Bachelor thesis

Comparison of Different Multi-Agent Path Finding Approaches Applied to Bottleneck Scenarios

Degree in Computer Science and Engineering, 180 ECTS

Isak Cetin

Supervisor: Franziska Klügl
Professor in Computer Science

Examiner: Lars Karlsson
Professor in Computer Science

Studies from the School of Science and Technology at Örebro University
Örebro 2025

Abstract

There is a growing demand for autonomous solutions within many industries, such as construction-, vehicle-, and mining industries. This thesis evaluates and compares a handful of multi-agent path finding algorithms for coordinating the traffic in a highly constrained online mining ramp scenario. The algorithms included are multi-agent A*, increased cost tree search, conflict-based search, and conflict-based search with priorities, of which the latter does not guarantee optimality.

This thesis demonstrates that increased cost tree search and conflict-based search are not suitable for the mining ramp scenario, as they are very limited in being able to find a solution within a reasonable time. A* manages to find solutions in more complex scenarios, but eventually begins to struggle as well. Conflict-based search with priorities manages to find solutions in even more complex scenarios, and much faster. However, the drawback is that it does not always find optimal solutions, whereas A* generally does. A*, therefore, is most suitable if the solution cost is critical and the scenario is not too complex. Conflict-based search with priorities is most suitable if the scenario is complex and finding a solution within a reasonable time is critical.

Keywords

Multi-agent path finding, Replanning, Mining ramp, Traffic coordination

Contents

Chapter 1
Introduction

1.1 Thesis introduction

This thesis was conducted on behalf of the Centre for Applied Autonomous Sensor Systems (AASS) team at Örebro university, which is a research team that focuses on autonomous systems. One of their multiple ongoing research projects is TeamRob (Teams of Robots Working for and with Humans), which this thesis was associated with. The core question TeamRob aspires to answer is how intelligent robots can work for and with us humans. TeamRob has several collaborators within the construction-, vehicle- and mining industries [16].

In recent time there has been an increasing interest in autonomous solutions, not least within the aforementioned industries. In the mining industry, mining ramps play an integral part as they connect the surface level to the underground level. They are, however, very constrained in the sense that space on the ramp is scarce which limits access to it, some vehicles have higher priority than others, different kinds of vehicles have different capabilities and restrictions, etc.

With this background, this thesis focused on traffic coordination in a scenario with a mining ramp through which vehicles travel between the surface and the mine. More specifically, this thesis sought to investigate different multi-agent path finding (MAPF) approaches to automating traffic coordination along the ramp.

1.2 Thesis aims and objectives

The overarching aim of this thesis was to apply several established MAPF approaches for coordinating the traffic on a mining ramp, and compare their performances. First, the MAPF research domain had to be surveyed to be able to select a handful of algorithms for coordinating the traffic. With the algorithms in place, the performances of the algorithms were evaluated and compared.

Concretely, this thesis sought to...

- ... implement and apply a handful of MAPF algorithms on the mining ramp scenario
- ... evaluate and compare the algorithm performances in terms of:
  - Successfully generating a traffic plan
  - Time required to generate the traffic plan
  - Cost of the traffic plan, measured as...
    - ... the sum of vehicle costs
    - ... the sum of prioritised vehicle costs (when relevant)
    - ... the sum of time spent waiting

  - Practical suitability for use in the mining ramp scenario
- ... create a tool for visualising the generated traffic plans

1.3 Outline of the thesis

Chapter 2 introduces the reader to the MAPF problem definition and several MAPF algorithms.

Chapter 3 defines the ramp scenario and the constraints it presents. Additionally, a discussion of the rationale behind what MAPF algorithms were selected is held.

Chapter 4 describes the methodology process, from surveying the MAPF research area to applying and comparing the algorithms, as well as tools used.

Chapter 5 includes choices made during the implementation of the mining ramp, the MAPF algorithms, and the visualisation tool.

Chapter 6 outlines the experiments conducted by applying the different MAPF algorithms on mining ramp and agent property configurations. The results of the experiments are also presented here.

Chapter 7 offers a discussion about the results, as well as limitations to this project and future work. A discussion is also held on the corporate and social impact of the results.

Chapter 8 states the conclusions of this thesis.

Chapter 9 includes a final reflection about the project and the course.

Chapter 2
Multi-agent path finding problem formulation and algorithms

This section introduces and defines the problem formulation for this thesis. It begins with a short introduction, followed by the MAPF problem definitions. The section dives into search-based MAPF algorithms. It briefly concludes by touching on other MAPF-solving approaches.

2.1 Topic introduction

As research in artificial intelligence progresses, so does the research within fields of automation – something that enables increased efficiency and reduced expenses. Many of the scenarios in which automation is desired involve multiple agents, which introduces a problem known as multi-agent path finding (MAPF). The concept of the MAPF problem is quite straightforward. Multiple agents, in a shared space, want to reach their own target destinations. The goal of MAPF is to procure a plan of actions for each agent such that they end up in their target destination without ever colliding with any other agent [12, 13].

Today there are several real-world scenarios where MAPF is being applied. These include automation of warehouses [5], management of unmanned aerial vehicles [4], and autonomous vehicles [1]. In this thesis, the MAPF problem is applied to a bottlenecked mining ramp scenario where multiple agents, either manned or unmanned, request access to a mining ramp for transportation between the surface and the mine. The scenario will be described more in-depth later.

2.2 The problem definition of classical MAPF

Classical MAPF is a term that describes a generalised and simplified setting of the MAPF problem. The classical MAPF problem takes place in a spatially discretised map represented as an undirected graph G, as seen in fig 2.1. The graph consists of vertices (V) and edges (E) between them. Thus, $G = (V, E)$. In this graph there are k agents, each with a source vertex and a target vertex (its target destination) [12, 13].

Not only space, but also time, is discretised. In each time step, the agents are free to perform an action which is either move or wait. Performing a move action lets an agent move along an edge from its current position v to a neighbouring vertex v', such that by the start of the next time step the agent is positioned in v' and is now free to perform an action anew. The wait action is an action where the agent simply remains in v during the current time step [12, 13].

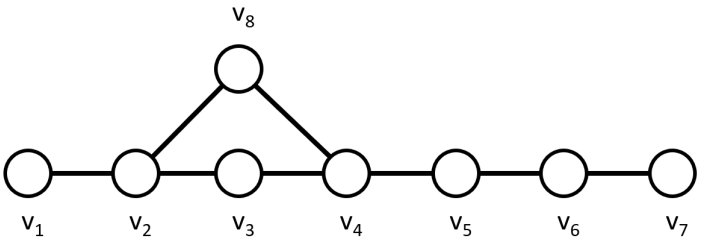

Figure 2.1: A simple, undirected, graph.

The journey of an agent from its source vertex to its target vertex is referred to as its plan. More specifically, this is a single-agent plan. A single-agent plan is a sequence of actions that, if an agent performs them in the specified order, takes the agent from its source vertex to its target vertex. A joint plan, on the other hand, is a set of all agents' single-agent plans that is conflict-free [12, 13].

A natural question that must always be considered is what happens if agents collide. Such events are referred to as conflicts. In classical MAPF, there are a number of conflicts that can occur. Depending on the scenario, some conflicts are allowed whereas others are prohibited. The types of conflicts relevant to this thesis are vertex conflicts, edge conflicts and following conflicts. A vertex conflict occurs whenever two or more agents occupy the same vertex in the same time step. An edge conflict occurs whenever two or more agents traverse along the identical edge in the same time step. This typically is the case whenever the agents swap positions, sometimes referred to as a swapping conflict. A following conflict occurs whenever an agent moves to a vertex simultaneously as another agent, previously having occupied that vertex, moves away from it. To conclude, the objective in MAPF is to find a valid solution, i.e. a plan for each agent, such that the procured joint plan is free from any prohibited conflicts [12, 13].

2.3 Going beyond classical MAPF

To summarise the previous section, classical MAPF assumes that (1) time is discretised, (2) agent actions take one time step to perform, and (3) an agent always occupies only one vertex at any given time. Naturally, these assumptions may not hold in the actual world. Although providing an easily comprehensible framework with which to work, classical MAPF fails to capture the nuances in the environments of the real-world MAPF problem scenarios. Thus, being able to go beyond these simplified rules may in many cases be necessary in order to address and capture the complexities that the real world poses. There are multiple ways in which classical MAPF can be extended to achieve this [12, 13]. Examples of particular relevance to this thesis are presented below.

Some scenarios may include agents of varying sizes. Depending on the graph used to model the scenario, agents may occupy more than one vertex, alternatively only occupy one vertex but prohibit other agents from occupying adjacent vertices [13].

Scenarios may also impose that actions take more than strictly one time step to perform. Such settings are often referred to as MAPF with non-unit edge cost, i.e. some edges cost more than others to traverse. To further extend the notion of time, in reality, time is continuous and not discrete. Discretising time is a major limitation in trying to model the real world, however treating time as continuous introduces added complexity [12].

An additional extension to classical MAPF is where agent velocities are not necessarily identical [13]. In the setting of a spatially and temporally discretised scenario, this means that agents with higher velocities travel more than one vertex per time step.

Lastly, classical MAPF is offline in the sense that no other agents than those present in the beginning of the scenario will enter the scenario later on. Thus, before anything is

executed, a plan can be procured, and that plan is fixed throughout the entire scenario. Conversely, online MAPF means that multiple plans might need to be procured. One type of online MAPF is called the intersection model. In this setting, new agents may appear throughout the elapsing scenario which requires a new plan to be generated. As the name suggests, this model is inspired by intersection scenarios where new agents come and go [13, 17].

2.4 Evaluation metrics of valid solutions

Naturally, a MAPF problem may have more than a singular valid solution. Depending on the situation, some attributes are more important than others and in those cases one would want to attain, and perhaps use, the most optimal solution for that situation. In classical MAPF problems, the most used evaluation metrics are makespan and sum of costs (SOC). The makespan simply is the number of time steps it takes for all agents to reach their target vertex. That is, the makespan corresponds to the elapsed time it takes for the joint plan to finish. SOC is the total sum of actions performed by the agents in the joint plan. As one can naturally deduce, makespan is a good evaluation metric for situations where one wants all agents to reach their destinations as fast as possible. SOC, on the other hand, is better whenever the total cost of operating the agents is of particular importance. To illustrate, if the agents are vehicles running on some kind of fuel and the desire is to minimise the fuel consumption, SOC would be a more fitting metric for evaluating the optimality of the plan solutions [12, 13].

Another key metric is the time taken to generate a solution. This is of particular relevance when coordinating traffic since, without a plan, agents have to wait until a plan has been generated. Thus, solutions taking too long to generate will cause traffic to be congested.

2.5 Search-based MAPF algorithms

In this section, some of the more established search-based MAPF approaches will be described. Search-based algorithms work by exploring (searching) the agent state space until a goal state is encountered.

2.5.1 Multi-agent A*

A* is a well-known single-agent path finding algorithm. In A*, each vertex has a heuristic (h) value which is a rough estimation of the cost required to reach a target vertex. h is calculated using a heuristic function $h(v)$ where v is the vertex. A* chooses the next course of action by comparing all options' f costs. For any given vertex, its f value is the sum of the cost to reach that vertex (g) and the h value of that vertex. A* guarantees optimality so long as $h(n)$ is admissible, i.e. if it is perfectly accurate or if it underestimates the actual remaining cost to a target vertex. A* starts with an initial node from which successor nodes are generated. A node holds information about the agent and its location. A* stores all successor nodes and always chooses to explore the node with the lowest f cost. When exploring the search space, A* keeps track of the nodes it has seen before to prevent duplicate nodes from potentially being explored again. The algorithm stops whenever a goal node is explored [3].

A* can be extended to also work in MAPF settings. Hereafter, multi-agent A* will simply be referred to as A*. Whereas a node previously consisted of the location of a single agent, a node is now a tuple of k agent locations. An A* node transition thus means that all k agents have taken an action. This, however, introduces the possibility of illegal nodes, i.e. nodes where at least one illegal action has been taken. This is the case, for instance, whenever agents end up at the same location or swap locations (or any other

type of conflict). Two extensions to multi-agent A* have been devised, both of which still maintain completeness [11].

The first is called operator decomposition (OD), where each individual agent action generates a new A* node. Thus, with OD, an operation consists of one agent's action, instead of all k agents' actions. The rationale behind using OD is that each node now has a lower branching factor. Assume that all agents can take four different actions at any given time. With OD, each node can result in four different successor nodes, whereas without OD, each node can result in 4^k successor states. However, whilst the branching factor decreases, the depth of the goal node increases since many more nodes must be created [11].

The second is called independence detection (ID). The idea is that agents are handled as if they are alone in the graph so long as they do not conflict with any other agents. Thus, every agent in the scenario is initially in a separate group. They are all given an individually optimal plan to their goal. All plans are then executed until a conflict occurs between two agents. If these agents are in different groups, group them together and generate a joint plan for them both. This is repeated until all agents receive conflict-free plans. In the best case, ID would result in A* being able to solve the MAPF problem by dividing it into k single-agent problems [11].

A* analysis

For each A* node, all its possible successor nodes are generated. The possible number of actions an agent can take is represented by its branching factor b. Thus, Sharon et al [10] show that the number of nodes generated is bounded by $O(X \times (b_{base})^k)$. Here, X is the number of nodes expanded by A*, b_{base} is the branching factor for a single agent, i.e. the number of possible actions to take by the agent, and k is the number of agents. Thus, $(b_{base})^k$ represents the average number of successor node generated by any given state.

2.5.2 Increased cost tree search (ICTS)

The increased cost tree search (ICTS) is a complete and optimal MAPF algorithm first introduced by Sharon et al [10]. The concept behind ICTS is that it attempts to find a solution with a cost equal to the individually optimal SOCs. Since this almost always fails, ICTS increases the plan costs for one agent at the time (in the form of child nodes) and re-attempts to find a solution with exactly the new plan costs. This continues until a conflict-free solution with the exact matching plan costs is found. ICTS employs a two-level strategy described in the following sections.

The ICTS high-level

The high-level constructs the increased cost tree (ICT) where each node consists of a vector of each agent's individual plan cost, as if each agent was alone in the ramp without competition. The root node vector contains the optimal plan cost for each agent. To check the ICT node for a solution, the low-level is invoked. If it fails to find a joint conflict-free solution, child ICT nodes are generated. In each child node, one of the agents' plan lengths have been incremented by one. This will result in identical child nodes at later generations, hence why any duplicate nodes are automatically pruned. As is perhaps apparent, the high-level searches the ICT for a goal node in a breadth-first search manner. An ICT node is considered a goal node whenever there is a conflict-free solution where each agent's plan cost matches the corresponding cost specified by the ICT node vector [10].

The ICTS low-level

The low-level is invoked to find a solution given the specified plan costs of each agent. The following is how the low-level functions. Assume that the ICT node vector's first value is

five. This means that the low-level will impose all allowed five-action combinations on the agent and check if any of them result in the agent successfully reaching its target vertex. If so, it continues with the next agent. If this succeeds for all agents without there being any, conflicts, the ICT node is declared a goal node. Otherwise, child ICT nodes are generated [10].

In practice, Sharon et al [10] make use of so-called multi-value decision diagrams (MDD) as a data structure to store the plan of an agent with a specified cost (see fig. 2.2). The idea is that every level of the MDD corresponds to a time step, where each MDD node in that level corresponds to a possible location of the agent at that time step. Thus, the top and bottom level of an MDD corresponds to the agent's source and target vertex, respectively. Any intermediate level i contains all possible agent locations after performing i actions. fig. 2.2 shows the MDD for an agent starting in v_1 whose goal is v_7 in the graph shown in fig. 2.1.

For an ICT node, the low-level begins with constructing an MDD for each agent by imposing a specified number of actions. To check if the ICT node is a goal node, the low-level looks if there exists at least one combination of plans in each MDD where the plans do not conflict. If so, a goal node is found [10].

