TU WIEN Informatics

Enhancing Meta-Agent Conflict-Based Search for the Multi-Agent Pathfinding Problem with Informed Merging and Heuristics

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Ian Ederer, BSc
Matrikelnummer 01404429

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.-Ass. Dipl.-Ing. Nikolaus Frohner, BSc

Wien, 4. Oktober 2022

Ian Ederer	Günther Raidl

TU WIEN Informatics

Enhancing Meta-Agent Conflict-Based Search for the Multi-Agent Pathfinding Problem with Informed Merging and Heuristics

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Ian Ederer, BSc
Registration Number 01404429

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Univ.-Ass. Dipl.-Ing. Nikolaus Frohner, BSc

Vienna, 4th October, 2022

Ian Ederer	Günther Raidl

Erklärung zur Verfassung der Arbeit

Ian Ederer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Oktober 2022

Ian Ederer

Danksagung

Ich möchte mich bei meinen Betreuern für die hilfreichen Ideen und das regelmäßige Feedback bedanken. Außerdem möchte ich mich auch bei meiner Familie, meinen Freunden und besonders bei meiner Freundin für ihre Unterstützung bedanken.

Kurzfassung

Das Multi-Agent Pathfinding (MAPF) Problem ist ein Planungsproblem mit dem Ziel, Pfade für mehrere Agenten zu finden. Diese Pfade ermöglichen es ihnen, gleichzeitig von ihren individuellen Startpunkten zu ihren jeweiligen Zielpunkten zu gelangen. Die Pfade müssen frei von Konflikten sein, was bedeutet, dass Agenten nicht zur selben Zeit am selben Ort sein dürfen. Der Optimierungsteil in der von uns betrachteten Variante des Problems besteht darin, die Summe der Längen aller Pfade zu minimieren. Computerspiele, Verkehrsregelung mit selbstfahrenden Fahrzeugen, und automatische Roboter in Lagerhallen gehören zu den Anwendungsfällen des MAPF Problems, da hier konfliktfreie Pfade für mehrere Agenten gefunden werden müssen.

Das Ziel dieser Diplomarbeit ist es, den Algorithmus Conflict-Based Search (CBS) zu verbessern. Dieser Algorithmus ist konzipiert um Instanzen des MAPF Problems, durch schrittweises Auflösen von Konflikten in einer Best-First Suche, optimal zu lösen. Unser Fokus liegt im Besonderen auf der Variante von CBS welche Meta-Agenten bilden kann, wobei Meta-Agenten Gruppierungen von einzelnen Agenten sind, für die die Pfadsuche gemeinsam stattfindet. Wir entwickeln Strategien, mit welchen wir vorteilhafte Gruppierungen bilden können, noch bevor der Suchvorgang von CBS gestartet hat. Unser erster Ansatz ist eine Strategie die auf paarweisen Symmetrien basiert. Für den zweiten Ansatz verwenden wir Machine Learning. Wir trainieren ein Machine Learning Modell, mit dem wir vorhersagen, welche Agenten in Meta-Agenten gruppiert werden sollen, um den Suchvorgang zu beschleunigen. Unsere letzten vorgestellten Verbesserungen sind heuristische Funktionen für CBS, welche auch mit Meta-Agenten umgehen können. Das Ziel dieser Heuristiken ist, durch genaueres Abschätzen verbleibender Kosten eines Suchknotens zu einem Zielknoten die Anzahl der expandierten Knoten der High-Level Komponente von CBS zu verringern.

Nachdem unsere Ansätze Erweiterungen für einen schon bestehenden Algorithmus sind, vergleichen wir unsere Lösungen mit ihren Basisversionen in unserer Studie. Wir basieren unsere Vergleiche auf die Laufzeit der Algorithmen, die daraus folgende Erfolgsrate, und, wenn sinnvoll, auf die Anzahl der expandierten Knoten der High-Level Komponente. Für unsere Experimente verwenden wir MAPF Instanzen welche entweder auf ein 20×20 großes Raster basieren, oder auf Szenarien aus der realen Welt, welche das Online Repository[1] für MAPF Instanzen bereit stellt.

[1] https://movingai.com/benchmarks/mapf.html

Wir untersuchen die Auswirkungen unserer Ansätze in Experimenten. Mit einer auf Rechteck-Konflikten basierenden Gruppierungsstrategie erhöht sich die Erfolgsrate, der Anteil gelöster Instanzen innerhalb eines Zeitlimits, gegenüber dem Basisalgorithmus substanziell auf dem Raster ohne Hindernissen. Die Machine Learning Ansätze zur Vorhersage, ob ein Merging zweier Agenten zu geringerer Laufzeit führt, führen insgesamt zu keiner Verbesserung, da wir mit den von uns getesteten Modell-Feature Kombinationen nicht ausreichend Vorhersage-Präzision erreichen konnten. Kombinieren wir unsere auf Rechteck-Konflikten basierende Mergingalgorithmen mit einer auf Multi-valued Decision Diagrams basierenden Heuristik für Multi-Agenten, so expandiert CBS durchschnittlich weniger High-Level Knoten in den Szenarien aus der realen Welt. Zusätzlich können mit dieser Heuristik Instanzen oft schneller gelöst werden.

Abstract

The multi-agent pathfinding (MAPF) problem is a planning problem with the goal to find paths for multiple agents. These paths allow them to concurrently navigate from their individual starting points to their goal points. The paths need to be conflict free, which means the agents are not allowed to occupy the same space at the same time. The optimization aspect of the MAPF variant we focus on is to find a solution where the sum of the lengths of the paths is minimized. Computer games, traffic control with autonomous vehicles, and autonomous warehouse robots are real world scenarios where the MAPF problem is relevant, as non-conflicting paths for multiple agents need to be found.

In this thesis we aim to improve an algorithm called conflict-based search (CBS), which is designed to solve the MAPF problem optimally, by iteratively resolving conflicts in a best-first search. Especially, we focus on the variant which is able to form meta-agents, which are groups of agents for which paths are searched for in a coupled manner. We provide strategies to find advantageous assignments to form meta-agents prior to the search process of CBS. Our first approach is to provide merging strategies based on pairwise symmetries. For the second approach we use machine learning. Instead of finding the meta-agent assignments ourselves, we train a machine learning model to predict beneficial merge candidates. Our last proposed improvements are heuristic functions which are capable of handling meta-agents. These heuristics are designed to decrease the number of high level node expansions needed to find a valid solution by calculating an informed estimate of the cost of getting from a specific node to a target node.

As our approaches are designed as extensions to an already existing algorithm, we compare our solutions to the base version in a computational study. We evaluate them based on the runtimes, the resulting success rates, and where suitable on the number of expanded high level nodes. For these experiments we choose MAPF instances which are based on either four-connected grids of size 20×20, or on real world scenarios taken from the online repository[2] for MAPF instances.

