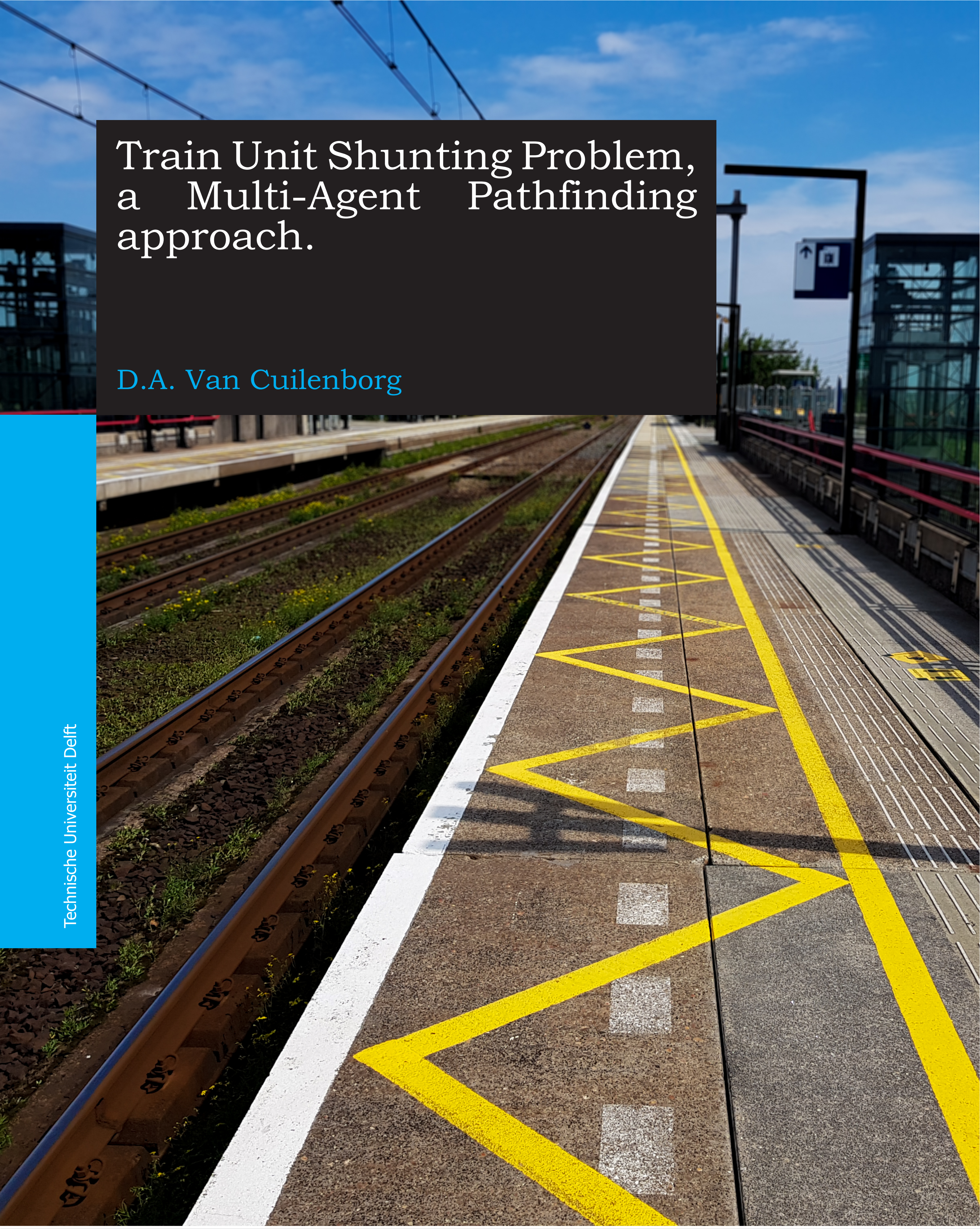


Train Unit Shunting Problem, a Multi-Agent Pathfinding approach.

D.A. Van Cuilenborg

Train Unit Shunting Problem, a Multi-Agent Pathfinding approach

by

D.A. Van Cuilenborg

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Thursday October 1, 2020 at 1:00 PM.

Student number:	4481321	
Project duration:	11 November, 2019 – September 14, 2020	
Thesis committee:	Dr. M.M. de Weerdt	TU Delft, supervisor
	Dr. ir. J.T. van Essen,	TU Delft
	Dr. ir. L. Bliek	TU Delft
	B. Huisman,	Nederlandse Spoorwegen

Delft University of Technology

Abstract

The Train Unit Shunting Problem (TUSP) is a well studied problem within the NS (Dutch Railways). TUSP is a problem where trains have to be routed on a shunting yard such that maintenance tasks can be performed and no collisions occur. Furthermore, incoming trains have to be matched to outgoing trains, since trains of the same type can be used interchangeably. Currently, the NS uses a local-search approach to solve TUSP; however, this approach is sometimes unable to find a simple detour route when the initial route of the train is blocked by another train. This research models the TUSP as a Multi-Agent Pathfinding (MAPF) problem to solve the routing issue. We solve this model of the TUSP using Conflict-Bases Search (CBS). To achieve this, CBS is altered to work with multiple trains on the same node, to work with directed edges, and to only allow wait-moves on certain nodes. In this work, we propose two ways of routing multiple trains on the same node; CBSL, where trains are given an exact location on the track; and CBSQ, where the order in which trains enter a track is recorded with a queue. Of these two methods, the queue model was deemed best. Finally, we propose a heuristic to prefer solutions where trains are more spread out on the shunting yard. This heuristic greatly improves the runtime at a loss of optimality.

Preface

This thesis marks the end of my academic career and, with that, my great time at Delft University of Technology. During this thesis, I undertook an internship at the Nederlandse Spoorwegen, which was nothing but a great experience. After the many projects I have already done on the Train Unit Shunting Problem, I was finally able to work closely with the team behind it all.

I would like to thank my entire team at the NS, but I would like to especially thank Joris den Ouden for his continues tech-support, especially during the COVID-19 times. Furthermore, Bob Huisman for his supervision from the NS side of the project.

I would also like to thank Jesse Mulderij, for his smart ideas and remarks during our countless brainstorm sessions. Laurens Bliek, for his supervision, especially during the first part of the project, and our attempt at the Flatland Challenge. Finally, I would like to thank Mathijs de Weerdt for guiding this project into the right direction whenever I went off-track or was lost.

I hope you enjoy your reading.

D.A. Van Cuilenborg
Delft, September 14

Contents

1 Introduction

The Dutch Railways (NS) manages a huge fleet of trains. Most of these trains are in service during rush-hour; however, at other times, such as night-time, these trains have to be parked. Additionally, to ensure reliability of these trains, maintenance has to be conducted regularly. The NS stores their *rolling stock*, the trains, in *shunting yards*, where these trains can be serviced.

However, many difficult problems arise when it comes to the planning of such shunting yards. The trains must be serviced with the capacity of the shunting yard in mind. The trains have to be routed and parked as to not cause collisions. Furthermore, the trains on the shunting yard have to be matched to the outgoing trains, since certain train types are required for its passenger railway network at certain times. The full problem is defined as the *Train Unit Shunting Problem* (TUSP)

Currently, a man-made planning is still used for these shunting yards. However, a lot of research has been done regarding automatic scheduling of trains on shunting yards. Most notably a local-search approach by Van den Broek [1], which is now the current state-of-the-art for solving the TUSP. However, this solution still does not solve all problems. One core issue that still occurs with this approach are the relaxations taken when routing trains. There are many cases where trains do route through each other and, while this is penalized in the objective function, the algorithm is often unable to find routing solutions to rather trivial problems.

In this thesis we focus on solving the TUSP from a routing perspective, rather than solving the planning problem first. This means we translate the TUSP to a Multi-Agent Path Finding problem (MAPF), where every train is an agent. By incorporating the various goals of a schedule, such as a valid matching of incoming and outgoing trains, in the goals of the pathfinding, we can use state-of-the-art MAPF algorithms to find a feasible routing.

The goal of this research is to model TUSP such that it is solvable by MAPF algorithms. With this research we show that MAPF can be used in other domains than what is currently researched. However, to do so the MAPF algorithm has to be altered to be able to work on a different domain. In this thesis, we show how such an alteration can be done for a shunting yard. While the alterations made to MAPF are domain-specific, it shows that it is possible to alter MAPF-algorithms in such a way that it can be applied to other, but similar, domains. To do so we require an answer to the following questions:

How to use MAPF in a railway setting?
In general, MAPF is used to route a set of agents in a grid-world, which would mean that we would have to discretize shunting-yards into a grid world such that it can be solved by MAPF-algorithms. This method is sadly not plausible, as is explained in subsection 4.3.2. Moreover, with general MAPF-problems there is no such thing as parking-spots, agents are simply free to wait and move as they please. When using shunting yards as setting, only certain tracks are available for parking and that has to be accounted for.

How can the Matching, Parking and Routing of TUSP be modeled in MAPF?
TUSP consists of various sub-problems, which all have to be solved to come to a feasible solution. While MAPF is generally used to solve the problem of routing agents, we propose a way to include the matching and parking of trains in a MAPF-algorithm.

How to analyze the MAPF algorithms in a TUSP setting?
MAPF-algorithms are generally only compared by their ability to route a certain amount of agents on a certain map. Due to the new setting, we propose new experiments which could be used to determine the efficiency of an algorithm. This would enable us to analyze the bottlenecks of the algorithm.

In this thesis, we will first start by giving the reader the relevant literature necessary to understand the contents of this thesis, this is done in chapter 2. We then continue by defining the model used in chapter 3. In chapter 4, we define how this model is solved using MAPF algorithms. In chapter 5, we go over the algorithms introduced in this paper and compare them. The assumptions and decisions made in this thesis are discussed in chapter 6. Then we conclude on the thesis in chapter 7.

2 Literature Review

In this chapter, we first provide an introduction of the literature on the Train-Unit Shunting Problem that is relevant to understand the ideas discussed in this paper. This is then followed by a similar review of the current state of the art of Multi-Agent Pathfinding. The goal of this chapter is to introduce a reader with sufficient material to understand the new algorithms discussed in this thesis.

2.1. Train-Unit Shunting Problem

The Train-Unit Shunting Problem (TUSP) was first formally introduced by Freling et al. [2]. The problem was defined as a matching of arriving trains and departing trains, since trains of the same type can be used interchangeably, and parking trains that were unneeded at the current time on parking tracks. Using a combination of MIP solvers and Dynamic Programming they generated a shunting plan in roughly 20 to 40 minutes for about 70 trains. However, this approach did not take the routing of the trains into account.

Lentink et al. [3] did take routing into account. The approach was quite similar to the approach by Freling et al.; however, as a final step of the algorithm the routes of the trains are computed. These routes are computed greedily after which they are then improved upon using a local search. The issue with this approach is that the sequential approach of this algorithm causes some solutions to be only deemed infeasible at the routing stage, after which the solution has to be recalculated.

Moreover, there is still one practical aspect missing in this model that is necessary to model shunting yards, which is the cleaning and maintenance of train units. This addition is solved by Lentink by modeling the cleaning problem as a job shop scheduling problem without preemption [4]. However, this solution requires all the sub-problems to be solved sequentially. Therefore, if the algorithm cannot find a solution in one of the sub-problems given the solution found in the previous sub-problem, all previous sub-problems would have to be recomputed.

Various other attempts have been made to solve this version of TUSP. The current state of the art for solving TUSP is a local search approach [1]. A local search can iteratively find a better solution by making small local changes on the order of the executed actions. The issue with this approach is that, although crossings (when two trains cross) are penalized in the objective function, a workaround is not searched for. Trains simply attempt to take one of the shortest paths that do not have a crossing. However, the issue lies with the fact that sometimes a train would have to be pushed aside to let another train pass, which is impossible to accomplish in the current algorithm. The local-search approach finds the best path for a single train; however, this can not always be the best path for the solution as a whole. Furthermore, this algorithm does not take into account the real-life time of its surroundings. When a track on the shunting yard intersects with a track used by the regular train service, it is impossible to take crossings on this track into account. Therefore, even the current state-of-the-art for TUSP still struggles with routing.

2.2. Multi-agent Path Finding

Since algorithms to solve TUSP still struggle with routing, it could be an interesting approach to look at the TUSP as a variant of a Multi-agent Path Finding (MAPF) problem. MAPF is a set of problems where a team of agents is routed from their initial position to their goal position without colliding with each other. The goal of these algorithms is to either minimize the makespan or minimize the sum of paths, the total distance traveled. Moreover, it has been shown that it can also optimize other types of objective function [5]. MAPF could very well be compared to TUSP, where the goal is to route trains from their initial entry to the exit of the shunting yard. While there are various differences between the two algorithms, they share a very similar goal.

TUSP would not be the first problem where MAPF is applied to the domain of trains. During the 2019 Flatland Competition[1], team JelDor applied MAPF as a potential solution [6], which got them third place. The goal of the Flatland competition was to route a set of agents from A to B between the various stations. During the competition they used a greedy approach where they planned the trains sequentially in the order of descending speed. Despite TUSP and Flatland being in the domain of trains, team JelDor's solution would not work for TUSP. The Flatland competition was merely a routing problem from A to B, while TUSP requires matching and parking to be solved. However, this shows us that MAPF is a valid option for the train-domain.

MAPF itself is a well-researched research-field in the AI community and it is still being actively researched. The general problem of finding an optimal path for every agent from their start-position to their goal-position without colliding with each other is NP-Hard [7]. Therefore, much work is being done on finding heuristics for this problem to solve it as fast as possible. Recent work focuses on robot routing in warehouses [8, 9]; non-player character routing in games [10]; and airplane taxiway scheduling [11]. In this section, the core ideas to solve MAPF are reviewed. We start with an algorithm that is sub-optimal but often used in practice. We then explain the concept of Conflict-Based Search, which is the current state-of-the-art for MAPF.

2.2.1. Cooperative A*

Cooperative A* (CA*) is an algorithm that sequentially plans individual agents [12]. The search for the shortest path is performed in space-time for every agent. An agent has the option to either wait at the vertex it is currently at or move to a neighbouring vertex, both of these actions advance the time forward. After the agent is done finding its shortest path, it reserves the vertices in its path in a reservation table. Entries in this reservation table are assumed to be impassible and other agents are then no longer able to use the vertex at that time. It is not always possible to find a path for all agents on all instances with this algorithm nor are the found paths optimal. This kind of solution is often used when routing NPCs in games [10, 13]; where computations have to happen fast and an optimal solution is not required.

2.2.2. Conflict-Based Search

Conflict-Based Search (CBS) was first introduced by Sharon et al. [7]. CBS is a search-tree algorithm that, unlike CA*, does find the optimal path for all agents. This algorithm can be found in Algorithm 1. Initially there is only the root node in the CBS search tree. In this node, CBS finds the shortest path for every agent using a shortest path algorithm, in general A* is used. This root node is then added to the OPEN set. CBS then continues to select the node with the lowest sum of paths until it finds a final solution. In this node with the lowest cost CBS checks whether there are any collisions; if there are none, the algorithm has found the optimal solution. However, if a conflict is found at node v at time t between agent a_i and a_j, CBS creates two constraints c_1 and c_2. With these constraints CBS prioritizes either of the two agents. For each constraint generated CBS creates a new node in the search-tree, this is done by adding them to OPEN. CBS then, for each new node, recalculates the route for the agent that has to find a new path, the constrained agent. When CBS finds a solution with no conflicts, we know that this solution is optimal, since CBS always takes the solution with the lowest cost from the OPEN set. This is sufficient, because when adding constraints it is impossible to find a shorter route, since every agent already found the shortest possible route given the current constraints.

[1] https://www.aicrowd.com/challenges/flatland-challenge/

Algorithm 1 CBS

```
R.Constraints ← ∅
R.Solution ← agent paths found through $A*$
R.Cost ← SumOfPaths(R.Solution)
OPEN ← {R}
while OPEN not empty do
    P ← $min$(OPEN)
    OPEN ← OPEN / P
    if P has no conflicts then return P
    Conflict ← first collision $(a_i, a_j, v, t)$ in $P$
    for agent $a_i$ involved in Conflict do
        Q.Constraints ← P.Constraints ∪ $(a_i, s, t)$
        Q.Solution ← P.Solution
        Update Q.solution by calling A*($a_i$, Q.Constraints)
        Q.Cost ← SumOfPaths(Q.Solution)
        OPEN ← OPEN ∪ {Q}
```

Many different CBS variations have been built using this CBS-framework as base. This ranges from speed-ups by using heuristics [14]; changing the objective function to minimize the amount of missed deadlines [5]; or allowing the algorithm to work with continuous time [15]. Noteworthy, all of these algorithms use the same CBS-algorithm as a base.

CBS has been shown to be able to route up to 8 agents consistently on warehouse-like grids, and even being able to solve some problems consisting of 12 agents [16]. We expect that the warehouse-like grid shares many commonalities with train tracks; you have various disjoint hallways, which are then connected by small passageways. These passageways can be seen as the switches between tracks and the hallways as the tracks themselves.