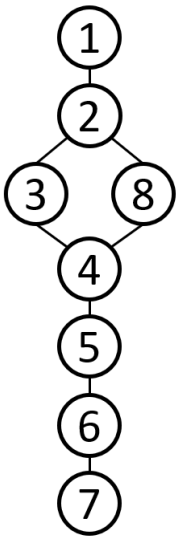

Figure 2.2: An MDD showing the possible paths from v_1 to v_7 in six actions.

ICTS analysis

Sharon et al [10] prove that for any given ICT node, the low-level search will expand at most X low-level (MDD) nodes. Furthermore, the number of generated ICT nodes is bounded by k^{Δ} where k is still the number of agents, and Δ is the depth of the ICT from the root node to the goal ICT node. Thus, the number of low-level nodes expanded by ICTS is bounded by $O(X \times k^{\Delta})$.

2.5.3 Conflict-based search (CBS)

Conflict-based search (CBS) is another complete and optimal MAPF algorithm first introduced by Sharon et al [9]. As the name suggests, the idea behind CBS is to apply constraints to agents whose plans collide, to prevent them from happening. These constraints are used when finding a plan for an agent, as they inform about the actions that the agent is prohibited from taking. If that plan ends up colliding with the plan of another agent, a new constraint is added to the agent.

CBS operates in two levels – the high-level and the low-level. The high-level is responsible for finding and applying conflicts, whereas the low-level performs the actual plan finding for the agents. It uses high-level conflicts to guide its process in finding plans for the agents [9]. The levels are described in more detail in the following sections.

The CBS high-level

The high-level CBS operates on a binary constraint tree (CT). Every CT node contains three kinds of data: (1) a set of constraints, (2) a joint plan solution (i.e. the set of all agents' individual plans), and (3) the SOC for the solution. The CT always starts off with a root node. The root always has an empty set of constraints, and its solution is the set of all agents' individual optimal plans. Thus, the root node disregards any conflicts between the current agent plans. The goal of the high-level is to find a CT goal node, which is a node whose solution is free from conflicts, upon which the algorithm is done [9].

The high-level always chooses to process the CT node with the lowest cost amongst those who have not yet been processed, i.e. open nodes, in a best-first search manner. When the high-level processes a CT node it invokes the low-level search. The low-level finds the shortest plan for each agent such that they all comply with the constraints associated with the CT node. If the low-level fails, the CT node will not generate any children and the branch is therefore terminated [9].

Next, the solution of the CT node is validated by checking the solution for any conflicts. As mentioned, should such a solution prove to void of conflicts, the CT node is a goal node. Otherwise, the generation of two CT child nodes is prompted. A child node always inherits the set of constraints of its parent, as well as is added with a new constraint relating to the conflict that prompted the child node generation. To illustrate this, assume there is a vertex conflict between agents A and B in vertex v at time t. The conflict can be resolved in two ways; either A is prohibited from occupying v at time t, or B is prohibited from it. The former implies a constraint which is denoted (A, v, t), meaning A is not allowed to occupy vertex v at time t. The latter constraint is conversely denoted (B, v, t). For each child node, the low-level is again invoked but only on the agent affected by the newly added constraint. The remaining agent plans remain the same. Finally, the child nodes are added to the list of open nodes, ordered by their solution costs [9].

Note that the same applies for edge conflicts. If agent A, positioned at vertex v, and agent B, positioned at vertex v', try to swap positions at time t, constraints prohibiting them will be created. More specifically, such constraints are denoted $(A, (v, v'), t)$ and $(B, (v', v), t)$, respectively [9].

The CBS low-level

The low-level CBS is responsible for searching for optimal individual plans for all agents, where the associated constraints are complied with. It does this one agent at the time, hence conflicts between the plans may exist. The algorithm typically used in the low-level is A* [9].

CBS analysis

Let Y be the number of CT nodes that are expanded. In the worst case, for each time step, each agent is constrained to avoid every available neighbour vertex but one. In other words, there is a maximum number of conflicts (and therefore constraints added) without the CT node being terminated due to the low-level search failing to find a solution satisfying the constraints. If C^* is the cost of the optimal solution, the total number of time steps is also C^*. Since each CT node can generate at most two children, in the worst case, 2^d CT nodes will be generated, where d is the depth of the CT from the root to the goal CT node. Therefore, the number of expanded CT nodes is bounded by $O(2^{(|V| \times C^*)})$, where $|V|$ is the number of vertices in the graph. Now, for each of these high-level nodes, the low-level

search expands at most $|V|$ nodes for each time step (which was C^*). Therefore, the total number of expanded low-level nodes is bounded by $O(2^{(|V| \times C^*)} \times |V| \times C^*)$ [9].

2.5.4 CBS with priorities (CBSw/P)

CBS with priorities (CBSw/P) is a variant of CBS first introduced by Ma et al [6]. CBSw/P is an algorithm that uses consistent prioritisation to search for a solution. It works identically to CBS but with a few modifications, mainly in that CT nodes now also contain a priority ordering (note that the root CT node starts off with an empty priority ordering). The priority ordering is a strict partial order on the agents and who has precedence over who. If a CT node is not a goal node and generates children, it will never generate children whose added constraint implies a priority ordering that contradicts the priority ordering of itself. To illustrate this, consider a CT where a CT node with two agents generates child nodes. The parent CT node has $a_2 < a_1$ (a_2 is prioritised over a_1) as its priority ordering set. The child node that is added a constraint which implies that a_1 has precedence over a_2 ($a_1 < a_2$) will not be generated since such a constraint contradicts the priority ordering of the parent.

This means that CBSw/P creates far less children than CBS. However, since not all possibilities are considered following a conflict, CBSw/P does not guarantee returning an optimal solution. Even so, CBSw/P is much faster than CBS since the tree is much sparser [6].

2.6 Constraint-based MAPF approaches

The algorithms described thus far fall under the category of search-based algorithms. However, it should be noted that there are other approaches, such as reduction-based solvers [7]. One reduction-based approach typically found in the literature is the constraint-based approach. Constraint-based approaches work in such a way that the MAPF problem is modelled as (reduced to) other known problems. These problems can then be solved with a general-purpose solver, which can then be used to find a solution [12]. Examples of constraint-based approaches are those reducing the MAPF problem to a constraint optimisation problem [8], boolean satisfaction problem [14], inductive logic programming [15] and answer set programming [2].

2.7 Replanning algorithms for online MAPF

Since online MAPF scenarios introduce the possibility of new agents entering throughout the course of the scenario, they necessitate the use of a replanning mechanism. Online scenarios generally work in such a way that the initial problem is offline. The MAPF solver must thus start off with solving the offline problem, and the generated solution can then be executed. However, as soon as new agents arrive, the MAPF solver must solve the problem anew, this time with the original agents at their current positions as well as with the newly entered agents. There are mainly two ways to go about replanning: replan single (RS) and replan all (RA) [17].

The RS algorithm looks at each new agent and tries to find an individually optimal plan that does not collide with the already existing agents [17].

The RA algorithm does the complete opposite; it looks at all agents, both the existing and the new, as equals in terms of planning. The RA algorithm thus discards any previously generated plan and generates a completely new one. Perhaps self-explanatory, the RA algorithm is more complex than the RS algorithm. However, it comes with a great advantage in that it always generates snapshot optimal plans [17].

In an online MAPF setting, snapshot optimality means that a generated plan is optimal, in terms of SOC, for all agents assuming no further agent entries will occur. However,

snapshot optimal plans do not guarantee that the complete online MAPF solution is optimal (in terms of SOC). This is because the choices made up until the point where the new agents arrive might not have been optimal, since there was no way of knowing that the new agents would enter to begin with. Since the agents up to this point have committed to the previous plan by executing it, any previous alternative plans that would have provided a better starting point for when the new agents arrive have been automatically discarded. Nonetheless, striving to generate snapshot optimal plans is the best way to achieve a low SOC solution [17].

Chapter 3
The mining ramp scenario

This section introduces the scenario on which the thesis is based. With a scenario presented, a brief discussion will be held regarding how and why certain algorithms were selected to include in the thesis.

3.1 The ramp scenario

As mentioned, this thesis is based on a mining ramp scenario. Mining ramps serve the purpose of facilitating transport of vehicles between the surface and the underground. Vehicles heading underground may for example transport workers that are beginning their shifts, and vehicles heading to the surface may for example carry excavated resources for further processing and use. The mining ramp imposes several constraints on the scenario.

1. The ramp only has one lane, which is the main bottleneck. This means that no two vehicles can pass by each other whilst simultaneously moving along the ramp. To alleviate this bottleneck, there are passing bays located at various positions throughout the ramp that vehicles may occupy to allow other vehicles to pass. These passing bays can fit only one vehicle at a time.

2. Whereas agents are allowed to wait inside a passing bay, they may not wait or turn directions whilst on the ramp itself.

3. Some vehicles must never stop once they enter the ramp. Such vehicles are therefore not able to enter passing bays, meaning that any oncoming vehicles must stop at a passing bay allowing them to pass without interference.

4. Implied by the previous constraint is that some vehicles have higher priority than others. An example of a prioritised vehicle would be one that carries workers to the underground whenever they are urgently needed. Another possible scenario is that there is a lack of free space underground that requires vehicles to leave at a rate higher than the number of vehicles arriving.

5. The queues must maintain a valid queue behaviour. More specifically, if an agent inside the queue stays in the same vertex with the vertex in front being free, this is an invalid behaviour. All agents must always move forward if such a move is possible.

As one may imagine, access to the ramp can be quite contested at times. Vehicles thus request access to the ramp by joining a queue at the surface or the underground. Only the vehicle at the head of the queue can enter the ramp at any given time. Following on the discussion from section 2.3, the mining ramp scenario is an online MAPF scenario.

Similarly to the intersection model, vehicles continuously enter and leave the scenario, with the difference being that instead of a classical intersection there is now a single-lane ramp.

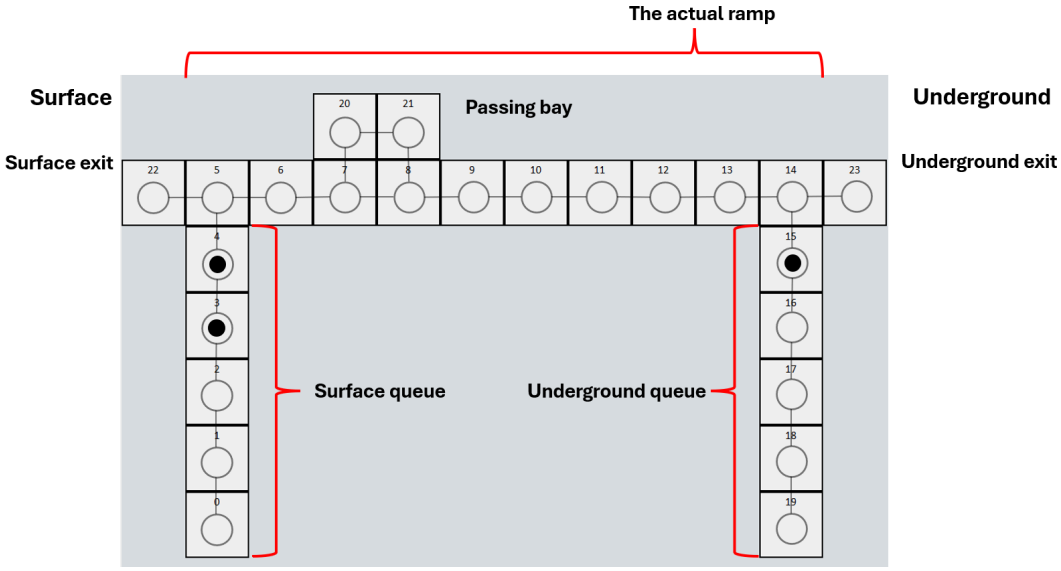

Figure 3.1: A representation of a mining ramp, with the different parts of the ramp labelled. Layered on top of the ramp is how it is represented as a MAPF graph, with agents occupying v_3, v_4, and v_{15}.

Fig. 3.1 shows the mining ramp and how it is represented as a traditional MAPF graph, where a vehicle occupies one vertex at any given time. Time is trivialised as it is divided into discrete time steps, as is space with the vertices. Moreover, the notion of momentum is disregarded in this scenario; moving is not affected by whether the vehicle was previously stationary or already in motion. Furthermore, as was described above, each vertex at either end of the ramp (denoted end vertex) is associated with a queue where vehicles wait to enter the ramp through that end vertex. At both ends of the ramp, there is an exit vertex, i.e. the target vertex of all agents starting at the opposite end of the ramp. If a vehicle reaches the start vertex at the opposite end of the ramp at time step x, at time step $x+1$ it moves to the exit vertex and stays there permanently. This simulates the agent having left the scenario. At time step $x+1$, the vehicle at the head of the queue for the same end vertex can enter. One vehicle may thus leave while another enters the ramp through the same vertex during the same time step.

To conclude, the mining ramp scenario incorporates aspects of both classical MAPF and extensions beyond classical MAPF. As for classical MAPF, the scenario assumes that all vehicles travel with equal speed, the vehicles are of equal size, and both time and space are discretised. The scenario extends the classical MAPF framework mainly by the fact that it is online, in that new agents can constantly enter and leave the ramp. Additionally, the graph itself goes beyond the assumptions of classical MAPF, because different sections of the graph constitute different parts of a mining ramp. These sections have unique rules and constraints, as described above.

3.2 How the MAPF algorithms were selected

As mentioned, there are two large categories of MAPF approaches: (1) search-based approaches and (2) reduction-based approaches. From surveying the literature, the search-based algorithms seemed to be most widespread [12, 13]. To keep the thesis project within a feasible scope, the work was therefore limited to the search-based approaches.

As for selecting what search-based algorithms were most appropriate for this scenario, a brief discussion is warranted. The performance of an algorithm on a specific problem scenario is very difficult to predict. This is because a scenario with a given set of factors, such as graph size and number of agents, can still look very differently. Thus, it is rarely obvious exactly how well an algorithm will perform on a specific problem. Indeed, as Stern [12] puts it "there are no clear guidelines to predict which of the MAPF algorithms [...] would work best for a given problem". Rather, the analyses of the algorithm performances, as described in chapter 2, may give hints to how well the algorithms perform given a specific problem. To complicate the matter more, as the previous section mentioned, the mining ramp is not an ordinary graph where all vertices are on equal terms. This is because different parts of the graph, notably the queues, passing bays and the ramp itself, constitute different parts of the complete ramp, and serve different purposes. With such added peculiarities, predicting the algorithms' performances becomes even more difficult.

When selecting what search-based algorithms to use, the original idea was to choose those who looked most promising in their performance analyses (sections 2.5.1, 2.5.2 and 2.5.3). However, with the introduced peculiarities of the mining ramp, making such predictions would be even more difficult than normal. Therefore, whilst analysing the algorithms gave hints to how they would perform, the decision was made to go with the algorithms most prominent in the literature. Since traffic efficiency is of the essence, the complete and optimal algorithms A*, CBS and ICTS were used. Moreover, the CBSw/P algorithm was also included to be able to compare a faster, albeit non-optimal, option to the optimal algorithms.

Chapter 4
Methods and tools

4.1 Work process methodology

The project process consisted of several distinct phases. The first part of the project was spent researching the MAPF research domain. This was done by initially reading review articles and later original papers. With an understanding of the field of MAPF, candidate algorithms were selected for evaluation. As was apparent in chapter 2, the research was mainly done on search-based MAPF algorithms since those seemed to dominate the field.

With the algorithms selected, the next phase was the implementation phase where the ramp and algorithms were developed. This phase was the most comprehensive phase throughout the project. The implementation part was divided into several steps, each with a certain goal. The first goal was to implement and get a good representation of the ramp graph. This was developed first since no algorithm would work without it. The next part was implementing the algorithms, one at a time. Naturally, this was where most of the time was spent. The final part of the implementation phase was spent developing a graphical user interface (GUI) with which the user could interact and get a graphical representation of the generated solutions.

With the code in place, experiments were conducted for evaluation and comparison between the algorithms. The final part of the project was writing and completing the thesis report. Fig. 4.1 shows the Gantt chart for outlining the work process.

