k = |B(T)|$ branches.

Proof. Given is an instance $I = (B, P, S, D, \gamma)$, such that there are k different colors, i.e. $k = |\{\gamma(p) \neq \gamma(q) \,|\, \exists p \in P, \forall q \in P\}|$. Suppose that $|B| \geq k$ and for each branch $b_i \in B$, $1 \leq i \leq k$

(or $B_i \subset B$) we know that $|b_i| = \sum_{\gamma(p)=i} 1$. Then, we can park all pebbles of the same color on one branch, for each of the k colors. We can always park the pebbles of an arriving sequence to match these colors because each incoming pebble can park in the designated branch. Furthermore, we can compute any departing sequence, with this color distribution, because some pebble of the correct color can always depart. If there are more than k branches, we can also combine branches into subsets B_i such that each subset provides enough space for all pebbles of a certain color to fit. □

Just like for the PPST, we have found a case that is polynomially solvable but we still expect the general problem to be more complex. Again, we have tried to prove the PPSTC to be $\mathcal{NP}$-*hard*, however, so far we have not been able to do so. The most promising attempt was from the 3D Perfect matching problem [2][Problem SP1], but the proof was not complete.

Chapter 7

Conclusion

In this thesis, we study the feasibility of the Train Unit Shunting Problem (TUSP). This is still an open research area as there does not exist an approach to determine the feasibility of TUSP scenarios. The current algorithms in use by NS focus on solving instances without knowing whether a solution exists in the first place, so they might use a lot of computing power to try and find a good solution to an infeasible scenario. We identified the most important elements of the problem that must be considered, and included these in a variant of the Pebble Motion problem (PM). The PM is known to be decidable in linear time, so this problem forms an interesting foundation for our feasibility study of the TUSP. For each of the created problem variants, we discussed the complexity and studied different solution approaches to determine the feasibility of TUSP instances. Each of the different problems considered different aspects of the TUSP and their solution approaches had their own drawbacks.

7-1 Problem relations

We now summarize the different approaches, and their relations are visualized in Figure 7-1. In this thesis, we are concerned with the TUSP, which is known to be $\mathcal{NP}$-*hard*. We initially created the Pebble Motion problem on a Tree with Arrival and Departure (PMTAD) as a relaxation of the TUSP. This is a variant of the PM that models the arrival and departure of pebbles, where the graph is assumed to be a Shunting tree. This restriction of the general PM adds the constraint of no reallocation. Without this constraint, the PMTAD is a special case of the PM where the graph has the specific structure of a Shunting tree. This problem would then be in $\mathcal{P}$, but we do not include this problem in this thesis because we do not consider reallocation. The PMTAD was studied with different approaches.

To determine what sequences are feasible, we defined a Partition of a sequence, which gives the totally ordered sets of pebbles that are in the correct order in the departing sequence compared to the arriving sequence. Given a valid Partition of a sequence, the Branch Set for Partition Problem (BSPP) determines in polynomial time whether a Branch set of a tree can match this partition. The BSPP was found to be in $\mathcal{P}$ although the approach does not always find a feasible solution if one does exist. On the other hand, given a branch set of a tree, the Partition for a Pebble Sequence on a Tree (PPST) problem tries to find a partition to match this branch set. Although we suspect this problem to be $\mathcal{NP}$-*hard*, we have thus far been unable to prove this, regardless of several attempts. There exist sub-exponentially many partitions for a given sequence and we have also not found an approach that efficiently traverses these. In Algorithm 4-1, a Directed Acyclic Graph (DAG) of the sequence is constructed and the algorithm then searches for a path cover that can represent the partition. However, in the worst case, all partitions still have to be considered, and the algorithm thus has an exponential worst-case runtime.

Besides the general PMTAD, we created two extensions that increase the affinity towards the TUSP.

First, we extended the model to the Pebble Motion problem on a Tree with Arrival, Departure, and Length inclusion (PMTADL), where train and track lengths were included to model the real-world scenario more closely. Since trains have different lengths, depending on their type, we need to ensure that the trains fit on the tracks, which can differ in length as well. Then, we extended the PMTAD such that each pebble has a color to represent the different train types, which resulted in the Pebble Motion problem on a Tree with Arrival, Departure, and Color matching (PMTADC). In this variant, we make sure that pebbles of the right color are matched in the departing sequence, compared to the arriving sequence, instead of the individual pebbles.

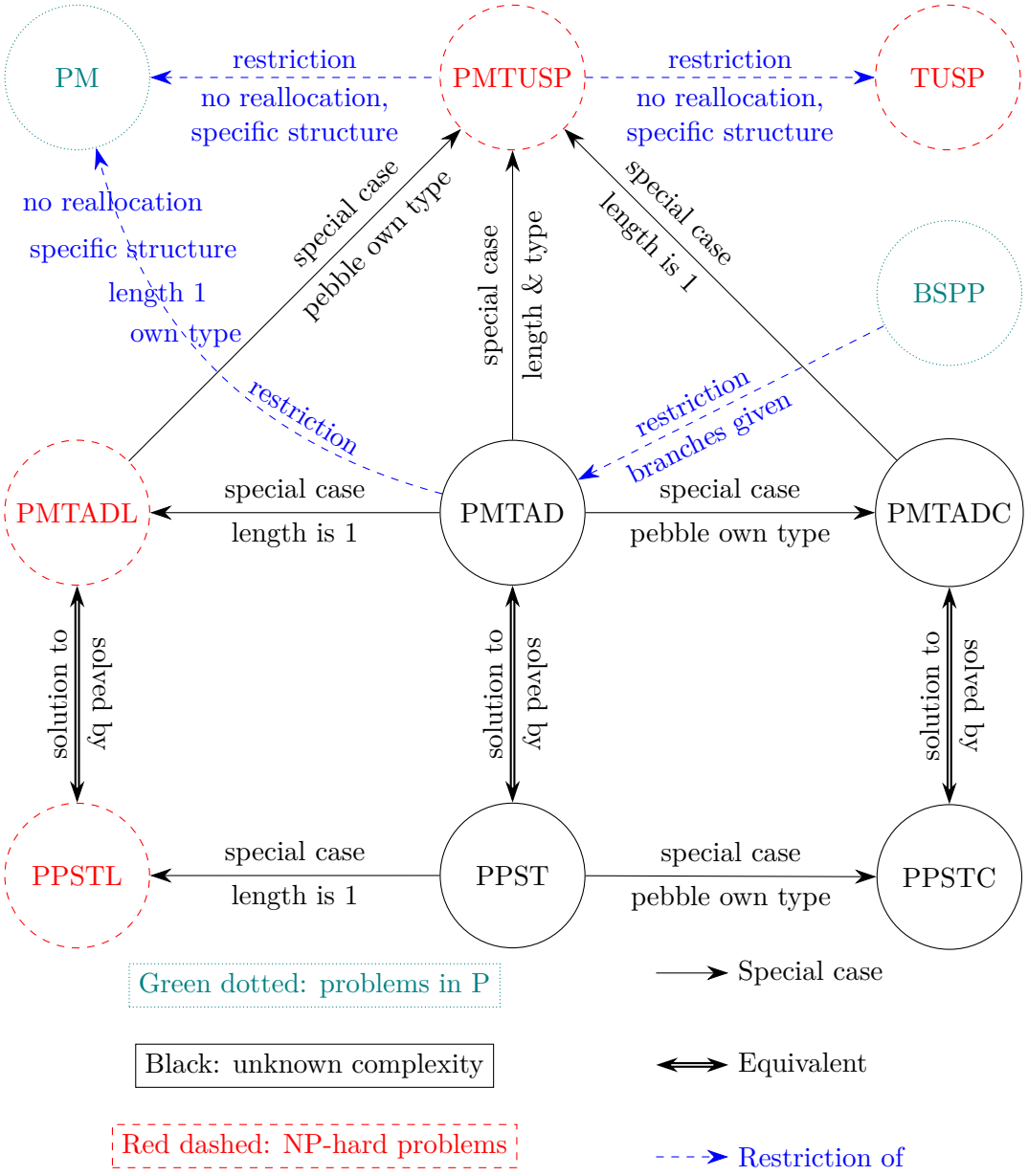
Both extensions have their respective extensions of the PPST problem: the Partition for a Pebble Sequence on a Tree with Length inclusion (PPSTL) problem and the Partition for a Pebble Sequence on a Tree with Color matching (PPSTC) problem. The first was proven to be $\mathcal{NP}$-*hard*, but the complexity of the latter remains to be shown. Finally, we can combine the two extended problems, PMTADL and PMTADC, to the Pebble Motion for the Train Unit Shunting Problem (PMTUSP).

Pebble Motion for the Train Unit Shunting Problem (PMTUSP)

Input: $I = (T_L, P, S, D, E_B, \ell, \lambda)$: given is a shunting tree T_L; a set P of n pebbles; an arriving sequence S; a departing sequence D; a set of branch edges E_B which connect a branch to its branching node; a function $\ell(e)$ which returns the length of an edge e; a function $\lambda(p)$ which returns the size of a pebble p; and a function $\gamma(p)$ which returns the color of a pebble p.

Question: Is there a sequence of moves, which each transfers a pebble p from its current position on $v \in T$ to an adjacent unoccupied node $v' \in T$, to transform S to D such that i) the edge capacity $\ell(e)$ for $e \in E_B$ is not exceeded by the sum of the sizes of the pebbles parked in the branch of this branching edge e, and ii) the color $\gamma(p)$ of a pebble in the sequence matches the color of the sequence at that position?

Since the PMTADL was shown to be $\mathcal{NP}$-*hard*, it is trivial that the PMTUSP is too. This problem gives the main variant of the general TUSP which includes all major components, as identified in Chapter 3, to closely relate it to the real-world scenario, while also being abstract enough to apply the theory from this thesis to it. By including the arrival and departure of pebbles in the graph, allowing pebbles to have different lengths, and matching the colors of pebbles in the arrival and departure sequences, we have extended the PM to study the feasibility of the TUSP. Although the original PM is polynomially solvable, we have found that the PMTUSP is not.