We present the effects of our proposed merging strategies and heuristics in a computational study. The merging strategy based on rectangle conflicts, provides a substantial improvement of the success rate, the fraction of solved instances within a given time limit,

[2]https://movingai.com/benchmarks/mapf.html

on the empty map. The merging approaches using machine learning models to predict whether a pair of agents should be merged upfront do not increase the success rates overall, but only could slightly reduce the runtimes on empty maps. In the end, we could not achieve sufficient prediction precision. On the other hand, enhancing multi-agent CBS with heuristics based on multi-value decision diagrams together with rectangle conflict based merging algorithms substantially decreases the average number of high level node expansions needed to find valid solutions for the instances based on real world scenarios. Additionally, the additional work to calculate the heuristics often pays off and instances are solved faster.

Contents

CHAPTER 1

Introduction

The multi-agent pathfinding (MAPF) problem, described by Stern et al. [22], is a fundamental planning problem, where multiple agents have to be navigated concurrently from their individual starting points to their respective goal points in a given environment potentially including static obstacles. The paths of the agents must not conflict with each other, which means that they are not allowed to occupy the same space at the same time. It is also an optimization problem with the goal to find solutions with either optimal costs or such of high quality in suboptimal settings. The most frequently studied objectives are to minimize either the sum or the maximum of the agents' individual costs, which is often the time needed to reach the goal.

The MAPF problem is relevant in many different real world scenarios. These include computer games, traffic control with autonomous vehicles, and autonomous robots in warehouses. For these scenarios solutions need to be computed quickly, but finding the optimal solution of a MAPF problem instance becomes computationally expensive with growing map size and number of agents. For this reason, numerous different exact and suboptimal algorithms have been proposed.

One of these algorithms is conflict-based search (CBS), introduced by Sharon et al. [20], which repeatedly computes individual optimal plans for each agent using an A*-based low level solver and resolves occurring conflicts in a high level constraint tree until an optimal conflict-free plan is obtained.

Sharon et al. also propose an enhancement to CBS which is the introduction of groups of agents, so-called *meta-agents*. This enhanced version of CBS is called meta-agent CBS (MA-CBS). These meta-agents actually represent small sub-instances of the actual problem scenario where plans for a group of agents are computed jointly. Merging the right agents (e.g., highly conflicting agents) into a meta-agent can decrease the search time of the algorithm substantially, while merging too frequently or the wrong agents might have negative impact on the runtime, since the low level solving time then becomes

excessive. Further enhancements, like restarting the whole search after merging, and prioritizing specific conflicts are introduced by Boyarski et al. [1], which result in an algorithm called Improved-CBS (ICBS). Currently, techniques which decide when to form meta-agents are rather uninformed. Improvements to this decision making could therefore substantially boost the performance of ICBS.

1.1 Aim of the Work

The current ICBS merging approach works by counting conflicts between (meta-)agents up until any pair has reached a predefined threshold. The search is then stopped and restarted with this highly conflicting pair of (meta-)agents merged into one meta-agent. In particular, this turned out to be beneficial for maps with low obstacle density. The main aim of this work is to investigate ways to quickly and efficiently decide which agents should be merged into meta-agents *from the start* given a concrete problem instance to reduce the overall runtime of the search.

To this end, our goal is to study a supervised learning approach to train a corresponding machine learning model which guides the decisions as to which agents to merge upfront. The key idea is that the model learns the impact on the runtime of a specific merge when a subsequent ICBS run would be performed compared to without performing the merge. To achieve this, features about the agents and the overall problem instances have to be collected, for example the distance between a pair of agents on the map or how "close" their individual optimal paths are to each other. Other features may focus on the number of obstacles that are close to the individual optimal paths of the agents. These type of features seem relevant, since Sharon et al. [20] point out that whether a merge is beneficial could also depend on the topology and the density of the map. Different models should be compared.

Huang et al. [10] present a supervised learning approach based on an oracle to learn on which conflicts to branch during the search. For our supervised learning process also an oracle is used to create the ground truth. This oracle predicts whether forming a specific meta-agent is beneficial regarding the solving time. For this, an existing solving algorithm like ICBS can be used. We let this algorithm solve an instance two times. The first time, we let it run with a given set of agents (and also possibly meta-agents). The second time, we form a new meta-agent before the search process starts. Then, we compare the runtimes to see whether the merge was beneficial. With this data and the previously mentioned features, we can train a model. The learning process has to be made robust in the sense that runtime measurements are subject to noise. The ICBS algorithm should then be extended by an initial merging algorithm which makes use of a previously trained model to perform the initial merging decisions. We seek to compare the performance combinations of different machine learning models (e.g., neural networks, support vector machines) on various instance classes.

ICBS uses Enhanced Partial Expansion A* (EPEA*) introduced by Goldenberg et al. [7] to solve the sub-instances the meta-agents represent, which itself is a strong MAPF

solver, if the number of agents is not too large. Since CBS is known to be inefficient at resolving pairwise symmetries as described by Li et al. [15], it could be beneficial to merge agents which are part of such conflicts and leave its resolution to the low level EPEA* search.

There are also other enhancements for CBS which, to the best of our knowledge, have not been applied to ICBS with meta-agents present yet. One of these enhancements is to use heuristics to guide the search, which is described by Felner et al. in [6]. These heuristics are based on the conflicts that are present in the agents' paths. For CBS without meta-agents there are elaborate techniques that reason about the conflicts. These techniques focus on finding out whether (and by how much) resolving a conflict will increase the overall solution cost or not, and prioritize the cost-increasing conflicts to more quickly close the optimality gap by reducing the number of search nodes to be considered. We suspect that ICBS with meta-agents would benefit from such heuristic values as well. Thus we aim to investigate in a preliminary study whether and how these techniques could be adapted efficiently to the ICBS algorithm while still using the meta-agent enhancement.

Our enhanced version of the ICBS algorithm will then be tested against the original version to study the impact of the machine learning guided merging algorithm and the heuristics on the performance over various instance classes. It will also be compared with state of the art algorithms like CBS with informed heuristics (CBSH2-WDG) designed by Li et al. [13] and CBSH2-WDG with pairwise symmetry reasoning techniques (CBSH2-RTC) introduced by Li et al. [15].

1.2 Methodological Approach

The methodical approach is as follows:

- First, a literature research will be conducted. We will gather information about the MAPF problem and which settings of it exist. Then we will find out which existing solving methods exist for our chosen setting. We will also research how heuristics have been applied to CBS.

- In a preparatory computational study, we will compare ICBS with the current state of the art approaches using benchmark instances. These benchmark instances are based on four-connected grids which can contain randomly placed obstacles. The focus of the comparison will be on the runtimes the solving algorithms will take to solve these benchmark instances. We will also compare them based on success rates, which will be calculated using the runtimes of these experiments.

- Then we will design and compute features for the supervised learning approach. With a suitable oracle, which will be based on an already implemented ICBS algorithm, we compute the ground truth.

- Different classification models will be trained and compared on the previously generated data. Hyperparameter tuning will be performed to combat possible overfitting. Afterwards, we incorporate selected well-performing models to be used in a merging algorithm for ICBS.