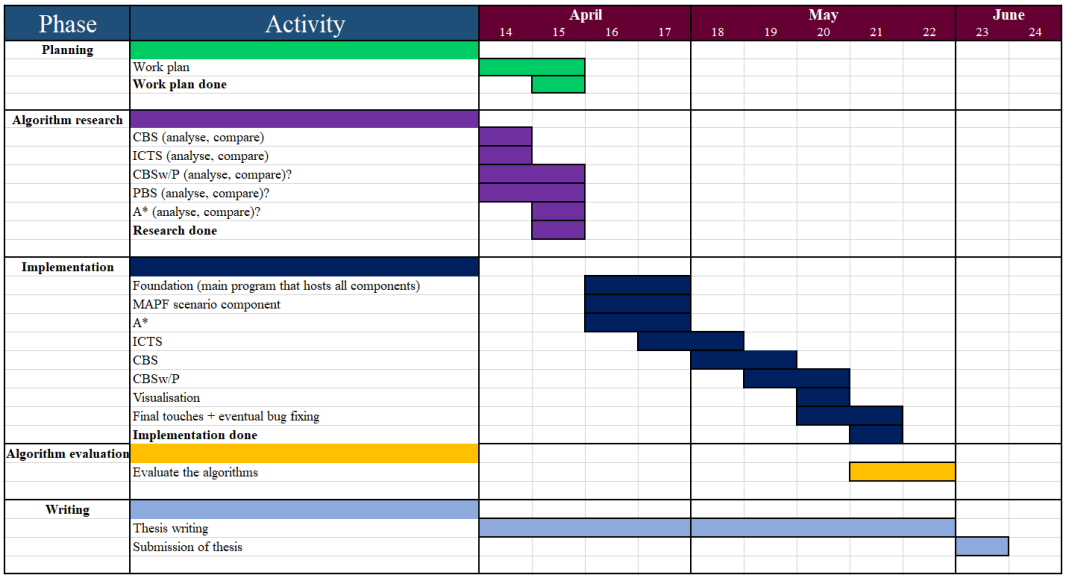

Figure 4.1: The work process Gantt chart used throughout the project, showing the project timeline with phases.

4.2 Programming language and libraries

The software was implemented using Java. All methods and classes were self-made without the use of any external libraries. The Java Swing framework was used for implementing the GUI.

4.3 Tools and resources

The IDE used for development was IntelliJ IDEA Community Edition 2024[1]. Figures and tables were created using Microsoft PowerPoint and Microsoft Excel. For running the experiments, a stationary PC running the Windows 10 operating system was used. The component specifications are listed below:

- CPU: Intel Core i5-6400 @ 2,7 GHz
- GPU: Nvidia GeForce GTX 1060 6GB
- RAM: 40GB

[1]IntelliJ IDEA. https://www.jetbrains.com/idea/

Chapter 5
System architecture and implementation

This chapter begins with describing the structure and architecture of the software. This is then followed by a deep-dive into how the software was implemented, what decisions were made and the rationale behind those decisions. The purpose of this chapter is to shed a light on what went into the creation of the software as a whole.

5.1 Overview of the system

On an overarching level, the system consists of several components as seen in fig. 5.1. At the heart lies the MAPF solver which is responsible for invoking a MAPF algorithm. This requires a MAPF scenario which specifies the ramp structure and times of agent entries. With a procured solution, the visualiser graphically presents it to the user.

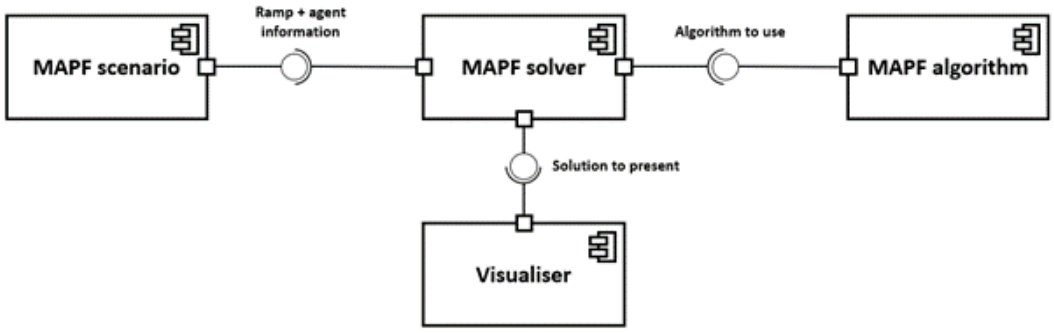

Figure 5.1: A mining ramp represented as a MAPF graph, with the different parts of the ramp labelled.

In essence, the components are responsible for the following:

- The **MAPF scenario** represents the scenario, including ramp and agent properties. It also specifies the latest time step at which new agents can arrive to the ramp.
- The **MAPF algorithm** constitutes any of the MAPF algorithms and contains the logic for them. When done, the solution is returned to the MAPF solver.
- The **MAPF solver** initiates the solution-finding process by invoking an algorithm. It is also responsible for checking for any arriving agents, at which point it invokes the algorithm to replan the traffic.

- With a solution from the MAPF solver, the **visualiser** constructs a GUI and allows the user to see the solutions along with statistics.

The following sections deep dive into the components as well as other integral structures.

5.2 The MAPF scenario and MAPF state

The MAPF scenario serves as the input to the MAPF solver. This structure stores information about the ramp, the initial MAPF state, agent entries and constraints. The agent entries contain information about at what time steps new agents, and what agents, will arrive. This means that all agent arrival information is specified before-hand. However, as is explained in section 5.4, only parts of the agent arrival information is accessed at a time. More details about the ramp and agents follow in section 5.3.

The initial MAPF state serves as the starting point from which the algorithms will later start searching. The MAPF scenario generates this state by inquiring from the agent entries what agents exist in the first time step. The MAPF state mainly stores a map between all agents and their locations, the time step it represents, as well as the total f, g and h values. The g value corresponds to the sum of travel costs for all agents to reach the current state. The h value is the sum of agent h values, and is used by A* as explained in section 2.5.1. The f value is the sum of the g value and h value. The relevance of these values are explained in section 5.5.1.

The relations between the MAPF scenario, MAPF state, ramp and the agents are illustrated in fig. 5.2

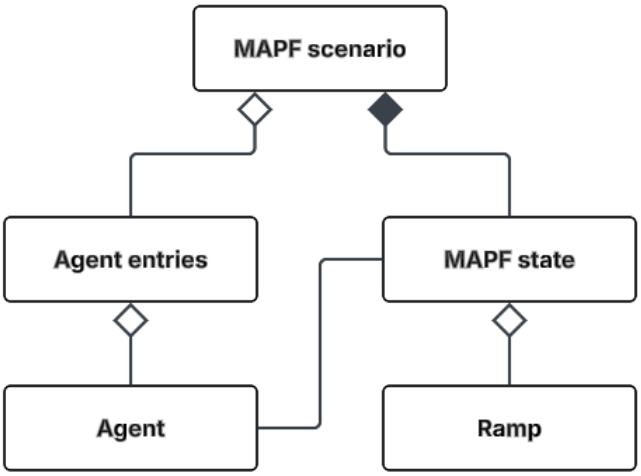

Figure 5.2: A simple class diagram illustrating how the MAPF scenario, MAPF state, ramp and agents are related.

5.3 The mining ramp and agents

As mentioned, the mining ramp was the first part of the system that was implemented, since it plays a fundamental part in the system. The mining ramp stores information about the structure of the ramp. This includes the ramp length, queue lengths and passing bay locations, all of which are specified by the user. Moreover, upon initialisation, it calculates the h value for each vertex in both directions. This is possible since the target vertices are always the same, i.e. at the ends of the ramp. This is a critical detail, since this ensures an

admissible heuristic. More accurately, the heuristic is perfect in that it neither over- nor underestimates the remaining cost to reach the target vertex. This gives A* its guarantee of optimality.

A few words on how the passing bays are constructed are warranted. In initial versions of the ramp, a passing bay only consisted of one vertex which connected two vertices to bypass the vertex in between them. The problem with this was that agents would consider using the passing bays to be equally efficient as not using them. To make the scenario more realistic, the passing bays were modified to consist of two vertices connecting two adjacent ramp vertices. With this implementation, agents would not use passing bays unless necessary.

The ramp itself is an unweighted directed graph. Two alternatives were considered for representing the ramp; as an adjacency matrix or as an adjacency list. In an adjacency matrix, each row represents the vertex from which an edge goes. Conversely, the columns represent the vertices to which edges go. In the case of an unweighted graph, if there is an edge going from vertex *a* to vertex b, the entry in the matrix at position *[a][b]* is 1, else 0. On the other hand, in an adjacency list, each element represents a map between a vertex and its neighbours, i.e. the vertices to which the current vertex is connected. Both alternatives would be viable but ultimately the decision was made to use an adjacency list as the ramp representation, mainly by the fact that maps are easily created in Java.

The ramp makes a distinction between edges in the down-going and up-going directions, as well as storing information about what vertices are part of what ramp structures. This is required by the algorithms when determining the eligible neighbouring vertices for an agent. Each agent stores information about its direction, ability to use passing bays, and priority status. Whenever determining the eligible neighbouring vertices, its direction is inquired. Moreover, if the neighbouring is in a passing bay, the agent's ability to use passing bays is inquired.

Exactly how the ramp and the agents are used in the system is shown in fig. 5.2.

5.4 The MAPF solver and MAPF solution

The MAPF solver is the hub of the system that brings all components together. Given a MAPF scenario, it is responsible for invoking an algorithm and later returning a MAPF solution if one has been obtained.

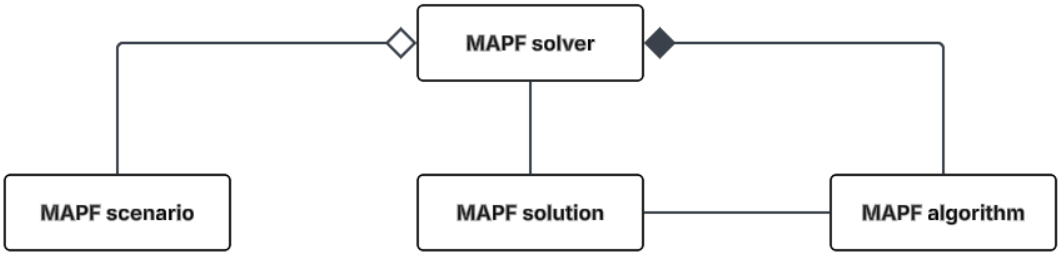

Figure 5.3: A simple class diagram illustrating how the MAPF solver, MAPF scenario and MAPF algorithm are related.

The relationship between the MAPF solver, MAPF solution and MAPF algorithm is shown in fig. 5.3. The MAPF solver starts with solving the initial offline problem by simply feeding the MAPF scenario it was given to an algorithm. The algorithm, if successful, returns a MAPF solution. A MAPF solution contains a series of MAPF states, one for each time step, that together represent the solution from start to finish. With a solution to the offline problem, the MAPF solver starts inquiring the agent entries, one time step at a time, for new arriving agents. Whenever new arriving agents are detected, the MAPF solver initiates its replan algorithm. To ensure the solutions are snapshot optimal, the MAPF

Algorithm 1 The replanning algorithm

Require: *MAPFScenario*

```
1: currentSolution = algorithm's solve(MAPFScenario)
2: for timeStep in duration do
3:    if MAPFScenario.agentEntries has an entry at timeStep then
4:       newInitialState = currentSolution.get(timeStep)
5:       add new arriving agents to newInitialState
6:       create newMAPFScenario
7:       currentSolution = algorithm's solve(newMAPFScenario)
8:       starting from the last MAPFState in currentSolution, traverse backwards to con-
         struct the full series of MAPFStates and assign to currentSolution
9:    end if
10: end for
11: return currentSolution
```

solver employs a RA algorithm for replanning. The pseudocode is shown in algorithm 1. Assume that new agents arrive at time t. RA discards any MAPF states of the previous (offline) solution that represents a time step later than t. The MAPF state representing the state at time t will be used as the initial state for the new MAPF scenario. RA modifies this MAPF state by inserting the new agents into the ramp. With a new MAPF scenario constructed, the MAPF algorithm is invoked anew. This process repeats itself until no further agent arrivals occur.

5.5 The MAPF algorithms

5.5.1 A*

Algorithm 2 shows the pseudocode for how the implemented A* works. It follows the classical A* very closely with a few modifications. A frontier is used to keep track of all current A* nodes not yet expanded. Note that an A* node is represented as a MAPF state with all agents and their locations. The input MAPF scenario's initial state is first added to the frontier, which is a priority queue where nodes with the lowest f values are first. This is to ensure that the cheapest options are considered first, which is what gives A* its optimality guarantee (assuming an admissible h, see section 5.3). Until a goal node is expanded, or the frontier is empty, the following is done. A* first checks if the currently polled node is a goal node. A goal node is a node where all agents are in an exit vertex. If the node is not a goal node, it is added to the expanded list. This is used to prevent duplicate nodes from being expanded. Next, the possible moves for all agents are inquired. Recalling back to section 5.3, this is where the agents' direction and ability to use passing bays are inquired to rule out what moves are legal for each agent. With all possible moves attained, a cartesian product is generated to get all possible successor nodes to the current node.

Before adding the successor states to the frontier, they must be validated. The validation is divided into several steps since the mining ramp imposes several constraints. First, the queue behaviour of each successor state is validated. As mentioned in chapter 3, an agent inside a queue must always move forward if possible. Thus, any successor nodes with an invalid queue behaviour are automatically discarded. The remaining candidate successor nodes are now validated in terms of vertex and edge conflicts. It is here that passing bay violations are checked. All of this is done by looking at the agent locations and marking them as prohibited (if an agent is in a passing bay, all vertices of that passing bay are automatically marked as prohibited). As soon as an agent is seen moving to an occupied vertex, or making a swapping move with another agent, the successor node is immediately discarded. Finally, the remaining successor nodes are added to the frontier,

Algorithm 2 The implemented A* algorithm

Require: *MAPFScenario*

```
1: initialState = MAPFScenario.initialState
2: frontier = an priority queue ordered by MAPFState.f-cost (h-cost if tied)
3: expanded = an empty set
4: enqueue initialState to frontier
5: while not empty(frontier) do
6:    currentState = poll(frontier)
7:    if isGoal(currentState) then
8:       solution = buildSolution(currentState)
9:       return solution
10:   end if
11:   add currentState to expanded
12:   for each remaining moveCombination do
13:      if moveCombination is free from conflicts and invalid passing bay and queue be-
         haviour then
14:         create newMAPFState
15:      end if
16:      if not expanded.contains(newMAPFState) then
17:         enqueue newMAPFState to frontier
18:      end if
19:   end for
20: end while
21: return null
```

given that they do not exist in the expanded list, after which the cheapest node in the frontier is polled.

This thesis also incorporates the option to prioritise some agents over others. To implement A* in a way where this is possible, the way the frontier works is modified. In classical A*, the g, h and f values are based on all agents. To enable A* to prioritise higher priority agents, separate g, h and f values that only look at higher priority agents are used to evaluate the nodes. If, however, there is a tie between two nodes, the evaluation resorts to comparing to classical f values (and h values if there is a tie between the f values).

Two extensions developed by Standley [11] were mentioned in section 2.5.1, OD and ID. The main benefit of OD was that it reduced the branching factor if the A* nodes. However, since the branching factor per agent is low in the mining ramp scenario (1-2 possible moves per time step), the reasoning was that the time investment of implementing OD would outweigh its benefits. For this reason, OD was not implemented. The idea behind ID was that it treats agents as independent if possible to minimise the use of the multi-agent aspect of multi-agent A*. In the mining ramp scenario, however, agents going in different directions are bound to collide since the ramp only has one lane, and hence agents would ultimately be grouped together. For this reason, it was decided that ID would not be implemented.

Lastly, a few words on tie-breaking are warranted. In many cases, there will be multiple A* nodes sharing the same lowest f value in the frontier. Tie-breaking was implemented in a way where, if two nodes share the same f value, their h values will be compared next and the one with the lowest h value will be polled first. Again however, the nodes might also share the same h value. In those cases, no further comparison is made and any of the nodes are polled.