2.2.3. Push and Rotate

Push and Rotate (PR) is a MAPF algorithm that is proven to be complete if there are at least two empty vertices [17]. The algorithm starts by finding the shortest path for every agent to its destination, after which it then attempts to move the agent towards each goal for each timestep. If two agents collides when moving towards their destination, the agent with the lower priority is *pushed* out of the way. The algorithm then attempts to *swap* the position of the two agents, such that the two agents occupies the position the other agent had at the time of the collision. This involves pushing the agents to a node with a degree of three or higher, such that they can perform the *exchange* operation. This *exchange* operation moves the agents to the neighbors of the vertex of degree three, such that the agents can pass each other, as is seen in Figure 2.1.

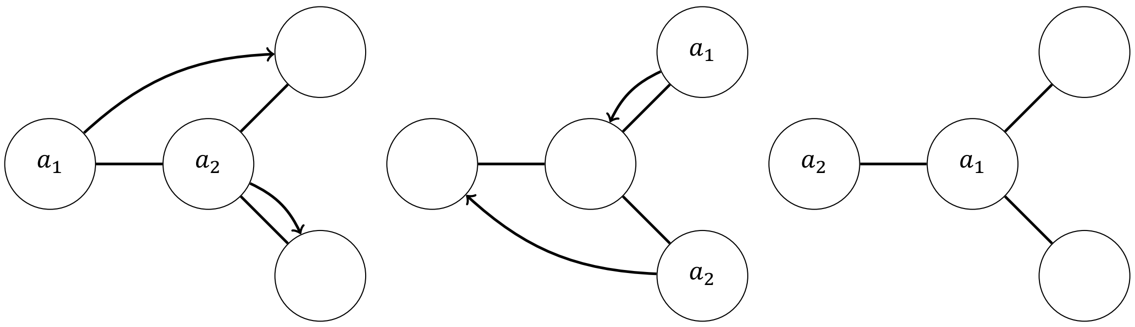


Figure 2.1: The *exchange* operation on a vertex of degree three.

The main drawback of this approach is that undirected edges are a requirement for this algorithm, since directed edges could put the system in an unrecoverable configuration. The exchange operation requires agents to be able to pass each other on the same node by going back and forth on the same edge. While a potential solution to this problem could be found, it could prove to be very complex.

2.2.4. Branch-and-Cut-and-Price

Branch-and-cut-and-price (BCP) combines branch-and-cut and branch-and-price into a single algorithm. BCP was previously used for Vehicle Routing Problems, but it was not until recently that this algorithm was adapted to work in an MAPF context [18]. BCP is a decomposition of the problem, where the master problem selects paths from a large set of paths P. This set P is initially empty, all variables of the LP problem are set to 0. The pricer enlarges the solution space by allowing more of the omitted variables into the solution, which the master problem can then select from. Finally we have a separator, which finds conflicts caused by paths selected by the master problem. The separator then adds constraints to the master problem, such that the conflicts are resolved.

BCP was shown to work better than CBS on instances involving large and open maps. Unfortunately, no benchmarks were ran using the warehouse-like benchmark set and we can therefore not clearly see whether it works great in situations containing many corridors. While the solving methods for BCP are different from CBS, models are adaptable and the constraints generated are very similar. BCP still requires the notion of edge- and node-conflicts, which have to be translated into LP constraints. Therefore, the different constraint ideas proposed for edge- and node-conflicts for CBS can be translated into BCP by devising the LP constraints.

2.3. MAPF and TUSP

While MAPF and TUSP are very similar, they both want agents to go from A to B, both have some major differences as well. In this section we describe what alterations have to be made for an MAPF algorithm to solve a TUSP instance.

There are three core aspects of TUSP which require alterations to the general MAPF problem. First of all, MAPF is usually used in a grid-world where an agent can only move into the cardinal directions. The freedom of movement of a train is severely limited in a shunting-yard. Movement requires the use of switches to move to other tracks and these switches have limited degrees of freedom. Furthermore, a train is not able to reverse (or set-back) instantaneously. For a train to reverse, a train-driver manually has to turn off the train on one side and walk to the other. This is a very costly operation and cannot be ignored when modelling TUSP to be solved by MAPF. This would require the use of directed edges rather than the undirected edges which are generally used in MAPF.

Secondly, a train can only perform a wait-move on certain tracks, as opposed to regular MAPF where a train can wait on any node. This means that trains have to be restricted from performing a wait-move when a track does not allow such moves.

Lastly, trains have a certain length. This means that these trains occupy a certain amount of track whenever they are parked. In MAPF an agent usually occupies a single node and does not care about its size. However, some research has been done on CBS working with larger agents [19]. Here they used 3D collision detection algorithms to check whether an agent ever crosses bounds with another agent. However, this approach is deemed excessive for our purpose, since trains can only collide with each-other when they are residing on the same track. Furthermore, only a single train can traverse a switch at a time-step regardless of size. Therefore, the size constraints are only relevant when trains are parked on the same track. Using a collision detection algorithm between every agent in the shunting-yard regardless of location is therefore unnecessary.

2.3.1. Conclusion

With this literature review we have seen that algorithms which solve the Train-Unit Shunting Problem (TUSP) still struggle when routing trains on the shunting yard. Our proposal is to have a dedicated Multi-Agent pathfinding (MAPF) algorithm solve the TUSP.

The current state of the art of MAPF is Conflict-based search (CBS), Push and Rotate (PR), and Branch-and-Cut-and-Price (BCP). While other solutions exist, these closely resemble CBS and any model proposed in this thesis could be adapted for these other **CBS**-algorithms. In this thesis we opt to use CBS over BCP and PR, due to the easily available source-code and the ease-of-use. BCP would provide a much greater challenge to link to the multi-agent simulator provided by the NS, which is not the goal of this thesis. Rather, the goal is to focus on the model such that TUSP can be solved as a MAPF problem. Furthermore, PR is not a valid candidate, since it might be difficult to fit to a TUSP model, due to undirected edges being a requirement.

For TUSP to be solved as a MAPF problem, some changes will have to be made to the MAPF model. These changes involve having to allow multiple trains on the same node, only allow trains to perform a wait-move on certain tracks, and have the algorithm work on a directed graph-world rather than a grid-world. How these changes are made will be discussed in the following chapter.

3 Problem Definition

To solve the Train Unit Shunting Problem (TUSP) we need to define our model. In this chapter, we define the routing, parking, and matching aspect of the TUSP. Moreover, we explain the relaxations that we are applying to TUSP for it to be solvable as a MAPF-problem. In general, the definition follows the definitions of the local-search method[1]; however, due to the different solving method we introduce new variables to define the routing of the trains. The usual TUSP defines 4 sub-problems: Matching, Routing, Parking and Servicing. The main goal of units on a shunting yard is to be served and leave on time, the matching aspect of TUSP. The other aspects are to ensure that we can service as many trains as possible at the same time on the shunting-yard. We start with introducing the concept of Routing and Parking, these are heavily intertwined and therefore explained alongside each other. We then finish up with the Matching of incoming to outgoing train units. For this thesis, the sub-problem Servicing is omitted for the sake of simplicity.

3.1. Routing & Parking

The aim of Routing & Parking is to guide a train through the track to find a path from A to B without colliding with any other trains. Before we start we need to define how the various tracks interact with the agents. A shunting-yard consists of tracks $\mathcal{T} \in \mathcal{T}r$. A track has a length $l_{\mathcal{T}}$, which is the upper-bound of the total length of trains it can harbor. Furthermore, a track always has an A- and B-side which connect with the neighbouring track. The set $A_{\mathcal{T}}$ and $B_{\mathcal{T}}$ define the neighbors of $\mathcal{T}$ on the A-side and B-side respectively. A train can only park on a track if $\mathcal{T} \in P$ and set-back if $\mathcal{T} \in S$, where P is the set of parkable tracks and S is the set of tracks where set-backing is allowed. There are various different types of tracks, which we have to model into a graph-structure where every track is a node and these nodes are connected by edges.

On these tracks we have train units. In the general TUSP, a train can consist of multiple train units. For simplicity we assume that a train is only 1 train unit at most. The train units are defined as $tu \in TU$ and have a type (ty_{tu}) and a subtype (st_{tu}), which tells you the amount of carriages a unit has. This can be used to infer the length of a train unit, which is defined as l_{tu}. When a train unit is on the shunting yard, they have a current location at time t $(\mathcal{T}^t_{tu})$ and a direction (d^t_{tu}). Furthermore, when a train enters a shunting yard it has an initial position, direction and arrival time. However, a train does not have an end-goal, this is defined by the matching in section 3.2.

3.1.1. Direction

Before we can explain the various tracks and their differences we first need to introduce the notion of Direction. A direction of a train unit is the set of nodes it can directly access from its current position, that is without having to reverse. Reversing is only allowed on certain tracks and therefore we need to restrict the movement of a train when it is moving through nodes which do not allow for reversing. When a train is unable to reverse its current direction is decided by the current track and the track the train was previously on. A visualization of this concept can be seen in Figure 3.1. Here the label of a node is set to the identifier of the node itself and the direction of that node. When reversing on a track is allowed we add an edge between the nodes with the same identifier, but different direction. In this case, only reversal on node 2 is allowed. This concept allows for shortest-paths algorithms to work, while keeping in mind that reversal is only allowed on certain nodes. Note that in the actual algorithm, the direction of a train is a property of the state of a train rather than a node; however, this is merely syntax and does not influence the semantics. This is done to keep the amount of nodes in memory low, since if we were to model the entire graph with reversals, it could quickly grow costly to store the entire graph. If we simply add direction in the state of a train we only have to keep a one-dimensional version of the graph, with no direction, in memory.

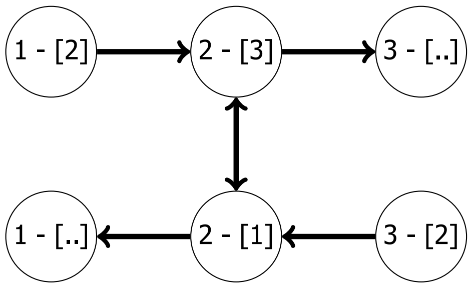


Figure 3.1: Graph representation of reversing

3.1.2. Tracks

Now that we have explained the notion of direction on a node, we can explain the various different tracks which we have to account for in our model. We use a similar notation as in Figure 3.1, where the direction is part of the node.

- **Regular Track** has 1 neighbor on each side: $|A_\mathcal{T}| = |B_\mathcal{T}| = 1$, in general trains can park on this track, but this is not always the case due to shunting yard specific rules. These restrictions are defined in the location-data of the shunting yard. In Figure 3.2, a visual representation can be seen of a regular track where reversing is allowed.

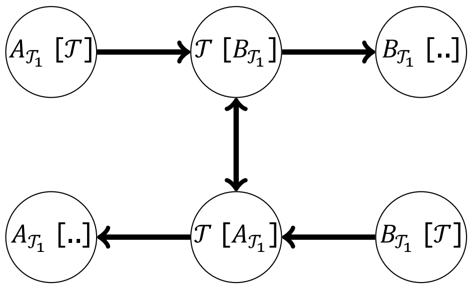
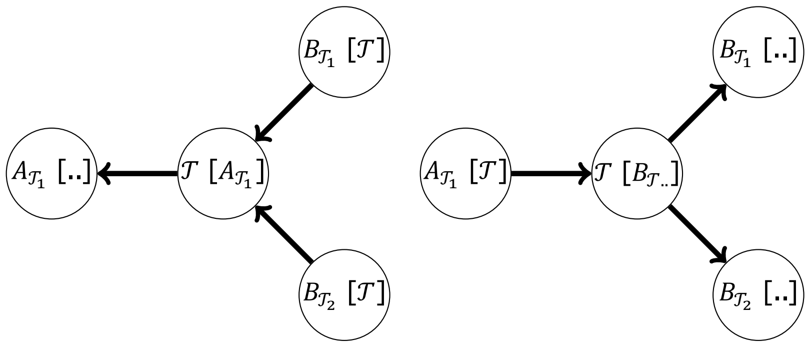

$A_{\mathcal{T}_1}$ [$\mathcal{T}$]
$\mathcal{T}$ [$B_{\mathcal{T}_1}$]
$B_{\mathcal{T}_1}$ [..]
$A_{\mathcal{T}_1}$ [..]
$\mathcal{T}$ [$A_{\mathcal{T}_1}$]
$B_{\mathcal{T}_1}$ [$\mathcal{T}$]

Figure 3.2: Graph representation of a regular track where reversing is allowed

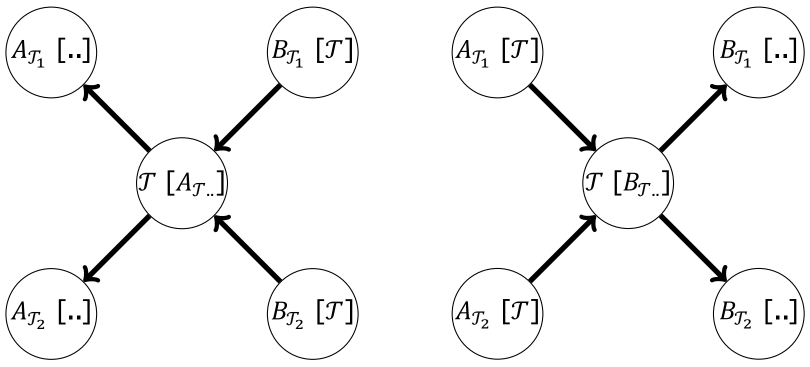
- **Switch** has, in general, 2 neighbors on one side and 1 neighbor on the other: $|A_\mathcal{T}| + |B_\mathcal{T}| = 3$, with $|A_\mathcal{T}| > 1$ and $|B_\mathcal{T}| > 1$. In general, trains cannot park on these tracks. Other than the regular switch, there are switches with different restrictions to the amount of neighbors they have. In Figure 3.3, a schematic illustration of switch $\mathcal{T}$ can be seen. In this example, a train can go from $A_{\mathcal{T}_1}$ through $\mathcal{T}$ to either $B_{\mathcal{T}_1}$ or $B_{\mathcal{T}_2}$ and back. However, a train cannot travel between $B_{\mathcal{T}_1}$ and $B_{\mathcal{T}_2}$, even with the use of $\mathcal{T}$.

- **English Switch** is very similar to a regular switch; however, in this case we have 2 neighbors on both sides: $|A_\mathcal{T}| + |B_\mathcal{T}| = 4$, with $|A_\mathcal{T}| = 2$ and $|B_\mathcal{T}| = 2$. A train cannot move between the



$B_{\mathcal{T}_1}$ $[\mathcal{T}]$
$A_{\mathcal{T}_1}$ [..]
$\mathcal{T}$ $[A_{\mathcal{T}_1}]$
$B_{\mathcal{T}_2}$ $[\mathcal{T}]$
$B_{\mathcal{T}_1}$ [..]
$A_{\mathcal{T}_1}$ $[\mathcal{T}]$
$\mathcal{T}$ $[B_{\mathcal{T}_{..}}]$
$B_{\mathcal{T}_2}$ [..]

Figure 3.3: Graph representation of a Switch



$A_{\mathcal{T}_1}$ [..]
$B_{\mathcal{T}_1}$ $[\mathcal{T}]$
$\mathcal{T}$ $[A_{\mathcal{T}_{..}}]$
$A_{\mathcal{T}_2}$ [..]
$B_{\mathcal{T}_2}$ $[\mathcal{T}]$
$A_{\mathcal{T}_1}$ $[\mathcal{T}]$
$B_{\mathcal{T}_1}$ [..]
$\mathcal{T}$ $[B_{\mathcal{T}_{..}}]$
$A_{\mathcal{T}_2}$ $[\mathcal{T}]$
$B_{\mathcal{T}_2}$ [..]

Figure 3.4: Graph representation of an English Switch

tracks on the same side without reversing. In Figure 3.4, a train on the $A_{\mathcal{T}}$-side can only move to a track on the $B_{\mathcal{T}}$-side and vice versa.

- **Bumper** has 1 neighbor in total: $|A_{\mathcal{T}}| + |B_{\mathcal{T}}| = 1$. Furthermore, a train cannot cross through a Bumper. The goal of a Bumper is to simply ensure that every Regular Track has at least two neighbors. In the graph, a train can only go towards a bumper, this means that visiting this node always results in a dead end and is therefore never added to the final solution. This closely resembles real-life, where a bumper is a physical bumper that a train cannot pass.

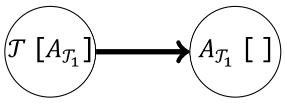


Figure 3.5: Graph representation of a bumper

3.1.3. Space-Time

So far we have only mentioned the model of the graph; however, we require a notion of time to be able to fully model TUSP. Since whenever a train moves, time advances. For regular moves, the graph is simply extended with an extra time-dimension. In this case the edges do not directly connect to its neighbor, but rather to its neighbor in the next time-step. A visual representation of this model can be found in Figure 3.6. When it comes to parking, we only connect nodes that are allowed to be parked on to itself in the future. In the figure, this can be seen for node 2. The other nodes are not allowed to be parked on and are therefore not connected with themselves in the future.