- We will investigate how to make use of symmetry reasoning techniques for our merging algorithms and machine learning models. One such technique is rectangle reasoning [14], with which agents in rectangle conflicts can be detected.

- The next step is to add heuristics to ICBS which are able to handle the presence of meta-agents. We will use the knowledge gained about heuristics used in CBS to adapt these techniques and add them to ICBS.

- The last step will be to evaluate our improvements. We will compare our enhanced ICBS algorithm with its base version and also state of the art approaches. This will be done again by using the benchmark instances and comparing the runtimes.

1.3 Key results

We test our approaches in a computational study. Of the merging strategies based on pairwise symmetries, the strategies which merge agents in cardinal rectangle conflicts perform best. With these, we are able to improve the success rate for the empty map substantially compared to the base variants ICBS(∞) (no dynamic merging) and ICBS(25) (dynamic merging of agent pairs after 25 conflicts). On the dense map, there are no notable improvements when using our rectangle conflict strategies, since there rectangle conflicts are less likely. The merging strategies based on target conflicts are also not able to improve the success rate substantially, while with a strategy based on corridor conflicts, we experience a small improvement on the dense map on instances with more than 15 agents.

The approaches based on machine learning provide no or only marginal improvement on the tested instances. Using our models for predictions on instances for the dense and medium map can lead to a slight decrease in success rates, for the empty maps we could achieve some minor improvements in terms of runtime. This is likely due to insufficient precision of our classifiers between 80–90%. We found out later about the strength of pairwise symmetry features and believe that they allow precision to be improved in future work.

Regarding our heuristic approaches, the multi-valued decision diagram based heuristic (MAMDD) outperforms an existing lookahead method on most instances, as its heuristic calculation process is faster but provides mostly equal values. On the real-world maps, the combination of the MAMDD heuristic with a rectangle conflict based merging algorithm reduces the average number of high level nodes expanded in the solved instances and also solves them faster.

1.4 Structure of the Work

In Chapter 2 we discuss state-of-the-art approaches to solve the MAPF problem optimally. The focus is on CBS and all its enhancements. In Chapter 3 the methodologies we use for the binary classification and the heuristics are explained. Chapter 4 provides a formal description of the MAPF problem. Chapter 5 describes in detail the solution approach for the merging of agents in pairwise symmetries and the solution approach for machine learning guided merging. Also described is how heuristics are employed in the informed search. Chapter 6 contains a computational study of our solution approaches which is mostly be based on their runtimes for different benchmark instances. We conclude in Chapter 7 and discuss directions for future work.

CHAPTER 2

State of the Art

There are many different variants of the multi-agent pathfinding problem. They differ mostly in how many aspects of a real-world MAPF scenario are abstracted away. To our knowledge the most common version assumes a world where agents move in discrete time steps on a four-connected grid and have one unique goal point, while the the objective is to minimize the sum of time steps each agent needs to reach its goal node.

In the following, we briefly describe various approaches to this MAPF variant. Since our work focuses for the most part on algorithms based on CBS, we split this chapter into algorithms not based on CBS, and algorithms which are variants of CBS.

2.1 Non-CBS Approaches

Silver [21] presents different approaches of how the A* search algorithm introduced by Hart et al. [9] can be employed effectively to solve the MAPF problem in a suboptimal setting. In his most advanced approach, windowed hierarchical cooperative A*, agents' paths are planned cooperatively within a radius (the window for conflict resolution) and use optimal single agent distances to the target as heuristics at the border of the radius to guide the agents' movement on a high level.

Wagner and Choset [23] improve the A* approach used to solve the MAPF problem further. Their algorithm is called M* and is actually an A* algorithm where the agents' configuration space is decomposed into single agents and only coupled agents are then solved jointly to reduce the dimensionality of the search space.

Goldenberg et al. [7] an algorithm also based on A* called Enhanced Partial Expansion A* (EPEA*) where only nodes corresponding to single agent moves with the same f-value as the parent node are expanded immediately to reduce the branching factor and therefore the growth of the open list.

Sharon et al. [19] introduce the increasing cost tree search for the MAPF. On the high level the increasing cost tree is explored. It consists of nodes representing all the possible solutions where an agent has a specific path cost, starting from single-agent optimal values. The child nodes are generated by increasing the cost of one path by exactly one. The low level component is used to find the actual conflict-free solutions. For each node the high level component visits, it invokes the low level component. This low level component then tries to find a valid solution restricted to paths for each agent which have the exact cost the nodes in the high level component specify. In essence, the low level component performs goal checks for each node. It does this by building multi-value decision diagrams (MDDs) for each agent. An MDD is a directed acylic graph which represents all possible and valid paths of a specific cost. Using these MDDs, the low level component tries to find paths for each agent which do not conflict with each other.

2.2 CBS Variants

In 2015, Sharon et al. [20] propose CBS, which is similar to the increasing cost tree search in terms of dividing the search into a high level and a low level component. The high level component searches a binary constraint tree while the low level component is used to find the actual paths which make up the solutions. These solutions need to be consistent with the constraints the high level component generates. These constraints are built by resolving conflicts which happen when agents want to occupy the same vertex at the same time.

They also proposed an enhancement for CBS, which is the MA-CBS algorithm, where agents can be merged to meta-agents, which are solved jointly in the low-level search. The decision on when agents are merged is based on the number of conflicts they already have with each other. If the number of conflicts exceeds a certain threshold, the agents are merged into a meta-agent. This threshold is a fixed value throughout the solving process. Sharon et al. propose to use different values for the threshold depending on the topology of the MAPF instance.

Boyarsky et al. [3] provide a further enhancement for CBS, which they call *bypass*. Instead of immediately splitting a high level node with a conflict, they first try to find a bypass of this conflict. They check whether the conflicting agents have other valid paths, which still satisfy the constraints and have the same cost. If such paths exists and the conflict is avoided using these paths, the original paths are replaced with these in the high level node. Therefore, the conflict is resolved without the need of splitting the node.

Additional enhancements to CBS have been introduced by Boyarski et al. [1]. One of these improvements is to restart the whole search after a merge happened. In most cases, this also improves the overall performance. Another enhancement for CBS also proven effective is to prioritize conflicts which will increase the cost of the solution when resolved, which are called *cardinal conflicts*. With these additional techniques, they refer to the resulting algorithm as Improved-CBS (ICBS).

This concept of prioritizing cardinal conflicts is used and developed further by Felner et al. [6]. For each high level node, they construct a graph where each vertex is an agent which is in a cardinal conflict and an edge exists between vertices if the agents are part of the same cardinal conflict. Then, they find a minimum vertex cover of this graph and use the size of this vertex cover as a lower bound of how often the cost of the current solution still has to increase, as each cardinal conflict has to be resolved. They use this estimate as a heuristic value for the high level node and thus guide the search, which reduces the overall number of nodes the algorithm needs to expand. They call the algorithm *ICBS plus h* (ICBS-h), but in the subsequent literature it is referred to as CBSH.