5.5.2 ICTS

Algorithm 3 shows the pseudocode for the implementation of ICTS. ICTS starts by handling the root node. To attain the individual optimal plan for the agents, a MAPF scenario is created for each agent. For each MAPF scenario, the same A* implemented in the previous section is invoked. Note that this is effectively a single-agent A* since each scenario only contains one agent. The solution costs are stored in the root ICT node's cost vector. The solutions themselves are used to construct the MDDs representing the plan for each agent. The MDD is represented as a linked list of MDD nodes. Each MDD node, in turn, stores information about the vertex it represents, as well as its parent and children MDD nodes. The parent is the predecessor from which the current MDD node was generated. Conversely, the children are the immediate successor nodes of the current MDD node. Their purpose is described shortly. The root ICT node contains one MDD for each agent, as well as the cost vector specifying their lengths. The root ICT node is immediately inquired to see if it is a goal node. This is done by converting the MDDs of the ICT node to a set of solution plans (described shortly), if one exists. If found, a MAPF solution is constructed and returned.

If not, child ICT nodes are generated, where each child increases one agent's plan cost by one. These children are then added to an ordinary first-in-first-out queue from which ICT nodes will be polled. This queue ensures the ICT being searched in a breadth-first search manner. With a polled child ICT node, a number of actions are imposed on all agents. This number corresponds to the cost specified in the cost vector of the ICT node. Note that for an agent, there might be multiple plans of equal cost that take the agent to its target. For each of these plans, an MDD is created. With all possible MDDs per agent generated, the ICT node is inquired to see if it is a goal node, again by converting the MDDs to a set of solution plans should one exist.

The process of generating a set of solution plans from an ICT node's MDDs works as follows. First, since each agent may have multiple MDDs representing the different plans that the agent can follow to reach the target vertex, the cartesian product of all possible MDD combinations is generated. This ensures all joint plan combinations are considered. For each MDD combination, the following is done. One level in the MDDs at a time, all MDD nodes therein are added as a list to a queue, starting with the root MDD nodes. If all MDD nodes represent target vertices, a valid joint plan solution has been found. This will most likely not be the case for the root MDD nodes. Instead, the cartesian product of all MDD node children is generated, where a child represents the next vertex in the plan. This is to ensure all possible next move combinations are considered. Every move combination is checked to see if it has any vertex conflicts, edge conflicts or passing bay violations. If so, it is discarded. If not, it is added to the queue. Whenever the queue is polled a set of MDD nodes representing only target vertices, the agent plans leading up to those MDD nodes are constructed. This is done by backtracking via each MDD node's parent until reaching the root MDD nodes. Finally, before being returned, the joint plan solution is checked to see if it contains any queue behaviour violations. If so, it is discarded and the queue is polled anew. If not, a valid goal ICT node has been found, and the solution is returned.

Worth noting is that in their paper, Sharon et al [10] merge multiple agent MDDs into a k-agent MDD where each MDD node is a representation of all agent locations. k-agent MDDs were not used in this implementation, mainly due to the added complexity it would introduce. Instead, the MDDs were kept separated and cartesian products between them were generated.

ICTS differs from the other agents in that its high-level search is performed in a purely breadth-first search manner, compared to a best-first search manner. Since ICTS fundamentally looks at the costs for all agents to search (and build) the ICT, introducing priorities would drastically alter the way in which ICTS works. For this reason, ICTS is not modified to be able to prioritise higher priority agents.

Algorithm 3 The implemented ICTS algorithm

Require: *MAPFScenario*

```
1: root = an empty ICT node
2: initialState = MAPFScenario.initialState
3: initialSolutions = an empty set of MAPFSolutions
4: costVector = an empty set of solution costs
5: for each agent in initialState do
6:    run A* on agent in isolation
7:    store cost of solution in costVector
8: end for
9: root.costVector = costVector
10: for each MAPFSolution in initialSolutions do
11:    currentMDD = createMDD(MAPFSolution)
12:    root.agentPaths.add(currentMDD)
13: end for
14: if a conflict-free set of agent paths is found then
15:    solution = buildSolution(root)
16:    return solution
17: end if
18: generateChildren(root)
19: ictQueue = queue of ICT nodes
20: enqueue root.children to ictQueue
21: while not empty(ictQueue) do
22:    currentNode = poll(ictQueue
23:    for each agent in initialState do
24:       get all possible valid paths for agent
25:       convert agent paths to MDDs
26:    end for
27:    if a conflict-free set of agent paths is found then
28:       solution = buildSolution(currentNode)
29:       return solution
30:    end if
31:    generateChildren(currentNode)
32:    enqueue currentNode.children to ictQueue
33: end while
34: return null
```

5.5.3 CBS

Algorithm 4 shows the pseudocode for how the implemented CBS works. Like ICTS, CBS starts with handling the root node by initialising it with agent plans and a total cost. For each agent, an independent optimal plan is obtained by invoking A* (see section 5.5.1). The set of constraints in the root node is empty. This root node is then immediately enqueued to a priority queue. The priority queue prioritises based on the sum of costs amongst the agent plans, thus giving the CBS high-level its best-first search property. If multiple CT nodes share the same lowest cost, any of them will be polled.

Algorithm 4 The implemented CBS algorithm

Require: *MAPFScenario*

```
1:  root = an empty CT node
2:  initialState = MAPFScenario.initialState
3:  for each agent in initialState do
4:      run A* on agent and store solution in root.allPaths
5:  end for
6:  if a path was not found for an agent then
7:      return null
8:  end if
9:  ctPrioQueue = an empty priority queue ordered by CT node cost
10: enqueue root to ctPrioQueue
11: while not empty(ctPrioQueue) do
12:     currentNode = poll(ctPrioQueue)
13:     conflict = validate(currentNode)
14:     valid = arePathsValid(currentNode.allPaths)
15:     if valid and no conflict then
16:         solution = buildSolution(currentNode)
17:         return solution
18:     end if
19:     if not valid then
20:         discard currentNode and continue
21:     end if
22:     generateChildren(currentNode)
23:     for each child in currentNode.children do
24:         run A* on the constrained agent to get a new path
25:         if the solution is not null then
26:             enqueue child to ctPrioQueue
27:         end if
28:     end for
29: end while
30: return null
```

Whenever a CT node is polled from the priority queue, its solution is checked to see if it contains any vertex or edge conflicts. This is done by going through the agent plans, in a pair-wise manner, to see if any conflicts exist. If so, the CT node is not a goal node and appropriate constraints are added to its child nodes. Special care is taken with passing bays since they consist of two vertices effectively working as one in the sense that if one is occupied by an agent, the other must be free. For this reason, special constraints are constructed. Consider a scenario where an agent is waiting inside a passing bay. If another agent, going in the same direction, decides to enter the passing bay, a traditional vertex conflict would not identify it as a conflict. If two agents, going in opposite directions, enter a passing bay simultaneously, both agents, in different child nodes, will be added a vertex constraint preventing them from entering the passing bay at that time step. Therefore, the validation process is modified to detect such scenarios as a vertex conflict.

Immediately after being checked for any conflicts, the node is checked to see if its solution violates any queue or passing bay behaviour. If the node is both conflict-free and does not violate any rules imposed by the ramp, it is considered a goal node and a solution is built from the agent plans. However, if the node is conflict-free but violates any ramp-imposed rules, it is discarded. This is because no conflicts have been encountered and therefore no children can be generated.

Otherwise, if the node contains a conflict, two child nodes are automatically generated. Depending on the nature of the conflict, the appropriate constraints are created and added to the children. For example, an edge conflict must contain a prohibited move which

involves two vertices, whereas a vertex conflict only involves one vertex. With two child nodes, and a new constraint added to each, A* is invoked to generate a new plan for the agent affected by the newly added constraint. The remaining agents remain untouched since no further constraints are imposed on them. The child nodes are then assigned a new cost, after which they are enqueued to the priority queue.

Enabling CBS to prioritise higher priority agents is very similar to how it is done in A*. Instead of comparing the CT nodes' SOC for all agents, CBS can compare only the higher priority SOC. Thus, the only part where CBS is modified is the nature of the priority queue.

5.5.4 CBSw/P

CBSw/P inherits the entire implementation of CBS, with only a few things modified. First, the CT node used in CBSw/P (which inherits the implementation of the regular CT node) is equipped with a priority ordering set. Second, the way in which child nodes are generated is different in that the parent node must be checked to ensure that no child violates the priority ordering of the parent before generating it.

5.5.5 The MAPF visualiser

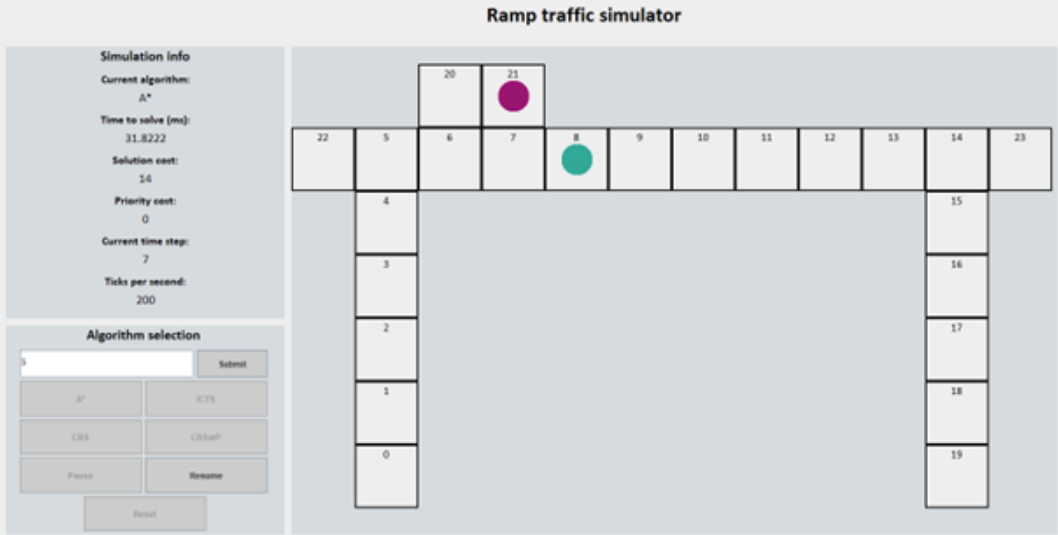

Figure 5.4: A view of the MAPF visualiser.

The MAPF visualiser was implemented as a GUI for the software. The GUI appears as soon as the algorithms have found a solution. Fig. 5.4 shows what the GUI looks like in action. To the right is a graphical representation of the ramp. On the top left is a statistics panel showing information relevant to the current scenario and solution being displayed. On the bottom left is a control panel with which the user interacts and controls the visualiser. The user is able to select the rate at which the solution is displayed, and also select what algorithm's solution to show.

The solution is displayed using a timer, the tick length of which is decided by the user. The timer can be paused and resumed by having the user click the corresponding buttons.

The ramp is painted on a JPanel container. Each vertex (ramp segment) represents a separate VertexPanel object. They are dynamical in size depending on the length of the ramp to ensure that the whole ramp always fits in the JPanel. As an agent moves to occupy a new vertex, its VertexPanel object is prompted to paint a circle, signalling to the user that the agent has moved. As a solution has been displayed, the user has the option to reset the ramp, which clears the agents (circles) from the ramp.

As mentioned in the previous chapter, the Java Swing framework was used to build the graphical components. Swing was mainly used due to previous experience with it.

Finally, fig. 5.5 shows the complete class diagram of the system, where the visualiser class is also included.

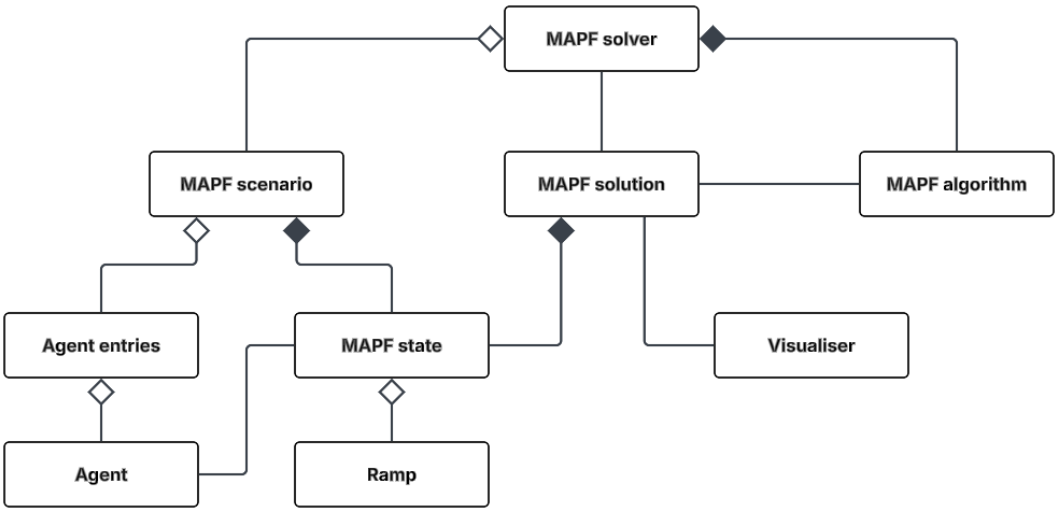

Figure 5.5: A complete class diagram of the system.

Chapter 6
Experiments and results

In this chapter, the series of conducted experiments and their results are presented. Each experiment focuses on certain aspects of the mining ramp scenario. All numbers were obtained from doing cold starts, i.e. starting the program from new. This was done to better mimic real-world circumstances, since running the same scenario multiple times in a loop went much faster than running the scenario multiple times between program restarts. Furthermore, all numbers were obtained by calculating the average from five runs, unless stated otherwise. A run was automatically aborted if no solution was obtained within two minutes of running. This decision was mainly made to make the experiments timely feasible, but also since an efficient traffic coordination demands that a solution is found relatively fast. Finally, unless explicitly mentioned otherwise, the experiments were set up such that every other agent went in the down-going direction, and every other agent in the up-going direction.

6.1 Experiment A: 5x5 grid

Before any other experiment, the algorithms were tested on a 5x5 grid scenario. Each agent could move in any non-diagonal direction or stay still, thus amounting to a maximum of five possible actions. The purpose of this experiment was to see how the algorithms would behave in less constraining circumstances, and furthermore in a scenario often seen in other works. The agents' source vertices were randomly decided, but they shared a fixed target vertex.

AVERAGE RUN-TIME (MS)

K	A*	ICTS	CBS	CBSw/P
2	4,8	12,8	31,6	3,4
3	23	12,2	29,9	4
4	4617*	438,4	45,3	11,9
5	1280*	257,6	52,8	14,8
6	11514	19376*	77,1	19,5
7	NA	10266*	171,5	34,1

Table 6.1: The average run-times on a 5x5 grid using the different algorithms over ten instances. k indicates agent count. *A solution was not found in all instances.

Table 6.1 presents the average run time for each algorithm over ten instances, where k indicates the number of agents. NA indicates that no result was obtained within two min-

utes. As the table shows, the average run-time varies quite a lot and does not necessarily follow a pattern all the time. This can be explained by the fact that for each run, a different agent starting location configuration is used, due to the starting locations being randomly decided. Some configurations therefore can result in more collisions than others, hence the disparity in run-times. However, the purpose of experiment A was mainly to see how the algorithms' performances scale with an increased agent count, rather than to focus on the numbers themselves.

A* suffers the most from an increased agent count, followed by ICTS. Since the agent starting locations differ, in many of the cases, A* and ICTS are unable to always generate a solution within two minutes. With seven agents, A* never generates a solution within the time limit. CBS and CBSw/P handles the scenarios the best. Towards the end of the table, the CBS run-times seem to begin to take off, with CBSw/P still managing to find solutions fast.

6.2 Experiment B: Ramp length and agent cost

Experiment B sought to investigate the impact of the ramp length and agent count on the performance of the algorithms. For this reason, the ramp was devoid of passing bays to keep the scenario simple.