(a) Relations between the studied problems

Short	Name	Page	Short	Name	Page
PM	Pebble Motion problem	15	PMTUSP	Pebble Motion for the Train Unit Shunting Problem	56
TUSP	Train Unit Shunting Problem	15	BSPP	Branch Set for Partition Problem	27
PMTAD	Pebble Motion on a Tree with Arrival and Departure	18	PPST	Partition for a Pebble Sequence on a Tree	31
PMTADL	Pebble Motion on a Tree with Arrival and Departure, and Length inclusion	47	PPSTL	Partition for a Pebble Sequence on a Tree with Length inclusion	49
PMTADC	Pebble Motion on a Tree with Arrival and Departure, and Color matching	52	PPSTC	Partition for a Pebble Sequence on a Tree with Color matching	53

(b) Problem abbreviations

Figure 7-1: Overview of studied problems

7-2 Contributions

The most important finding in this work is that the PMTADL is $\mathcal{NP}$-*hard*. This result allows us to extend this complexity result to establish that the very simplistic variant of the TUSP, the PMTUSP, is not decidable in polynomial time. This result provides researchers at the NS the insight that scenarios, where lengths are included, cannot be decided in polynomial time. So, for problem instances with the lengths included, there is no quick way to determine the feasibility. Since a fast solution will thus never be completely optimal, resources at NS can rather be spent on new ways of solving the problem. Since determining the feasibility is $\mathcal{NP}$-*hard*, an approximation method could be used, or the feasibility question could be answered while trying to find an (optimal) solution.

Another contribution is Algorithm 4-1, presented in Subsection 4-7-3, which solves the PPST by using the sequence DAG to find a path cover that represents a valid partition. For roughly $n \leq 25$, the algorithm can determine whether the given instance is feasible or not within five minutes. This approach is complete and will thus always give a correct response on the feasibility if it finishes before the timeout. Since real-world scenarios at this time usually consider about 20-30 trains, this can still be useful in practice when constructing a daily plan.

We established several conditions that can be used to quickly determine the (in)feasibility of a problem instance. These conditions were also partially used in Algorithm 4-1 and are presented in Section 4-3. Based on the number of pebbles and the capacity of the tree, simple cases can be ruled out. For example, if there are fewer empty nodes than there are pebbles, then the scenario is infeasible. Or, if there are as many branches as pebbles, then the scenario is always feasible, independent of the sequences.

Finally, we have identified two special cases of the mentioned problems, for which we have shown that the problem is in the class $\mathcal{P}$. We restricted the PPST problem to a special case, the 2-PPST (Theorem 4-6.1), where there are m branches of size two and $n = 2m$ pebbles. Furthermore, the k-PPSTC (Theorem 6-2.1) is a special case of the PPSTC where there are k different colors of pebbles used, and there are k branches in the tree, such that for each color there is a branch with a capacity that is at least equal to the number of pebbles of that color.

7-3 Future work

The most important work that is yet to be completed is to determine the complexity of the PPST and the PPSTC. We suspect both to be $\mathcal{NP}$-*hard* and although proof of this would not lead to any new contributions, it would give the guarantee that any further attempts to exactly solve the problem in polynomial time can be ignored. On the other hand, if it can be shown that the PPST is in $\mathcal{P}$, then this would be a very interesting result that could provide fast feasibility results of the TUSP and help to solve its instances.

Furthermore, other approaches for studying the feasibility of the TUSP can still be explored. The use of arrival and departure sequences might give rise to the intuition of using group theory to express the permutation between S and D. This approach was in fact the first instinct for this thesis as well. The idea was to find the cycles in the permutation and use these to determine the necessary tree structure for a feasible solution. However, it resulted in many subcases which clouded the overview of how to solve general problem instances. Furthermore, the theory was not complete as it lacked a definition of the direction of a cycle and the order in which multiple cycles must be applied. In the end, we decided not to continue this approach, and instead, we looked into the definition of a partition. The original findings of the cycle approach can be found in Appendix C which can be read as a continuation of the introduction of Chapter 4.

In this thesis, all the work was focused on trees, as they provide a simple and recursive structure to form a solid base for the theory. However, there are also shunting yards that do not have a tree struc-

ture, see Appendix A, so further research could be done to extend this work to general graphs. Since general graphs are allowed to contain cycles, there is no longer one way to exit a shunting yard from a given parking location. This enlarges the search space for feasible solutions, because a blocked pebble may be able to depart from a different route. Since this will result in more feasible solutions, and thus also in more possible partitions, the complexity of the problem is likely to be equal to or more difficult than the tree version.

Although different train lengths are included in this work, trains are always considered to stay together. In real-world scenarios, trains can be split into different train units, and train units can be combined to form new train compositions. Allowing different train units to make up the required compositions, as long as the train types match, will provide a more realistic approach.

The conditions for feasibility or infeasibility that we established throughout this thesis are not a complete approach for determining all problem instances. However, these conditions can be used in combination with different approaches. This was demonstrated in Algorithm 4-1, but other applications can be seen in Branch-Cut-and-Price algorithms, for example, where the conditions can be used to define cuts. Furthermore, these conditions might apply to subcases of the problem. By dividing a problem instance into multiple subproblems, each one can be tackled individually, and especially infeasible subproblems can be caught quickly. Such an approach would require that the subproblems are created in such a way that an infeasible subproblem guarantees an infeasible instance overall. For example, by assigning trains that can definitely not park on the same track to a different part of the shunting yard such that parts of the shunting yard form different subproblems. Otherwise, a feasible solution could be ruled out if those trains could be parked on the same track.

One of the assumptions that we stated in Chapter 4 was prohibiting mixed traffic (Assumption 4-0.2). However, in real-world scenarios, it happens that some train has to depart before all trains have entered. Therefore, considering mixed traffic is still an interesting direction. With this extension, the notion of one track used for arrival and departure (Assumption 4-0.1) is also no longer applicable and a more general scenario with several tracks should be considered. Mixed traffic will probably make the problem more complicated because routing and parking are done in parallel, so some parking locations might be unavailable because another train is routed for departure again. The addition of extra arrival/departure tracks will also open up the search space for more feasible solutions, which can add to the complexity of the problem.

Another assumption ruled out the possibility of reallocation (Assumption 4-0.3). Therefore, in this thesis, we only considered finding a parking configuration such that the sequences can be realized. However, with the option of reallocation, the feasible solution space becomes much larger because a single pebble that is blocked might be unblocked when we reallocate the blocking pebble. Reallocation was not considered in this thesis because it opens up the solution space to such an extent that defining conditions to determine the feasibility would require a completely new approach, and we started with the most simple solution to define the feasibility. Although reallocation would more closely resemble real-world scenarios, the work in this thesis still provides relevant results as allowing reallocation will always result in more costly shunting plans, because drivers have to make extra moves to reallocate a train. So, solutions without reallocation can be favorable as they limit the costs. However, reallocation is still an interesting direction for future research.

Appendix A

Shunting yard types

The NS has two main types of shunting yards: the carousel and the shuffleboard. The first name implies that there is some sort of cycle, and the graph that represents this shunting yard indeed has cycles. An example of a carousel shunting yard layout is shown in Figure A-1. We clearly see that cycles exist in this yard and there are multiple ways in and out of the shunting yard. The shuffleboard on the other hand is named after the game shuffleboard because all trains are parked in a corridor, and the graph of such a shunting yard can thus be represented by a tree structure. Figure A-2 gives an example of a shuffleboard layout. Trains can now enter only from one side, park on one of the tracks, and leave again on the same side.

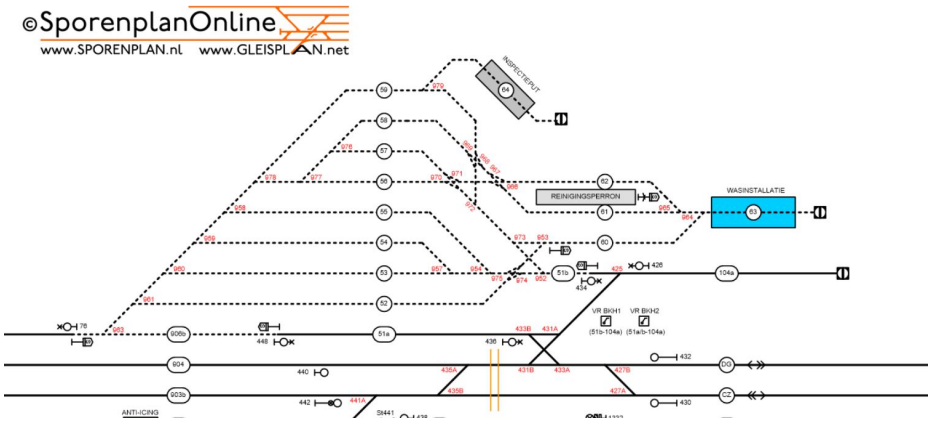


Figure A-1: Carousel shunting yard

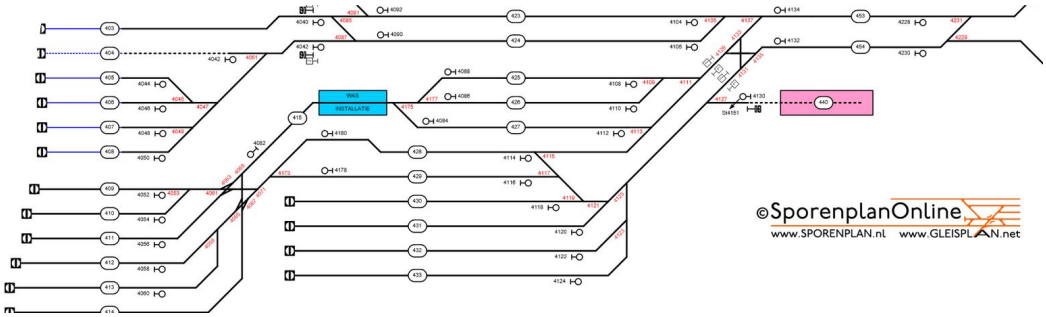


Figure A-2: Shuffleboard shunting yard

Appendix B

Algorithm pseudocode

Here, we give the pseudocode of three algorithms that were mentioned in Chapter 4 but were not very relevant to include within the chapter.