Li et al. [13] improve this heuristic technique further to compute more informed heuristic values based on tighter lower bounds. The resulting algorithm is called CBSH2. They describe two new approaches to calculate heuristic values for the high level nodes of CBS. The better performing heuristic function is based on weighted pairwise dependency graphs (WDG). The graph is identical to the conflict graph used by CBSH in terms of edges and vertices, but they add weight values to the edges. For each agent pair connected by an edge, they completely resolve the conflicts they have with each other by solving the two-agent MAPF subproblem, and observe how much the sum of their path costs increase. This value then is used as the weight of their edge in the dependency graph.

The minimum vertex cover is now replaced by edge-weighted minimum vertex cover. The problems is to assign each vertex a non-negative integer x_i so that the overall sum is minimized while ensuring that the following constraints hold for every edge (i, j):

$$x_i + x_j \geq \Delta_{ij},$$

where Δ_{ij} is the weight of the edge connecting the vertices with the assignments x_i and x_j. The resulting sum is then used as an admissible heuristic.

Li et al. [14] further introduce an additional technique to improve CBS. They add detection mechanisms to find a specific type of conflict which they call rectangle conflict. These conflicts appear when the optimal paths of two agents go through the same rectangle area of the grid. The agents also have to enter this rectangle at the same time. Since CBS resolves these rectangle conflicts inefficiently, they introduce additional constraints which help the algorithm to solve them more quickly.

These rectangle conflicts are due to instance symmetries and part of broader class of symmetries, as described by Li et al. [15]. They additionally identify target conflicts and corridor conflicts, which are two other types of pairwise symmetries that CBS also solves inefficiently. Target conflicts occur when an agent repeatedly conflicts with another agent which is stationary at its target. Corridor conflicts can only occur if there are obstacles on the map. Then, corridor conflicts happen when two agents have to traverse a corridor in opposite directions at the same time. If there is no space for an agent to move aside and let the other agent pass, CBS will solve these conflicts inefficiently.

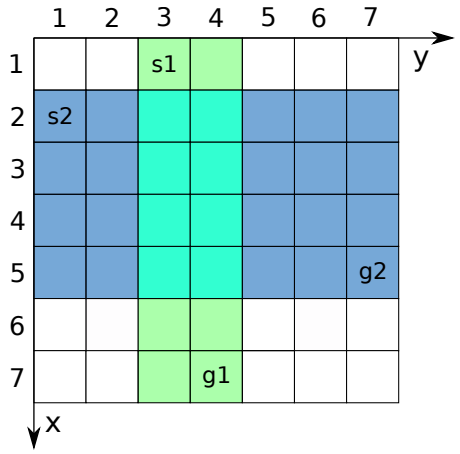


Figure 2.1: A MAPF instance on a 7×7 grid with two agents a_1 and a_2. The green and blue cells represent the rectangles for agents a_1 and a_2 respectively. The turquoise cells are the intersection of both rectangles and form the rectangle area of the rectangle conflict.

To resolve rectangle conflicts, they make use of so-called *barrier* constraints. Their aim is to resolve the rectangle conflict with only one split, by adding a set of constraints in one branching step. The constraints are created for the cells on the edges of the rectangle area of the conflict. This rectangle area is found by building two additional rectangles, one for each agent. These rectangles range from s_i to g_i, where i is an agent and s_i and g_i represent its start and goal cells, respectively. Then, the intersecting cells of these two rectangles form the rectangle area of the rectangle conflict. This is depicted in Figure 2.1. Then green and the blue cells represent the rectangles of agent a_1 and a_2 respectively, and the turquoise colored cells represent the rectangle area of the corresponding rectangle conflict.

The barrier constraints now target the border of the rectangle area through which the respective agent exits. For agent a_1, it consists of the two cells (5, 3) and (5, 4). If agent a_1 takes its optimal path, it reaches one of these two cells at timestep four. As CBS is designed, the high level node with this rectangle conflict is split, but instead of only creating one new constraint for each child node, a set of new constraints is created. In the left child, agent a_1 is prohibited from occupying cells (5, 3) and (5, 4) at timestep four, imposing the eponymous barrier, and in the right child, the same is done for agent a_2, but with the cells of its own exit edge. With these constraints the rectangle conflict is resolved. In this case, the rectangle conflict is called cardinal, as both barrier constraints increase the sum of costs, i.e., both children nodes have a higher cost. It is semi-cardinal if only one barrier constraint increases the cost and non-cardinal if both children nodes have the same sum of costs as their parent.

Recently, Boyarski et al. [2] introduce an enhancement of the merging strategy for CBS algorithms which support meta-agents. The focus of their work is not on the question

Table 2.1: The abbreviations for different CBS variants and their descriptions.

Name	**Description**
CBS	The first version of CBS with no additional improvements.
MA-CBS	CBS with the capability of forming meta-agents by merging conflicting agents.
ICBS(B)	B is the threshold which decides after how many conflicts agents are merged. Additionally, after merging, the search restarts with these agents premerged. Also, instead of splitting on conflicts, ICBS(B) first tries to find a bypass, and cardinal conflicts are prioritized.
ICBS-h	The same enhancements as ICBS, but does not support meta-agents. A heuristic function is added to guide the high level search.
CBSH2(-WDG-R)	The heuristics function is further refined (WDG). Additionally, rectangle reasoning (R) is added.
CBSH2-RTC	The rectangle reasoning is generalized to pairwise symmetry reasoning, which also includes target and corridor conflicts.

whether conflicting agents should be merged, but rather, if there are multiple candidates for merging, which agents should be merged. They noticed a disadvantageous aspect of the already existing merging strategy proposed Sharon et al. [20], which is to merge agents which collided more often than a certain threshold value B. The problem is that large meta-agents occupy more cells than single agents at once and therefore are more likely to conflict with other agents. This leads to large meta-agents growing even larger. The problematic aspect of this is that the low level solving process for large meta-agents is fairly expensive and thus the whole search gets slowed down.

Their proposed strategy to combat this is to choose all suitable merging pairs, i.e., agents which conflict more often than B, and find the sizes of the meta-agents which would result if these pairs are merged. Then, they only keep the ones which would result in the smallest meta-agent. If there are still multiple conflicting agents which fit these criteria, they pick the merge candidates which have the most unresolved conflicts with other meta-agents. The idea behind this is that after the merge new paths for the agents in the newly formed meta-agent have to be found, and the hope is that conflicts with other meta-agents are hereby resolved indirectly.

As there are numerous different variants of CBS, we provide a summary of all their names as abbreviations and a short description of their features in Table 2.1.