Table 6.2 and table 6.3 show the average run-times using ICTS and CBS, respectively. Even with a very short ramp, they quickly begin to struggle as the agent count increases. Interesting to note is that ICTS handles an increasing ramp length much better than CBS. As the length increases, ICTS' performance suffers very little. Even with a length of 20, it manages to quickly find a solution, given that the agent count is low. CBS, however, suffers heavily from both an increased length, as well as increased agent count.

Table 6.4 shows that A* is similar to ICTS in that it handles an increasing ramp length quite well, given a low agent count. Nonetheless, A* can handle a much higher agent count, up to 19 concurrent agents with the shortest ramp length tested. Best of all algorithms is CBSw/P, as table 6.5 clearly shows. Even with the maximally tested ramp length and agent count, CBSw/P finds a solution in only 45 seconds. It is worth to note that for low agent counts, A* finds solutions faster than CBSw/P. However, the run-time of A* scales much faster, hence why it fails to find solutions for higher agent counts, especially as the ramp length increases.

In terms of total costs and waiting costs, where the algorithms managed to find a solution, the solutions were always of equal quality. The solution costs and waiting costs are presented in table 6.6 and table 6.7, respectively.

ICTS AVERAGE RUN-TIME (MS)

LENGTH / AGENTS	4	6	8	10	12	14	16	18	20
2	24	42	77	107	173	204	353	368	511
3	82	126	360	692	1307	2114	3185	5044	7785
4	847	4312	25363	115534	NA	NA	NA	NA	NA
5	16658	NA	NA	NA					
6	NA								

Table 6.2: The average run-times over five instances using ICTS, with differing ramp lengths and agent counts. Further attempts for a ramp length were aborted as soon as a solution was not found.

CBS AVERAGE RUN-TIME (MS)

LENGTH / AGENTS	4	6	8	10	12	14	16	18	20
2	28	98	529	1461	7397	86826	NA	NA	NA
3	67	163	654	2528	20473	NA			
4	631	49800	NA	NA	NA				
5	1777	NA							
6	NA								

Table 6.3: The average run-times over five instances using CBS, with differing ramp lengths and agent counts. Further attempts for a ramp length were aborted as soon as a solution was not found.

A* AVERAGE RUN-TIME (MS)

LENGTH / AGENTS	4	6	8	10	12	14	16	18	20
2	11	7	40	28	50	40	63	146	170
3	13	12	27	34	111	171	337	641	1216
4	22	49	94	245	1414	4989	16095	54418	NA
5	22	46	117	708	3643	19868	97849	NA	
6	61	85	340	3249	29035	NA	NA		
7	94	127	517	4945	62052				
8	149	189	1317	13604	NA				
9	226	272	1523	16484					
10	407	763	2696	33476					
11	547	1005	3362	40168					
12	962	1748	7570	89690					
13	1406	2482	11366	119388					
14	2386	5830	32938	NA					
15	4016	10342	45569						
16	9337	23970	116427						
17	18224	46867	NA						
18	45836	115835							
19	106053	NA							
20	NA								

Table 6.4: The average run-times over five instances using A*, with differing ramp lengths and agent counts. Further attempts for a ramp length were aborted as soon as a solution was not found.

CBSW/P AVERAGE RUN-TIME (MS)

LENGTH / AGENTS	4	6	8	10	12	14	16	18	20
2	20	38	63	86	109	133	148	158	221
3	21	44	63	98	203	135	175	235	184
4	42	113	161	159	199	215	266	321	388
5	64	107	132	193	199	231	334	398	324
6	102	228	203	278	300	441	506	552	659
7	153	181	227	381	297	383	553	605	612
8	160	267	301	432	550	702	908	1096	1499
9	243	297	453	431	593	842	1119	1336	1461
10	470	417	537	760	1130	1276	1770	2089	2115
11	779	670	891	998	1157	1504	1996	2156	2637
12	904	919	1224	1282	1626	1995	2443	3453	3622
13	942	1142	1288	1435	1688	2382	2947	3810	4702
14	1683	1629	1757	2217	2372	3322	4305	5883	7183
15	2664	1996	1893	2688	2899	3898	5331	6811	7687
16	4080	3502	3317	4099	4430	5940	8459	10190	11894
17	6067	4948	3953	4979	5560	6773	9881	11950	13362
18	13695	9166	7360	8095	8881	11354	14529	17900	21874
19	27252	12362	10313	10779	11856	14039	17777	19242	24785
20	60495	27881	20141	18018	21037	24625	30639	35953	45087

Table 6.5: The average run-times over five instances using CBSw/P, with differing ramp lengths and agent counts. Further attempts for a ramp length were aborted as soon as a solution was not found.

SOLUTION COST (TIME STEPS) FOR THE ALGORITHMS

LENGTH / AGENTS	4	6	8	10	12	14	16	18	20
2	14	20	26	32	38	44	50	56	62
3	21	29	37	45	53	61	69	77	85
4	32	44	56	68	80	92	104	116	128
5	41	55	69	83	97	111	125	139	153
6	54	72	90	108	126	144	162	180	198
7	65	85	105	125	145	165	185	205	225
8	80	104	128	152	176	200	224	248	272
9	93	119	145	171	197	223	249	275	301
10	110	140	170	200	230	260	290	320	350
11	125	157	189	221	253	285	317	349	381
12	144	180	216	252	288	324	360	396	432
13	161	199	237	275	313	351	389	427	465
14	182	224	266	308	350	392	434	476	518
15	201	245	289	333	377	421	465	509	553
16	224	272	320	368	416	464	512	560	608
17	245	295	345	395	445	495	545	595	645
18	270	324	378	432	486	540	594	648	702
19	293	349	405	461	517	573	629	685	741
20	320	380	440	500	560	620	680	740	800

Table 6.6: The solution costs for the algorithms. The cost is the sum of all agent actions from source to target vertex, including the wait action. Note that not all costs necessarily apply to all algorithms, since not all algorithms manage to find solutions for every scenario configuration. Costs outside the orange border could not be obtained with CBS. Costs outside the blue border could not be obtained with ICTS. Costs outside the green border could not be obtained with A*.

WAITING COST (TIME STEPS) FOR THE ALGORITHMS

LENGTH / AGENTS	4	6	8	10	12	14	16	18	20
2	4	6	8	10	12	14	16	18	20
3	5	7	9	11	13	15	17	19	21
4	10	14	18	22	26	30	34	38	42
5	12	16	20	24	28	32	36	40	44
6	18	24	30	36	42	48	54	60	66
7	21	27	33	39	45	51	57	63	69
8	28	36	44	52	60	68	76	84	92
9	32	40	48	56	64	72	80	88	96
10	40	50	60	70	80	90	100	110	120
11	45	55	65	75	85	95	105	115	125
12	54	66	78	90	102	114	126	138	150
13	60	72	84	96	108	120	132	144	156
14	70	84	98	112	126	140	154	168	182
15	77	91	105	119	133	147	161	175	189
16	88	104	120	136	152	168	184	200	216
17	96	112	128	144	460	176	192	208	224
18	108	126	144	162	180	198	216	234	252
19	117	135	153	171	189	207	225	243	261
20	130	150	170	190	210	230	250	270	290

Table 6.7: The waiting costs for the algorithms. The waiting cost is the sum of all agents' wait actions. Note that not all costs necessarily apply to all algorithms, since not all algorithms manage to find solutions for every scenario configuration. Costs outside the orange border could not be obtained with CBS. Costs outside the blue border could not be obtained with ICTS. Costs outside the green border could not be obtained with A*.

The tables thus far have shown the absolute run-times from the different cases. Fig. 6.1A and fig. 6.1B show how the algorithm run-times scale with increasing ramp lengths (when the agent count is four) and agent counts (when the ramp length is six), respectively. Fig. 6.1A reveals that the CBS run-time is most affected by the ramp length, followed by ICTS and later A*. CBSw/P, however, as is more clearly shown in fig. 6.2 has its run-time increasing roughly linearly with the ramp length. Fig. 6.1B shows similar patterns to fig. 6.1A where CBS and ICTS are majorly affected by an increased agent count. The A* run-time shoots off as the agent count increases further. Whereas CBSw/P was linearly affected by the ramp length, its run-time increases exponentially with an increased agent count, like the other algorithms.

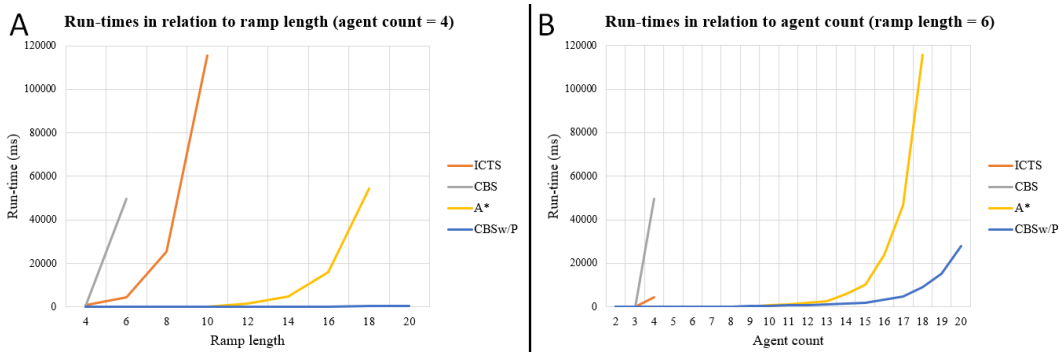

Figure 6.1: (**A**) How the algorithm run-times are affected by the ramp length, where the initial agent count is four. (**B**) How the algorithm run-times are affected by the initial agent count, where the ramp length is six. In both (**A**) and **B**, the graphs end prematurely if no solutions were found as the ramp length and agent count, respectively, increased further.

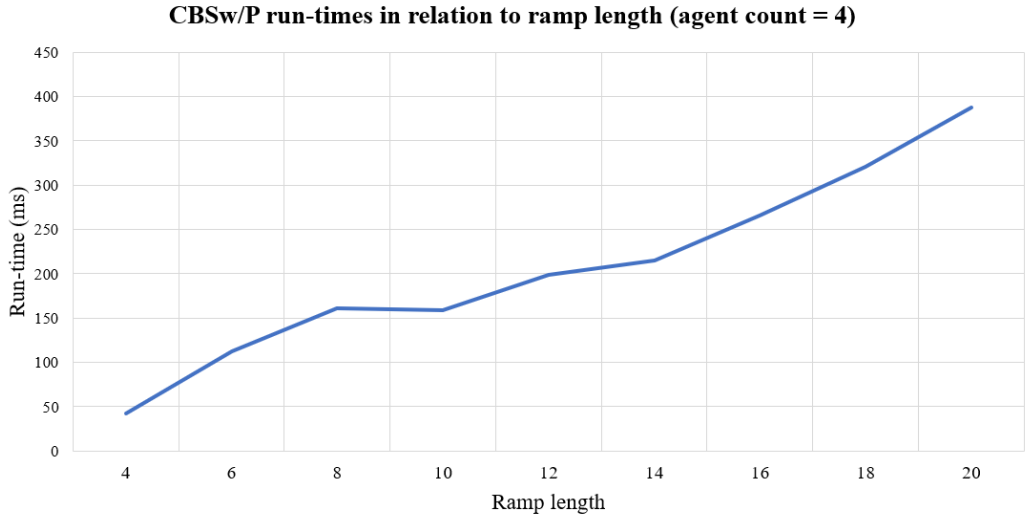

Figure 6.2: A clearer view of how the CBSw/P run-times are affected by the ramp-length, where the initial agent count is four.

6.3 Experiment C: Passing bays

Experiment B used a very simple ramp structure. To investigate the effects of passing bays, both in terms of managing to find a solution, and their quality, experiment C was about introducing passing bays in varying locations throughout the ramp. The ramp throughout all runs was of length ten. This length was chosen based on the results from experiment A, in such a way that every algorithm would be able to find solutions with the lowest agent counts, without the ramp being unrealistically short. Three different passing bay distributions were tested: (1) clustered close to the surface, (2) clustered around the middle, and (3) evenly spread. For the clustered distributions, both one, two and three passing bays were tested. For the spread distribution, two and three passing bays were tested. Fig. 6.3 shows the ramp structures with the different passing bay distributions.

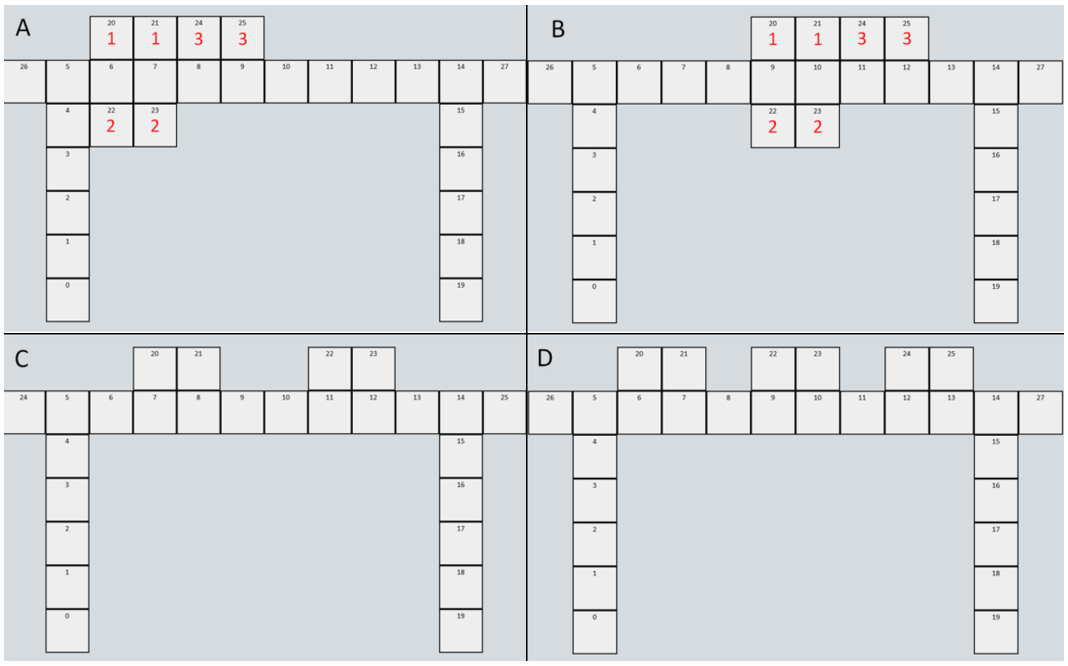

Figure 6.3: The different passing bay distributions used in experiment C. (A) Passing bays clustered close to the surface, with numbers indicating in what order they were added in the experiments. (B) Passing bays clustered in the middle, with numbers indicating in what order they were added in the experiments. (C) Two evenly distributed passing bays. (D) Three evenly distributed passing bays.

The average run-times are presented in tables 6.8 to 6.11 for ICTS, A*, CBS and CBSw/P, respectively. None of the passing bay configurations improve the number of agents that ICTS can handle. However, with two to three passing bays clustered in the middle, the run-times show improvements. ICTS and CBS always benefit from having passing bays, at least with the tested distributions. Having them clustered in the middle enables CBS to handle five agents instead of three, where no passing bays are used. Furthermore, the more passing bays, up to the tested amount, the more CBS seems to benefit. The performance of A* is significantly impaired with passing bays. Having them clustered in the middle seems to improve run-times with low agent counts, but it then quickly exceeds the run-times seen when passing bays are absent. Furthermore, contrary to ICTS and CBS, more passing bays have a more negative effect on the performance. CBSw/P suffers very similarly to A*. The difference seems to lie in that CBSw/P prefers having the passing bays at the surface of the ramp.

ICTS AVERAGE RUN-TIME (MS)

	No passing bays	Clustered Surface			Clustered middle			Spread	
# BAYS / AGENTS	0	1	2	3	1	2	3	2	3
2	24	94	78	22	18	21	23	62	21
3	82	1617	2672	145	28	30	44	237	40
4	847	NA	55024	3228	10987	572	751	NA	1727
5	16658		NA	NA	NA	8922	2882		NA
6	NA					NA	NA		

Table 6.8: The average run-times over five instances using ICTS, with a ramp length of ten and differing passing bay distributions. Further attempts for a distribution were aborted as soon as a solution was not found.