Algorithm B-2 creates a partition by using the longest path approach on a Directed Acyclic Graph (DAG). First, we ConstructDAG for both the negative and positive edges (as we discussed in Method 4-3.1). Then, the FillBranches method iterates over the branches, starting with the largest one. Each branch is filled with a toset that is preferably the same size as that branch. These tosets are created by using the longest path in the remaining graph, as edges corresponding with the pebbles that are selected for a toset are removed from the graph. Finally, we check if all pebbles were selected, and if so, a valid partition was found. Otherwise, the feasibility remains unknown. Now, we consider the runtime of the algorithm. First, the depth-first search on a DAG can be done in $O(n+m) = O(n^2)$ time, where n is the number of nodes and m the number of edges [15, p.92-94], and we know that there are $O(n^2)$ edges in the DAG. This procedure is repeated on each iteration of FillBranches. We traverse $|B| = O(n)$ branches, where each might have to traverse over $\ell = |b| = O(n)$ possible tosets for that branch. So, the total complexity is $O(n^3)$.

Secondly, Algorithm B-3 tries to find a branch set to match the given partition. The DFS method uses a depth-first search to establish the order of the nodes in the tree from the root down. This order is then converted to branches in PossibleBranches. These branches are then matched to the partition, which in general finds most of the feasible cases. However, sometimes we need to merge branches to fit all the pebbles. We add a method MergeBranch that can merge two small branches together to park a larger toset. We now analyze the runtime of the algorithm. Again, a depth-first search can be done in $O(n+m)$ time. The PossibleBranches procedure traverses all nodes to determine their subtrees, which is done in $O(n)$ and returns the set *branches* of size $O(nm)$. There are at most n tosets in a partition, so the sorting of this takes $O(n \log n)$, while the sorting of the branch set takes $O(nm \log nm)$. The MergeBranch procedure runs in at most $O(n^2m^2)$, so the complete matching is done in $O(n \cdot nm \cdot n^2m^2) = O(n^4m^3)$.

Finally, in Algorithm B-4, we give an algorithm that runs FindPartition to determine if there exists a partition that fits the given set of branches B. Here, we assume that $|\Pi(S)| = |B(T)|$, so we want the partition to exactly match the branch set. We construct the Integer Linear Program (ILP) as in Equation 4-2, and CheckPartition verifies if we have found a fitting partition with the correct toset sizes. Then, we keep adding the previous solution as a constraint in each iteration if we have not found the solution. Once the ILP returns an infeasible solution, we terminate and conclude that sequence S is infeasible. In the worst case, we have to traverse all possible partitions, of which there may exist exponentially many. Furthermore, solving an ILP also requires an exponential runtime.

Algorithm B-2 Approach for filling the branches based on longest paths

procedure SOLVE(S, B)
  if $|B| = |S|$ **then**
    return TRUE
  if $|S| > \sum_{b \in B} |b|$ **then**
    return FALSE
  $E^+, E^- \leftarrow$ CONSTRUCTDAG(S)
  $P \leftarrow$ ALLPATHS(E^-, S)
  if $|E^-| > 0 \wedge \max_{p \in P} |p| > |B|$ **then** ▷ If longest negative path longer than number of branches
    return FALSE
  return FILLBRANCHES(B, E^+, S)
procedure CONSTRUCTDAG(S)
  $E^+ \leftarrow \emptyset, E^- \leftarrow \emptyset$
  for $i \in [0, n]$ **do**
    for $j \in [i+1, n]$ **do**
      if $S[i] > S[j]$ **then**
        Add edge $(S[i], S[j])$ to E^+
      else
        Add edge $(S[i], S[j])$ to E^-
  return E^+, E^-
procedure FILLBRANCHES(S, B, E)
  Sort B in descending order
  selected $\leftarrow \emptyset$
  $\Pi \leftarrow \emptyset$
  for $b \in B$ **do**
    $P \leftarrow$ ALLPATHS(E, S)
    Add a 1-node path to P for each pebble that is not yet in *selected*
    $\pi \leftarrow \emptyset$
    $\ell \leftarrow |b|$
    while $\pi = \emptyset$ **do**
      if There is a path of length ℓ in P **then**
        $\pi \leftarrow$ random path of length ℓ from P
      else
        $\ell \leftarrow \ell - 1$
    Add π to Π
    Add pebbles in π to *selected*
    Remove all edges from any of the nodes in π from E
  return $S \setminus \Pi = \emptyset$ ▷ Check if all pebbles are in Π
procedure ALLPATHS(E, S)
  $P \leftarrow \bigcup_{p \in S} \mathrm{DFS}(p, \emptyset, [p])$ ▷ Get all paths starting from any node in the $DAG(S)$
  return P
procedure DFS($node, seen, p$) ▷ Depth-first search to find paths down from *node*
  Add *node* to *seen*
  paths $\leftarrow \emptyset$
  for $t \in$ neighbors($node$) **do**
    if $t \notin seen$ **then**
      $p_t \leftarrow [p, t]$
      Add p_t to *paths*
      paths $\leftarrow$ *paths* $\cup$ DFS($t, seen, p_t$)
  return *paths*

Algorithm B-3 Approach for finding the branch set to match partition $\Pi(S)$

```
procedure DFS(node, dfsOrder, position)        ▷ Depth-first search to find paths down from the node
    Add node to dfsOrder
    Increment position
    Store start position for node
    for n ∈ children(node) do
        if n ∉ dfsOrder then
            position ← DFS(n, dfsOrder, position)
    Store end position for node
    return position
procedure PossibleBranches(T)                  ▷ Construct the branches from the dfsOrder
    dfsOrder ← DFS(root(T), [], 0)
    branches ← ∅
    for node ∈ T do
        if |children(node)| > 1 then
            Add branch for each of the children, rooted by that child, to branches
        else if |children(node)| > 0 ∧ |branch rooted by node| > 1 then
            Add branch rooted by node to branches
    return branches
procedure MatchPartitionToTree(T, Π(S))
    branches ← PossibleBranches(T)
    Sort branches in descending order of branch size
    Sort Π(S) in descending order of toset size
    selection ← ∅
    for π ∈ Π(S) do
        found ← FALSE
        for b ∈ branches do
            if |b| ≥ |π| ∧ b ∩ selection = ∅ then        ▷ All branches must be disjoint
                selection ← selection ∪ b
                found ← TRUE
        if ¬found then
            if ¬ MergeBranch(selected, branches, π) then
                return FALSE
    return TRUE
procedure MergeBranch(selection, branches, π)
    for i ∈ range(branches) do
        for j ∈ range(branches) do
            if branches[i] ∩ branches[j] = ∅ then
                branch ← branches[i] ∪ branches[j]
                if selected ∩ branch = ∅ ∧ |branch| ≥ |π| then
                    return TRUE
    return FALSE
```

Algorithm B-4 Approach for finding a partition to match the branch set B

procedure FINDPARTITION(B)
  Construct *variables* and *constraints* according to Method 4-6.1
  $X \leftarrow$ ILP(*constraints*)
  $\Pi, X' \leftarrow$ GETPARTITION(X)
  while $\neg$ CHECKPARTITION(Π, B) **do**
    ▷ The combination of variables that led to Π cannot be used again
    Add $\sum_{x \in variables(X')} x < \sum_{x \in X} x$ to *constraints*
    $X \leftarrow$ ILP(*constraints*)
    if X is infeasible **then**
      return FALSE
    $\Pi, X' \leftarrow$ GETPARTITION(X)
  return TRUE
procedure GETPARTITION(X) ▷ Generate the partition from the ILP solution
  $\Pi \leftarrow \emptyset$
  $X' \leftarrow \emptyset$
  for $p \in S$ **do**
    $\Pi_p \leftarrow \emptyset$
    if $X((s, p_{in})) > 0$ **then**
      $v \leftarrow p_{in}$
      while $v \neq t$ **do**
        for $u \in$ neighbors(v) **do**
          if $X((v, u)) > 0$ **then**
            $v \leftarrow u$
            $X' \leftarrow u$
            $\Pi_p \leftarrow \Pi_p \cup u$ ▷ If u is an out pebble node
      $\Pi \leftarrow \Pi \cup \Pi_p$
procedure CHECKPARTITION(Π, B)
  Sort B and Π according to length of their items
  for $i \in [1, |\Pi|]$ **do**
    if $|\Pi_i| \neq B_i$ **then**
      return FALSE
  return TRUE
procedure ILP(*constraints*) ▷ This runs the ILP with the given constraints
  Run ILP to find a Π **return** Π

Appendix C

Group theory cycle approach

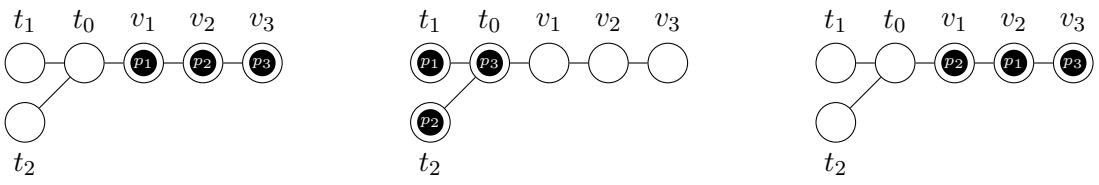
With the model of an incoming path connected to a tree as presented in the Pebble Motion problem on a Tree with Arrival and Departure (PMTAD), we can derive conclusions about the structure and size of the tree that determine the feasibility of an instance of the Train Unit Shunting Problem (TUSP). In this section, we study the feasibility instances of the PMTAD, which we define in terms of feasible sequences. The departing sequence is feasible if there exists a tree T such that the arriving sequences of the pebbles can be permuted to the necessary order for the departing sequence. We use the notion of a feasible sequence to indicate that the related instance of the PMTAD is feasible. The theorems hereafter only apply to a Shunting tree, and more specifically, we only consider binary trees, where each node has no more than two children.