For a train unit to traverse a track, some approximations about the duration of the move have to be made. For this algorithm it is preferred to discretize the time. While the generated instances of a shunting yard measure actions in seconds, we opt to go for a discretization of a minute. This has to do with limiting our search space. For the following tracks this discretization could bring inaccuracies:

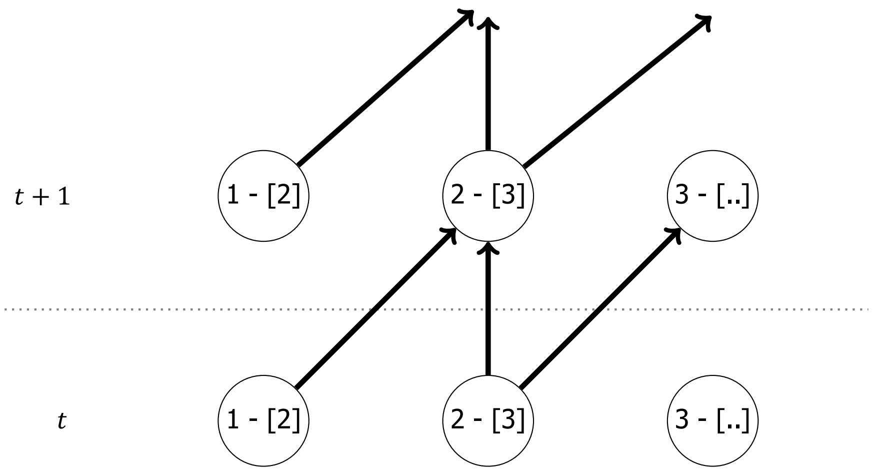


Figure 3.6: Advancing Time and Parking

- If a switch is set to the correct configuration already, traversing it should be instant; however, if the track is not set to the correct configuration the switch has to be operated. The time it takes to operate manual switches is about two minutes and for automatic switches it takes around one minute. To simplify the algorithm we assume that crossing any switch takes 1 minute.
- Setting a train back (reversing) can take quite long, since a train-driver has to walk from one side to the other, which could take 5 minutes. Therefore, assuming that this costs a minute is quite a big underestimation, but for simplicity sake we assume every step has the same duration.
- A train can park for multiples of 1 minute. In the real world it could possibly park for less than a minute, but since all time is discretized it only makes sense to do so for parking too.

3.2. Matching

When a train unit enters a shunting yard, it does not have an end-goal. It simply requires service and has to find a path to the exit at a time where we expect a train to leave. Furthermore, this departing train has to be of the same type and sub-type to leave. We define a departing train as a tuple (t, v, d, ty, st). We are looking for a train at time t; on track v; with direction d; of type ty; and of sub-type st. When a train unit tu matches this tuple, if the values of the departing train and the shunting unit tu are the same, it is allowed to leave the shunting yard. The end-goal of a train in the routing algorithm is to match with a departing train and be allowed to leave the shunting yard.

4

Multi-Agent Pathfinding approach

In this chapter, we discuss the Multi-Agent pathfinding (MAPF) approach taken to solve our version of the TUSP, which was defined in the previous chapter. We start with the bare-bones version of our approach, where trains can only occupy a single node. We then extend the model by allowing multiple trains to park on the same track and give two different solutions on how to solve this problem. We opt to use the bare-bones version of Conflict-based search, with no additional heuristics. This is to keep the complexity of the algorithm at a minimum, while being able to be expanded upon in the future. Most, if not all, heuristics found for CBS could potentially be applied here, but this would only complicate the algorithm explained in this thesis.

4.1. The Base

The core of the algorithm is very similar to the original Conflict-Based Search (CBS) as used in MAPF. The alterations made to this algorithm will be discussed in this section. We will refer to this algorithm as TUSP-CBS. In this section, we first explain how the low-level search works and how constraints are used to find a possible route. We then explain how these constraints are generated given a path for every agent.

4.1.1. Low-Level Search

The low-level search resembles mostly an A*-algorithm. Due to the way this problem has been modeled as a graph in the previous chapter, we could freely run an A*-algorithm on this model. However, some changes were made to keep the size of the graph manageable, even with the addition of time-space. This means that we do not explicitly generate the entire space-time graph; rather, we opt for using a state which can keep track of additional information such as direction and time. This state can then be used to see which node the A* algorithm can then visit next. This causes us to end up with three different cases when generating our next move, namely moving to a neighbor, parking, and set-backing. These various cases will be explained individually in this section.

In Algorithm 2, an overview of the altered A* algorithm can be seen, we name this A*-TUSP. The alterations made are as follows: first of all a match function is used to check whether a train has reached its goal; a more thorough explanation will be provided in section 4.2. Furthermore, we need to find a way to guide the search algorithm in the right direction. We do this using the $gScore$. Usually one would use the total time spent to get from the start to the current location as the $gScore$. However, since only movements are costly, and parking is not, we only increase the $gScore$ when moving and not when parking. This means that parking edges are seen as 0-cost; however, they do still advance the time of the state. This means that trains will prefer to park for as long as possible, as opposed to moving around aimlessly.

Finally, there is the $validMove(t, \mathcal{T}, C)$ function. This function checks whether the current location v of train tu at time t is valid according to the given constraints C. It does so by checking if the input values match any of the constraints, if any are matched it means that the train is not allowed to be at this location and we therefore forfeit this node in the low-level search.

For clarity, the visited list is omitted from the algorithm; however, this list would function as it would in any A*-algorithm.

Algorithm 2 A*-TUSP

```
start.time ← tu.arrival_time
start.location ← tu.initial_location
start.direction ← tu.initial_direction
gScore[start] ← $0$
OPEN ← {start}
came_from[start] ← ∅
while OPEN not empty do
    current ← node in OPEN with lowest gScore
    if validMove($tu$, current.time, current.location, constraints) then
        continue
    if match(current, $tu$, constraints) then
        return reconstruct_path(came_from, current)
    OPEN ← OPEN / current
    OPEN ← OPEN ∪ move_to_neighbor(current)
    OPEN ← OPEN ∪ park(current)
    OPEN ← OPEN ∪ setback(current)
```

The first move a train should be able to make is to move to a neighbouring track, see Algorithm 3. A train can only move towards a track it is currently facing, in other words, the train can move to any track in its direction set. We thus have to make search-nodes for any track it is looking at, since a potential shortest path could go through any of these neighbors. As was previously explained in section 3.1, a train takes 1 minute to move to a new track. Therefore, the time is forwarded by 1 minute in the newly created state. To find the new direction of the train we need the track the train is currently on and the track the train is moving to. This new direction will then be tracks on the opposite side of our current track. Finally, the algorithm has to update the $gScore$ and update from which state the train came from to allow for a reconstruction of the taken path.

Algorithm 3 Moving a train to a neighbouring track

```
function move_to_neighbor(current)
    new_states ← ∅
    for neighbor in current.direction do
        new_state.time ← current.time + 1
        new_state.location ← neighbor.location
        new_state.direction ← newDirection(current, neighbor)
        gScore[new_state] ← current.time + 1
        came_from[new_state] ← current
        new_states ← new_states ∪ new_state
    return new_states
```

A train can also decide to park on the current track, the algorithm for this is very similar to the previous algorithm an is therefore not included here. If one so desires, it can be found in the Appendix: Algorithm 9. In the case of parking the new state simply contains the same location and direction, but the time is advanced by 1 minute. This move is only allowed if $\mathcal{T} \in P$, in other words, if a track is allowed to be parked on. When a train parks, the algorithm does not increase the *gScore* and therefore this new search-node does not cost more than the previous search-node.

Set-backing is the final move to be explained, in this case a trains set-backs and reverses its direction. This move is very similar to the parking move above, the direction of the train is now reversed by this move. That means that if the train is currently looking towards the A-side, its direction is changed to the B-side and vice versa. However, since changing the direction of a train is costly, the algorithm increases the *gScore* by 1. This algorithm can once again be found in the Appendix: Algorithm 10.

For any of these moves it holds that the algorithm does not add the new search-node to the queue if we have already seen that node before. In that case, the algorithm has already explored this possibility in a previous step and it is therefore pointless to cover this node again. It is, however, important to realize that one can only skip a node if the node has been seen at the same time before.

4.1.2. Constraint Generation

For the A*-TUSP search to work correctly, we need to be able to generate constraints. Every constraint C is a tuple of (t, v, tu): time, location and unit. Simply said, with a constraint we disallow a certain train unit to be at a specific location at that time. These constraints are generated whenever a collision occurs to prioritize one of the two involved train units in its own search-node. In section 4.3 we go into more detail on having multiple trains on the same track, where parking is allowed. For this section, we focus on the case where two trains collide on a non-parking track. There are two different kinds of collisions, so called edge-collisions and node-collisions.

- **Node-collisions:** This type of collision occurs when two trains tu_i, tu_j are on the same track $\mathcal{T}$ at the same time t. We then create a search-node with the constraint $(t,\ \mathcal{T},\ tu_i)$ and another search-node with the constraint $(t,\ \mathcal{T},\ tu_j)$.
- **Edge-collisions:** This type of collision occurs when two trains tu_i, tu_j take the same edge at the same time. Simply said, they would swap places without actually being on the same node at the same time. When tu_i is at track x, tu_j is at track y at time t and they would switch location on the next time-step, we have to create two constraints: $(t+1,\ y,\ tu_i)$ and $(t+1,\ x,\ tu_j)$. This would stop the train from crossing the edge at that time-step.

With these collisions found, it is then the responsibility of CBS to replan these trains given these new constraints using the low-level search.

4.2. Matching

The algorithm is currently able to route trains on the shunting yard, but it is not yet able to find an end-goal for the trains. A train has found its end-goal when it finds an outgoing train matching its description, as seen in Algorithm 4. The goal of matching in this algorithm is to let the constraint generator solve the occurring conflicts. That is, if two trains match with the same outgoing-train we simply let the constraint generator generate the necessary constraints to come to a solution. The constraint generator would find two train units to be at the same location at the same time and would therefore create new search-nodes to prioritize the one or the other. Since the current node was already validated with the constraints in the low-level search (Algorithm 2), we can be sure that if this match occurs, there are no conflicts. The advantage of this approach is that we have now integrated the matching into the pathfinding algorithm, rather than finding a matching prior to the pathfinding algorithm.

Algorithm 4 Matching outgoing trains

```
function Match(current, $tu$, constraints)
    for $(t, v, d, ty, st) \in$ outgoing_trains do
        if $(t, v, d, ty, st) =$ (current.time, current.location, current.direction, $ty_{tu}$, $st_{tu}$) then
            return True
    return False
```

4.3. Multiple Trains on a Track

While the previously mentioned methods work when a single train occupies a single track, it becomes more difficult when we allow multiple trains to be parked on the same track. This problem differs from the regular MAPF problem in that we now need to keep train length in mind. A simple way of modelling this would be to create a search node for every few meters of track. The main issue of this is the huge increase in the amount of low-level search nodes and the lack of precision due to rounding. Therefore, any suitable solution needs to provide precision and minimize the amount of both low-level (A*-TUSP) and high-level (CBS) search-nodes. In this section, we propose two different solutions. One of them uses a queue-based approach which keeps track of the order and length of the trains one a track. The second approach keeps track of the exact location of a train on a track. Both approaches build upon the previously mentioned framework, they simply expand on certain aspects that are mentioned in this section.

4.3.1. Queue-approach

The first approach we discuss will be the queue-like approach, which we will call **TUSP-CBSQ**. This approach first plans the individual trains, after which it then finds all conflicts that occur when trains try leave a track when it is blocked by another train. We start by explaining how crossings are detected by TUSP-CBSQ. We then follow up with tracking the total capacity used on a track. The low-level A*-TUSP algorithm is unchanged, we merely add more constraints.

Detecting Crossings

To detect crossings between trains we run a small-scale simulation. For crossings it is necessary to compare every train with the others, but this does not necessarily have to happen at the same time. Therefore, we check only for crossings of two trains at a time. In this simulation we go through every move of the train and see whether a crossing would occur between these two trains. Algorithm 5 shows the algorithm to do so.

We provide the algorithm with train units as input, tu_1 and tu_2. We then find the first time t they can collide, which is when both units are on the shunting yard. We then keep track of the occupation of all locations v, this is done by keeping a list of the current occupation on that location at that current time. The idea is to insert a train coming from the left to the left-side of the list and vice versa for the right side.

We continue while both trains are still on the shunting-yard, in other words, when $v_{tu_i}^{t+1}$ exists. We then take a step for both units. When taking this step, we check for which direction the train is going. To do so, we check on which side new_v is of old_v. This is accomplished by checking whether new_v is in A_{old_v} or B_{old_v}. When this new location is on the A-side, it means the left-side of the track has to be clear and the train should be the left-most unit on the track. Therefore we can check whether the train is left-most in the occupy list, or right-most when the next location is on the B-side. We then append the train on the new location to the side from which we arrived from; on the left if we came from the A-side and on the right if we came from the B-side. This ensures that we catch all movements.

If we catch a collision between two trains, we create two constraints. Just like in the rest of the CBS algorithm, we allow one train the right of way, while we disallow another train to be there. The high-level search then decides which search-node is the most promising to search through.

Detecting Track Usage

In the previous section we only explored the notion of crossings; however, when comparing two trains we cannot infer the total amount of the length of the track used. To do so, we do have to compare every train at the same time.

A simple approach might seem to use a time-space list, where we sum the total amount of length used on a track at a certain time. However, this kind of list would use an enormous amount of space, especially when our instances start having a higher duration.

Algorithm 5 Crossing simulation

```
function Simulate($tu_1$, $tu_2$)
    $t \leftarrow max$(time of entry of $tu_1$, time of entry of $tu_2$)
    occupies[$\mathcal{T}$] ← [], $\forall \mathcal{T} \in \mathcal{T}r$
    while $\mathcal{T}_{tu_1}^{t+1}$ and $\mathcal{T}_{tu_2}^{t+1}$ exist do
        $t \leftarrow t + 1$
        $c_1 \leftarrow$ Take_step(occupies, $tu_1$, t)
        $c_2 \leftarrow$ Take_step(occupies, $tu_2$, t)
        if $c_1$ or $c_2$ is Collision then
            return $c_1$, $c_2$
    return False

function Take_step(occupies, $tu$, t)
    $old\_v \leftarrow \mathcal{T}_{tu}^{t-1}$
    $new\_v \leftarrow \mathcal{T}_{tu}^{t}$
    if $old\_v = new\_v$ then return
    if $new\_v \in A_{old\_v}$ then
        if occupies[$old\_v$][0] $\neq tu$ then return Collision
        else
            Remove $tu$ from occupies[$old\_v$]
            Add $tu$ to the right of occupies[$new\_v$]
    else if $new\_v \in B_{old\_v}$ then
        if occupies[$old\_v$][1] $\neq tu$ then return Collision
        else
            Remove $tu$ from occupies[$old\_v$]
            Add $tu$ to the left of occupies[$new\_v$]
    return Clear
```

Instead we opt for the algorithm in Algorithm 6, where a single list (*track_usage*) containing every location once is used. We then start at time 0 and simulate the total amount of length used for each time-step sequentially. After we have simulated all track occupations for a single time-step, we check the usage of the track. If the usage of the track is higher than allowed on that location, the algorithm has found a conflict. In this case it creates constraints for every single train that occupies this track, these constraints are then handled by the high-level search of CBS. The usage of this algorithm does mean that we lose the branching factor of 2, which is generally seen with CBS. Instead, we have as many branches as there are trains on the track. If we go over every single time-step without conflicts, it means that the solution is free of track-usage conflicts.

Algorithm 6 Track Usage

```
function Validate_track_length
    $t \leftarrow 0$
    while $t <$ end_time_scenario do
        track_usage[v] ← [], $\forall \mathcal{T} \in \mathcal{T}r$
        for $tu \in$ train_units do
            if $\mathcal{T}_{tu}^{t}$ exists then
                track_usage[$\mathcal{T}_{tu}^{t}$] ← track_usage[$v_{tu}^{t}$] + $l_{tu}$
        for $\mathcal{T} \in \mathcal{T}r$ do
            if track_usage[$\mathcal{T}$] > $l_v$ then
                return Conflict
```

4.3.2. Analysis of CBS vs CBSQ

In comparison with regular CBS, where we would have to split the tracks into many different nodes, we can now have many trains on a single node. To illustrate the gain of this approach, we provide an example. If we were to split a track $\mathcal{T}_1$ of 300 meters into 3 tracks of 100 meters, we could get a graph-layout as seen in Figure 4.1. This layout is already sub-optimal, since many trains are shorter than 100 meters. The unused space on every node is therefore wasted, which is not the case in a continues approach, like with CBSQ. If you were to discretize the track even further, the problems described below would only get more severe.