CHAPTER 3

Methodology

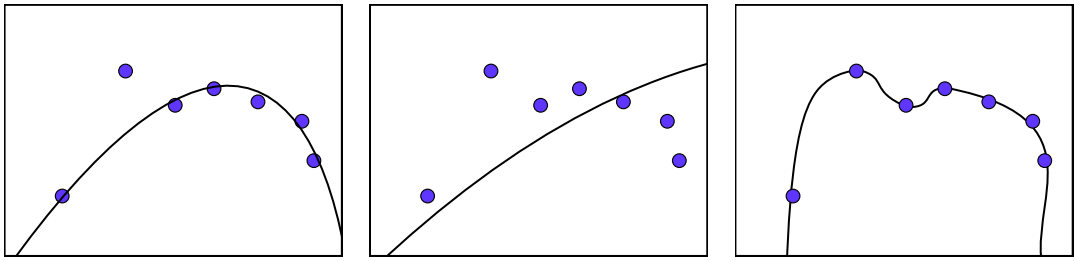
In this chapter we explain and discuss fundamental methods related to our solution approaches. First, we discuss the methodology of the machine learning part of our thesis, which involves binary classification with supervised learning. Then, we describe informed search in detail and how heuristics are applied in the search process.

3.1 Binary Classification with Supervised Learning and Noisy Labels

This section about binary classification using supervised learning is based on Chapter 5 of the "Deep Learning book" by Goodfellow et al. [8]. In machine learning, we approach a specific task in a data-driven fashion instead of concretely specifying algorithmic steps. We first discuss supervised learning, a paradigm in which labeled data—a *ground truth*—is available. Then, we describe a concrete task to be solved, which is binary classification.

3.1.1 Supervised Learning

Before a machine learning model is able to solve any tasks with sufficient accuracy, it first needs to learn how to do it. This learning process can either be supervised or unsupervised. The difference between these two learning methods lies in the data set on which they learn on. This data set consists of many entries, where each entry can again be a vector of multiple values. Each of these values are measurements of a certain object which we want the model to process. These values, which are also called input, are needed for both supervised and unsupervised learning. But the difference is that for supervised learning an additional output value is needed. This output value is also called *label* or *target*. Thus, for supervised learning, each entry in the data set is actually a pair of input values and an output value. The goal for the model is to learn a function which



(a) Balanced model. (b) Underfitted model. (c) Overfitted model.

Figure 3.1: We fit a model to learn a function which best generalizes from the learned data to new data.

maps the input vector to the desired output. In an ideal scenario, the model is then able to correctly predict the label for an input vector it has not seen previously.

The input values are measurements of characteristics of the object our model is designed to process. These characteristics are also referred to as *features*. These features should be present in every instance of the object type we plan to process with this model, so there are no missing values in the data set. Also, the features should be chosen in a way that the values resulting from measuring these features should be different if the individual objects also differ. If the value for a specific feature is equal for all instances of the object type we process, it does not provide any benefit at all.

The overall goal of employing machine learning is to have a model which is able to perform well on data it has not seen during the learning phase, which is called *generalization*. Capacity refers to the complexity of the function a model is able to learn, e.g., linear functions, polynomials, etc. Learning is an interative process where a training error shall be minimized. A model with low capacity may be unable to learn a complex function and therefore underfits no matter how long it learns. In contrast, models with too much capacity may learn aspects of the training data which are not present in the test set. Both underfitting and overfitting are disadvantageous.

To further explain underfitting and overfitting, consider the example in Figure 3.1. We assume a iterative learning strategy where capacity—expressive power—is added over time to increase its performance on a given training data set. In Figure 3.1a, the model has stopped learning at a justifiable time. It captures the essence of the distribution of the elements and thus is able to generalize well. In Figure 3.1b, we show an underfitted model. This can be the result of stopping the learning process too early. Therefore, the line is not correctly fit into the data points and the predictions of the model will be off. In contrast we show an overfitted model in Figure 3.1c. There, each element is placed on the line, since the model has a high capacity (e.g., arbitrary polynomials) This model is able to correctly label the entries of the training data, but it is unlikely that it generalizes well, as the learned line is too specialized on the training data which are often subject to noise.

For the evaluation of the learned model, we need data it has not seen during the learning process, which is referred to as the *test set*. This is because we want to evaluate how well the model is able to generalize. To evaluate the performance of the model, we only let it access the input part of the test set. With the input, it generates a prediction of the label for each entry. Since this is a supervised learning method, we already have the correct labels and can compare them to the predicted values.

However, finding a suitable function to get an actual value for the performance of a model is not as straightforward, as the desired behavior of the system heavily depends on the use case. There are many factors to consider when choosing an evaluation metric. For example, if a wrongly predicted label is only rated as being wrong, or rather graded by how wrong it is. Also, the task which the model is trained to solve influences this decision.

Such an evaluation metric is also used in the learning process. During the learning process an algorithm is employed which aims to minimize a loss function, which is based on a performance evaluation done on the training data. The loss function gives an indicator of how good the current iteration of the model is at predicting correct outputs for the training data. After each step in the learning process the parameters, or weights, of the model are tuned by minimizing the chosen loss function. Also, a regularization strategy can be employed as well. This regularization can be in the form of penalties added to the loss function with the goal to prefer a simpler model function, which may result in better generalization.

3.1.2 Binary Classification

The aim of machine learning is to solve a specific task using data. An example for such a task where machine learning can be effectively employed is classification. In the classification problem the goal is assign classes to elements which are specified by feature vectors. Thus, a machine learning model requires to learn a function:

$$f : \mathbb{R}^d \to \{1, \ldots, k\}, \tag{3.1}$$

where d is the dimension of the input vector and k the number of classes. In the case that $k = 2$ it is called *binary classification*, as there are only two classes.

In Figure 3.2 we show an example of a binary classification problem. There are numerous elements which either are blue and belong to class zero or are red and belong to class one. For each element the values of two features are known, which are x_1 and x_2. Using these values, they are placed in a Cartesian coordinate system. The goal for the model is to find a hyperplane which separates the elements by their classes. There are multiple such hyperplanes and again, there is no universal best measurement to select the best one, as it again depends on the desired behavior of the algorithm. Also, the hyperplanes could be replaced by more general hypersurfaces for more capacity.

Using these hyperplanes, the model is able to predict the class of elements it did not encounter while learning. As there are only two classes, only one hyperplane is needed.

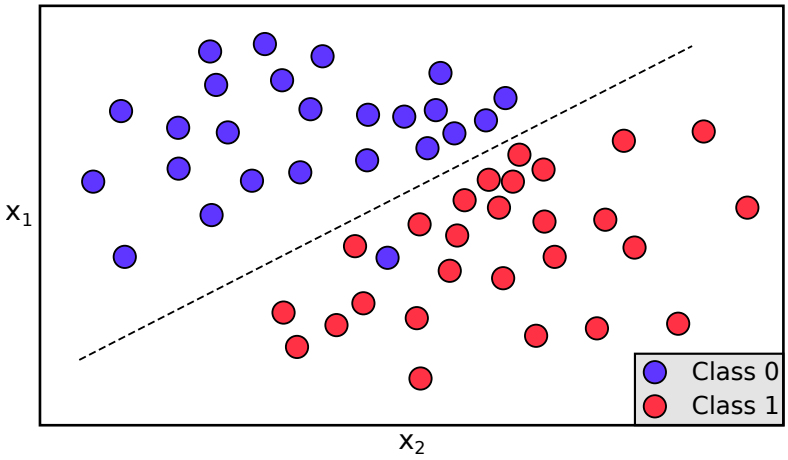


Figure 3.2: An instance with a group of objects which are either part of class zero or class one. Each object is defined by two values of features, which are x_1 and x_2. The dotted line is learned by a model to separate the two classes. This separation is not perfect as one element of class zero is on the wrong side of the line. The model can classify a previously unseen element by checking on which side of the line it is located.