C-1 Single inversion

We start with one of the most simple cases; when there are just two positions that are swapped, which we call an Inversion. This is in fact a simple case of the more complex scenarios that we discuss next (Section C-2). However, it is important to first understand how a single inversion affects the arriving and departing sequences before we consider more complex scenarios.

Definition C-1.1 (Inversion) Given an arriving sequence $S = (s_1, .., s_n)$ and a departing sequence $D = (d_1, .., d_n)$, an **inversion** is a pair of arriving pebbles p_1 and p_2, such that p_1 arrives before p_2, and also departs earlier. More formally, an inversion is a pair of positions $(j, k) : 1 \leq j < k \leq n$, such that $i \notin \{j, k\} : s_i = d_i, s_j = d_k$ and $s_k = d_j$. We refer to an **adjacent inversion** if the two pebbles are located next to each other in the sequence, i.e. $k = j + 1$. In the literature, we also sometimes speak of a **transposition** instead of an inversion [6]. An **exchange** of (j, k) consists of exchanging the pebbles at positions j and k, which resolves the inversion.

To illustrate this definition of an Inversion, see Figure C-1. Pebbles p_1 and p_2 have to be exchanged, to resolve the adjacent inversion $(1, 2)$.



(a) Arriving sequence **(b)** Intermediate configuration **(c)** Departing sequence

Figure C-1: Example of an adjacent inversion

We now derive a theorem on the structure of T to determine the feasibility of a PMTAD instance. First, we consider the following definitions that will help us refer to nodes in the tree and the relations between nodes. The **number of children** of a node t is denoted by $c(t)$. A **branching node** has more than one child, which means that $c(t) = 2$ since we only consider binary trees. Furthermore, a **leaf node** has no children. We define the **ancestors** of a node t, to be all the nodes included in the path from t to the root, starting with the parent node of t. We define the **descendants** of a node t to be all the nodes to which t is an ancestor; so all children, grandchildren, grand-grandchildren, etc.

Furthermore, we use the following two functions for defining the size of a (sub)tree, which will help to determine the necessary tree size for the different scenarios that we consider. The Offspring of a node in a tree is defined by $o(t)$, which are all the descendants of the node t. We use the Width of a tree, denoted by $w(t)$, which counts the number of leaf nodes.

Definition C-1.2 (Offspring) Let $o(t)$ recursively return the offspring, which is the total number of descendants of node t in the tree.

$$o(t) = \begin{cases} 0 & \text{if } c(t) = 0 \\ c(t) + \sum(t' \in c(t))o(t') & \text{else} \end{cases}$$

Definition C-1.3 (Width of a tree) Let $w(t)$ recursively return the width of the tree rooted by node t, which counts the number of leaves.

$$w(t) = \begin{cases} 1 & \text{if } c(t) = 0 \\ \sum_{t' \in c(t)} w(t') & \text{else} \end{cases}$$

These definitions imply the following corollaries on the width of a tree (Corollary C-1.1) and the size of the tree (Corollary C-1.2). The latter is used in all further lemmas and theorems, although we do not explicitly state each time that this condition is necessary to park n pebbles in a tree.

Corollary C-1.1 (Width of branching node) The width of a tree rooted at node t_0 is 2 if and only if there is exactly 1 node t' with $c(t') = 2$ and $\forall t \neq t' : c(t) < 2$.

Proof. Proof of ($\Longrightarrow$): Suppose $w(t_0) = 2$. The,n by the Width of a tree definition, there must be exactly two nodes with $c(t) = 0$. Then, there must be a node t' with $c(t') > 1$, otherwise, the tree is a path, and there is only 1 leaf. Since T is a tree, there are no cycles so $c(t') = 2$, otherwise, there would be more than two nodes with $c(t) = 0$.

Proof of ($\Longleftarrow$): If there is exactly 1 node t' with $c(t') = 2$ and $\forall t \neq t' : c(t) < 2$, there will be exactly two nodes with $c(t) = 0$, since there are no cycles in a tree. By the Width of a tree definition, $w(t_0) = 2$ if there are two nodes with $c(t) = 0$. □

Corollary C-1.2 (Minimal subtree size) Any sequence of length n requires $o(t_0) \geq n - 1$.

Proof. By the proposition of the Minimal tree size, we know that $|T| \geq n$ for $D = S$ to be feasible. Moreover, as all pebbles need to park in the tree T (Property 4-0.1), this is a necessary, though not sufficient, requirement for any D to be feasible.

Using the Offspring definition, we know that the tree T rooted by t_0 has $o(t_0) = |T|-1$ since $o(t)$ counts all descendants of t_0, and not the root itself. Therefore, a sequence of length n requires $o(t_0) \geq n-1$. □

Now, we can determine the necessary conditions of T, to make sure a departing sequence with a single adjacent inversion is feasible. These conditions are defined in Lemma C-1.1, and state that the tree must have a branching node with at least two children, such that the exchange is possible, as demonstrated in Figure C-1.

Lemma C-1.1 (Single adjacent inversion) A departing sequence $D = (d_1, .., d_n)$ with only one adjacent inversion (j, k) is feasible if and only the tree T rooted at t_0 has $w(t_0) \geq 2$ and there is a branching node $t' : c(t') = 2$ such that $o(t') \geq k$.

PROOF. PROOF OF ($\Longrightarrow$): Suppose $D = (d_1, .., d_n)$ has only one inversion (j, k) and is feasible. By Definition C-1.1, we have that $j < k$. By Corollary C-1.1, if $w(t_0) = 2$, then there is exactly one node t' with $c(t') = 2$. If $w(t_0) > 2$, there is more than one node with $c(t') = 2$, and we take t' to be the node with $c(t') = 2$ which is the highest up the tree, i.e. closest to the root t_0. In both cases, t' is the node in the tree closest to t_0 with $c(t) = 2$. Therefore, we can assume that all ancestors of t' form a simple path in which no exchange is possible. So, for the exchange, we can focus solely on the part of the tree rooted by t'. We use a proof by contradiction, either i) $w(t_0) < 2$ or ii) branching node t' has $o(t') < k$. We prove both cases separately to show that both conditions are necessary for a feasible sequence because if either condition is not met the sequence is incompatible.

CASE I: Suppose $w(t_0) < 2$, then we know that $\forall t \in T : c(t) < 2$ (Definition C-1.3). Then, the tree is a simple path. In this case, the only possible departing sequence is the same as the arriving sequence. This is one of the basic cases, demonstrated in Proposition 4-1.1. Since the departing sequence must equal the arriving sequence, there is no exchange possible. However, as we have taken D to have one inversion with S, we have arrived at a contradiction.

CASE II: Suppose $o(t') < k$. Then, the pebbles at positions $s_1, .., s_j$ in the arriving sequence can park in a descendant of t'. The pebble p_1 at position j in the arriving sequence is also able to park in a descendant of t'. However, the pebble p_2 at position k then has to park in t'. However, p_1 has to depart before p_2 (Definition C-1.1). Since p_2 is parked in an ancestor of the parking node of p_1, the exchange is not possible. Thus, p_1 cannot depart before the p_2 and we have arrived at a contradiction. So, we know that the tree T rooted at t_0 must have that $w(t_0) \geq 2$ and there is a branching node $t' : c(t') = 2$ such that $o(t') \geq k$.

PROOF OF ($\Longleftarrow$): Suppose the tree T rooted at t_0 has $w(t_0) \geq 2$ and there is a branching node $t' : c(t') = 2$ such that $o(t') \geq k$. If there is a single adjacent inversion (j, k), the pebble located at position j, which we call p_1, can park in a descendant of one child of t', while the pebble at position k, called p_2, can park in a descendant of the other child of t', so the exchange is possible. As $o(t') \geq k$, all pebbles that arrive before p_1 will park lower down the tree than where p_1 is parked and can thus depart later. So, the departing sequence is feasible.

□

Using Lemma C-1.1, we can derive Corollary C-1.3.

Corollary C-1.3 (Different branches for an exchange) For any inversion to be possible, the two pebbles involved must be able to park in a different branch. So, there must be a branching node t with two children t_1, t_2 that each form a subtree of size at least 1 such that the subtrees rooted by t_1, t_2 can each fit one of the pebbles of the inversion.

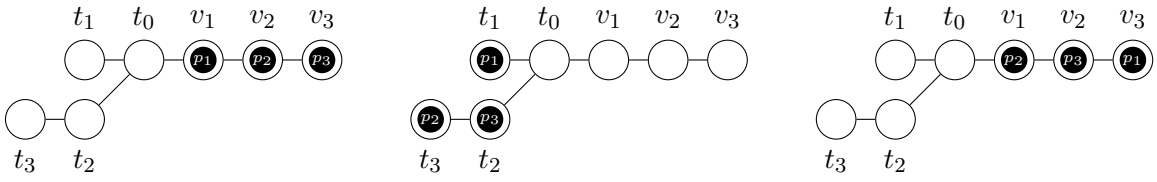
C-2 Cycles in a sequence

We now consider more complex scenarios, which involve multiple inversions between S and D. A departing sequence is a permutation of the arriving sequence and can be defined as a series of exchanges to the arriving sequence. Formally, a **permutation** $\sigma(S, D) = \binom{S=(s_1,..,s_n)}{D=(d_1,..,d_n)}$ defines the shift between two sequences, where $\sigma(s_1)$ defines the position of the pebble at position s_1 in the departing sequence.