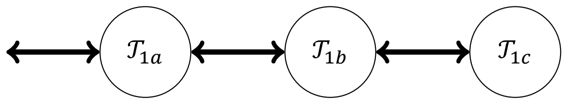


Figure 4.1: A way to split $\mathcal{T}_1$ into three nodes.

There are 3 important cases we have to analyze in this situation. Two of those cases are related to the order in which trains enter and leave the track. This can be in either LIFO (Last In, First Out) or FIFO (First in, First Out) order. The other case is related to a train leaving the track on the right-side. For the below examples we use three trains to show the exponential growth of the problem. However, the below holds for any amount of trains. This is due to CBS comparing every two pairs of trains to find conflicts and if no two trains have a conflict, there is no conflict between any of the paths.

LIFO

When trains enter and leave in LIFO order, the queue-model does not create any constraints. There are no crossings when trains enter and leave from the left-side, if they leave in the reversed order as they entered.

For the other method, where the tracks are split, we have the issue that all trains would initially be parked on $\mathcal{T}_{1a}$. This would then have the constraint generator create constraints for this node and create 3 additional search-nodes, each of them constraining one train to leave $\mathcal{T}_{1a}$. However, each of these 3 search-nodes still has 2 trains left on $\mathcal{T}_{1a}$, which requires more constraints to be generated. This will then result in a total of 9 search-nodes, since all 3 search-nodes create an additional 2 search-nodes. Now in each search-node, the end result is that 2 trains will be pushed to $\mathcal{T}_{1b}$, where this process repeats again. This process scales exponentially with the amount of trains and will therefore only get worse.

FIFO

When trains do leave in FIFO order, CBSQ has to generate constraints. When the rightmost train wants to leave, it requires the trains to the left to leave the track. This move generates an additional 4 search-nodes. Two where the train is not allowed to move and two where the 2 left-most trains are moved away from the track.

For the other method, we now have both the issue presented in the *LIFO* section, since we still have to park the trains that enter the track, and the additional issue that was presented above for CBSQ. For a train to leave it still requires the two left-most trains, the ones on $\mathcal{T}_{1a}$ and $\mathcal{T}_{1b}$ to leave the track. This then causes the same constraints as explained in the previous paragraph.

Double-Sided Tracks

Double-sided tracks are a combination of both problems described above. Either a train on one end wants to leave on the other, or trains are already at the right side of the track where they have to leave. Regardless, the queue approach generates fewer constraints because of the arguments provided above.

Conclusion

With this we have covered every possible case possible with CBSQ. We have shown why this is better than using plain CBS, with discretized tracks. Due to CBS comparing every possible pair of trains to find constraints, we know this holds when it involves more trains. However, one issue still remains, when many trains get parked on the same track, many constraints are generated. Therefore, this is not the end-all solution. **CBSQ** is exponential in the amount of conflicts that occur. Furthermore, it has a branching-factor bounded by the amount of trains on the shunting-yard. This is due to the fact that when the length-constraint of a track is violated, we create as many constraints as there are trains on the track.

4.3.3. Exact Location

Another approach we could take is to keep track of the exact location on a track for any given train, we call this method **TUSP-CBSL**. The general idea is similar to the doubly-linked queue; however, in this case, we move the train on the track itself rather than only handling conflicts when leaving or the length constraint is violated. When a train enters a track, it sticks to the side of entry as much as possible and only moves when required. This means that a train does not move unless another train is parked on top of the current train. In that case we slide the train further along the track.

This idea could be better than the algorithm of the previous section, since we do no longer need to create a node for every train when the length constraint is violated; rather, since every train has a location on the track we know exactly which train cannot fit on the track. Every option is still explored due to the search-tree still containing every possible solution to the parking problem.

A potential issue we might come across is the very same issue described in subsection 4.3.2, the section above. While we do not discretize the track, we still have to move trains on the track; rather than just taking into account the order in which they entered the track.

In this section we first explain how a new location for a train is found given the constraints, since this differs from the previously explained algorithm and needs to be understood before being able to understand the constraint generation, which will be explained later.

Search

Every train will initially stay on the side of arrival on a track. If a train unit tu_i is then parked on a track on interval $[s_i, e_i]$ at time t, we need to check whether this interval is still a valid parking interval on $t+1$. To do that we have to extend the definition of constraints. Previously a constraint was defined by a tuple of (t, v, tu). We now extend this tuple with two additional parameters: (t, v, tu, s, e), where s and e are the left-most and right-most value of an interval where a train should not be parked on. Therefore, it should always hold that $s \leq e$ for every constraint C. If a track cannot be parked on then these values can simply be set to 0, since the location on such tracks is irrelevant; either the entire track is blocked, or it is entirely open.

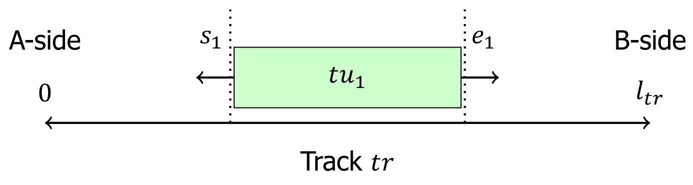


Figure 4.2: A visualization of a train on a track

For these intervals it holds that 0 is closest to the A-side and l_{tr} (track length) is closest to the B-side. For shunting unit tu_i this is done by checking the constraints to see if $s_c < e_i$ and $s_i < e_c$ does not hold for all constraints $c \in C_{t+1}$ on time. If this holds for all constraints, the train does not have to move from this position and we can create a search-node where this unit is parked on the interval $[s_i, e_i]$ on time $t+1$.

However, if a train is not to be parked on the same interval at $t+1$, the train has to be moved. There are two cases when moving a train, depending on the direction the train is facing:

- The train is currently facing to the A-side, towards 0, and we need to find an interval between $[0, e_1]$
- The train is currently facing to the B-side, towards l_{tr}, and we need to find an interval between $[s_1, l_{tr}]$

This concept is visualized in Figure 4.3. Here we find the train to be blocked on the next time-step and looking towards the B-side. Therefore we find the next open spot at $t+1$ and move the train there.

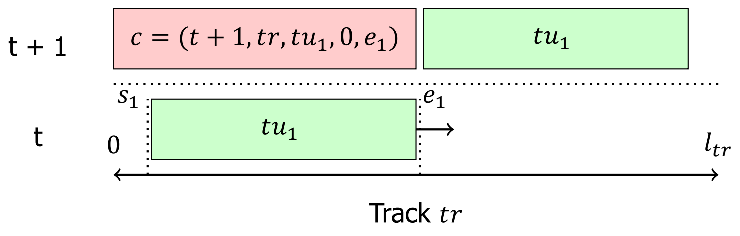


Figure 4.3: A visualization of moving a train on the same track

Algorithm 7 Finding Next Parking

```
1: $s, e \leftarrow s_i, e_i$
2: for $c \in C_{t+1}$ do
3:     if $e_c \geq e \geq s_c$ then
4:         $e \leftarrow s_c + 1$
5:     else if $s_c \leq s \leq e_c$ then
6:         $s \leftarrow e_c + 1$
7: if $e - s < unit\_length$ then return Infeasible
   return $[s, e]$
```

To find the next possible parking interval on the track, we have to go through all constraints C_{t+1}. Using Algorithm 7, we can find the parking spot nearest to this train. To minimize the amount of space used on a track we need to minimize the amount of empty space between trains. By moving only just past the constraint for this train, we know that we are optimally using the space allocated for this train to park on.

Furthermore, due to the double-ended queue property of the track, we know that when a train is found further ahead on the track, we cannot overtake this train. Therefore, only the closest interval free of constraints is a possible candidate for this train to park on. This idea is visualized in Figure 4.4. c_2 can only exist if a train entered the track from the right side. Otherwise, there would have to be a constraint covering x, since every train is sticking to their entry-side as much as possible. Since this train already covers the right side, there is no way for tu_1 to pass c_2 to go to a further free location on the track. Therefore, x is the only possible location for tu_1 to park on, if sufficient space is available.

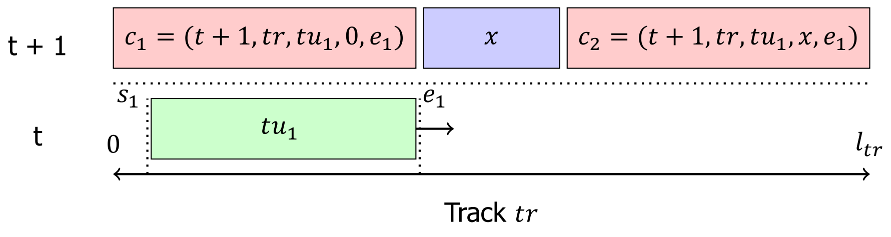

t + 1
$c_1 = (t+1, tr, tu_1, 0, e_1)$
x
$c_2 = (t+1, tr, tu_1, x, e_1)$
s_1
e_1
t
0
tu_1
l_{tr}
Track tr

Figure 4.4: A visualization of the gap between 2 constraints.

Given the interval provided by the algorithm, we need to check whether it is possible to reach this interval at the current time t. When the train is facing the B-side, this can be done by checking the interval $[s_i, s + unit\_length]$, where s_i is the left-most location at t and s the left-most location on $t + i$.

When the train is facing the A-side, we check the interval $[e - unit\_length, e_i]$. If this is impossible, there is no way for the train to move away from its current location and we can therefore conclude that taking this path in the low-level search is infeasible.

When trains want to leave the track, there needs to be an empty track from the train towards the exit. This once again causes us to research two different cases:

- The train is currently facing to the A-side, there must be no constraint for $[0, e_1]$

- The train is currently facing to the B-side, there must be no constraint for $[s_1, l_{tr}]$

When there is no constraint for the given interval, the train can freely leave the track.

Constraint Generation

To enable the low-level search to execute the above algorithm, constraints have to be generated. There are three types of constraints we have to consider.

- A train is parked on top of another train.
- A train tries to leave the track and another train is parked between it and the exit.
- A train moves to another location on the same track and another trains is in the way.

The first two cases are trivial, but the last case requires some additional explanation.

Parked on top When train tu_1 and train tu_2 are parked on top of each other at time t, it means their intervals $[s_1, e1]$ and $[s_2, e_2]$ overlap. This happens when both $s_1 < e_2$ and $s_2 < e_1$ hold. In this case we create 2 nodes, with different constraints:

- Constrain tu_1 to not be at $[s_2, e_2]$ at time t
- Constrain tu_2 to not be at $[s_1, e_1]$ at time t

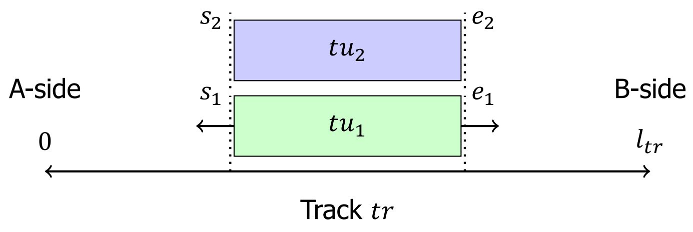


Figure 4.5: A visualization of trains parked on top of each other at time t

Leaving the track When train tu_1 wants to leave the track at time t, there must be no train between it and the exit. This calls for two cases, one where the train leaves on the A-side and when where the train leaves on the B-side. This means we have 2 different intervals to check; however, the collision detection remains the same for both intervals.

- The train is currently facing to the A-side, we need to check $[0, e_1]$ for collisions
- The train is currently facing to the B-side, we need to check $[s_1, l_t]$ for collisions

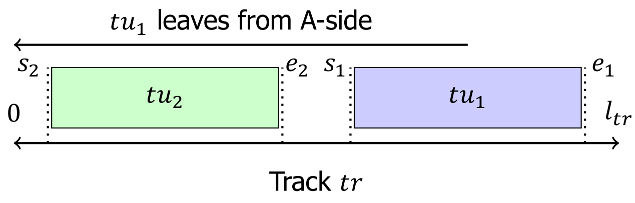


Figure 4.6: A visualization of train tu_1 leaving the track but being blocked by tu_2.

For these intervals it means, that no other train than tu_1 must be present at time t. If there is, we create the following two constraints:

- tu_1 is not allowed to leave the track at time t. We constrain tu_1 to not be at $[0, e_2]$ or $[s_2, l_t]$ at time t, for leaving the A-side and B-side respectively.
- tu_2 is not allowed to park at its current location. Therefore, we constrain tu_2 to not be at $[0, e_1]$ or $[s_1, l_t]$ at time t, for when tu_1 leaves either the A-side or B-side respectively.

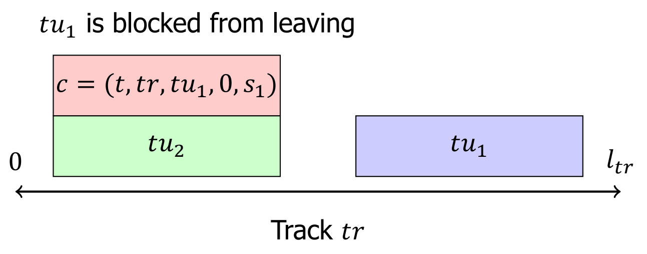


Figure 4.7: tu_1 is not allowed to left the track from the A-side at time t.

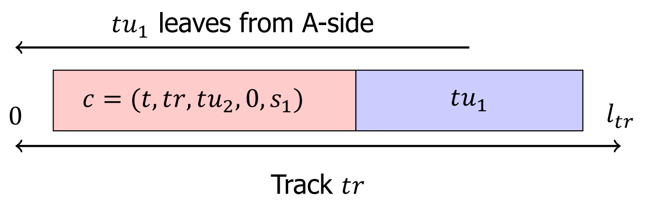


Figure 4.8: tu_2 was not allowed to park on track tr at time t and has therefore left the track beforehand.

Moving on same track When train tu_1 and tu_2 move on the same track, we have to make sure they do not cross each-other. For this we need to check both the location at time t and $t+1$. The new location at $t+1$ of u_1 and tu_2 are defined as $[s_1^*, e_1^*]$ and $[s_2^*, e_2^*]$ respectively. There are now two cases for which intervals we have to compare.

- The train is facing the A-side, $[s_i^*, e_i]$ has to be free at time t for tu_i to move.
- The train is facing the B-side, $[s_i, e_i^*]$ has to be free at time t for tu_i to move.

We can then use the same interval checking as used in Figure 4.3.3. We now show how to create the constraints when u_1 is facing to the A-side and u_2 is facing to the B-side.

- Constrain tu_1 to not use $[s_2, e_2^*]$ at time t.
- Constrain tu_2 to not use $[s_2^*, e_2]$ at time t.

The constraint generation for the other cases is very similar.

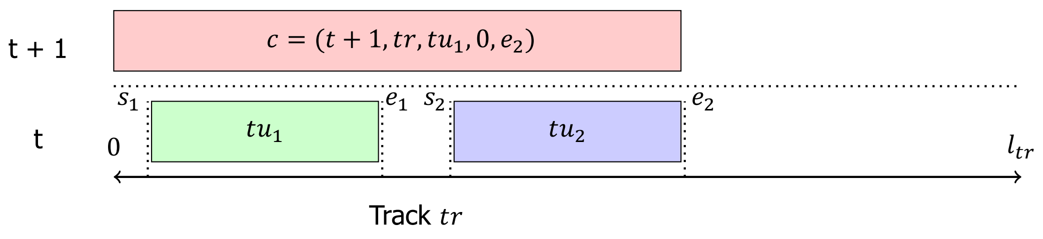


Figure 4.9: A visualization of a case where tu_1 could cross with tu_2

Figure 4.9 shows a situation where this occurs. If tu_2 leaves on the A-side at time $t+1$, the constraint c would be generated for tu_1. In that case, tu_1 has to be rerouted. When tu_1 is rerouted it would like to move to a location between e_2 and l_{tr}; however, tu_2 blocks this movement, so we have to ensure that during constraint generation these kind of movements are caught and a constraint is generated.

Conclusion

We have previously shown three different cases where collisions can occur and how to handle the constraint generation of these cases.

- Trains are parked on-top of each other

- Trains cross each other to move to another location on the same track
- Trains cross each other to leave the track

These cases cover all moves a train could make on a track. With these constrains we only remove the search-nodes that are unable to become a final solution, since multiple trains would have to be at the same location, and generate constraints such that all possible solutions are explored. Due to the nature of CBS, where we could go through both branches of the search-tree, we know that every solution has the potential to be explored. Furthermore, due to the nature of Algorithm 7, we know that we use the space on the track as efficient as possible. Finally, **CBSL** is exponential in the amount of conflicts that occur. It has a branching-factor of 2, since every conflict causes two new search-nodes to be created. This is in line with the general time-complexity of CBS.

4.4. Heuristics

In this section we introduce the heuristics which were used to speed up the algorithm. First and foremost we discuss how it is possible to examine conflict-heavy areas of the path-planning as early as possible. This could potentially solve a lot of conflicts before getting to conflicts occuring in less conflict-heavy areas.