Table 3.1: A confusion matrix for the binary classification problem.

		Prediction	
		Positive	Negative
Actual	Positive	True Positive	False Negative
	Negative	False Positive	True Negative

Then, we check on which side of the hyperplane the newly encountered element is located and consequently find out to which class it belongs.

In many binary classification problems the two classes represent *positive* and *negative*, or *yes* and *no*. A prime example of such a classification problem is medical testing, where the goal is to correctly predict if a patient has a specific disease. In this example, the input vector is data about the patient, and the output is either positive or negative.

Comparing the predicted value with the actual value can result in four possible outcomes. We show these four outcomes in Table 3.1 as a confusion matrix. The first case is that both the actual and the predicted value are positive. The result is called *true positive* and is a desired outcome. In the medical testing example this would mean that a disease present in the patient is correctly predicted by the model. On the other hand, if the disease is present but the model predicts negative, it is a *false negative* and the disease goes undetected. The second unwelcome outcome is the *false positive*, which means the actual value is negative and the model predicts positive. In our example this implies that the patient is healthy but the model thinks that the particular disease is present. Then, the last case is *true negative*, which is when the model correctly predicts the absence of

the disease. Again, the decision on which outcome should be prioritized most depends on the use case. Clearly, true positives and true negatives are preferred over false positive and false negatives.

With these four possible outcomes additional metrics for the evaluation of machine learning models are designed. We will focus on the metrics *precision* and *recall*, which is also called sensitivity. Precision is the number of true positives divided by all predicted positives:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{3.2}$$

and recall is the number of true positives divided by sum of the numbers of true positives and false negatives, which are all actual positives:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{3.3}$$

Individually, precision and recall as metrics are not helpful. For example, it is possible to achieve perfect recall by classifying every object as positive. In binary classification, often there is an inverse relationship between precision and recall. For example, if we increase the threshold which is used to decide whether the model predicts positive or negative based on a score or a probability, we increase precision while simultaneously decreasing recall. A high threshold makes the model only predict positive if it is certain. Therefore, it is less likely to return a false positive and the precision increases. In contrast, the number of returned false negative also increases, which results in a decrease in recall. Thus, using this threshold, a suitable balance between precision and recall can be found. But again it depends on the use case which of these two metrics is more important.

The tradeoff between precision and recall mentioned above is depicted in Figure 3.3, where we show an example for a precision-recall curve. This curve helps in the selection of a suitable threshold, since this curve makes it easier to compare specific precision values to their respective recall values. For each threshold, there is a precision value and a recall value, which are used to plot the curve. Additionally to aiding in finding an appropriate threshold, the area under the curve can be used as a performance measure to compare different models. The bigger the area under the curve, the better the model is at predicting correctly, as then both the precision values and the corresponding recall values are high.

3.1.3 Noisy Labels

As described earlier, the training data needs to have labels for supervised learning. These labels can either be provided by humans or are computed by algorithms. As with most other tasks, humans are prone to make mistakes by mislabeling entries of the data set. Having wrong labels in the training data is referred to as *noise*. Not only humans can produce noisy labels, algorithms may be susceptible to make such errors as well. For example, if the label is the result of comparing runtime measurements of algorithms which

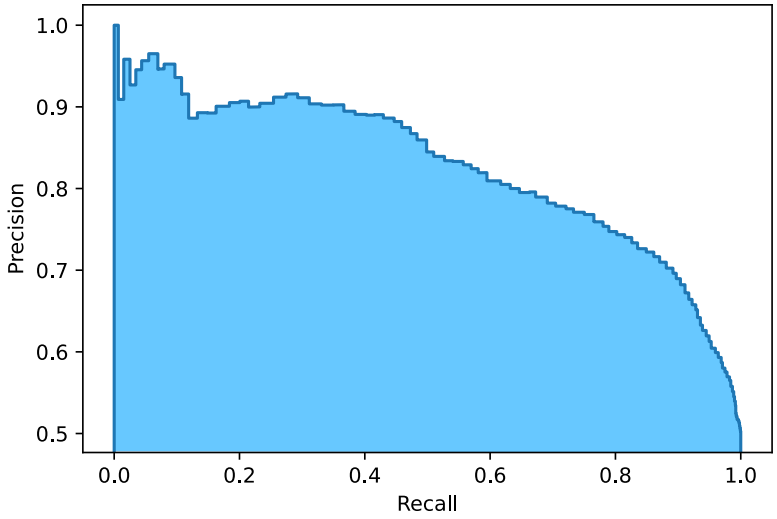


Figure 3.3: In this precision-recall curve the inverse relationship between precision and recall can be seen.

only differ by milliseconds, it is possible for slight runtime fluctuations to completely change the outcome.

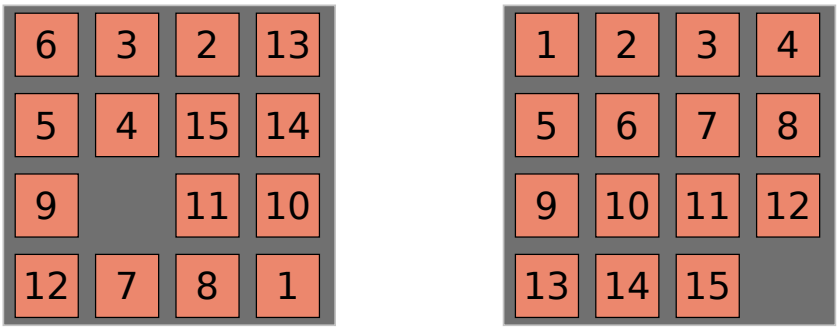
Noisy labels also do not have to be the result of an error as described by Sculley and Cormack in [18], who analyze noisy labels for emails. In their study, user feedback is used for labeling training data, where emails are to be labeled as either spam. For some emails, it is subjective if it is spam or not. For example, some users experience newsletter emails as spam and some do not. Also, since the users are not specially trained to classify the correct emails as spam, some users contribute wrong labels. Other sources of wrong labels are misunderstanding how the feedback mechanism works and accidentally choosing the wrong label.

Having noisy labels in the training data results in worse classification performances, even if the noise is artificially added by flipping a label with a uniform probability. To counteract this, it is either required to use machine learning models which are more robust regarding noisy labels or cleanse the training data from the noisy labels.

3.2 Heuristics in Informed Search

In this section we discuss informed search and how heuristics are used to enhance the search process. We base this section on Chapter 3 of the work of Russel and Norvig [17].

Search processes are used to solve complex problems. The desired outcome of performing a search is to find a sequence of actions that lead from a given initial state to a goal state.