An inversion is defined as two positions in S that are different in D (Definition C-1.1). In Section C-1, we studied a single adjacent inversion in a sequence. Now, we look at a **cyclic permutation**, or cycle, which is a subsequence C of D of length r (an r-cycle) such that $\sigma(S_C, D_C) = \binom{j_1,j_2,..,j_{r-1},j_r}{j_2,j_3,...,j_r,j_1}$ [40, Chapter 4]. A cyclic permutation can be written as a product of adjacent inversions: $(j_1, j_2)(j_2, j_3)..(j_{r-1}, j_r)$. We note that an r-cycle can be expressed in $r - 1$ inversions. An illustrative example of a 3-cycle is shown in Figure C-2.

C-2-1 Single cycle

As mentioned earlier, an inversion was the base case of a cycle. In fact, an adjacent inversion can be seen as a 2-cycle. From now on, we only discuss cycles, which can be expressed as products of inversions.



(a) Arriving sequence **(b)** Intermediate configuration **(c)** Departing sequence

Figure C-2: Example of a cyclic permutation of length 3

Any permutation can be expressed as a product of disjoint cycles [6], therefore the notion of cycles is enough to solve any permutation D of S. However, in this theory we use a slightly different definition of a cycle. In traditional group theory, a cycle can be between any position, for example, the permutation $\sigma = \binom{4,2,3,5,1}{1,2,3,4,5}$ has a cycle of the positions $(1, 4, 5)$. In this section, we state that a cycle always considers only adjacent positions, and thus this example would consist of several cycles: $C_1 = (1, 2, 3, 4, 5)$ moves 1 to the front, then cycle $C_2 = (2, 3, 4)$ moves 2 and 3 to their correct positions as it transfers 4 to the end. Two cycles are **disjoint** if they concern different positions, i.e. the sets of positions are disjoint. In each cycle C_i, there is one position that is involved in all inversions, and we call the pebble at this position in S the **cycling pebble**, denoted with $p(C_i)$. We continue to show the necessary structure for realizing a single cycle of length $r > 2$.

Lemma C-2.1 (Single cycle) A departing sequence $D = (d_1, .., d_n)$ with a single r-cycle ($r > 2$), with inversions $(j_1, j_2)..(j_{r-1}, j_r)$, is feasible if and only if the tree T rooted at t_0 has $w(t_0) \geq 2$ and there is a branching node $t' : c(t') = 2$ such that i) $o(t') \geq j_r$ and ii) at least one child t'' of t' has $o(t'') \geq r - 2$.

Proof. Proof of ($\Longrightarrow$): Suppose that D has a single r-cycle and is feasible. We assume that $r > 2$, otherwise the cycle is an adjacent inversion and Lemma C-1.1 suffices. An r-cycle can be expressed as the product of $r - 1$ inversions $C = (j_1, j_2)(j_2, j_3), .., (j_{r-1}, j_r)$. Since this is the only cycle in the sequence, there are no other inversions than the one included in C. So, the inversions of C can be exchanged independently, in sequence with each other. Each inversion (j, k) requires that there is a branching node $t' : c(t') = 2$ such that $o(t') \geq k$ (Lemma C-1.1). Since j_r is the highest position in any of the inversions in C, we have that $o(t') \geq j_r$.

For each of the exchanges, the two pebbles must be able to park in a subtree of at least size 1 Corollary C-1.3. Since there is one pebble p is involved in all $r - 1$ exchanges (the cycling pebble), it can park in a child of t', while the other pebbles involved in the exchanges park in the subtree rooted by the other child t'' of t'. Since these $r - 1$ pebbles must park in the subtree rooted by t'', this subtree must have at least $r - 2$ descendants, because t'' can also be used for parking. So, we have $o(t'') \geq r - 2$.

Proof of ($\Longleftarrow$): Suppose that the tree T rooted at t_0 has $w(t_0) \geq 2$ and there is a branching node $t' : c(t') = 2$ such that at least one child t'' of t' has $o(t'') \geq r - 2$. If there is a single r-cycle, then there is one pebble p which is involved in all inversions in $C = (j_1, j_2)(j_2, j_3)..(j_{r-1}, j_r)$. So, there are $r - 1$ pebbles that exchange with p. These can park in the subtree rooted by t'', which has $r - 2$ descendants. Then, the departing sequence is feasible. □

C-2-2 Disjoint cycles

Now, we consider multiple cycles in a sequence. We start with the case of multiple disjoint cycles, where the necessary structure of a tree to realize a feasible departure sequence depends mostly on the length of the cycles (Lemma C-2.2). For visualization, see Figure C-3, where two disjoint cycles are shown. The cycles of positions $(1, 2, 3)$ and $(4, 5, 6)$ consider different positions of the sequence and are thus disjoint. In this case, both cycles have a length of 3. Also note, that the sets of pebbles concerned in the cycles are disjoint.

Lemma C-2.2 (Disjoint cycles) The departing sequence $D = (d_1, .., d_n)$ with m disjoint cycles $(C = \{C_i = (j_{1_i}, j_{2_i})..(j_{r-1_i}, j_{r_i}) \,|\, 1 \leq i \leq m\})$ is feasible if the tree T rooted at t_0 has $w(t_0) \geq m$ and

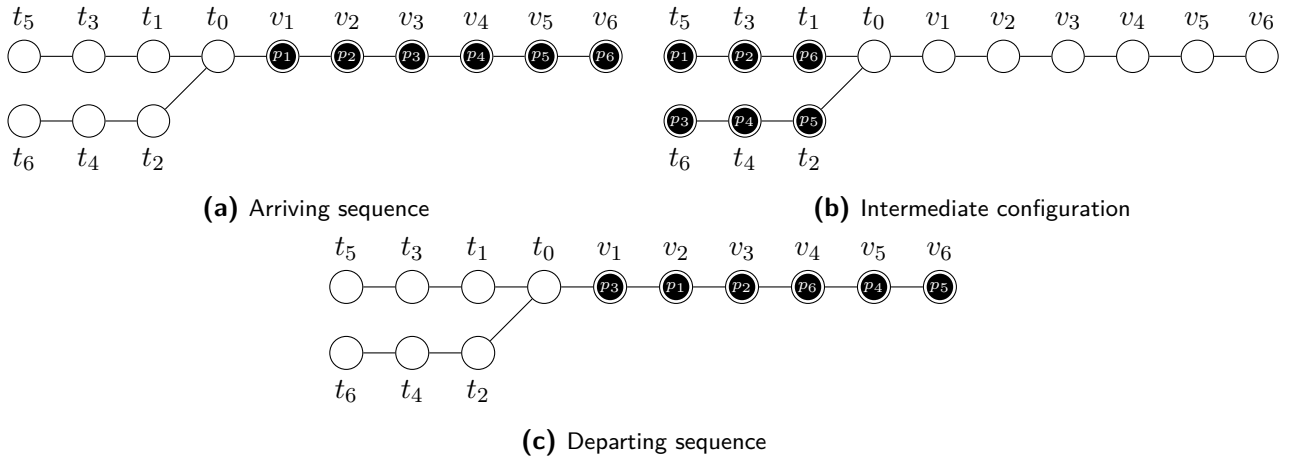


Figure C-3: Example of two disjoint 3-cycles

for each cycle C_i with length r_i there is a branching node $t_i : c(t_i) = 2$ such that i) $o(t_i) \geq j_{ri}$ and ii) at least one child t_i' of t_i has $o(t_i') \geq r_i - 2$.

Proof. Suppose that the permutation $\sigma(S, D)$ has m disjoint cycles. We refer to the cycles as $C = \{C_1, .., C_m\}$, with length $r_1, .., r_m$, and order them from longest to shortest, such that $r_1 \geq r_2 \geq .. \geq r_m$, so C_1 is the longest cycle. From Lemma C-2.1, we know that for any cycle C_i to be feasible there must be a branching node t' such that $o(t') \geq j_r$ and at least one child t'' of t' has $o(t'') \geq r_i - 2$. Since the m cycles are disjoint, the sets of pebbles concerned in each cycle are disjoint. So, there are no pebbles that need to be exchanged with pebbles from other cycles, only the cycles they are involved in.

We thus know that for each cycle there must exist some branching node t that deals with this cycle. So, the tree T could have m branching nodes such that each branching node t_i is sufficient to deal with that respective r_i-cycle, $C_i = (j_{1i}, j_{2i})..(j_{r-1i}, j_{ri})$, i.e. $o(t_i) \geq j_{ri}$ and there is at least one child t_i' of t_i with $o(t_i') \geq r_i - 2$. □

Lemma C-2.2 gives a sufficient condition for m disjoint cycles. However, there do not necessarily need to be m branching nodes, as one branching node can also resolve multiple disjoint cycles. One branching node can also deal with multiple disjoint cycles at once. We refer to this branching node as t', with children t_1 and t_2. There are several ways to distribute the pebbles over the subtrees rooted by t_1 and t_2, respectively. One of these ways is to index the cycles from largest r to smallest and park the cycling pebble in the subtree rooted by t_1 if the cycle index is odd, and in the subtree rooted by t_2 if the cycle index is even. The other pebbles of each cycle are then parked in the subtree that is not hosting that cycle's cycling pebble. This scenario is sketched in Lemma C-2.3.

Lemma C-2.3 (Disjoint cycles with one branching node) The departing sequence $D = (d_1, .., d_n)$ with m disjoint cycles $(C = \{C_i = (j_{1i}, j_{2i})..(j_{r-1i}, j_{ri}) \,|\, 1 \leq i \leq m \,|\, r_i \leq r_{i+1}\})$ is feasible if the tree T rooted at t_0 has $w(t_0) \geq 2$ and the branching node $t' : c(t') = 2$ that is the highest up in the tree has children t_1 and t_2 such that $o(t_1) \geq \left(\lfloor \frac{m}{2} \rfloor + \sum_{\substack{i=0 \\ i=even}}^{m} (r_i - 1) \right) - 1$ and $o(t_2) \geq \left(\lceil \frac{m}{2} \rceil + \sum_{\substack{i=0 \\ i=odd}}^{m} (r_i - 1) \right) - 1$.