CBS AVERAGE RUN-TIME (MS)

# BAYS / AGENTS	No passing bays	Clustered surface			Clustered middle			Spread	
	0	1	2	3	1	2	3	2	3
2	1461	258	208	18	11	14	14	51	22
3	2528	2090	2653	74	21	23	33	59	16
4	NA	NA	31571	642	13275	32	176	NA	371
5			NA	NA	79844	495	149		114047
6					NA	NA	NA		NA

Table 6.9: The average run-times over five instances using CBS, with a ramp length of ten and differing passing bay distributions. Further attempts for a distribution were aborted as soon as a solution was not found.

A* AVERAGE RUN-TIME (MS)

# BAYS / AGENTS	No passing bays	Clustered Surface			Clustered middle			Spread	
	0	1	2	3	1	2	3	2	3
2	28	15	14	4	4	4	5	10	3
3	34	91	190	24	7	5	9	30	6
4	245	548	677	310	144	35	86	1247	73
5	708	6046	30750	25891	368	122	207	5774	2799
6	3249	20648	88351	NA	8789	43737	28944	NA	25130
7	4945	NA	NA		18396	NA	NA		NA
8	13604				112576				
9	16484				NA				
10	33476								
11	40168								
12	89690								
13	119388								
14	NA								

Table 6.10: The average run-times over five instances using A*, with a ramp length of ten and differing passing bay distributions. Further attempts for a distribution were aborted as soon as a solution was not found.

CBSW/P AVERAGE RUN-TIME (MS)

# BAYS / AGENTS	No passing bays	Clustered Surface			Clustered middle			Spread	
	0	1	2	3	1	2	3	2	3
2	86	45	68	32	35	30	32	37	34
3	98	91	137	54	38	48	52	96	34
4	159	207	127	128	156	362	176	166	138
5	193	269	245	181	204	457	134	198	229
6	278	327	376	539	531	1431	756	558	472
7	381	518	846	729	751	2048	1477	708	1507
8	432	964	1019	1234	1977	6780	7330	2348	2960
9	431	1644	2114	2870	2432	16227	22123	2780	8236
10	760	2307	2644	5132	5343	83625	NA	10028	23390
11	998	4351	4820	25887	10928	NA		18284	NA
12	1282	6847	8422	38446	33581			111965	
13	1435	14439	18477	NA	79921			NA	
14	2217	25593	36611		NA				
15	2688	60230	101583						
16	4099	NA	NA						
17	4979								
18	8095								
19	10779								
20	18018								

Table 6.11: The average run-times over five instances using CBSw/P, with a ramp length of ten and differing passing bay distributions. Further attempts for a distribution were aborted as soon as a solution was not found.

Tables 6.12 and 6.13 present the solution and waiting costs, respectively, for ICTS, CBS and A*. They clearly show that the presence of passing bays allows for the algorithms to generate cheaper solutions, both in terms of the total solution cost and the time spent waiting. For the agent counts where the algorithms managed to find solutions, the costs were always equal amongst the optimal algorithms (ICTS, CBS, and A*), with the only exception being the scenario with three passing bays clustered in the middle. In this scenario, ICTS finds a solution where no agent waits, whereas CBS and A* finds a solution with a waiting cost of two. CBSw/P manages, for the most part, to find solutions of equal costs as the other algorithms, as seen in tables 6.14 and 6.15. The only observable exceptions are where two passing bays clustered in the middle are used, with an agent count between four and six.

SOLUTION COST (TIME STEPS) FOR ICTS, CBS AND A*

# BAYS / AGENTS	No passing bays	Clustered Surface			Clustered middle			Spread	
	0	1	2	3	1	2	3	2	3
2	32	28	28	24	24	24	24	26	24
3	45	45	45	39	36	36	36	39	36
4	68	63	60	55	58	52	52	61	54
5	83	83	82	74	72	68	66	76	68
6	108	102	98	NA	96	92	86	NA	91
7	125	NA	NA		113	NA	NA		NA
8	152				139				
9	171				NA				
10	200								
11	221								
12	252								
13	275								
14	NA								

Table 6.12: The solution costs for ICTS, CBS and A*, with passing bays. The cost is the sum of all agent actions from source to target vertex, including the wait action. Note that not all costs necessarily apply to all algorithms, since not all algorithms manage to find solutions for every scenario configuration. Costs outside the orange border could not be obtained with CBS. Costs outside the blue border could not be obtained with ICTS. All costs could be obtained with A*.

WAITING COST (TIME STEPS) FOR ICTS, CBS AND A*

# BAYS / AGENTS	No passing bays	Clustered Surface			Clustered middle			Spread	
	0	1	2	3	1	2	3	2	3
2	10	4	4	0	0	0	0	2	0
3	11	11	11	1	0	0	0	3	0
4	22	15	10	5	10	2	2*	13	4
5	24	24	19	9	11	5	3	15	5
6	36	28	22	NA	22	16	8	NA	15
7	39	NA	NA		25	NA	NA		NA
8	52				37				
9	56				NA				
10	70								
11	75								
12	90								
13	96								
14	NA								

Table 6.13: The waiting costs for the algorithms. The waiting cost is the sum of all agents' wait actions. Note that not all costs necessarily apply to all algorithms, since not all algorithms manage to find solutions for every scenario configuration. Costs outside the orange border could not be obtained with CBS. Costs outside the blue border could not be obtained with ICTS. Costs outside the green border could not be obtained with A*. * ICTS found a solution with a waiting cost of 0.

CBSW/P SOLUTION COST (TIME STEPS)

# BAYS / AGENTS	No passing bays	Clustered surface			Clustered middle			Spread	
	0	1	2	3	1	2	3	2	3
2	32	28	28	24	24	24	24	26	24
3	45	45	45	39	36	36	36	39	36
4	68	63	60	55	58	58	52	61	54
5	83	83	82	74	72	72	66	76	68
6	108	102	98	90	96	96	89	100	91
7	125	125	122	113	113	113	106	117	113
8	152	145	140	131	139	139	131	143	133
9	171	171	166	156	158	158	150	162	158
10	200	192	186	176	186	186	NA	190	179
11	221	221	214	203	207	NA		211	NA
12	252	243	236	225	237			241	
13	275	274	266	NA	260			NA	
14	308	298	290		NA				
15	333	331	322						
16	368	NA	NA						

Table 6.14: The solution costs for CBSw/P, with passing bays. The cost is the sum of all agent actions from source to target vertex, including the wait action. The yellow costs differ from those from ICTS, CBS and A* (see table 6.12).

CBSW/P WAITING COST (TIME STEPS)

# BAYS / AGENTS	No passing bays	Clustered surface			Clustered middle			Spread	
	0	1	2	3	1	2	3	2	3
2	10	4	4	0	0	0	0	2	0
3	11	11	11	1	0	0	0	3	0
4	22	15	10	5	10	10	2	13	4
5	24	24	19	9	11	11	3	15	5
6	36	28	22	12	22	22	13	26	15
7	39	39	32	21	25	25	16	29	25
8	52	43	36	25	37	37	27	41	29
9	56	56	47	35	41	41	31	45	41
10	70	60	52	40	54	54	NA	58	45
11	75	71	64	51	59	NA		63	NA
12	90	79	70	57	73			77	
13	96	93	83	NA	79			NA	
14	112	100	90		NA				
15	119	115	104						
16	136	NA	NA						

Table 6.15: The waiting costs for CBSw/P, with passing bays. The waiting cost is the sum of all agents' wait actions. The yellow costs differ from those from ICTS, CBS and A* (see table 6.13).

The test runs above assumed that all agents were able to use the passing bays. As mentioned in section 3.1, some vehicles are unable to stop once they have entered the

ramp, thus being unable to use passing bays. To see how this would affect the algorithms, a scenario with five and ten initial agents, with the same passing bay distributions as before, was created, where only 50% of the agents would be able to use the passing bays. In the case of five initial agents, the first and third down-going agents, and the second up-going agent, were able to use passing bays. In the case of ten initial agents, the first, third and fifth down-going agents, and the second and fourth up-going agents were able to use passing bays. The ramp was ten vertices long.

Neither ICTS nor CBS ever managed to find a solution, hence why there are no tables for them. No significant changes were seen for A* run-times. Changing the agent ability to use passing bays did not result in A* being able to find solutions for ten initial agents. No significant changes were seen for CBSw/P either, except for when there was one passing bay in the middle, which increased the run-time with roughly two seconds. As for the solution costs, having fewer agents with the ability to use passing bays made no difference for any of the algorithms when the passing bays were clustered at the surface. However, in the cases where the passing bays were clustered in the middle, solution and waiting costs increased. As for when the passing bays were evenly spread, an increase in costs were seen for both algorithms when the agent count was five. With ten agents, however, CBSw/P saw no increased costs with fewer agents being able to use the passing bays. All run-times and costs are presented in tables 6.16 to 6.21 below.

A* AVERAGE RUN-TIME (MS)

	Clustered Surface			Clustered middle			Spread	
# BAYS / AGENTS	1	2	3	1	2	3	2	3
2	6673	27076	18276	321	1552	1637	7345	5746
3	NA	NA	NA	NA	NA	NA	NA	NA

Table 6.16: The average run-times over five instances using A*, with a ramp length of ten and differing passing bay distributions. Roughly 50% of the agents were unable to use passing bays. Further attempts for a distribution were aborted as soon as a solution was not found.

CBSW/P AVERAGE RUN-TIME (MS)

	Clustered Surface			Clustered middle			Spread	
# BAYS / AGENTS	1	2	3	1	2	3	2	3
2	252	286	195	2177	424	167	265	303
3	4067	3065	4432	3515	4134	5460	2511	4515

Table 6.17: The average run-times over five instances using CBSw/P, with a ramp length of ten and differing passing bay distributions. Roughly 50% of the agents were unable to use passing bays. Further attempts for a distribution were aborted as soon as a solution was not found.

A* SOLUTION COSTS (TIME STEPS)

# BAYS / AGENTS	Clustered Surface			Clustered middle			Spread	
	1	2	3	1	2	3	2	3
2	83	82	74	81	75	75	83	76
3	NA	NA	NA	NA	NA	NA	NA	NA

Table 6.18: The solution costs for A*, with passing bays, where roughly 50% of the agents were unable to use passing bays. The cost is the sum of all agent actions from source to target vertex, including the wait action.

A* WAITING COSTS (TIME STEPS)

# BAYS / AGENTS	Clustered Surface			Clustered middle			Spread	
	1	2	3	1	2	3	2	3
2	24	19	9	20	12	12	24	13
3	NA	NA	NA	NA	NA	NA	NA	NA

Table 6.19: The waiting costs for A*, with passing bays, where roughly 50% of the agents were unable to use passing bays. The waiting cost is the sum of all agents' wait actions.

CBSW/P SOLUTION COSTS (TIME STEPS)

# BAYS / AGENTS	Clustered Surface			Clustered middle			Spread	
	1	2	3	1	2	3	2	3
2	83	82	74	81	81	81	83	76
3	192	186	176	186	186	186	190	179

Table 6.20: The solution costs for CBSw/P, with passing bays, where roughly 50% of the agents were unable to use passing bays. The cost is the sum of all agent actions from source to target vertex, including the wait action.

CBSW/P WAITING COSTS (TIME STEPS)

# BAYS / AGENTS	Clustered Surface			Clustered middle			Spread	
	1	2	3	1	2	3	2	3
2	24	19	9	20	20	20	24	13
3	60	52	40	54	54	54	58	45

Table 6.21: The waiting costs for CBSw/P, with passing bays, where roughly 50% of the agents were unable to use passing bays. The waiting cost is the sum of all agents' wait actions.

6.4 Experiment D: Replanning

Experiment D sought to investigate how well the algorithms are able to replan as new agents arrive to the ramp. The ramp used throughout the experiment was one of length

ten, as in experiment C, with one passing bay in the middle (see fig. 6.3B). This ramp structure was chosen as it is neither too long nor too short, whilst allowing all algorithms to at least find a result given a low agent count. To keep the experiments feasible, all scenarios included three agents arriving throughout the scenario. Four different online scenarios were designed: (1) early burst – agents arrive early within a short window; (2) middle burst – agents arrive in the middle of the scenario within a short window; (3) late burst – agents arrive late within a short window; and (4) evenly spread – agents arrive in regular intervals. The exact arrival times of the agents was partly randomised in the burst scenarios. In the early burst, the three agents were able to arrive anywhere within time step one to three. In the middle burst, they were able to arrive anywhere within time step five to eight. In the late burst, they were able to arrive anywhere within time step ten to thirteen. Consequently, depending on the agents' arrival times, different runs of the same burst can give vastly different run-times. Thus, although the run-times in tables 6.22 to 6.25 show the average results from five separate instances, the individual results may differ. The evenly spread scenario, on the other hand, had agents arrive at time steps one, five and nine, and therefore had consistent run-times.

Tables 6.22 to 6.25 show the run-times for ICTS, CBS, A* and CBSw/P, respectively. The run-times represent the total run-time, including both the initial run of the algorithm as well as the replanning runs. In one case, the total run-time exceeds two minutes (see table 6.25). This run-time is valid since no individual solution exceeded the two-minute limit.

For both ICTS and CBS, agents arriving later affect the performance negatively. The algorithms do not manage to handle the online scenario better than the offline scenario, in any of the test scenarios. Having the agents arrive late in the scenario seems to have the least negative impact on the performance of the algorithms. A* handles replanning quite well. Worth to note is that it handles early arriving agents much better than ICTS and CBS. Nonetheless, A* does not manage to handle more agents in any of the online scenarios than in the offline scenario. CBSw/P also suffers from having to replan, but it still performs better than any of the other algorithms in terms of handling higher agent counts, regardless of scenario.

ICTS AVERAGE RUN-TIME (MS)

AGENTS	OFFLINE	EARLY BURST	MIDDLE BURST	LATE BURST	SPREAD
2	18	11462	14049	299	579
3	28	NA	NA	445	2976
4	10987			58640	NA
5	NA			NA	

Table 6.22: The average run-times over five instances using ICTS, with a ramp length of ten and one passing bay in the middle. The run-times are the sum of all individual run-times, including both the initial offline scenario as well as the replans. The columns refer to the different patterns of agent arrivals tested. The offline column does not have agents arriving after the start. Further attempts for a pattern were aborted as soon as a solution was not found.

CBS AVERAGE RUN-TIME (MS)

AGENTS	OFFLINE	EARLY BURST	MIDDLE BURST	LATE BURST	SPREAD
2	11	50	51	37	109
3	21	NA	6490	29	108
4	13275		15634*	14407	62903
5	79844		115041*	93595	NA
6	NA		NA	NA	

Table 6.23: The average run-times over five instances using CBS, with a ramp length of ten and one passing bay in the middle. The run-times are the sum of all individual run-times, including both the initial offline scenario as well as the replans. The columns refer to the different patterns of agent arrivals tested. The offline column does not have agents arriving after the start. Further attempts for a pattern were aborted as soon as a solution was not found. *Only recorded in one of the instances, where the remaining failed to find a solution.

A* AVERAGE RUN-TIME (MS)

AGENTS	OFFLINE	EARLY BURST	MIDDLE BURST	LATE BURST	SPREAD
2	4	33	19	21	29
3	7	11	31	14	16
4	144	197	273	337	181
5	368	356	341	288	306
6	8789	9067	8614	8766	8737
7	18396	18230	16705	15172	16120
8	112576	100101	NA	101431	106320
9	NA	NA		NA	NA

Table 6.24: The average run-times over five instances using A*, with a ramp length of ten and one passing bay in the middle. The run-times are the sum of all individual run-times, including both the initial offline scenario as well as the replans. The columns refer to the different patterns of agent arrivals tested. The offline column does not have agents arriving after the start. Further attempts for a pattern were aborted as soon as a solution was not found.