(a) Initial state: scrambled. (b) Goal state: solved.

Figure 3.4: An example of a 15-puzzle.

This search process can be done by informed and uninformed algorithms. An informed search algorithm has access to information about the problem it aims to solve. It can use this information to estimate if performing a specific move brings it closer to a goal state. These estimates are calculated by a heuristic function.

One of these problems where an algorithm performing an informed search using heuristics can be applied is the game 15-puzzle, which is depicted in Figure 3.4. The 15-puzzle consists of 15 tiles which are numbered from one to 15. They are placed on a four by four board which has space for 16 tiles, which means there is one free spot. Then, the tiles need to be ordered without removing them from the board. The only way allowed to move them is to slide them either horizontally or vertically into the free space. The goal of the search is to find a sequence of sliding-moves which lead from the initial state where the puzzle is scrambled, depicted in Figure 3.4a, to a goal state. In a goal state every tile is at its correct position, as illustrated in Figure 3.4b. This sequence is also referred to as a solution to this problem and the cost of a solution is the number of moves in the sequence.

A commonly used informed search algorithm is the A* algorithm introduced by Hart, Nilsson, and Raphael in [9]. It searches through graphs by calculating an f-value for each node n it encounters. The f-value is calculated as follows:

$$f(n) = g(n) + h(n) \tag{3.4}$$

As shown in Equation 3.4, the f-value is the sum of the g-value and h-value of a node. The g-value is the lowest cost from the start to this node found so far. This g-value changes if the A* algorithm finds a better path with a lower cost to the node. The h-value is the estimated cost from this node to a goal node and is calculated using heuristics.

A* maintains a priority queue called *open list*. All generated nodes are placed in this queue where they are sorted by their f-value. Using the open list, the algorithm always chooses the node with the lowest f-value to expand next. This approach is called *best-first search*. Before expanding a node, a check whether the state of the chosen node is a goal

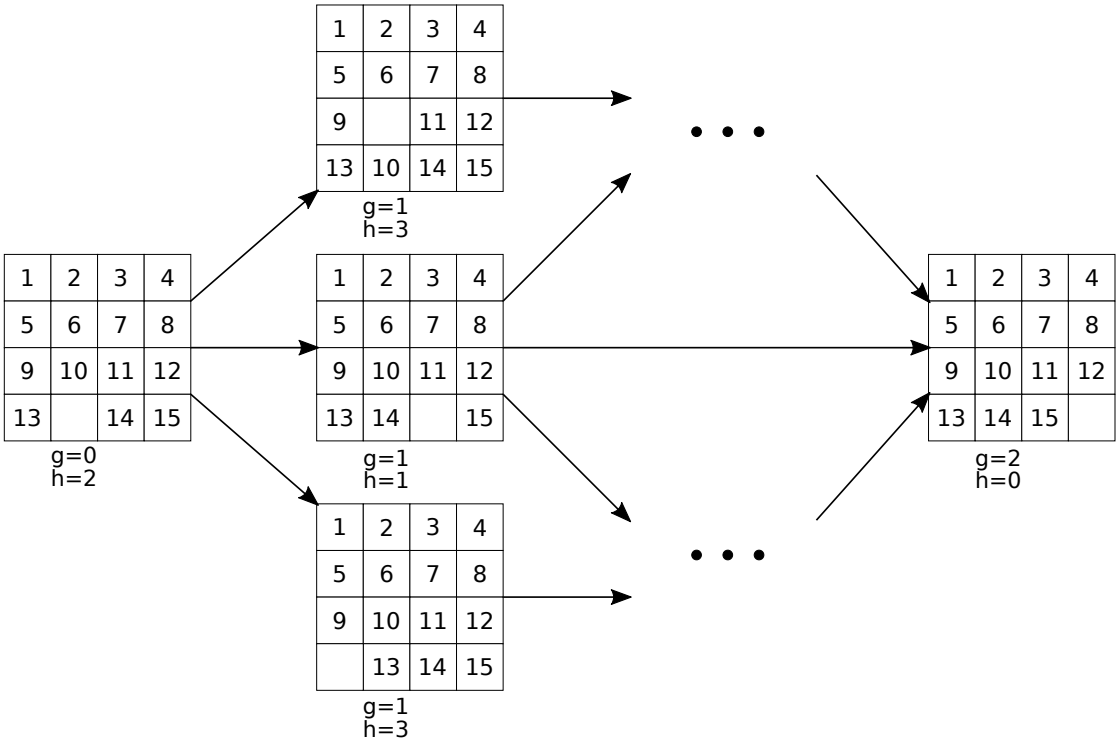


Figure 3.5: A search tree derived from a scrambled 15-puzzle. In the initial state, only tiles 14 and 15 are out of place. Also shown are the g- and h-values below their respective states.

state is performed. If it is, this solution is returned and the search process is finished. Otherwise, the node is expanded and added to the *closed list*. This list contains all nodes which already got expanded. If the algorithm encounters a node that is already in the closed list with an equal or smaller f-value, it will not be put into the open list to avoid duplicated computational work. Similarly, if a node is already in the open list with an equal or smaller f-value, it will not be added as well.

The A* search algorithm can be employed to solve the 15-puzzle, as described in [12]. A suitable property of the problem needs to be selected for the heuristic calculation. It is necessary for the chosen heuristic to be admissible to guarantee that the search finds an optimal solution. For the 15-puzzle, a solution is a sequence of moves leading from the initial state to the goal state. A solution is called optimal, if it consists of the minimum number of moves it takes to solve this 15-puzzle instance. And for a heuristic function to be admissible, it must not overestimate the cost from the current node to the goal node. Given the function $h^*(n)$, which returns the optimal cost of reaching a goal node from node n, the following equation must hold for the heuristic to be admissible:

$$h(n) \leq h^*(n) \tag{3.5}$$

This means that the h-value of a node always has to be smaller or equal to the lowest

possible cost to the goal node.

Often, heuristics are calculated by looking at a relaxed version of the problem at hand. An example of an admissible heuristic for the 15-puzzle based on a relaxation is to count the number of tiles that are currently in a wrong position. This is a relaxation of the problem, since this assumes that we are allowed to pick up tiles and place them at their target. So even in this simplified version, we still need to perform this number of moves.

We will explain how A* employs this heuristic based on the example depicted in Figure 3.5. Figure 3.5 shows parts of a search tree which is derived from a rather simple instance of the 15-puzzle. The search tree consists of nodes and directed arcs which connect the nodes. A node represents a state and an arc represents an action, which is the moving of a tile in the case of the 15-puzzle. Also, each action has a cost of one. In the initial state, only two tiles are not on their target position. Tiles 14 and 15 both need to move one position to the left for the puzzle to be in a goal state.