Proof. Suppose D has m disjoint cycles, and each cycle C_i can be expressed in $r_i - 1$ inversions: $C_i = (j_{1i}, j_{2i})..(j_{r-1i}, j_{ri})$. From Lemma C-2.1, we know that for any cycle C_i to be feasible, there must be a branching node $t' : c(t') = 2$ such that $o(t') \geq j_{ri}$ and at least one child t'' of t' has $o(t'') \geq r_i - 2$. We use a proof by induction. The base case concerns $m = 2$ because if there are zero cycles it means $S = D$ and we refer to Proposition 4-1.1, and with only one cycle Lemma C-2.1 suffices.

Base case: If $m = 2$ there are only two disjoint cycles. We refer to the cycling pebble of C_1 as p_1, and the cycling pebble of C_2 as p_2. Suppose that p_1 parks in the subtree rooted by t_1, then $r_1 - 1$ pebbles must park in the subtree rooted by t_2. If p_2 then parks in the subtree rooted by t_2, $r_2 - 1$ pebbles must park in

the subtree rooted by t_1. So, we have $o(t_1) = (1+r_2-1)-1$ and $o(t_2) = (1+r_1-1)-1$, because one pebble can park in the node t_1 and t_2 respectively. Therefore, it is sufficient if $w(t_0) \geq 2$ and the branching node has two children such that $o(t_1) \geq \left(\lfloor \frac{m}{2} \rfloor + \sum_{\substack{i=0 \\ i=even}}^{m} (r_i - 1) \right) - 1$ and $o(t_2) \geq \left(\lceil \frac{m}{2} \rceil + \sum_{\substack{i=0 \\ i=odd}}^{m} (r_i - 1) \right) - 1$.

Induction hypothesis: Suppose that the sequence is feasible for $k = m$ such that $o(t_1) \geq \left(\lfloor \frac{k}{2} \rfloor + \sum_{\substack{i=0 \\ i=even}}^{k} (r_i - 1) \right) - 1$ and $o(t_2) \geq \left(\lceil \frac{k}{2} \rceil + \sum_{\substack{i=0 \\ i=odd}}^{k} (r_i - 1) \right) - 1$. Then, it will also be feasible if $k = m + 1$.

Inductive step: Suppose the departing sequence is feasible for $k = m$ and $o(t_1) \geq \left(\lfloor \frac{k}{2} \rfloor + \sum_{\substack{i=0 \\ i=even}}^{k} (r_i - 1) \right) - 1$ and $o(t_2) \geq \left(\lceil \frac{k}{2} \rceil + \sum_{\substack{i=0 \\ i=odd}}^{k} (r_i - 1) \right) - 1$. Now, we consider $l = m + 1$. We consider the new cycle C', which is disjoint with all existing cycles $C_i \in C$, and take p' to be the cycling pebble of C'. If k is even, then l is odd. Then, $\lfloor \frac{l}{2} \rfloor = \lfloor \frac{k}{2} \rfloor$ and $\lceil \frac{l}{2} \rceil = \lceil \frac{k}{2} \rceil + 1$. Furthermore, $\sum_{\substack{i=0 \\ i=even}}^{k} (r_i-1) = \sum_{\substack{i=0 \\ i=even}}^{l} (r_i-1)$ and $\sum_{\substack{i=0 \\ i=odd}}^{k} (r_i-1) = \sum_{\substack{i=0 \\ i=odd}}^{l} (r_i-1)+r'-1$. So, $o(t_1)_l = o(t_1)_k+r'-1$ and $o(t_2)_l = o(t_2)_k + 1$. This means that p' can park in the subtree rooted by t_2 while the $r' - 1$ pebbles that it has to exchange with can park in t_1, so the departing sequence is feasible. □

Although this method describes a way to distribute the pebbles over the branching node, this is not the only way to park the pebbles such that the departing sequence is feasible, thus Lemma C-2.3 is not necessary to show the infeasibility if this condition is not met. To do this, the minimally necessary structure for a departing sequence with m disjoint cycles should be derived, so we could check if this structure is present in a tree. When trying to find the minimal size of the subtrees rooted by t_1 and t_2, we could give a more general description using a decision variable x_i which is 0 if cycle C_i parks its cycling pebble in the subtree rooted t_1 and $x_i = 1$ if the cycling pebble of C_i is parked in the subtree rooted by t_2. More formally, $x_i = \begin{cases} 1 & \text{if } p(C_i) \in o(t_1) \cup \{t_1\} \\ 0 & \text{if } p(C_i) \in o(t_2) \cup \{t_2\} \end{cases}$. Finding the necessary structure for a departing sequence with cycles to be feasible, is a difficult problem, although it does not scale up with the number of pebbles but with the number of cycles. So, a practical situation with many trains though relatively few cycles is still relatively easy to solve. Moreover, we can derive the proposition on the necessary offspring of a branching node for m disjoint cycles to be feasible (Corollary C-2.1).

Corollary C-2.1 (m Disjoint cycles) The departing sequence $D = (d_1, .., d_n)$ with m disjoint cycles is feasible if and only if the tree T rooted at t_0 has a branching node t' such that i) $o(t') \geq \max_{C_i \in C} j_{ri}$ and ii) children t_1 and t_2 of t' such that $x_i = \begin{cases} 1 & p(C_i) \in o(t_1) \cup \{t_1\} \\ 0 & p(C_i) \in o(t_2) \cup \{t_2\} \end{cases}$ assigns the subtrees such that $|\sum_{C_i \in C} x_i + (1 - x_i) \cdot (r_i - 1) - \sum_{C_i \in C} (1 - x_i) + x_i \cdot (r_i - 1)| = k$.

Although this corollary provides a way to determine whether a certain departing sequence is feasible for a given tree, it might take some time to find a feasible distribution of pebbles over the branching node. However, it is easier to determine the infeasibility of a given sequence with m disjoint cycles, as shown in Lemma C-2.4.

Lemma C-2.4 (Infeasible disjoint cycles) The departing sequence $D = (d_1, .., d_n)$ with m disjoint cycles $(C = \{C_i = (j_{1i}, j_{2i})..(j_{r-1i}, j_{ri}) \,|\, 1 \leq i \leq m\})$ is infeasible if the tree T rooted at t_0 has $w(t_0) \leq 2$ and the branching node $t' : c(t') = 2$ such that i) $o(t') < \max_{1 \leq i \leq m} j_{ri}$, or ii) the child t'' of t' that roots the largest subtree has $o(t'') < \max_{1 \leq i \leq m} r_i$.

Proof. Suppose D has m disjoint cycles, and each cycle C_i can be expressed in $r_i - 1$ inversions: $C_i = (j_{1i}, j_{2i})..,(j_{r-1i}, j_{ri})$. From Lemma C-2.1, we know that for any cycle C_i to be feasible, there must be a branching node $t' : c(t') = 2$ such that $o(t') \geq j_{ri}$ and at least one child t'' of t' has $o(t'') \geq r_i - 2$.

So, if $w(t_0) < 2$ the departing sequence is always infeasible because there need to be two branches with at least one child for any inversion to be possible Corollary C-1.3. We do not consider cases where $w(t_0) > 2$, so for $w(t_0) = 2$ we show both conditions lead to an infeasible result.

Case I: Suppose $w(t_0) = 2$ and there is only one branching node t' such that $o(t') < \max_{1 \leq i \leq m} j_{ri}$, then the cycle C_i is infeasible because the inversion (j_{r-1_i}, j_{ri}) requires a branching node t' such that $o(t') \geq j_{ri}$ (Lemma C-1.1).

Case II: Suppose $w(t_0) = 2$ and there is one branching node t' in T with children t_1 and t_2. We take t_1 such that $o(t_1) \geq o(t_2)$. If $o(t_1) < \max_{1 \leq i \leq m} r_i$, then the cycle C_i is infeasible because the pebble that is involved in all C_i inversions cannot be in a separate subtree from the other $r_i - 1$ pebbles (Lemma C-2.1). □

C-2-3 Intersecting cycles

Not all cycles in a sequence are disjoint, and we speak of **intersecting cycles** when the sets of pebbles they concern intersect. An example of intersecting cycles is shown in Figure C-4, where there are three cycles ($m = 3$) on the positions $\{(3, 1, 2), (2, 1), (5, 4)\}$. This clearly shows that two cycles intersect, while both are disjoint from the third cycle.