4.4.1. Match Conflicts

One such area where lots of conflicts occur is the exit of the shunting yard. Since many trains match to the same departing train, we find that lots of conflicts occur. What could potentially save us a lot of unnecessary calculations is to find these match conflicts before we start looking at the remainder of the path. In general, we want to look for conflicts from start to finish. However, in this case, we check whether two trains are on the same location $\mathcal{T}$ at the latest time t where they are still on the shunting yard. If these trains are at the same location, we can create the constraints as done before. This then saves us looking at conflicts in the remainder of the path caused by unfortunate matching, but rather solves the issue at the root of the problem, which is the matching. CBS variations with this algorithm will be suffixed by "**-Matching**".

4.4.2. Spreading Trains

A major issue that occurs with the CBS-algorithm is that, when many trains have the same entry- and exit-point, the trains have the same shortest-path. This causes a lot of collisions around the first few available parking-tracks. These first few parking-tracks are in general overcrowded, there are more trains on the track than the track can handle, and this will require many constraints to distribute the trains. For example, if we were to use **CBSQ**, we would create a constraint for every train currently parked on that track. If every train is parked on that track, we would create as many search-nodes as there are trains. If there are then still too many trains on the track, we create more constraints in every newly created search-node according to the amount of trains still parked on that track. This causes the amount of search-nodes to explode when many trains are parked on a single track.

To combat this issue we propose a heuristic that prefers solutions with fewer trains parked on a single track. This heuristic causes a loss of optimality, but should be better at spreading the trains between the tracks. Note that feasibility is more important than optimality when it comes to TUSP and this could therefore greatly improve the results achieved by CBS-algorithms.

The general idea behind this heuristic is to change the cost-function CBS uses to decide with search-node to explore next. This cost-function is changed to take into account the amount of trains that are parked on the busiest track at the busiest moment. The function takes the maximum amount of trains parked on a track at any time and multiplies the cost of a search-node by this value. This causes the algorithm to greatly prefer solutions where the trains are more spread-out in favor of only minimizing the sum-of-paths of the trains. To accomplish this we replace the SumOfPaths function in Algorithm 1, the basic CBS-algorithm, with the algorithm shown in Algorithm 8.

Due to the nature of CBS, where every search-node has the potential to be explored, we know that the optimal solution, if one exists, is still in the search-tree. This heuristic simply changes the order in which these search-nodes are explored. Therefore we know that, with this heuristic, the algorithm still provides a solution in due time, if one exists.

Algorithm 8 Heuristic CBS Cost-Function

```
function CostHeuristic(Solution)
    $Multiplier \leftarrow max_{\mathcal{T} \in \mathcal{T}r}(|tu \in \mathcal{T}_i^t|)$
    $Cost \leftarrow$ SumOfPaths(Solution) * $Multiplier$
    return $Cost$
```

Algorithms using this heuristic will be suffixed by a "**D**" (**CBSD**).

5 Experiments

In this chapter we discuss the experimental analysis of the previous mentioned algorithms and their results. The goal of these experiments is to find which algorithms perform best and find what future improvements should focus on. To answer these questions we need an answer to the following:

- Which algorithms are able to run the biggest instances in the least amount of time?
- What other criteria can be used to judge the CBS-algorithms?
- What do these criteria tell us about potential bottlenecks found when using the CBS-algorithms?

We start the remainder of this chapter by explaining our methodology and explain what kind of experiments were ran. All experiments were ran on the following machine:

- Intel I5-4690
- 8GB DDR3@1600MHz RAM

The algorithm itself was written in *Python 3.7*. This decision will be further explained in chapter 6. Furthermore, while it is possible to run the CBS algorithm multi-threaded we opt, for simplicity and fairness, to run the algorithm on only a single core. The benchmark-set used for these experiments is provided by a proprietary instance generator, provided to us by the NS. This instance generator is able to generate realistic instances for a varying amount of train units. For this experiment, we run bigger instances until an entire instance size has no successful runs These instances were generated for the location *Kleine Binckhorst*, which is generally used as benchmark for TUSP. A schematic representation of this shunting yard can be found in Figure 5.1.

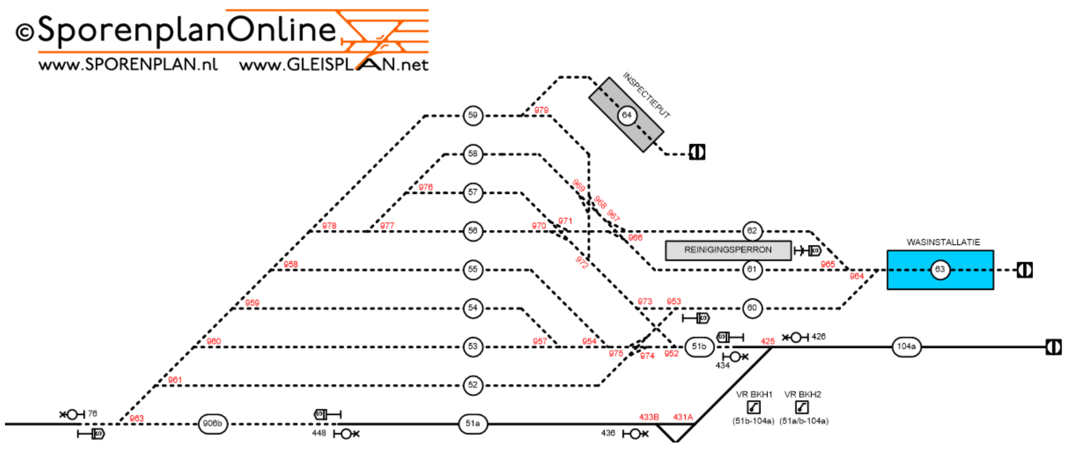


Figure 5.1: De Kleine Binckhorst, retrieved from www.sporenplan.nl

To answer which version of the algorithm performs best, we need to have a combination of the various heuristics and **CBSQ**- and **CSQL**-algorithms. Moreover, to provide a baseline performance and to explore what can cause a bottleneck for the CBS-algorithms, we want to add some CBS-Algorithms that do not use any of the more complex changes made to CBS. The selection of algorithms for these experiments can be found below:

- **CBS-Simple** - This algorithm is a version of CBS that has no way of dealing with multiple trains on the same track. Therefore, all trains are parked on their own individual tracks.
- **CBSQ-ParkScore** - This algorithm consists of the queue variant of the CBS algorithm, TUSP-CBSQ. Furthermore, this version increases the *gScore* of the low-level algorithm when parking, this means that, for the low-level algorithm, moving and parking have the same cost. This means that the train no longer has a preference between parking or moving and this will therefore cause a lot of unnecessary movement. This version is added to showcase the necessity of not increasing the *gScore* when parking.
- **CBSQ-NoMatching** - This algorithms consists of the queue variant of the CBS algorithm, TUSP-CBSQ, with no other heuristics.
- **CBSQ-Matching** - This algorithms consists of the queue variant of the CBS algorithm, TUSP-CBSQ, with the match conflict heuristic as explained in subsection 4.4.1.
- **CBSQ-BFS** - This algorithms consists of the queue variant of the CBS algorithm, TUSP-CBSQ. Furthermore, with this solution we opt for the high-level CBS search to do a Breadth-first-search rather than taking the node with the lowest value. With this approach we do lose optimally; however, since with TUSP feasibility is more important than optimally, it could prove to be an interesting benchmark to compare against CBS-algorithms optimizing for the minimum amount of movement.
- **CBSL-NoMatching** - This algorithms consists of the location variant of the CBS algorithm, TUSP-CBSL, with no other heuristics.
- **CBSL-Matching** - This algorithms consists of the location variant of the CBS algorithm, TUSP-CBSL, with the match conflict heuristic as explained in subsection 4.4.1.
- **CBSQD-Matching** - This algorithms consists of **CBSQ**, combined with the spreading heuristic explained in subsection 4.4.2 and the matching heuristic explained in subsection 4.4.1.
- **CBSQD-NoMatching** - This algorithms consists of **CBSQ**, combined with the spreading heuristic explained in subsection 4.4.2. No other heuristic was used.
- **HIP (Local Search)** - This algorithm is the local search algorithm currently in use at NS.

The initial goal is to find which algorithms perform the best out of these algorithms. Since the amount of computation time we have is limited, we decided to run every algorithm on a shorter time-out to find the most promising algorithms. To do so we first ran 10 instances per amount of trains with a **5 minute** time-out. The output of the algorithm provides a list of routes for every train unit, which can then be verified on the multi-agent simulator. This experiment provides us with an initial overview of which algorithms look promising and which do not.

After this initial experiment, we then decide on the most promising algorithms to run on 30 instances per amount of trains with a **30 minute** time-out. This can then provide us with a clearer picture on how MAPF is able to solve TUSP.

We then continue with an analysis of what part of the **CBS**-algorithm is causing a bottleneck. We do this by looking into the constraint generation to see how many and where the constraints are generated. Finally, we look at a grid-world example to see how other **CBS**-algorithms handle a situation commonly found in TUSP. We finish this chapter with a discussion on the results found during these experiments.

5.1. 5-minute Time-out

In this first section we run the experiments with a 5 minute time-out on 10 instances per amount of trains. With this experiment we intend to answer the question which algorithm performs fastest on bigger instances.

5.1.1. Success Rate

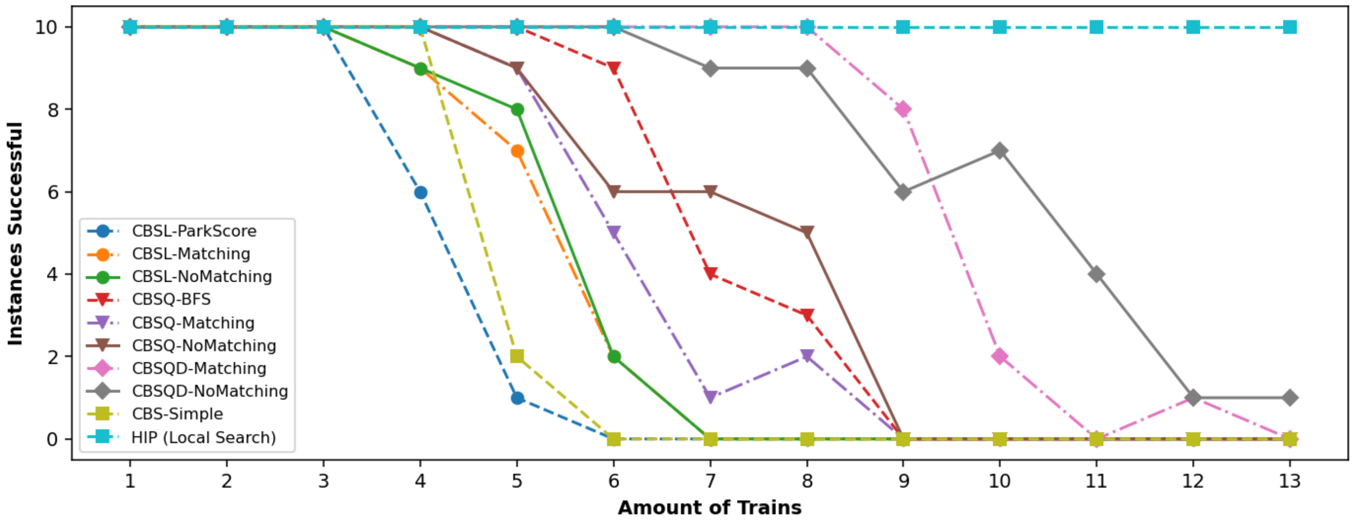


Figure 5.2: Benchmark results comparing the algorithms with a 5-minute time-out based on the amount of instances completed for each amount of trains. CBSQD-algorithms can solve significantly more instances than their CBSL- and CBSQ-counterparts.

In Figure 5.2, it can be seen how many of the instances were completed before the time-out for a certain amount of trains. It can be seen that **CBSQ-ParkScore** quickly fell off when the amount of trains increased. Most other algorithms performed comparably until the 6-train benchmark. Here **CBSQ**-algorithms started performing considerably better. However, **CBSQ-Matching** started to to struggle considerably more than the others when increasing the amount of trains further. Most algorithms were unable to route more than 8 trains on the shunting yard. Finally, **CBSQD**-algorithms were still able to reliably solve instances containing 8 trains. The **CBSQD-NoMatching** outperformed its Matching counterpart slightly when it comes to instances with a higher amount of trains. Finally, it can also be seen that **HIP** is able to solve all instances before the time-out.

With this we can conclude that **CBSQ** manages to solve bigger instances than **CBSL** with a 5-minute time-out. This could be explained by the fact that **CBSL** has to resolve more collisions than **CBSQ**, due to having to find the exact location on a track where a train is parked. With **CBSQ**-algorithms we omit this exact location and focus on merely the length of the track and the collisions occurring when a train leaves a track.

Furthermore, adding the spreading heuristic of **CBSQD** improves the amount of trains that the algorithm can route on a track; however, at the loss of optimality.

Notably, **CBSQ-ParkScore** and **CBS-Simple** perform exceptionally bad. The bad performance of **CBSQ-ParkScore** can be explained by the fact that this algorithm generates a lot of constraints. Trains do not have a preference for staying parked, which causes them to move around a lot. These movements then cause a lot of crossings which are costly to resolve. **CBS-Simple** most likely performs badly due to the slow propagation of trains towards the next track, and being unable to have multiple trains on the same track without conflicts. A similar problem was explained in subsection 4.3.2.

5.1.2. Time

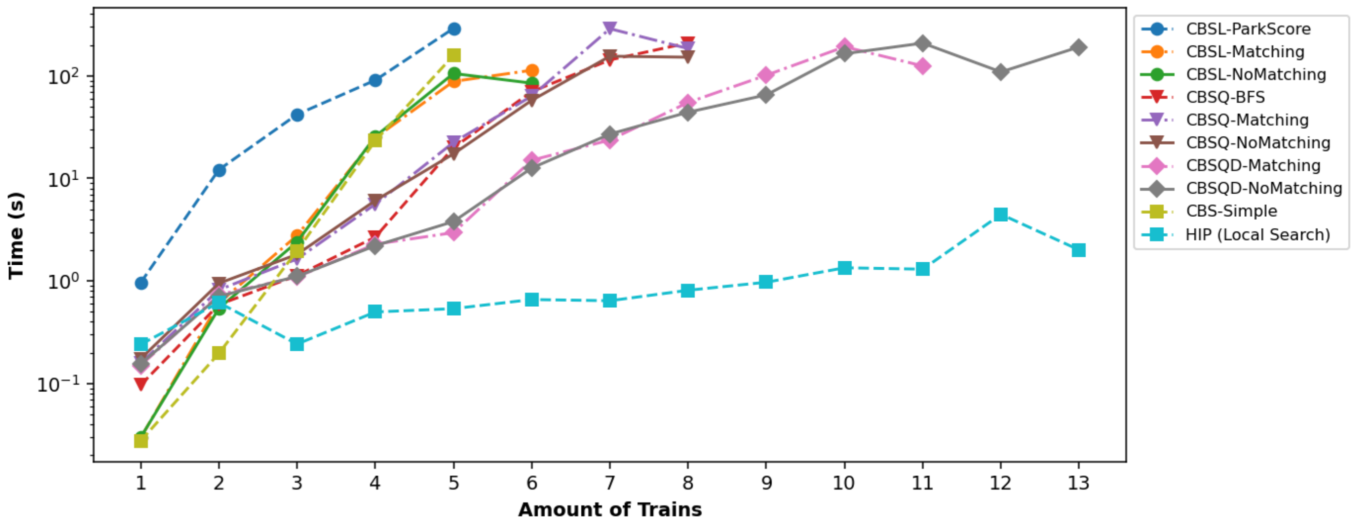


Figure 5.3: Benchmark results comparing the algorithms with a 5-minute time-out on the runtime for each amount of trains. CBSQD-algorithms perform significantly better than their CBSL- and CBSQ-counterparts.

To get a better answer to our question which algorithm performs the fastest on the bigger instances, we need to look at the runtime of the algorithm.

In Figure 5.3, it can be seen how much time is spent on average by the algorithm to find a result. Time-outs do not influence the average, only successful runs influence the outcome. One should keep in mind that this skews the results when many unsuccessful results are completed for a certain instance-size. It can be seen that all algorithms are exponential in the amount of trains. Furthermore, as was expected when looking at the results of the previous section, the **CBSQ** algorithms performed considerably better when the amount of trains was increased. The **CBSQD**-algorithms with the spreading heuristic perform exceptionally well and are able to finish their instances much faster than the other **CBS**-algorithms. However, it can also be seen that **HIP** is beating all other **CBS**-algorithms.

As before, the **CBSL-ParkScore** algorithm performed particularly bad, while the matching heuristic has almost no influence on the time it took to finish an instance. Interestingly enough the **CBS-Simple** algorithm does not perform significantly slower than the **CBSL**-algorithms, despite the worse results in the previous section. This is most likely caused due to the fact that when it finds a solution, it finds one rather fast. However, it reaches the time-out a lot more often than the other algorithms, and those runs are not accounted for in this graph.