CBSW/P AVERAGE RUN-TIME (MS)

AGENTS	OFFLINE	EARLY BURST	MIDDLE BURST	LATE BURST	SPREAD
2	35	139	106	116	29
3	38	156	214	132	113
4	156	455	337	254	418
5	204	421	1141	495	373
6	531	1234	581	536	919
7	751	1180	888	942	1424
8	1977	2021	2197	2148	1939
9	2432	4228	4676	2601	3043
10	5343	10270	6101	6225	8611
11	10928	22374	11188	12490	16014
12	33581	NA	38537	36399	50930
13	79921		87733	89431	137311*
14	NA		NA	NA	NA

Table 6.25: The average run-times over five instances using CBSw/P, with a ramp length of ten and one passing bay in the middle. The run-times are the sum of all individual run-times, including both the initial offline scenario as well as the replans. The columns refer to the different patterns of agent arrivals tested. The offline column does not have agents arriving after the start. Further attempts for a pattern were aborted as soon as a solution was not found. *None of the individual run-times exceeded two minutes.

Tables 6.26 to 6.29 present the solution and waiting costs for ICTS, CBS, A* and CBSw/P, respectively. Compared to previous experiments, replanning causes the algorithm solution costs, and their waiting costs, to differ more regularly from each other. Between ICTS, CBS and A*, the differences in costs are not very dramatic. Although CBSw/P occasionally finds a better solution than the others (e.g. spread agent arrival, with seven initial agents), in other cases its solutions are more expensive (e.g. middle burst arrival, with two to four initial agents).

ICTS SOLUTION AND WAITING COSTS

AGENTS	OFFLINE		EARLY BURST		MIDDLE BURST		LATE BURST		SPREAD	
	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting
2	24	0	69	11	75	13	64	2	68	9
3	36	0	NA	NA	NA	NA	64	4	82	10
4	NA	NA					103	21	NA	NA
5							NA	NA		

Table 6.26: The solution and waiting costs for ICTS, with different agent arrival patterns. The solution cost is the sum of all agent actions from source to target vertex, including the wait action. The waiting cost is the sum of all agents' wait actions.

CBS SOLUTION AND WAITING COSTS

AGENTS	OFFLINE		EARLY BURST		MIDDLE BURST		LATE BURST		SPREAD	
	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting
2	24	0	69	11	72	14	64	2	68	9
3	36	0	NA	NA	90	18	77	4	82	10
4	58	10			112*	26*	103	21	106	21
5	72	11			124*	25*	112	13	NA	NA
6	NA	NA			NA	NA	NA	NA		

Table 6.27: The solution and waiting costs for CBS, with different agent arrival patterns. The solution cost is the sum of all agent actions from source to target vertex, including the wait action. The waiting cost is the sum of all agents' wait actions. *Only recorded in one of the instances, where the remaining failed to find a solution.

A* SOLUTION AND WAITING COSTS

AGENTS	OFFLINE		EARLY BURST		MIDDLE BURST		LATE BURST		SPREAD	
	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting
2	24	0	69	11	72	14	64	2	68	9
3	36	0	88	18	90	18	64	4	82	10
4	58	10	107	23	113	27	103	21	106	21
5	72	11	132	33	125	26	112	13	124	25
6	96	22	160	46	153	39	139	26	150	37
7	113	25	179	50	171	44	158	30	172	42
8	139	37	200	57	NA	NA	196	44	NA	NA
9	NA	NA	NA	NA			NA	NA		

Table 6.28: The solution and waiting costs for A*, with different agent arrival patterns. The solution cost is the sum of all agent actions from source to target vertex, including the wait action. The waiting cost is the sum of all agents' wait actions.

CBSW/P SOLUTION AND WAITING COSTS

AGENTS	OFFLINE		EARLY BURST		MIDDLE BURST		LATE BURST		SPREAD	
	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting	Cost	Waiting
2	24	0	69	11	75	13	68	10	68	9
3	36	0	88	18	95	25	81	12	82	10
4	58	10	107	23	118	34	103	21	106	21
5	72	11	132	33	125	26	112	13	124	25
6	96	22	152	41	153	39	139	26	150	37
7	113	25	179	50	173	46	158	30	170	42
8	139	37	200	57	202	62	196	55	199	56
9	158	41	231	72	221	62	209	50	221	62
10	186	54	255	79	255	80	241	65	252	77
11	207	59	284	90	274	83	264	72	276	84
12	237	73	NA	NA	307	104	306	101	309	100
13	260	79			333	107	324	97	335	108
14	NA	NA			NA	NA	NA	NA	NA	NA

Table 6.29: The solution and waiting costs for CBSw/P, with different agent arrival patterns. The solution cost is the sum of all agent actions from source to target vertex, including the wait action. The waiting cost is the sum of all agents' wait actions.

6.5 Experiment E: Priorities

Experiment E used a fixed ramp scenario. The ramp was of length ten with one passing bay in the middle, as in previous experiments (see fig. 6.3B). The schedule of agent arrivals was as follows. Initially, at scenario start, there were either three or four agents – two down-going and one up-going, or two down-going and two up-going, respectively. The up-going agents had higher priority. The rest of the scenario was very similar to the evenly spread scenario from experiment D, where one agent arrived at time step one (down-going), five (up-going, higher priority) and nine (down-going, higher priority).

As described in section 5.5.2, ICTS is unable to handle priorities since it only considers the SOC for all agents, regardless of priority status. Therefore, ICTS was excluded from experiment E.

Tables 6.30 to 6.32 present both run-times, and solution and waiting costs, both with and without prioritisation. Prioritisation significantly decreases the run-time of CBS when the initial agent count is four. The SOC amongst the higher priority agents also significantly decreases. When the initial agent count is four, the algorithms manage to find a solution where none of the higher priority agents have to wait at all. It should be noted, however, that prioritising the higher priority agents comes at the cost of increasing the overall SOC for all agents. A* seems to also benefit from prioritisation when the initial agent count is four, as the run-time is decreased. All costs are identical to those of CBS. CBSw/P is less affected in terms of run-times, seeing very small increases. Again, all costs are identical to those of CBS and A*.

CBS RUN-TIMES, SOLUTION AND WAITING COSTS

INITIAL AGENTS	No prioritisation	With prioritisation	No prioritisation Cost		No prioritisation Waiting		With prioritisation Cost		With prioritisation Waiting	
	Run-time (ms)	Run-time (ms)	Ord	Prio	Ord	Prio	Ord	Prio	Ord	Prio
3	108	1668	82	44	10	10	109	38	37	6
4	62903	1546	106	52	21	18	121	31	37	0

Table 6.30: The run-times, solution and waiting costs using CBS, with and without prioritisation. "Ord"(ordinary) refers to all agents.

A* RUN-TIMES, SOLUTION AND WAITING COSTS

INITIAL AGENTS	No prioritisation	With prioritisation	No prioritisation Cost		No prioritisation Waiting		With prioritisation Cost		With prioritisation Waiting	
	Run-time (ms)	Run-time (ms)	Ord	Prio	Ord	Prio	Ord	Prio	Ord	Prio
3	10816	16	82	44	10	10	109	38	37	6
4	181	44	106	52	21	18	121	31	37	0

Table 6.31: The run-times, solution and waiting costs using A*, with and without prioritisation. "Ord"(ordinary) refers to all agents.

CBSW/P RUN-TIMES, SOLUTION AND WAITING COSTS

INITIAL AGENTS	No prioritisation	With prioritisation	No prioritisation Cost		No prioritisation Waiting		With prioritisation Cost		With prioritisation Waiting	
	Run-time (ms)	Run-time (ms)	Ord	Prio	Ord	Prio	Ord	Prio	Ord	Prio
3	113	259	82	44	10	10	109	38	37	6
4	418	450	106	52	21	18	121	31	37	0

Table 6.32: The run-times, solution and waiting costs using CBSw/P, with and without prioritisation. "Ord"(ordinary) refers to all agents.

Chapter 7
Discussion

This chapter begins with discussing the conducted experiments and their results. The discussion then continues with touching on the general implications of the findings, the limitations of the project, and ends with an outlook on what can be done in future works.

7.1 Discussion of the experiments

7.1.1 Experiment A: 5x5 grid

Experiment A mainly served the purpose of comparing the properties of the implemented algorithms with those from previous work. Since the mining ramp is a very peculiar scenario, results from trying the algorithms on the ramp would not be comparable to what others have tried the algorithms on. A common approach is to use a grid of any length and test the algorithms on it. For example, Sharon et al [10] tested ICTS, CBS and A* on an 8x8 grid. The findings from experiment A align with theirs quite well in terms of what algorithm suffers the fastest, and what performs the best. The fact that A* suffers very quickly is expected since the number of nodes generated gets out of hand very fast. After A*, ICTS seems to be next in line to struggle, whereas CBS and CBSw/P manage to find solutions very fast. The results showing that CBSw/P is faster than CBS also was expected by observing the results from Ma et al [6], albeit the fact that they run their experiment on a 20x20 grid with up to 50 concurrent agents.

7.1.2 Experiment B: Ramp length and agent count

The results from experiment B show how well the algorithms can handle different scenario configurations where the ramp is very simple in its structure. Table 6.2 shows that ICTS works fine with a very short ramp and low agent counts. It manages to handle all ramp lengths if there is a maximum of three agents. From there, it starts to struggle. CBS struggles even more as it fails to find solutions in longer ramps even with only two agents. Interestingly, as seen in table 6.3 and fig. 6.1A, the struggle of CBS significantly increases as the ramp length increases, more so than ICTS. Looking back at the analysis of CBS (see section 2.5.3), the number of nodes generated partly depends on the number of vertices in the ramp. ICTS, on the other hand, primarily depends on the depth of the tree, which indirectly depends on the number of obstacles that increase the plan length of an agent, compared to its individually optimal plan. An increased agent count results in more obstacles for the agents, hence why it is heavily affected by it (see fig. 6.1B). The same figure also shows that CBS is significantly affected by an increased agent count. This is because more agents increase the total solution cost, which, recalling back to section 2.5.3, increases the total number of expanded low-level CT nodes. Nevertheless, the results from

ICTS and CBS indicate quite clearly that they might not be suitable in a realistic mining ramp scenario.

A* manages to find solutions for even the longest ramps if the agent count is not more than three, as seen in table 6.4. Its run-times starts to shoot off as the ramp length reaches around ten vertices. However, it is worth noting that, although A* manages to find solutions in many cases, the computational run-times are many times high. Applying this to a real-world scenario may cause issues since the traffic must wait long before a solution is found. It is also worth keeping in mind that if the agents have more possible actions, i.e. an increased branching factor, the curve in fig. 6.1B would be expected to increase at a much lower agent count.

CBSw/P, as seen in table 6.5, is the clear winner in experiment B. It manages to find solutions in all tested scenarios, and in most of them, within a reasonable time. The suitability of CBSw/P in these simple scenarios becomes even more evident as its solutions, cost and waiting-wise, are of equal quality as those of the other algorithms, at least in the cases where any of the optimal algorithms manage to find a solution. Since CBSw/P cannot be compared to the other algorithms, in terms of solution costs, in the scenarios with long ramps and many agents, evaluating the quality of the CBSw/P solutions becomes more difficult. Nonetheless, one can argue that finding a solution, albeit not guaranteed to be optimal, is much better than not finding one at all.

Overall, experiment B indicates that ICTS and CBS struggle to operate on a mining ramp, whereas A* and CBSw/P handle it better, especially the latter.

7.1.3 Experiment C: Passing bays

The introduction of passing bays only had negative effects on the ICTS run-time performance as seen in table 6.8. One might have expected the opposite since passing bays allow for the agent plan lengths to be closer to those of their individual optimal plans. The problem, however, is that agents might still have to wait in queue, especially when the agent count increases. In other words, the presence of passing bays do not guarantee agents not having to wait in queue. Because ICTS is primarily hurt by agents having to wait, since every time step spent waiting creates a one-time-step deviation from its individual optimal plan, the ICT becomes very large very quickly. This is confirmed by the fact that ICTS benefits from having more passing bays, as table 6.8 shows. Having more passing bays means less agents having to wait in queue, which in turn results in the ICT finding a goal node much faster. Thus, having more passing bays in the experiments would most likely benefit ICTS even further.

Unlike ICTS, CBS generally benefits from having passing bays in the ramp (see table 6.9). This result is expected since passing bays, by their definition, are a way to avoid conflicts along the ramp, which in turn allows a goal node to be found much faster. Like ICTS, CBS seems to benefit more whenever the passing bay count increases. It is worth to mention that passing bays do not make everything better, as is apparent in the A* and CBSw/P results. This is because every passing bay introduces a heap of new branches to search along. The main culprit here is the fact that an agent technically can wait indefinitely inside a passing bay, resulting in many more new possibilities for each time step explored. However, in the case of CBS, the benefits of passing bays outweigh the negatives.

For A*, however, things look dramatically different. As discussed, passing bays introduce a lot of new agent-location possibilities, resulting in A* having to generate more nodes. Recalling the A* analysis from section 2.5.1, the number of generated A* nodes are affected by both agent count and the agent branching factor. Passing bays mainly affect A* by increasing the agent branching factor, hence the negative effects of passing bays on A* seen in table 6.10.

Like A*, the run-time of CBSw/P does not benefit from having passing bays. Some would expect CBSw/P be affected similarly to CBS since they share many common properties. It is difficult to pin-point the exact reasons for these results, but one probable factor is how CBSw/P searches the CT. Since it does not generate two child nodes upon detecting

a conflict, the number of conflicts does not affect the size of the CT as it does in classical CBS. Instead, the added possibilities having to be considered, brought by the presence of the passing bays, outweighs the conflict-resolving benefits. This is further backed-up by the fact that CBSw/P suffers even more as the number of passing bays increases.

It is also worth mentioning that the location of the passing bays has an impact. For ICTS and CBS, having the passing bays clustered in the middle is more beneficial than having them clustered at the surface, or evenly spread throughout the ramp. For ICTS, this is most likely since having passing bays in the middle results in minimal waiting for the agents since they are bound to collide at the middle of the ramp. Having a passing bay located close by thus allows for the agents to smoothly pass each other by. Similarly, for CBS, having the passing bays located near the site of conflict means that the following conflicts in the CT can be resolved much faster than if the passing bays would be located at either end of the ramp. The same goes for A* which appreciates having the passing bays clustered in the middle. This is most likely due to the fact that, the later in an agent's plan that new possibilities are introduced (such as deciding on whether to stay inside or move from a passing bay), the smaller the impact of those added possibilities, since the agent is less time steps away from its target. Conversely, if a passing bay is located at the start, the added possibilities grow very early on, and have more time steps to expand on. Having the passing bays in the middle thus maximises the total delay, all agents considered, of the arrival of the passing bay-induced possibilities. CBSw/P differs from the other algorithms in that it prefers having one passing bay close to the surface.

However, the greatest benefit of passing bays is the reduction in solution-related costs. In virtually all instances both the solution cost and waiting cost are reduced. Naturally, the waiting cost is especially reduced. However, here the optimality properties of the algorithms can be noticed, where CBSw/P in a few cases fails to return the optimal solutions, highlighted as yellow in tables 6.14 and 6.15. Note, that CBSw/P returns solutions in scenarios where the others fail. Therefore, more of the costs than those highlighted might be suboptimal.

It must be noted that the optimal algorithms do not guarantee finding solutions with optimal waiting times. An example of this is seen in table 6.13 where ICTS finds a solution with equal cost but lower waiting cost than CBS and A*. The reason this happens is that there are multiple optimal solutions. Since the different optimal algorithms use different strategies when searching, they may find different, but cost-wise equivalent, solutions. However, because the only heuristic the algorithms use is the solution cost, they leave no guarantees regarding the waiting costs.

Finally, with fewer agents being able to use passing bays, little to no changes were observed for the run-time performances of the algorithms. However, the reduced ability to use passing bays had a negative effect on the solution qualities, especially whenever the passing bays were clustered in the middle. From the limited experiments conducted, it seems that as fewer agents are able to use the passing bays, solution costs may be affected, but only relatively little.

7.1.4 Experiment D: Replanning

Introducing replanning was expected to affect the algorithm performances negatively, since there are more agents having to be planned. It was also expected that having agents arriving later in the scenario would affect the algorithms less, since it means that the previous agents are much closer to their destinations. Thus, the number of time steps where all agents are active in the scenario is smaller, which makes finding a solution easier and faster.