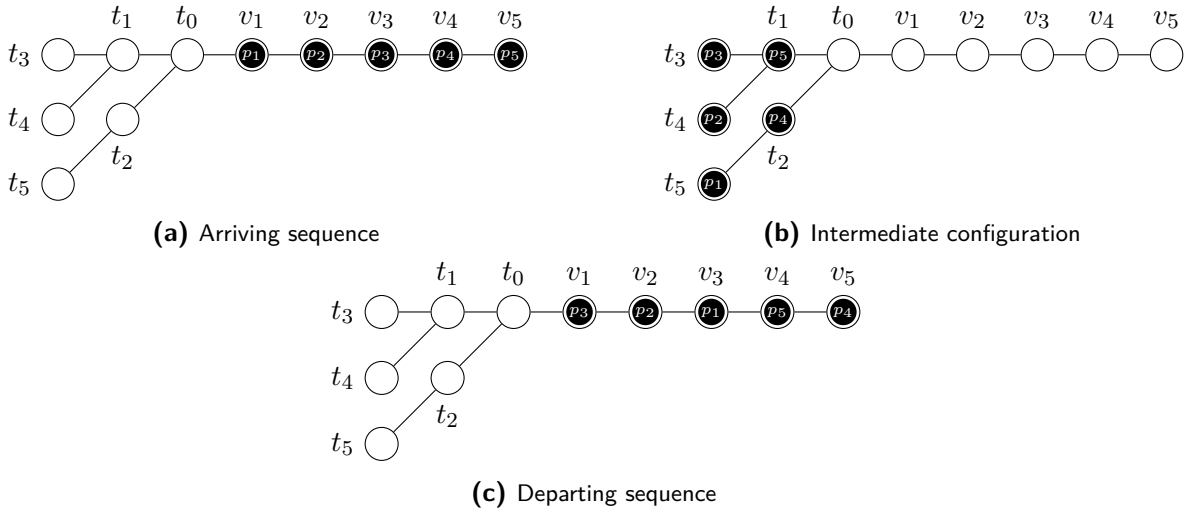


Figure C-4: Example of three cycles, of which two intersect and the third is disjoint

It becomes clear now that we need to define the cycles on the positions that are involved and not directly on the pebbles. When we refer to the pebbles of a cycle, we mean the pebbles that are located on the positions of the cycle in the arriving sequence. Furthermore, the order of intersecting cycles matters. From a group theory perspective, the cycles must be applied in order to turn the arriving sequence into the departing sequence, and in reversed order to turn the departing sequence into the arriving sequence. When we look from the pebble motion perspective, we transform the arriving sequence into the departing sequence by parking the pebbles in a smart way in the tree. For example, in Figure C-4, first, we deal with the cycle of $(3, 1, 2)$. Note that the pebble positioned at i in the arriving case is p_i. So, by parking the pebble in the third position of the arriving sequence, which is p_3, in a different branch than the pebbles p_1 and p_2, we can resolve this cycle. Next, we deal with the cycle $(2, 1)$, by parking the pebbles p_1 and p_2 in different branches. Finally, the pebbles p_4 and p_5 are split over the branching node t_0, to resolve the last cycle. On the other hand, a group theory perspective would take the departing sequence $(3, 2, 1, 5, 4)$ and first perform the cycle $(5, 4)$ resulting in $(3, 2, 1, 4, 5)$. After which the second cycle from the reversed list, $(2, 1)$ is applied to yield $(3, 1, 2, 4, 5)$. Finally, the last cycle is applied to shift the first three positions one to the left: $(1, 2, 3, 4, 5)$.

We now consider the scenario where a departing sequence has multiple intersecting cycles, compared to the arriving sequence, in Lemma C-2.5. It is important to note that the condition we derive here is both sufficient and necessary for intersecting cycles, while it is only a sufficient condition for disjoint

cycles. This is because the same pebbles are concerned in intersecting cycles, so the cycles need to be dealt with recursively.

Lemma C-2.5 (Intersecting cycles) The departing sequence $D = (d_1, .., d_n)$ with m intersecting cycles $(C = \{C_i = (j_{1i}, j_{2i})..(j_{r-1i}, j_{ri}) \,|\, 1 \leq i \leq m\})$ is feasible if and only if the tree T rooted at t_0 has $w(t_0) \geq m + 1$ and for each cycle C_i with length r_i there is a branching node $t_i : c(t_i) = 2$ such that $o(t_i) \geq j_{ri}$, and at least one child t_i' of t_i has $o(t_i') \geq r_i - 2$.

Proof. Proof of ($\Longrightarrow$): Suppose D has m intersecting cycles, and each cycle $C_i \in C$ can be expressed in $r_i - 1$ inversions: $C_i = (j_{1i}, j_{2i})..,(j_{r-1i}, j_{ri})$. From Lemma C-2.1, we know that for any cycle C_i to be feasible, there must be a branching node $t_i : c(t_i) = 2$ such that $o(t_i) \geq j_{ri}$ and at least one child t_i' of t_i has $o(t_i') \geq r_i - 2$. Therefore, there must be such a branching node t_i for each cycle. From the Width of a tree definition, we know that m branching nodes will determine that $w(t_0) \geq m + 1$. If m cycles intersect, this means that the longest of these cycles, C' with $r' = \max_{1 \leq i \leq m} r_i$, contains the other cycles as subsets. The branching node t that resolves this longest cycle has a child t' with $o(t') \geq r' - 2$. We take t to be the branching node the highest up in the tree. There is at least one pebble in C' that is not included in any of the other intersecting cycles because intersecting cycles are subsets of each other. So, all pebbles that are considered in the other intersecting cycles, can park in the subtree rooted by t'. Now, we can resolve the remaining intersecting cycles similarly by looking only at the tree rooted by t' and the cycles $C \setminus C'$.

Proof of ($\Longleftarrow$): Suppose that the tree T rooted at t_0 has $w(t_0) \geq m + 1$ and for each cycle $C_i : 1 \leq i \leq m$ with length r_i there is a branching node $t_i : c(t_i) = 2$ such that $o(t_i) \geq j_{ri}$, where j_{ri} is the highest position in any of the inversions in C_i, and at least one child t_i' of t_i has $o(t_i') \geq r_i - 2$. If there are m intersecting cycles, then each cycle $C_i : 1 \leq i \leq m$ can be resolved by the branching node t_i, because this node t_i meets all the conditions for a cycle (Lemma C-2.1). □

C-2-4 Combining cycles

Using the lemmas we have derived, we are now able to state the conditions that a tree needs to meet to resolve a permutation with any number of cycles, whether they be disjoint or not. We determine the number of intersecting cycles between S and D with $\mu(S, D) = |\{C_i \,|\, C_i \cup C_j \neq \emptyset;\, C_i, C_j \in C;\, C_i \neq C_j\}|$. Lemma C-2.6 gives the conditions on a tree for a general permutation to be feasible.

Lemma C-2.6 (Cycles) The departing sequence $D = (d_1, .., d_n)$ with m cycles $(C = \{C_i = (j_{1i}, j_{2i})..(j_{r-1i}, j_{ri}) \,|\, 1 \leq i \leq m\})$ is feasible if the tree T rooted at t_0 has $w(t_0) \geq \mu(S, D)+1$ and for each cycle C_i with length r_i there is a branching node $t_i : c(t_i) = 2$ such that $o(t_i) \geq j_{ri}$, and at least one child t_i' of t_i has $o(t_i') \geq r_i - 2$; and the top branching node t' with children t_1 and t_2 meets $o(t') \geq k = \max_{C_i \in C}(j_{ri})$ and $o(t_1) = -1 + \sum_{C_i \in C} x_i + (1 - x_i) \cdot (r_i - 1), o(t_2) = -1 + \sum_{C_i \in C}(1 - x_i) + x_i \cdot (r_i - 1)$

where $x_i = \begin{cases} 1 & \text{if } p(C_i) \in o(t_1) \cup \{t_1\} \\ 0 & \text{if } p(C_i) \in o(t_2) \cup \{t_2\} \end{cases}$.

Proof. All intersecting cycles require a branching node, therefore $w(t_0) \geq \mu(S, D) + 1$ (Lemma C-2.5). Disjoint cycles do not require their own branching node, because they can fit in subtrees formed by other branching nodes as long as there is enough space for the cycle to distribute its pebbles (Lemma C-2.3). The top branching node should supply the necessary conditions for all disjoint cycles because then we can be certain that also the longest cycle is feasible.

Consider the pebble with the highest index k in the departing sequence that is considered in some cycle C'. In other words, this is the first pebble to depart the shunting yard that arrived later than all pebbles, which depart later. This is the first cycle that needs to be resolved. Therefore, the top branching node t' must have $o(t') = k$, so all pebbles that depart later than the pebble at index k of D can park in the offspring of t'. Furthermore, we know that for each cycle, the cycling pebble must be in a different subtree than the other pebbles in the cycle (Lemma C-2.1), which can be established by the sizes of the

subtrees rooted by t_1 and t_2 such that their offspring $o(t_1) = -1 + \sum_{C_i \in C} x_i + (1 - x_i) \cdot (r_i - 1)$ and $o(t_2) = -1 + \sum_{C_i \in C} (1 - x_i) + x_i \cdot (r_i - 1)$ where $x_i = \begin{cases} 1 & \text{if } p(C_i) \in o(t_1) \cup \{t_1\} \\ 0 & \text{if } p(C_i) \in o(t_2) \cup \{t_2\} \end{cases}$ (Corollary C-2.1). □

C-3 Conclusion

Although Lemma C-2.6 suggests a solid approach for determining the feasibility of the PMTAD, this is not a complete approach. It turns out that expressing a sequence in these cycles is quite complex. In this chapter, we considered very simple examples, which were the starting point of this thesis. However, real-world scenarios can be much more complex. Consider, for example, the sequence $D = (p_5, p_3, p_6, p_1, p_4, p_2, p_7)$, then expressing the cycles of this sequence is not so straight-forward. Furthermore, it turns out that the order and direction of a cycle are also very important and should be included in the definition. The permutation $\sigma_1 = \binom{1,2,3}{3,1,2}$ is usually considered a left-to-right, and the permutation $\sigma_2 = \binom{1,2,3}{2,3,1}$ is considered a right-to-left cycle, but the latter can also be expressed as two left-to-right cycles which are applied consecutively. This direction complicates the definition of cycles of more complex sequences. Finally, the approach is not complete because, as we established earlier, there are different ways to park pebbles of disjoint cycles and the inability to park them in a certain way does not necessarily imply that the sequence is infeasible. So, we conclude that this approach is not complete and should not be used in this way.

Bibliography

[1] R. M. Wilson. "Graph puzzles, homotopy, and the alternating group". In: *Journal of Combinatorial Theory, Series B* 16.1 (1974), pp. 86–96.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. United States of America: Bell Telephone Laboratories, Incorporated, 1979. ISBN: 0-7167-1044-7.

[3] W. Colman. "The number of partitions of the integer N into M nonzero positive integers". In: *Mathematics of Computation* 39.159 (1982), pp. 213–224.

[4] D. M. Kornhauser, G. Miller, and P. Spirakis. "Coordinating pebble motion on graphs, the diameter of permutation groups, and applications". MSc Thesis. M. I. T., Dept. of Electrical Engineering and Computer Science, 1984.