Since the CBS-Algorithm is exponential in the amount of collisions that occur, it was to be expected that the algorithm runs in exponential time. When we increase the amount of trains, which all have the same start and end-location (at a different time), it is not far-fetched that many of these trains end up on similar tracks and cause collisions. In subsection 5.3.4, we explore this problem in more detail.

With these runtime results and the results found in the previous section, we can conclude that **CBSL**-algorithms, **CBSL-Parkscore** and **CBS-Simple** are significantly worse than **CBSQ**-algorithms, they are unable to solve the bigger instances and take significantly more time on the smaller instances. Out of all **CBS**-algorithms, **CBSQD** performed significantly better. However, **HIP** outperforms all of the **CBS**-algorithms.

5.2. 30-minute Time-out

In this section we continue the experiments with the most promising algorithms. These experiments consist of running each algorithm on 30 instances per amount of trains with a time-out of 30 minutes each. Our goal is to find the algorithm that runs the biggest instances possible before the time-out, therefore we selected the algorithms that showed the greatest promise on the bigger instances in the previous section. We now analyze these algorithms more in-depth on a longer time-out to see how they scale on bigger instances. Based on the results in the previous section, we decided to use the following algorithms for this experiment:

- **CBSQ-BFS**
- **CBSQ-Matching**
- **CBSQ-NoMatching**
- **CBSQD-Matching**
- **CBSQD-NoMatching**

There are no **CBSL**-algorithms being tested with a 30-minute time-out due to their bad performance and having no merit over the other algorithms in any of the criteria studied.

5.2.1. Success Rate

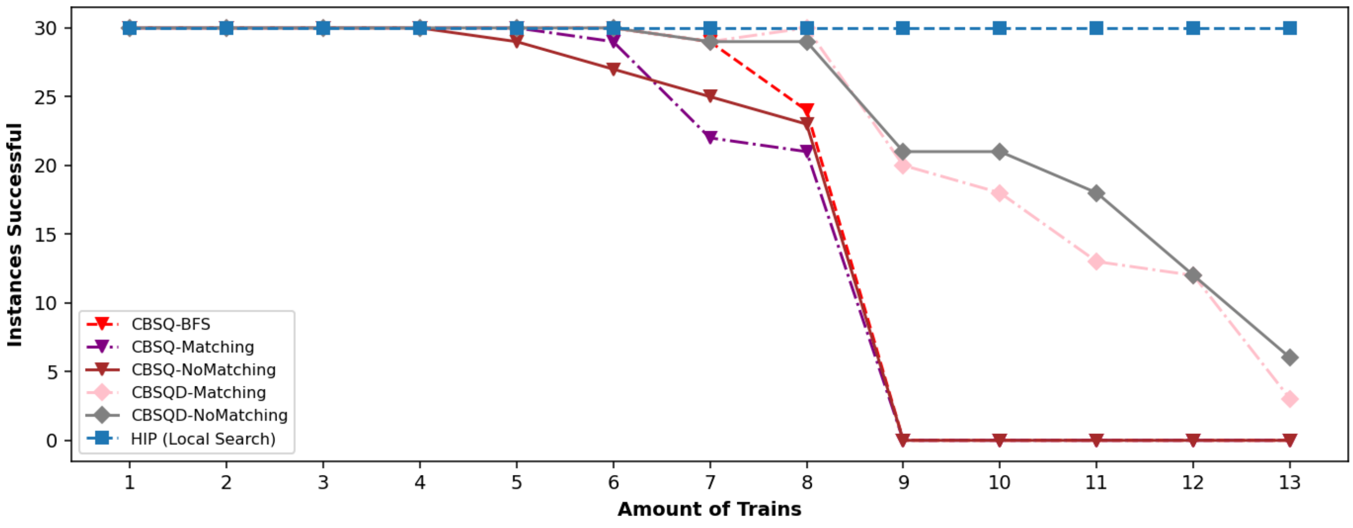


Figure 5.4: Benchmark results comparing the algorithms with a 30-minute time-out based on the amount of instances completed for each amount of trains. **CBSQD**-algorithms can solve significantly more instances than their **CBSQ**-counterparts. All **CBS**-algorithms are outperformed by **HIP**.

In Figure 5.4, we see the amount of trains the algorithms were able to route before the time-out of 30 minutes. We can see that the **CBSQ**-algorithms were able to route up to 8 trains quite consistently, but failed to route any instance consisting of 9 trains. Furthermore, the **CBSQ-BFS** algorithm seems to perform slightly better than its non-BFS counterparts. The **CBSQD**-algorithms were able to route problems up to a size of 11 trains somewhat consistently and even managed to solve a few instances of size 13, outperforming its non-heuristic counterparts. None of the CBS-algorithms managed to solve a problem instance with 14 trains. Furthermore, **CBSQD** performs slightly better when no matching heuristic is used.

5.2.2. Time

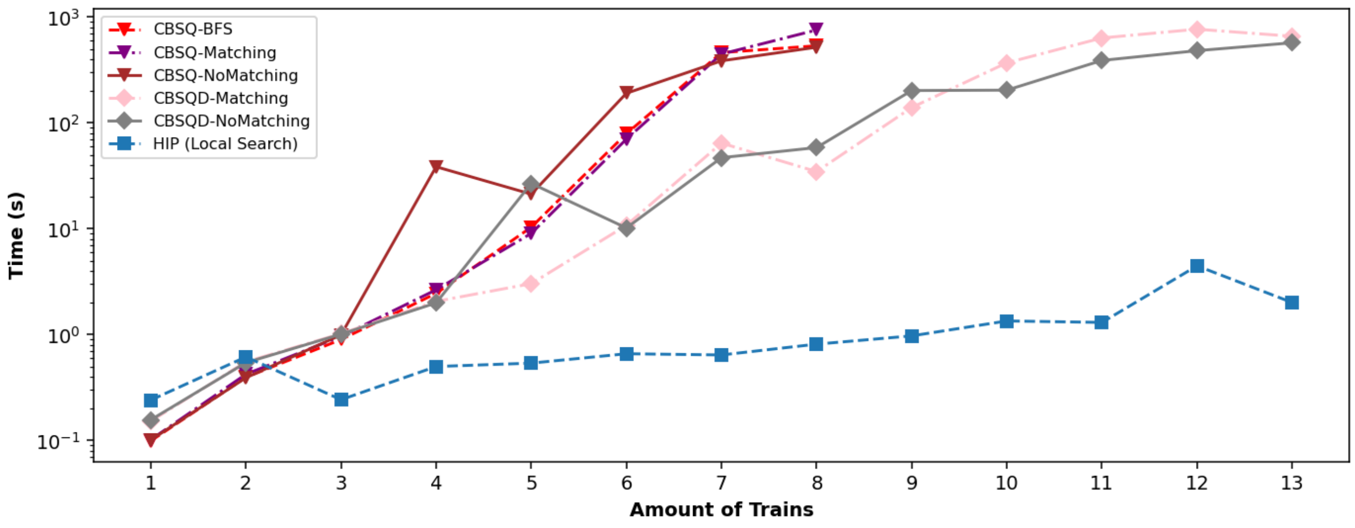


Figure 5.5: Benchmark results comparing the algorithms with a 30-minute time-out on the runtime for each amount of trains. **CBSQD**-algorithms perform significantly better than their **CBSQ**-counterparts. **HIP** is able to solve the instances significantly faster than the other algorithms.

In Figure 5.5, we can see the time it took to finish an instance with a certain amount of trains. Like in the previous section, if the time-out was reached, the time will not be taken account in this plot.

In accordance with the results in the previous section, **CBSQD**-algorithms outperform the **CBSQ**-algorithms. The **CBSQD**-algorithms are orders of magnitude faster than the **CBSQ**-algorithms on the bigger instances. However, there is no clear difference among the various algorithm types themselves.

In conclusion, with a 30-minute time-out the **CBSQD**-algorithms perform better than their non-heuristic counterpart, they are able to solve bigger instances in less time. Furthermore, it seems that not using the matching heuristic provides better performance for the **CBSQD**-algorithms. Finally, it can be seen that **HIP** performs much better than any **CBS**-algorithm. To further analyze the performance bottlenecks of the **CBS**-algorithms, a more in-depth analysis is necessary, which will be done in the next section.

5.3. Bottlenecks

In the previous sections we have seen the performance of the **CBS**-algorithms, however it was not yet clear what caused them to perform the way they did. In this section we seek to answer the question of what kind of bottlenecks we can find for our CBS-algorithms. We suspect the bottleneck to be in the constraint generation, since CBS is said to scale exponentially with the amount of constraints generated, and will therefore analyze how many constraints are generated, how much time is spent on constraint generation, and where these constraints generated. We finish this section with a showcase of a general bottleneck which can be found with CBS-algorithms, even in a grid-world.

5.3.1. Search-Tree Size

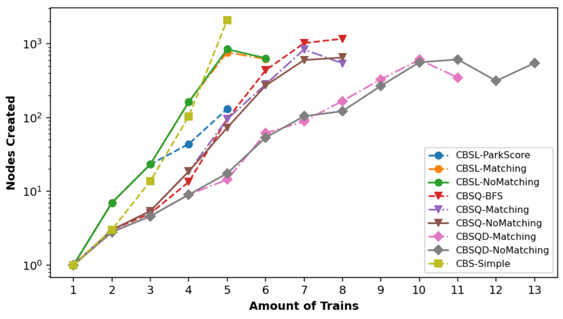


Figure 5.6: Amount of nodes created in the search-tree for all **CBS**-algorithms. It can be seen that algorithms creating many nodes are unable to solve the bigger instances.

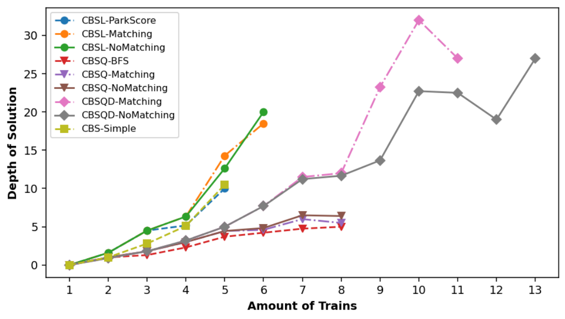


Figure 5.7: Depth of node where solution is found in the search-tree for all **CBS**-algorithms. A lower value means fewer constraints were necessary to construct the final solution.

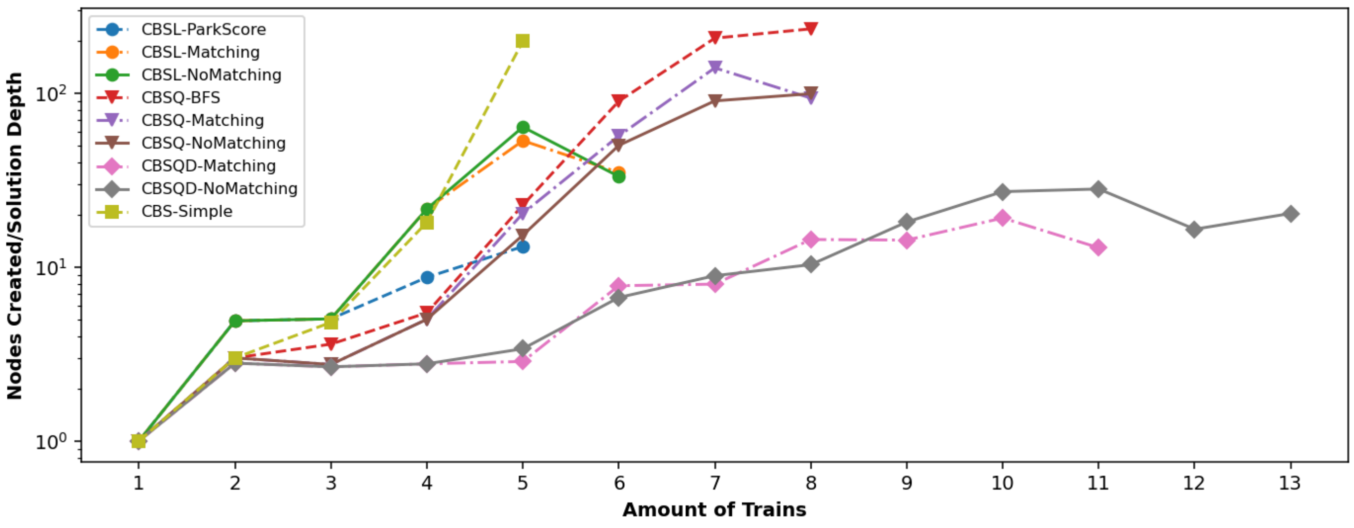


Figure 5.8: Amount of nodes created for every node in the final solution in search-tree. A value closer to 1 means that the algorithm creates fewer nodes that are not necessary to reach the solution. It can be seen that **CBSQD**-algorithms create fewer nodes for every node leading to the final solution.

To get a better picture on what is bottle-necking the CBS-algorithms, it is necessary to see how many constrains are generated in total and how many of these constraints are deemed useful. The total amount of constraints generated is equal to the amount of nodes in the search-tree when the result is found. This tells us how efficient the algorithm is in guiding itself towards the correct solution. When fewer nodes are created it means that the algorithm was better at guiding itself towards the best solution. The depth at which the final solution is found tells us how many constraints were necessary to get to this solution.

With this information we can also define how many of these search-nodes are deemed useful. Many of the search-nodes created are not used to reach the path to the final solution. We can infer how many *useless* nodes, nodes not leading to the final solution, are created using the following formula:

$$r = \frac{\text{nodes created}}{\text{solution depth}}$$

This formula tells us how many nodes are created for every node that guides us towards us to a solution. In general, a value closer to 1 is preferred; however, in some cases this can be misleading if the depth at which the solution is found is unusually deep. This is especially an issue with heuristics, where a potential solution might have been available at depth 2, but the heuristic guides you to a much deeper depth at which another feasible solution is found.

For this experiment we used a 5-minute time-out. In Figure 5.6, it can be seen that the amount of nodes of the **CBSL**-algorithms increased much faster than that of the the **CBSQ**-algorithms. Among the **CBSL** and **CBSQ** algorithms themselves, no significant difference was measured. However, the **CBSD**-algorithms greatly outperform the other algorithms when it comes to amount of search-nodes created. Noteworthy is the fact that **CBSQ-BFS** finds its solution on a similar depth as its non-BFS counterparts.

In Figure 5.7, we see that **CBSQ**-algorithms find their solution at a lower depth than their **CBSL**-counterpart. This is most likely explained due to the fact that **CBSQ** has a higher branching factor than **CBSL**. Where **CBSL** creates 2 child-nodes for every constraint found, **CBSQ** can create multiple constraints if the length-constraint on a track is invalidated. **CBSL** has to add many levels to its search-tree to solve the length-constraint, **CBSQ** is able to do so in a single level of the search-tree.

Interestingly enough, we see that **CBSQD**-algorithms find their solution on a greater depth than their non-heuristic counter-part, **CBSQ**-algorithms. This implies that **CBSQD**-algorithms do indeed get to a deeper part of the search tree faster.

This hypothesis is supported by Figure 5.8, here we see that the amount of search-nodes created per search-node necessary to reach a solution is substantially lower than with the other solutions. This implies that **CBSQD**-algorithms create relatively fewer nodes that are not required to reach the final solution. Therefore, it could prove useful to research more heuristics that guide an algorithm to a greater depth faster, as we see solutions with a better ratio be better at routing bigger instances in the same amount of time.

The results for the 30-minute time-out is quite similar to the 5-minute timeout and therefore omitted. However, the results can still be found in the Appendix, section 8.2.

5.3.2. Pathfinding vs Constraint Generation

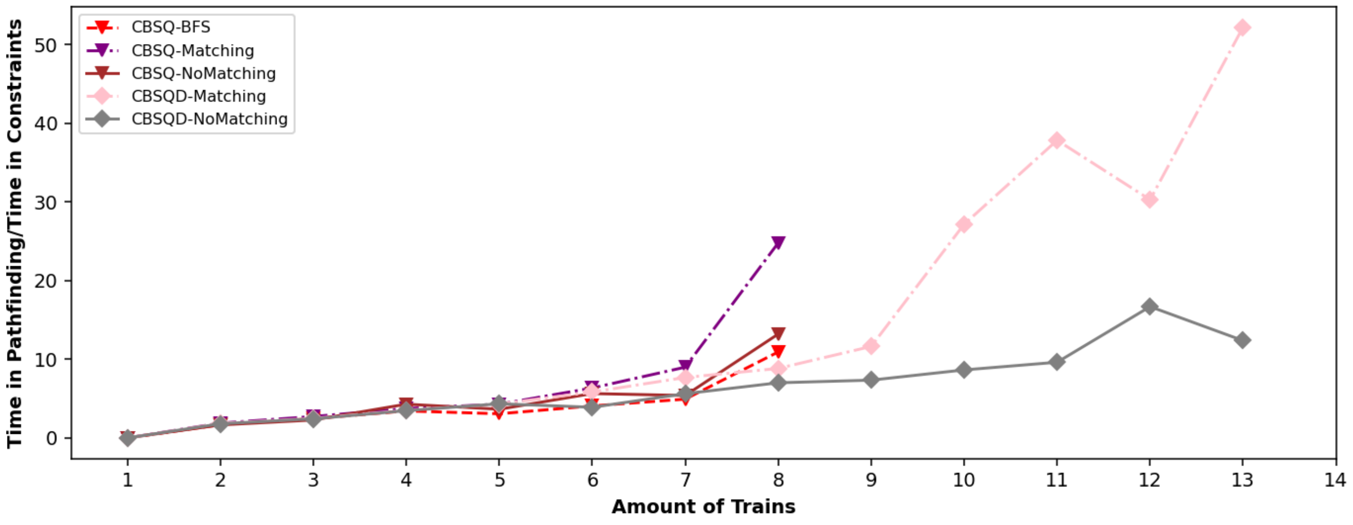


Figure 5.9: Ratio of time spent pathfinding vs constraint generation.