From the conducted experiment, A* and CBSw/P display the greatest levels of resilience when faced with having to replan. Table 6.24 shows that A* is only negatively affected with agents arriving in burst in the middle of the scenario. CBSw/P, like ICTS and CBS, is most affected when the agents arrive in burst early. However, since the exact time steps at which the new agents arrived in the burst-arrival scenarios, and only

five instances were tested per initial agent count, not all arrival scenarios were examined. Therefore, the results may have looked slightly different if the agents would have arrived differently.

As has been previously mentioned, replanning removes to optimality property of the algorithms. By using RA to replan, snapshot optimality is ensured (amongst the normally optimal algorithms), but this only ensures that the sub-solutions are optimal [7] (see section 2.7). Experiment D demonstrates this since the solution costs amongst the different algorithms occasionally differ, even amongst the optimal algorithms. CBSw/P even manages to find better solutions than its peers at times (for instance, compare table 6.29 early burst with six initial agents to table 6.28). Again, however, since the burst-arrival scenarios were not fixed, their solutions cost are the average from the five recorded instances. The results from experiment D do not seem to show a clear winner in terms of consistently finding the cheapest solutions. However, similar to the previous experiments, A*, and especially CBSw/P, at least manage to find solutions with higher agent counts than ICTS and CBS. Since the normally optimal algorithms cannot ensure optimality in online MAPF scenarios, the use of CBSw/P may even be more warranted, especially considering that it sometimes manages to find the best solutions out of all the algorithms.

7.1.5 Experiment E: Priorities

As previously mentioned, some vehicles are in greater need of reaching their destinations, hence why priorities were implemented. The results in section 6.5 are very promising in that the costs related to the higher priority agents significantly decrease. This is especially the case for the waiting costs, which has a massive impact in how fast the agents can reach their destinations. The results therefore confirm that, if needed, some agents can travel through the ramp faster than normal. Two things must be noted, however. First, agents must enter the ramp via the queues, regardless of their priority status. Therefore, if the queues are the main bottlenecks, the priority status may have less of an impact. Thus, modifying queues to be able to also prioritise higher priority agents would bring down the costs even more. Second, although the higher priority agent costs are reduced, the overall solution costs are increased, as seen in the tables in section 6.5. Before enforcing priorities, one must therefore carefully consider both the benefits and the costs that prioritisation brings.

Worthy mentioning is that experiment E was not as exhaustive as the other experiments. The findings for the tested agent counts may not necessarily hold when the agent count increases further (or when the ramp length is different).

7.2 Social, economic, and ethical implications

Effective traffic coordination is essential in minimising the costs associated with mining ramp transport. Cost is not only measured in time, but also by the resources required to transport the heavy vehicles. To reduce the solution costs, but also the waiting costs (since waiting is a complete waste of resources, strictly speaking), is therefore paramount from both an economical and environmental perspective.

Something that has not been discussed thus far is how realistic the generated traffic plans are in practice. Questions such as what happens if a manually operated vehicle deviates from its plan must be considered and addressed to avoid potential collisions or other problems. The solutions provided by the algorithms assume total vehicle compliance. In some cases, however, executing a given plan might be difficult. Consider, for example, the scenario in fig. 7.1 where a vehicle enters a passing bay just as a vehicle inside the passing bay leaves. At the same time, a third vehicle drives by the passing bay. In this scenario, there are three instances of following conflicts. Albeit allowed in the mining ramp scenario, they might be difficult to execute in practice. Such difficult-to-execute plans may pose a risk for the involved vehicles and drivers since they leave small margins of error. In

order for a traffic coordinator to be sound in regards to both safety and ethics, such plans may need to be addressed.

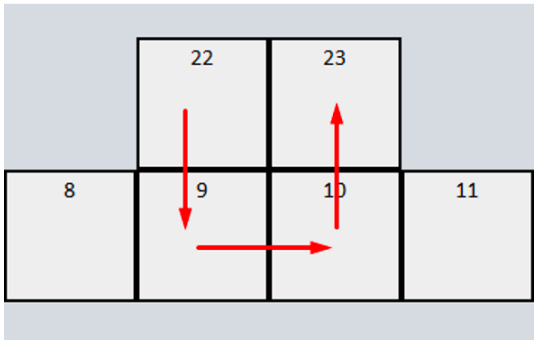

Figure 7.1: A scenario involving three simultaneous following conflicts.

The matter becomes even more complicated if the vehicles are autonomous since compliance with the traffic plans becomes less predictable. If an autonomous vehicle deviates from its plan the consequences might be worse than if the vehicles are manually operated. To conclude, the aspect of traffic safety must be thoroughly considered and addressed before relying on autonomous traffic coordinators to make the decisions.

7.3 Limitations of the thesis

There are several limitations to this project that are important to acknowledge. First and foremost, the mining ramp scenario used in this thesis is a gross simplification of the real world. As mentioned, both time and space are discretised, all vehicles travel in constant speed where momentum is disregarded, all vehicles are assumed to perfectly adhere to the plans assigned to them, etc. Strictly speaking, every aspect of the scenario that is simplified can be considered a limitation.

Another limitation concerns the implementation of the algorithms. This thesis focused primarily on implementing the algorithms, and not as much on optimisation. The implementations could therefore be further optimised to achieve better run-time performances. Moreover, since the mining ramp scenario is very predictable in that agent behaviour can be predicted, the algorithms could be implemented in a way where they are tailored specifically for this scenario. For instance, if the ramp has no passing bays, one would not have to manually simulate through every time step, since, with the agents' initial positions, their imminent collision, and the location of it, can be predicted.

Another limitation concerning the algorithm implementations is how ties are resolved when deciding what node to poll from a priority queue. This is relevant for A*, CBS and CBSw/P since they use priority queues. If two (or more) equivalent nodes can be chosen from, the implementations made in this thesis does not provide the algorithms with any guidance. Instead, any of the nodes can be polled. One approach to handle tie-breaking is by using a so-called conflict avoidance table (CAT) as described by Standley [11]. The CAT contains entries of all agent locations at every time step. If two (or more) nodes have identical f values, instead of first resorting to comparing their h values, one can instead begin by comparing how much the nodes violate the CAT. This is done by having each node track the number of times the plan leading up to that node has violated the CAT. By favouring the node with the fewest violation, the number of future conflicts is kept minimal. This approach of breaking ties was ultimately not implemented in this project. The rationale for this decision was that, since the mining ramp is a unique scenario which vastly differs from traditional MAPF scenarios, the benefit of using the CAT approach

was not easily foreseeable. Because the benefits that the CAT approach would bring were uncertain, implementation of other features was prioritised.

Important to also note is that an algorithm returns the first solution it finds. This means that a solution, although having the lowest cost (in the case with an optimal algorithm), may have other cost-equivalent solutions, potentially more preferred in practice.

Finally, there were limitations pertaining to the experiments. This scenario entailed many different factors, making it difficult to thoroughly investigate their impacts on the algorithm performance. Thus, the experiments are not exhaustive in that all possible scenarios were tested. Additionally, the two-minute threshold at which an attempt is aborted, is a limitation since any behaviours outside of the two-minute window are not captured by the experiments. As mentioned, the motivation for this threshold was to make the experiments feasible to perform, but also to act as a requirement for the algorithms to be able to find a solution within a practically reasonable time. If traffic is controlled by a central coordinator, solutions must be distributed in a timely manner to prevent vehicles from having to wait, essentially wasting time. This is especially important in the cases where replanning is done. A replan is triggered if new agents arrive in the middle of an ongoing scenario, at which point there may already be vehicles moving on the ramp. To know their next course of actions requires the replanned solution to be distributed fast, ensuring that the moving vehicle does not potentially miss a passing bay that the new solution will tell it to use. Since vehicles must not stop whilst on the ramp, every time step spent on finding a replanned solution is a time step during which a vehicle already on the ramp keeps moving. Thus, the initial state of the scenario that the algorithm is fed with would not align with the actual real-world scenario. This is a problematic scenario that must be addressed if a replanning-capable traffic coordinator is to be applied in practice.

7.4 Future works

Since there are limitations to this thesis, it leaves much room for improvements in future works. First, any scenario modification making the scenarios more realistic is a major improvement when investigating the algorithms' suitability in practice. Second, as mentioned in the previous section, the algorithms can be tailored for a mining ramp to make them even better, primarily in terms of run-time performance.

Additionally, this thesis only focused on a handful of search-based MAPF algorithms. There exists other algorithms in the same category, many of them with various extensions for optimisation. As was briefly described in section 2.6, there are other types of algorithms with fundamentally different approaches, such as constraint-based algorithms. To investigate such algorithms' suitability for a mining ramp scenario is more than warranted.

Making use of a tie-breaking mechanism may have an impact. As mentioned, this was not explored in this thesis, and is therefore another excellent example of what can be improved in future works.

This thesis only focused on a mining ramp scenario. There are many other candidate bottleneck settings that the algorithms in this thesis can be applied on. For instance, in a mining setting, efficient traffic is also paramount inside the mine itself.

Finally, since this thesis experimented with looking at multiple scenarios with different ramp properties, none of the experiments managed to capture their full implications on the algorithm performances. A future improvement could therefore be to focus on a certain property of the ramp and conduct more exhaustive experiments than those conducted in this thesis.

Chapter 8
Conclusions

This thesis sought to achieve multiple objectives. The first was to implement a handful of MAPF algorithms. This objective was well achieved, with the algorithms working as intended. The second objective was regarding evaluation of and comparison between the algorithm performances. Since there is a plethora of scenarios with unique ramp and agent property configurations, testing all such scenarios would be very difficult. Therefore, selecting a universal winner amongst the algorithms is not trivial. With this said, there are things from the experiments conducted that hint at what algorithms may be more or less appropriate for bottleneck scenarios.

ICTS and CBS consistently struggle with the mining ramp scenario. They are always more limited in being able to successfully generate traffic plans at all (especially as the agent count and ramp length increase), which makes them unfit for use in realistic mining ramp scenarios. Thus, the real candidates are A* and CBSw/P. CBSw/P looks very promising in scenarios where the ramp is very simple in its structure, in terms of passing bays. If no, or a limited number of, passing bays exist, CBSw/P almost always seems to be on par with A*. Moreover, in such scenarios, CBSw/P manages to find solutions with higher agent counts than A* manages to. Thus, if there is no critical requirement that a solution must always be perfectly optimal (primarily in terms of SOC and secondly in terms of waiting cost), CBSw/P would probably be the best choice. Another strength of CBSw/P is that it also manages to find its solutions faster than A*, making it perfect for situations where traffic plans must be generated fast.

However, the mining ramp scenario is rarely this simplistic, especially since it is an online scenario where new vehicles can arrive continuously. As discussed, no algorithm managed to consistently find the best solutions in an online setting. Again, however, CBSw/P is able to find solutions in scenarios where the others do not; none of the other algorithms find solutions when the initial agent count is nine or more, whereas CBSw/P, although not all the time, can handle up to thirteen initial agents. Since what makes A* a reliable algorithm, i.e. that it normally guarantees optimality, is removed in online settings, the argument for using CBSw/P becomes more convincing. However, as seen in section 6.4, A* might be better if the initial agent count is very low, assuming a limited number of later arriving agents.

In settings where there are higher priority agents, all algorithms, at least in the tested scenarios, find equivalent solutions. A* benefits immensely from having higher priority agents, as its run-time is significantly decreased. A*, therefore, is the clear winner amongst the algorithms.

The final objective was to implement a tool for visualising generated traffic plans. This objective was also well achieved since it was primarily meant to facilitate testing and validation of the software. The current visualiser gives the user agency to adjust the solution display speed, pause, resume, and monitor statistical information, which allows the user to thoroughly dissect and analyse the proposed solutions from the algorithms.

Chapter 9
Reflection

This project gave me a brilliant insight into the world of MAPF, its approaches and applications. Most of the algorithms implemented were completely new to me and had to be researched through their original papers. A* is a classical algorithm that has been introduced in our academic studies, but in such instances it has always been in single-agent settings. To expand A* to work with multiple concurrent agents was a great learning experience in how traditional algorithms and approaches can be expanded to solve problems in more complicated scenarios.

Reading studies and papers within the field of computer science was a relatively new endeavour which in the beginning was difficult to get into. This is why the first weeks were allocated solely to researching. In hindsight, I appreciate my supervisor advising me to not rush the research phase, since it paved the way for the implementation phase. Had I not spent the many hours reading, implementation would have become an issue. This is especially the case since these algorithms are not widely known on the Internet, making it difficult to find easy-to-digest information about them on platforms such as YouTube.

Weekly meetings were held together with my supervisor, Franziska Klügl. During these sessions, I had the opportunity to voice my thoughts and any concerns. These discussions were thought-provoking and inspiring, which guided me in the decisions made throughout the entire project. From start to finish, the support from my supervisor has been invaluable, and made the project even more enjoyable.

Working alone on this thesis forced me to really get an understanding of what I was implementing. I now feel confident in navigating myself within the field of MAPF, at least amongst the search-based algorithms. As mentioned in section 2.6, there are other approaches to solving MAPF scenarios. Had I worked together with a colleague on this thesis, perhaps implementing some completely different algorithms would have been possible as well. Working alone therefore had both its positives and its negatives.

Overall, I feel like this project allowed me to take a great level of responsibility over my own work which I welcomed from day one. Being able to, in collaboration with my supervisor, set up a working plan and striving towards adhering to it has been very rewarding. I have gained great respect for holding deadlines and not underestimating how much time different parts of a project need to take.

Last but not least, I want to extend my gratitude to my aforementioned supervisor, Franziska Klügl, for the guidance and support through the process of writing this thesis.

References

[1] K. Dresner and P. Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31:591–656, March 2008. (Cited on page 6.)

[2] Esra Erdem, Doga Kisa, Umut Oztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1):290–296, June 2013. (Cited on page 12.)

[3] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. (Cited on page 8.)

[4] Florence Ho, Ana Salta, Ruben Geraldes, Artur Goncalves, Marc Cavazza, and Helmut Prendinger. Multi-agent path finding for uav traffic management. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, page 131–139, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems. (Cited on page 6.)

[5] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(13):11272–11281, May 2021. (Cited on page 6.)

[6] Hang Ma, Daniel Harabor, Peter J. Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding, 2018. (Cited on pages 12 and 48.)

[7] Surynek Pavel, Felner Ariel, Stern Roni, and Boyarski Eli. *Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective*. IOS Press, 2016. (Cited on page 12.)

[8] Malcolm Ryan. Constraint-based multi-robot path planning. In *2010 IEEE International Conference on Robotics and Automation*, pages 922–928, 2010. (Cited on page 12.)

[9] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent path finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):563–569, September 2021. (Cited on pages 10, 11, and 12.)

[10] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, February 2013. (Cited on pages 9, 10, 24, and 48.)

[11] Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):173–178, July 2010. (Cited on pages 9, 23, and 52.)

[12] Roni Stern. *Multi-Agent Path Finding – An Overview*, page 96–115. Springer International Publishing, 2019. (Cited on pages 6, 7, 8, 12, 15, and 16.)

[13] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Kumar, Roman Barták, and Eli Boyarski. Multi-agent pathfinding: Definitions, variants, and benchmarks. *Proceedings of the International Symposium on Combinatorial Search*, 10(1):151–158, September 2021. (Cited on pages 6, 7, 8, and 15.)

[14] Pavel Surynek. *Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving*, page 564–576. Springer Berlin Heidelberg, 2012. (Cited on page 12.)

[15] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs, 2012. (Cited on page 12.)

[16] Örebro University. Teamrob - teams of robots working for and with humans. `https://www.oru.se/english/research/research-projects/rp/?rdb=p2388`, 2025. Accessed: May 20, 2025. (Cited on page 4.)

[17] Jiří Švancara, Marek Vlk, Roni Stern, Dor Atzmon, and Roman Barták. Online multi-agent pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):7732–7739, July 2019. (Cited on pages 8, 12, and 13.)