[5] P. Diaconis. *Group Representations in Probability and Statistics*. Vol. 11. Institute of Mathematical Statistics, 1988, pp. i–192. URL: http://www.jstor.org/stable/4355560.

[6] A. M. Gaglione et al. *An introduction to group theory*. Naval Research Laboratory, Identification Systems Branch, Radar Division, 1992.

[7] C. H. Papadimitriou, P. Raghavan, M. Sudan, and H. Tamaki. "Motion planning on a graph". In: *Proceedings 35th A nnual Symposium on Foundations of Computer Science*. IEEE. 1994, pp. 511–520.

[8] J.-F. Cordeau, P. Toth, and D. Vigo. "A survey of optimization models for train routing and scheduling". In: *Transportation science* 32.4 (1998), pp. 380–404.

[9] R. L. Rardin. *Optimization in operations research*. Vol. 166. Prentice Hall Upper Saddle River, NJ, 1998.

[10] V. Auletta, A. Monti, M. Parente, and P. Persiano. "A Linear-Time Algorithm for the Feasibility of Pebble Motion on Trees". In: *Algorithmica* 23 (0 1999), pp. 223–245.

[11] A. Barvinok and J. E. Pommersheim. "An algorithmic theory of lattice points in polyhedra". In: *New perspectives in algebraic combinatorics* 38 (1999), pp. 91–147.

[12] H. S. Wilf. *Lectures on integer partitions*. 2000. URL: http://www.cis.upenn.edu/~wilf (visited on 03/23/2022).

[13] R. M. Lentink, P.-J. Fioole, L. G. Kroon, et al. "Applying operations research techniques to planning of train shunting". In: *ERIM Report Series Reference* ERS-2003-094-LIS (2003).

[14] R. Freling, R. M. Lentink, L. G. Kroon, and D. Huisman. "Shunting of passenger train units in a railway station". In: *Transportation Science* 39.2 (2005), pp. 261–272.

[15] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[16] R. Lentink. "Algorithmic decision support for shunt planning". PhD dissertation. Erasmus Universiteit Rotterdam, 2006.

[17] R. Cavallo. “De spoorwegen en de Nederlandse stad”. In: *OverHolland* 3.5 (2007), pp. 43–59.

[18] L. G. Kroon, R. M. Lentink, and A. Schrijver. “Shunting of passenger train units: an integrated approach”. In: *Transportation Science* 42.4 (2008), pp. 436–449.

[19] P. Surynek. “A novel approach to path planning for multiple robots in bi-connected graphs”. In: *2009 IEEE International Conference on Robotics and Automation*. IEEE. 2009, pp. 3613–3619.

[20] P. Surynek. “An application of pebble motion on graphs to abstract multi-robot path planning”. In: *2009 21st IEEE International Conference on Tools with Artificial Intelligence*. IEEE. 2009, pp. 151–158.

[21] G. Goraly and R. Hassin. “Multi-color pebble motion on graphs”. In: *Algorithmica* 58.3 (2010), pp. 610–636.

[22] R. Luna and K. E. Bekris. “Efficient and complete centralized multi-robot path planning”. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2011, pp. 3268–3275. DOI: 10.1109/IROS.2011.6095085.

[23] K. D. Sedgewick Robert; Wayne. *Algorithms*. 4th edition. Addison-Wesley Professional, 2011, pp. 661–666.

[24] C. Expósito-Izquierdo, B. Melián-Batista, and M. Moreno-Vega. “Pre-marshalling problem: Heuristic solution method and instances generator”. In: *Expert Systems with Applications* 39.9 (2012), pp. 8337–8349.

[25] A. Krontiris, R. Luna, and K. E. Bekris. “From feasibility tests to path planners for multi-agent pathfinding”. In: *Sixth annual symposium on combinatorial search*. 2013.

[26] M. Sipser. *Introduction to the Theory of Computation, Third Edition*. Boston, MA, USA: Cengage Learning, 2013.

[27] B. De Wilde, A. W. Ter Mors, and C. Witteveen. “Push and rotate: a complete multi-agent pathfinding algorithm”. In: *Journal of Artificial Intelligence Research* 51 (2014), pp. 443–492.

[28] R. van den Broek. “Train Shunting and Service Scheduling: an integrated local search approach”. MSc Thesis. Utrecht University, 2016.

[29] R. Diestel. *Graph Theory*. 5th edition. Springer, 2016.

[30] F Wolfhagen. “The train unit shunting problem with reallocation”. MSc Thesis. Erasmus University Rotterdam, 2017.

[31] E Huizingh. “Planning first-line services on NS service stations”. MSc Thesis. Enschede: University of Twente, 2018.

[32] J. Erickson. *Algorithms*. University of Illinois, 2019. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/.

[33] M. Kulich, T. Novák, and L. Přeucil. “Push, stop, and replan: An application of pebble motion on graphs to planning in automated warehouses”. In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE. 2019, pp. 4456–4463.

[34] NS. *Jaarverslag 2019*. Report, *Nederlanse Spoorwegen*. 2019.

[35] R. Özčelik. *Solving Minimum Path Cover on a DAG*. Nov. 2019. URL: https://towardsdatascience.com/solving-minimum-path-cover-on-a-dag-21b16ca11ac0 (visited on 04/07/2022).

[36] J. Mulderij, B. Huisman, D. Tönissen, K. van der Linden, and M. de Weerdt. “Train Unit Shunting and Servicing: a Real-Life Application of Multi-Agent Path Finding”. In: *arXiv preprint arXiv:2006.10422* (2020).

[37] N. E. Ahangar, K. M. Sullivan, S. M. Spanton, and Y. Wang. “Algorithms and Complexity Results for the Single-Cut Routing Problem in a Rail Yard”. In: (2021).

[38] R. van den Broek, H. Hoogeveen, M. van den Akker, and B. Huisman. “A local search algorithm for train unit shunting with service scheduling”. In: *Transportation Science* (2021).

[39] F. Kamenga, P. Pellegrini, J. Rodriguez, and B. Merabet. “Solution algorithms for the generalized train unit shunting problem”. In: *EURO Journal on Transportation and Logistics* (2021), p. 100042.

[40] J. S. Milne. *Group Theory (v4.00)*. Available at www.jmilne.org/math/. 2021.

[41] NS. *Treinen van NS*. 2022. URL: https://www.ns.nl/over-ns/treinen-van-ns.

[42] NS. *Vervoerplan 2022*. Report, *Nederlanse Spoorwegen*. 2022.

Glossary

converse given a conditional statement $P \rightarrow Q$ (read P implies Q), the converse is the statement $Q \rightarrow P$. 32

corridor a part of the tree which forms a simple path and represents one track in the shunting yard. 47

feasibility determining whether a solution to a problem exists. 2, 3, 7

optimality finding a best solution, based on certain criteria. 3, 7

reallocation moving a train within the shunting yard to a new parking location, thus not considering the movement of arriving or departing. 3, 5, 11, 18, 44, 55, 59

relaxation a constraint relaxation of the problem weakens the conditions for a solution while making sure that feasible solution to the relaxed problem are still feasible for the original problem. 4, 5

rolling stock vehicles that operate on railway. 1, 38, 43

shunting yard a location where railway carriages are maneuvred to. 1, 7, 38

simple path a simple path is a path in a graph with no loops, such that all visited nodes are distinct. 17, 19, 21, 27, 32, 45, 46, 47, 69

train carriage a railway coach for passengers. 1, 16, 18, 43

train unit a fixed formation of train carriages, depending on the type of train. 1, 7, 16, 43, 59

Acronyms

m-PM m-color Pebble Motion problem. 9

m-PMT m-color Pebble Motion problem on a Tree. 9, 16

BSPP Branch Set for Partition Problem. 27, 28, 29, 41, 55

DAG Directed Acyclic Graph. 22, 23, 24, 26, 31, 36, 41, 55, 58, 63

GMPkR Graph Motion Planning with k Robots. 8

GMP1R Graph Motion Planning with 1 Robot. 8

ILP Integer Linear Program. 10, 11, 12, 34, 63

LIFO Last In, First Out. 17

LP Linear Program. 10, 11, 31, 34, 35

MAPF Multi-Agent Path Finding. 4, 5, 9, 10, 13, 46

MPP Multi-robot Path Planning. 9, 10

NS *Nederlandse Spoorwegen*, Dutch Railways. viii, 1, 4, 5, 7, 12, 16, 21, 27, 38, 43, 44, 51, 55, 58, 61

PM Pebble Motion problem. 4, 5, 6, 7, 8, 9, 10, 13, 15, 16, 41, 43, 45, 51, 55, 56

PMG Pebble Motion problem on a Graph. 8

PMT Pebble Motion problem on a Tree. 9, 15, 16, 17

PMTAD Pebble Motion problem on a Tree with Arrival and Departure. 18, 19, 21, 27, 29, 32, 33, 34, 41, 43, 44, 47, 48, 51, 52, 55, 56, 67, 68, 75

PMTADC Pebble Motion problem on a Tree with Arrival, Departure, and Color matching. 52, 53, 56

PMTADL Pebble Motion problem on a Tree with Arrival, Departure, and Length inclusion. 47, 48, 56, 58

PMTUSP Pebble Motion for the Train Unit Shunting Problem. 56, 58

PP Classic Partition Problem. 49, 50

PPST Partition for a Pebble Sequence on a Tree. 31, 32, 33, 34, 36, 37, 41, 49, 50, 53, 54, 55, 56, 58

PPSTC Partition for a Pebble Sequence on a Tree with Color matching. 53, 54, 56, 58

PPSTL Partition for a Pebble Sequence on a Tree with Length inclusion. 48, 49, 50, 56

PPT Pebble Permutation problem on a Tree. 8

TUSP Train Unit Shunting Problem. 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 15, 16, 18, 47, 51, 55, 56, 58, 67

TUSP Train Unit Shunting Problem on Trees. 15, 16, 17

TUSS Train Unit Shunting and Servicing problem. 1, 4, 11, 12, 13