In Figure 5.9, we show the ratio between the time spent in pathfinding (A*-TUSP) versus the time spent generating constraints. This might give us a better insight where most time is spent in the algorithm. A lower value on the y-axis means that the time spent is more evenly distributed. A higher value means that more time is spent in the pathfinding part of the algorithm.

As we can see most algorithms spent a similar amount of time pathfinding versus generating constraints. This trend continues for the first instance sizes, but increases quite swiftly when instances with more trains are ran. This tells us that the pathfinding portion becomes more taxing when more trains are added. This could be caused by a difference in complexity, where the complexity in the pathfinding part scales faster than the complexity in the constraint part.

Even with the time required to run the pathfinding portion of the algorithm, we do not necessarily know that this is indeed the bottleneck of this algorithm. Instead, it could be the case that the constraint generation creates many new search-nodes, which all require the pathfinding portion of the algorithm to run. Therefore, we cannot draw any conclusions whether the pathfinding part is too inefficient. However, we can conclude that the constraint generation algorithm itself scales better than pathfinding, since the pathfinding algorithm takes significantly more time than the constraint generation on bigger instances.

5.3.3. Constraint Generation

To get a better idea of the performance of the CBS-algorithms, we discuss for which tracks the constraints are generated. To do this we have selected an instance consisting of 8 trains, which was passed by all **CBSQ**- and **CBSQD**-algorithms. This instance was passed by **CBSQD**-algorithms in about 20 seconds and by the **CBSQ**-algorithms in about 200 seconds.

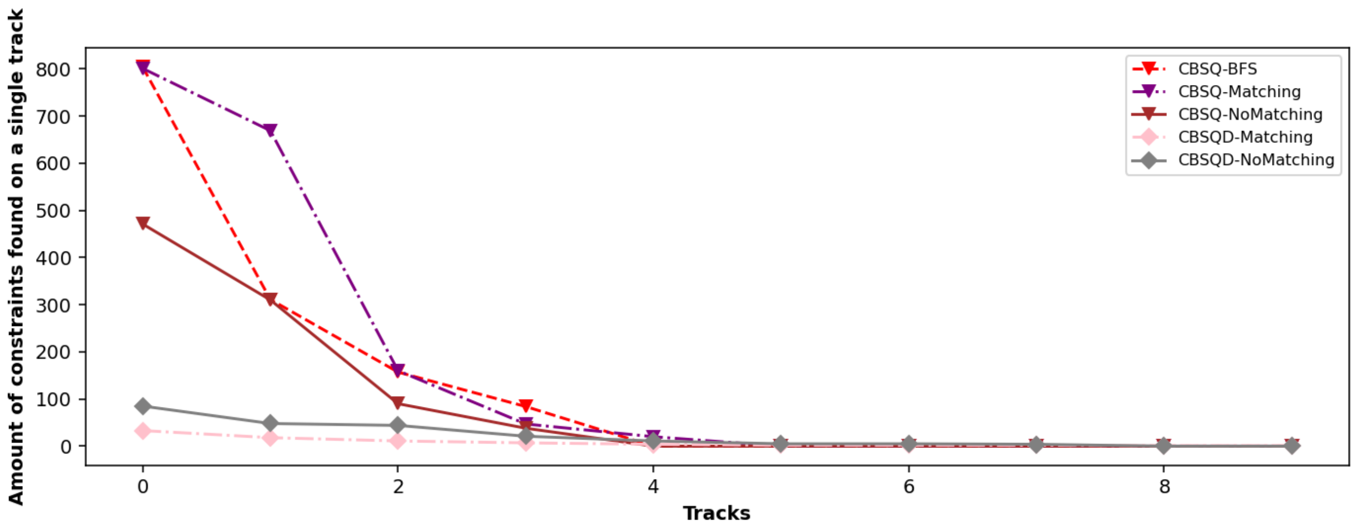


Figure 5.10: Benchmark showing the amount of constraints generated on a single track for each algorithm.

Figure 5.10 shows for every algorithm how many of the constraints are generated on a single track. There are close to 100 tracks on this shunting yard, this plot only showcases the 10 tracks with the most constraints. The tracks are sorted for every algorithm to have the track with the most constraints as *track 0*. This means that *track 0* of **CBSQ-Matching** might not be the same as *track 0* of **CBSQD-Matching**.

It can be seen that the **CBSQ**-algorithms create many more constraints than the **CBSQD**-algorithms. This was to be expected, since the results from subsection 5.3.1 show us that **CBSQ**-algorithms create many more search-nodes to find a solution.

It should also be noted that there are almost 100 tracks in total, this means that all conflicts occur on fewer than 10% of the tracks. This implies that the trains are not well spread out when they first enter the shunting yard, which causes many collisions on a select few tracks.

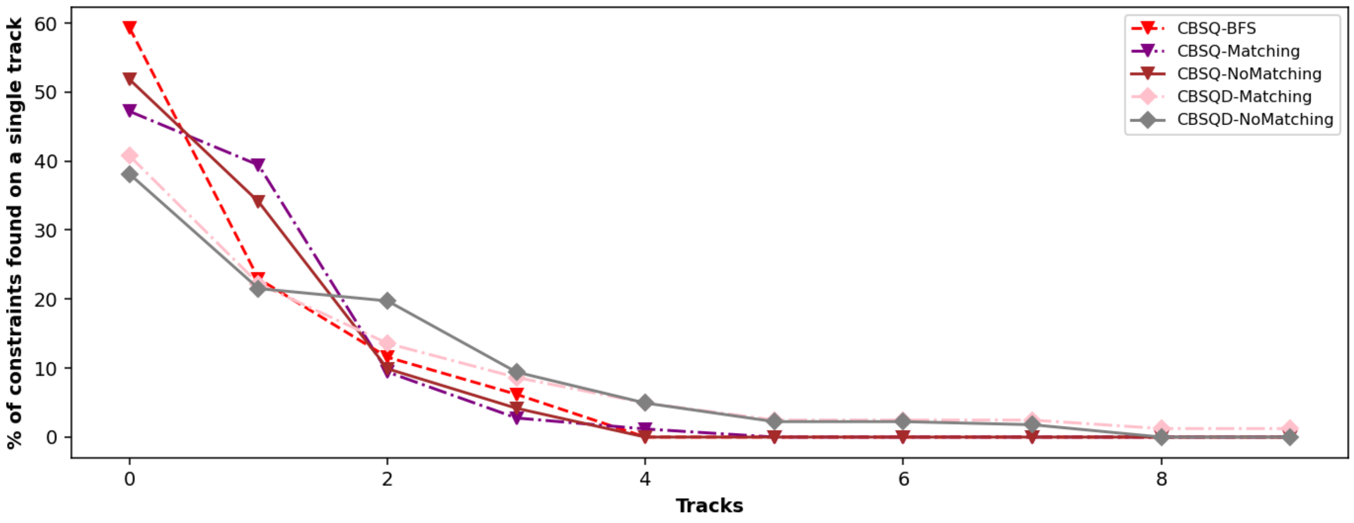


Figure 5.11: Benchmark showing the relative amount of constraints generated on a single track for each algorithm.

To get a better idea of the spread of the constraints we can look at the relative amount of constraints on a single track, this can be found in Figure 5.11. Here we see that **CBSQD**-algorithms have a lower percentage of conflicts on a single track than the other algorithms. Between the algorithm groups themselves, no big difference is seen.

These result imply that the conflicts are more spread out through the shunting yard for **CBSQD**-algorithms. This can also be seen due to the fact that, while **CBSQ**-algorithms only have conflicts on 4 or 5 tracks, **CBSQD**-algorithms have conflicts on up-to 10 tracks. This behavior is expected from **CBSQD**-algorithms, since their goal is to increase the spread of the trains on the shunting yard and, as a result, trains tend to not stick to a select few track as much, as can be seen with **CBSQ**-algorithms.

5.3.4. Grid CBS

In an attempt to showcase the weakness of CBS-algorithms, and with that a bottleneck of the algorithms in this thesis, we made a grid-representation of the parking of trains. With this, we show that CBS-algorithms are weak when many agents cross the same locations in a hallway and how this relates to the problems found with MAPF-TUSP. This grid can be found in Figure 5.12. This grid resembles a case where the trains have to leave a track in the reverse order in which they are parked, similarly to the issue explained in subsection 4.3.2.

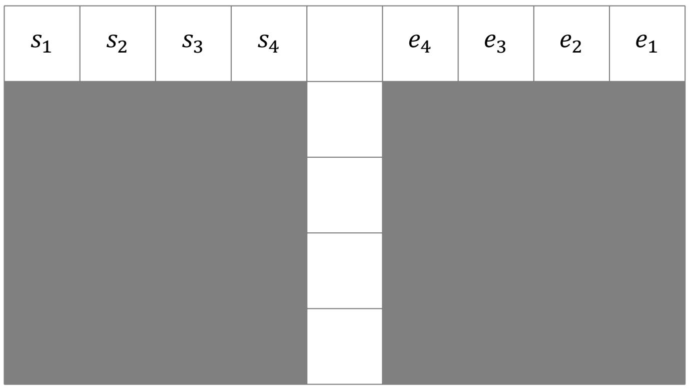


Figure 5.12: A grid version of the CBS-parking problem

The goal for the agent a_i starting on s_i is to find a path to e_i, without colliding with another agent. This grid represents a situation where the train want to leave the track LIFO order, where s_1 was the first to enter the track and is therefore in the outermost position.

To run this experiment we ran the C++ implementation used by Hoenig et al. [9]. This implementation can be found on GitHub[1]. This implementation was able to route 3 agents on the grid shown in Figure 5.12 within a few seconds. However, the algorithm failed to route the same problem with 4 agents with a time-out of 15 minutes. The map-file used can be found GitHub Gist[2].

This experiment provides evidence towards our previously mentioned hypothesis; **CBS**-algorithms can be very inefficient when routing in tight corridors, where many agents cross the same nodes. This could explain why **CBSQD**-algorithms perform so much better than their **CBSQ**-counterparts, since they attempt to spread the trains as much as possible in an attempt to avoid many collisions on the same tracks.

[1]https://github.com/whoenig/libMultiRobotPlanning
[2]https://gist.github.com/Dominique2509/9fa4ddb0f012f109597cbb18a75d2188

5.4. Discussion of Results

We have seen that **CBSQD**-algorithms greatly outperform their **CBSQ**-counterpart. Given the bottleneck analysis, the most likely hypothesis is that **CBSQD**-algorithms are able to go deeper down the search-tree to find a potential solution faster. Furthermore, it was seen that this algorithm prefers solutions with fewer trains on the same track, which causes fewer collisions in the search-nodes it analyzes.

CBSL-algorithms were shown to perform rather poorly. After our analysis we saw that these algorithms created many more nodes not leading to the final solution than the other algorithms, despite being able to go further down the search-tree. This is most likely caused by having to find the exact parking location of a train on a track. While this information is nice to have, it is apparently very costly to calculate.

We saw **CBSQ-BFS**, the bread-first search variation of **CBSQ**, perform just as well as its non-BFS counterparts. This is quite interesting, since the the non-BFS variations should supposedly be guided towards the solution with the lowest cost. This could imply that the non-BFS variations get stuck in local-optimum, while a potential solution could already be find at a much lower depth, which are easily found by a BFS-algorithm. To confirm this, more research would be necessary on the nodes selected in the high-level search by **CBSQ**-algorithms.

Furthermore, The *Matching* heuristic, first explained in subsection 4.4.1, has no clear result on whether it should be used. For the **CBSQ**-algorithms it seems that not using the heuristic is preferred; however, for the **CBSQD**-algorithms the opposite seems to hold true. More research would be necessary to know whether changing the order in which special conflicts are handled, such as match conflicts, matter for **CBS**-algorithms.

Finally, it was shown that CBS has difficulty routing agents in a similar setting as TUSP on a grid. This could explain why **CBS**-algorithms without a heuristic had a tough time routing trains on the shunting yard. For future research, one should consider focusing on the selection of search-nodes by the high-level algorithm as to avoid such situations.

6 Discussion

In this chapter, we discuss the various assumptions or choices made in this thesis. We give a justification for these decisions and propose alternatives which could be made.

6.1. Discrete Time

In this thesis, we decided the discretize the time-step of an instance at 1 minute. This has the implications that we lose some precision, since certain moves might only take 30 seconds or 90 seconds. We opted to use 1 minute, since this was deemed the time that was precise enough to model the movements of the trains, while still attempting to not create too many search-nodes in the low level search. If one wants more precision, one could opt to decrease the length of the time-steps at the cost of runtime.

If one wants to run instances with a longer duration, it might be a valid option to increase the time-steps of the model to 2 or more minutes. With these longer instances, the search-space of the low-level **A*-TUSP** algorithm becomes larger, as there is a longer time between the arrival and departure of trains. However, increasing the time-steps to 2 minutes would come at the cost of precision and should be well considered before opting for this change.

6.2. Python

In this thesis we used *Python* to implement our algorithm, which is obviously not the optimal choice when it comes to algorithms. It was decided to use python for a multitude of reasons. The algorithm in this thesis was written in cooperation with the NS and we therefore had a preference to use a programming language which is commonly used by them. This left us with the choice of *Python* and *C#*. We further restrict ourselves by using the Multi-Agent simulator developed by NS, which was written in *Python*. This implied that any language used should be able to easily work in tandem with this simulator, which is why we decided to use *Python* for this algorithm.

While this choice does hurt the run-time analysis part of this thesis, we think that the impact was minimal. It has been suggested that, when it comes to MAPF, the difference between Python and C++ is a speedup of 2.25 times [20]. Therefore, we can still use that analysis to analyse the growth-rate of time in comparison with the amount of trains. It should still be noted, however, that perhaps with an implementation in C++, it could be possible to push the algorithm to slightly bigger instance sizes.

6.3. Experiments

For the experiment section, we decided to have an initial run with a 5-minute time-out, to only have a 30-minute time-out for our best algorithms. This was done due to the very long runtime required for some of the bigger instances and a lack of time. While it would have been better to run the algorithm for 30 minutes on every 30 instances for every algorithm, it would simply have cost too much time to conduct such an experiment. Furthermore, due to the bad performance and exponential growth of

CBSL-algorithms we do not expect to have gotten a much better result with a longer time-out.

Another assumption we made during this thesis is to only run the experiments on the Shunting Yard *Kleine Binckhorst*, despite there being many more shunting yards owned by NS. This was done for both simplicity sake, time constraints and there being no visualizer for the other shunting yards at the time this decision was made. However, *Kleine Binckhorst* is the most used shunting yard for benchmarks and therefore our results are easily comparable to other solving methods by the NS.

6.4. Matching

In this thesis we opted to have all trains consist of only a single unit. In the real-life version of TUSP the coupling and decoupling of train units have to be modeled. Because, when a train enters the shunting-yard consisting of two units, it is possible for the units of this train to leave the shunting-yard as two individual trains or vice versa. This operation was deemed out-of-scope for this thesis, but should be considered if one continues research on using MAPF-algorithms to solve the TUSP.

7 Conclusion

In this thesis, we set out to show how parts of the Train-Unit Shunting Problem (TUSP) can be solved using Multi-Agent Pathfinding (MAPF). We proposed using CBS to solve the routing, parking and matching of TUSP. Furthermore, we proposed alterations to the way trains are parked in the CBS-algorithm to optimize it for a train-domain which we bench-marked on a realistic data-set.

To solve TUSP we have shown a model that can be used to translate a TUSP problem into a MAPF problem. This model can then be solved by state-of-the-art MAPF algorithms, such as Conflict-Based Search (CBS). CBS finds paths for each individual agent, after which it then finds collisions between the various paths. CBS then generates constraints in creates a node in a search-tree for each of these constraints. It then starts finding new solutions giving the constraints until it finds a solution, or terminates because of the lack thereof.

To accomplish this we have shown how to model the TUSP as a MAPF problem. We have demonstrated how to model directional tracks, where reversing is only allowed on certain tracks, which is done by adding the notion of direction to every node. Furthermore, we have shown how to allow parking only on certain nodes, which is done by only adding edges to the next time-step when parking is allowed. Additionally, we mention how a train can be matched to an outgoing train using this algorithm, without knowing the end-location of the train beforehand. Finally, we proposed two different ways to keep track of multiple units on the same node, namely location-based (TUSP-CBSL) and queue-based (TUSP-CBSQ).

To accompany these variations, we came up with two different heuristics. The first one handles the order in which constraints are detected; we opted to find conflicts in the matching, before we attempted to find conflicts in the route itself. The second heuristic has to do with the order in which search-nodes are selected by the CBS algorithm. With this heuristic we opt to find solutions where the trains are more evenly divided over the tracks, rather than just focusing on the shortest path.

To compare these algorithms we ran a benchmark set which contains realistic instances. For each algorithm we compared how many instances they were able to complete of the benchmark-set and how long it took them. Additionally, we have shown that many constraints are generated on the same few tracks, which is a problem that is also present in regular MAPF instances solved by CBS. Furthermore, we analyzed the amount of nodes created in the search-tree and how efficiently this search-tree is traversed.

With these algorithms we found that, out of two parking variations, it was seen that TUSP-CBSL was outperformed TUSP-CBSQ. There was no clear difference when it comes to the use of the matching heuristic. However, when it came to the heuristic TUSP-CBSQD, where trains are divided evenly, a significant improvement was seen in both runtime and instance size.

With this thesis, we have shown that it is possible to use MAPF in other domains than it was originally intended to solve. We have shown that is is possible to use MAPF in a directed-graph. Furthermore, we have shown that it is possible to model nodes as a queue to speed up the algorithm. Moreover, we have solved a matching problem of incoming and outgoing agents using MAPF.

Finally, it should be noted that this algorithm is currently not faster than the local-search method which is currently in use at the NS. However, with the rapid advances in the field of MAPF, the provided model can prove very useful when MAPF-algorithms get faster.

In conclusion, we have shown how to model a TUSP into a MAPF problem. Additionally, we have shown how **CBS**-algorithms can be altered to solve this new MAPF instance, which contains many problems specific to TUSP. Finally, we have analyzed the various **CBS**-algorithms in regards to TUSP to guide future improvements.

7.1. Future work

In this section, we discuss potential future work based on the contents of this thesis and the achieved results. For each proposal, we attempt to give a few pointers as to how these problems could be solved.

7.1.1. Modeling Service Tasks

The modeling of service tasks was something that was originally planned for this thesis, but was eventually scrapped. This could be done by using the method described by Grenouilleau et al. [21]. This method could be combined with the matching method explained in section 3.2. An agent would only be able to match with a departing train if it has acquired the label of ”being-cleaned”, which can only be acquired if the current path to the exit goes through a node that allows a train to be repaired or cleaned.

7.1.2. Mix and Match

We started this project with the knowledge that the current local-search method has issues with routing. In this thesis we proposed an entirely new algorithm which takes the routing as a first-class citizen, rather than an afterthought. It could prove useful to incorporate a similar pathfinding algorithm into the already existing local-search. As we have shown in this thesis, it is possible to model the TUSP, and especially a shunting-yard, as a MAPF problem. An open question would be whether one can model the routing problem of the local-search as a similar model, which could then solve using the ideas proposed in this thesis.

7.2. Simplifying the Model

Often times, there are multiple tracks alongside each-other, with no switch in between. This means that these tracks could potentially be used at the same time to store a single train. In the current version of the algorithm, these tracks are seen as separate tracks; However, one could consider combining such tracks for the algorithm to create a single longer track. This could then be used to stored more trains, such that the track-length constraint is not violated as easily.

7.3. Heuristics

As has been shown with this thesis, heuristics can greatly improve the run-time of MAPF algorithms. While some heuristics keep optimality, some others do not. However, for TUSP optimality is not a requirement; rather, heuristics should focus on feasibility. With smarter decisions on which search-nodes should be discovered next, it could be possible to push the algorithm even further.

One potential heuristic could be to keep trains of the same type on the similar tracks. This would reduce the amount of swapping around the trains would have to do, especially when the train-type one needs just so happens to be parked in the far back of a track. When keeping trains of the same type on the same track, one could simply pick the train closest to the exit and leave other trains stay on the track.

Furthermore, one could look into general heuristics used by CBS. These were omitted from this thesis for simplicity sake, since these heuristics would take away focus from the model. However, future research could focus on applying these general heuristics on the model that has been constructed in this paper.

7.4. Branch-and-Cut-and-Price

For future research, one could consider using Branch-and-Cut-and-Price (BCP), rather than CBS. To do so, I would advice to convert TUSP-CBSQ into a Linear Programming Formulation. One could do so by constraining the length on the track, such that the track-length is never exceeded. Furthermore, one should keep an order of every train on the track for every time-step. This can then be used to check whether no illegal move is made for every time step. Finally, one could use the guiding heuristic in BCP to ensure that the trains are well spread across the shunting yard.

8 Appendix

8.1. Potential moves

This section of the appendix shows two additional moves a train can make when using the A*-TUSP algorithm.

Algorithm 9 Parking a train on current track

```
function park(current)
    if current.parking_allowed then
        new_state.time ← current.time + 1
        new_state.location ← current.location
        new_state.direction ← current.direction
        gScore[new_state] ← current.time
        came_from[new_state] ← current
        return {new_state}
    return ∅
```

Algorithm 10 Setbacking a train on current track

```
function setback(current)
    if current.setback_allowed then
        new_state.time ← current.time + 1
        new_state.location ← current.location
        new_state.direction ← reverseDirection(current.location, current.direction)
        gScore[new_state] ← current.time + 1
        came_from[new_state] ← current
        return {new_state}
    return ∅
```

8.2. Additional Results

These are some additional results for the experiments with a 30-minute time-out. These results were too similar to the 5-minute time-out

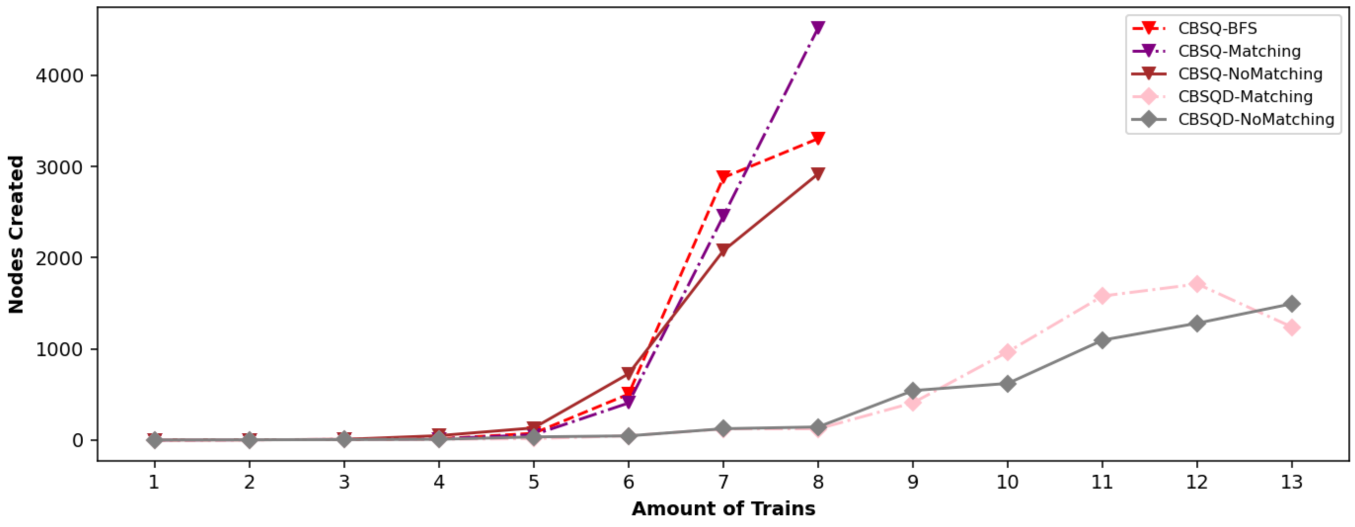


Figure 8.1: Total amount of nodes created when a solution is found in the search-tree. Fewer nodes means a more efficient algorithm.

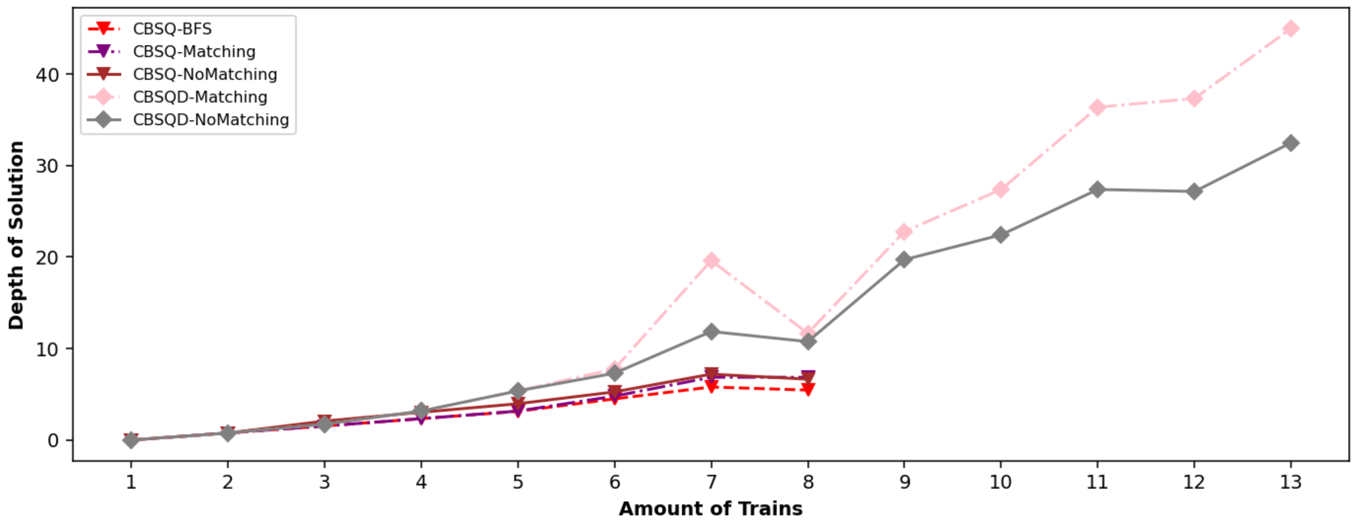


Figure 8.2: Depth of node where solution is found search-tree. A lower depth means fewer constraints were necessary to find this solution.

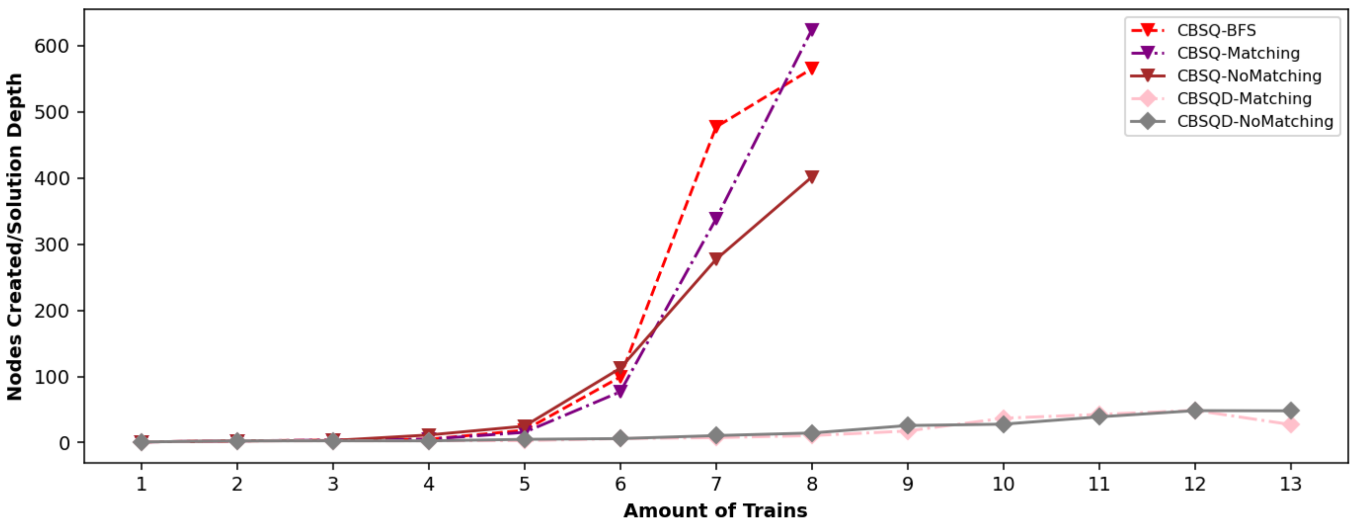


Figure 8.3: Ratio of nodes necessary to reach the final solution and the total amount of nodes created. A ratio closer to 1 means fewer nodes were created that are not part of the path to the final solution.

Bibliography

[1] R. W. van den Broek, "Train shunting and service scheduling: an integrated local search approach," Master's thesis, 2016. [Online]. Available: http://dspace.library.uu.nl/handle/1874/338269

[2] "Shunting of passenger train units in a railway station," *Transportation Science*, vol. 39, no. 2, pp. 261–272, 2005. [Online]. Available: http://www.jstor.org/stable/25769246

[3] R. Lentink, P.-J. Fioole, L. Kroon, and C. Woudt, *Applying Operations Research Techniques to Planning of Train Shunting*, 02 2003, pp. 415 – 436.

[4] R. Lentink, "Algorithmic decision support for shunt planning," Ph.D. dissertation, E, Feb. 2006. [Online]. Available: http://hdl.handle.net/1765/7328

[5] H. Ma, G. Wagner, A. Felner, J. Li, T. Kumar, and S. Koenig, "Multi-agent path finding with deadlines: Preliminary results," 05 2018.

[6] J. Rovnik, V. Andrew, and Y. Konstantin, "Flatland challenge team report," 01 2020. [Online]. Available: https://github.com/vetand/FlatlandChallenge2019/blob/master/Approach_description.pdf

[7] G. Sharon, R. Stern, A. Felner, and N. Sturtevant, "Conflict-based search for optimal multi-agent path finding," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, ser. AAAI'12. AAAI Press, 2015, pp. 563–569. [Online]. Available: http://dl.acm.org/citation.cfm?id=2900728.2900809

[8] W. Hoenig, S. Kiesel, A. Tinka, J. Durham, and N. Ayanian, "Persistent and robust execution of mapf schedules in warehouses," *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–1, 01 2019.

[9] W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian, "Conflict-based search with optimal task assignment," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '18. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2018, p. 757–765.

[10] D. Sigurdson, V. Bulitko, W. Yeoh, C. Hernandez, and S. Koenig, "Multi-agent pathfinding with real-time heuristic search," in *Proceedings of the 2018 IEEE Conference on Computational Intelligence and Games, CIG 2018*, vol. 2018-August. United States: IEEE Computer Society, 10 2018.

[11] J. Li, M. Gong, Z. Liang, W. Liu, Z. Tong, L. Yi, R. Morris, C. Pasearanu, and S. Koenig, "Departure scheduling and taxiway path planning under uncertainty," 06 2019.

[12] D. Silver, "Cooperative pathfinding." 01 2005, pp. 117–122.

[13] G. E. Mathew, "Direction based heuristic for pathfinding in video games," *Procedia Computer Science*, vol. 47, pp. 262 – 271, 2015, graph Algorithms, High Performance Implementations and Its Applications (ICGHIA 2014). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915004743

[14] G. Gange, D. Harabor, and P. J. Stuckey, "Lazy cbs: Implicit conflict-based search using lazy clause generation," in *ICAPS*, 2019.

[15] A. Andreychuk, K. Yakovlev, D. Atzmon, and R. Stern, "Multi-agent pathfinding (mapf) with continuous time," 01 2019.

[16] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T. K. S. Kumar, and S. Koenig, "Adding heuristics to conflict-based search for multi-agent path finding," in *ICAPS*, 2018.

[17] B. de Wilde, A. W. ter Mors, and C. Witteveen, "Push and rotate: Cooperative multi-agent path planning," in *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems*, ser. AAMAS '13. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2013, p. 87–94.

[18] E. Lam, P. Le Bodic, D. Harabor, and P. Stuckey, "Branch-and-cut-and-price for multi-agent pathfinding," 08 2019, pp. 1289–1296.

[19] J. Li, P. Surynek, A. Felner, H. Ma, T. Kumar, and S. Koenig, "Multi-agent path finding for large agents," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 7627–7634, 07 2019.

[20] T. Bestebreur, "Analysis of the influence of graph characteristics on mapfw algorithm performance," 2020. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3Ad602656e-4e27-4e36-8039-90497db5b905

[21] F. Grenouilleau, W.-J. van Hoeve, and J. N. Hooker, "A multi-label a* algorithm for multi-agent pathfinding," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 07 2019.