First, A* calculates the f-value by summing up the g-value and the h-value of the initial state. For the initial state the g-value is zero, as no moves have been done yet. Then, to calculate the h-value, we count the tiles which are out of position. As only tiles 14 and 15 are placed incorrectly, the f-value of the initial state is two. The next step is to generate the children of the initial state, which is also referred to as expanding a node. It is possible to perform three different actions in this state. We can either move tile 10, tile 13, or tile 14 into the open space. Since we need to perform an action to generate each child, all children will have a g-value of one. Moving tiles 10 and 13 into the open space reduces the number of correctly placed tiles by one. Thus, both children have a h-value of three and and an f-value of four. The third child, which is generated by moving tile 14 into the open space, only has one incorrectly placed tile. Thus, its f-value is two and will be chosen to be expanded next, as it has the lowest f-value and is thus the most promising unexpanded state.

This state again has three children, as tiles 11, 14, and 15 can be moved into the open space. Since we need to perform two actions to get to the newly generated children, their g-values are two. The children which moved tiles 11 and 14 into the open space both removed a tile that was already on its correct position. Both have two unsolved tiles and therefore h-values of two and f-values of four. On the other hand, the remaining child which moved tile 15 to the open space has now zero incorrectly placed tiles. Thus, its h-value is zero and its f-value is two. Since this node has the lowest f-value it will be chosen next for expansion. But before this node is expanded, a goal check is performed. This goal check is done before every node expansion. Since there are only correctly placed tiles, it is a goal node and the search can stop. The algorithm then returns the sequence of actions it takes to reach this goal node, as this is the actual solution to the problem.

The discussed heuristic is admissible, as every misplaced tile has to be moved at least once. Sliding a misplaced tile to its correct position will always require one move, but can also require more moves. But, as assumed for the heuristic, picking up a tile and placing it at its right place only requires one move. Also, sliding a tile does not change

the position of other tiles, so a move can only increase the number of correctly placed tiles by one. This means that the calculated f-value can never overestimate the required cost to finish the puzzle.

Another important aspect of heuristics is their monotonicity. If A* uses a monotone heuristic, it guarantees to expand every node at most once, which is important for performance. Given the cost function $c(n, a, m)$, which returns the cost of performing move a to get from node n to the successor node m, a heuristic is monotone, if it satisfies the following equation for every node n:

$$h(n) \leq c(n, a, m) + h(m) \tag{3.6}$$

This means that the h-value of a node has to be less than or equal to the sum of the cost of a move and the h-value of the node which results by performing this move. In the 15-puzzle, a move always has a cost of one. Consequently, to violate this requirement, the h-value of a successor node needs to be smaller compared to its predecessor with a difference larger than one. As stated earlier, it is not possible to move more than one tile to its correct position with one move. Therefore, if the h-value decreases, it can only decrease at most by one, which shows that the heuristic is also monotone. Also, monotonicity implies admissibility.

Using monotone heuristics in A* guarantees it to be optimally efficient. This means, that compared to other A*-like algorithms using the same heuristic information, it will not expand more nodes than the others. This can be shown by dividing all the nodes of a search problem into three groupings. Given the cost C^* of an optimal solution and the function $f(n)$, which returns the f-value of a node, we get the following three groupings:

- $f(n) < C^*$: Nodes with an f-value smaller than the optimal cost are guaranteed to be expanded. Any node in this group has the potential of belonging to an optimal solution.
- $f(n) = C^*$: Nodes in this group might be expanded.
- $f(n) > C^*$: All nodes in this group will not be expanded by A*.

Therefore, A* will expand all nodes from the first group, some of the second group, and none of the third group. Because all nodes of the first group might be part of an optimal solution, all the other A*-like algorithms need to expand them as well. For the second group, the tie-breaking rules decide whether a node will be expanded. For the purpose of defining optimal efficiency, this difference in expanded nodes is negligible and can be ignored.

Although A* with monotone heuristics is optimally efficient, it still has its drawbacks. One of these shortcomings is its space complexity. The space complexity of an algorithm refers to the amount of memory it needs to execute. By design, the A* algorithm stores

every generated node. In the worst case this leads to memory consumption which is exponential in the depth of the search.

Because of its space complexity, A* was generally not able to actually solve an arbitrary instance of the 15-puzzle [12] with the computational capabilities available in the year 2000. To combat the immense memory usage, Iterative-deepening A* (IDA*) is proposed by Korf [11]. In contrast to basic A*, IDA* has a space complexity which is only linear in the maximum depth of the search process. This is achieved by searching in a depth-first manner and backtracking once the f-value of a node exceeds a cutoff value. This cutoff value increases at every iteration and is chosen by taking the smallest value from all the f-values encountered in the previous iteration which exceeded the cutoff. This enables it to solve arbitrary instances of the 15-puzzle without running out of memory.

CHAPTER 4

Problem Formalization

In this chapter we formally describe the multi-agent pathfinding problem. We also describe the problem setting for our solution approaches. Then, we describe how we build meta-agent assignments to premerge agents in the ICBS algorithm.

4.1 Multi-Agent Pathfinding

In this section we formalize the multi-agent pathfinding problem based on the work of Stern et al. [22]. As there are numerous different settings for the MAPF problem, it is important to describe in detail using a unified terminology which variant is used.

First, we define the classical MAPF problem, which all MAPF variants are based on. An instance of the classical MAPF problem is defined by the tuple

$$\langle G, s, t \rangle,$$

where $G = (V, E)$ is an undirected graph with V being the set of vertices and E the set of edges, and s and t are the functions

$$s\colon [1, \ldots, k] \to V$$
$$t\colon [1, \ldots, k] \to V,$$

where k is the number of agents of the instance. The function s maps the agents to starting vertices and t the agents to their goal vertices. The time is discretized into time steps and each time step each agent performs an action $a\colon V \to V$ such that $a(v) = v'$, where v and v' are vertices. Performing such an action moves the agent from vertex v to vertex v'. If $v \neq v'$, it is called a *move* action. This action is only possible if the vertices are connected by an edge. Otherwise, if $v = v'$, the agent performs the *wait* action, where it stays at its current vertex.

Each agent i performs a sequence of actions $\pi_i = (a_1, ..., a_{n_i})$. For an agent i, which starts at vertex $s(i)$, a sequence of actions π_i is called a single-agent plan, if and only if executing the actions results in the agent being at vertex $t(i)$. Then, a solution consists of k such single-agent plans.

There are many solutions to an MAPF instance, but for a solution to be *valid*, there must not be any conflicts between the single-agent plans. In our setting, a conflict happens when two or more agents collide in any way. This can either be a collision on a vertex, which is called *vertex conflict*, or a collision while traversing an edge in opposite directions, which is referred to as *swapping conflict*.

Another factor to consider is how agents that reach their goal before others are handled. They can either stay at the target and therefore potentially block paths for other agents, or they disappear and do not have to be considered anymore for paths of other agents. We choose the variant where the finished agents stay at their target. This has implications on the objective function, which is used to evaluate MAPF solutions. We choose the *sum of costs* function

$$g^{\mathrm{SoC}}(\pi) = \sum_{i=1}^{k} |\pi_i|,$$

which is the sum of the costs of all single-agent plans, which we also call lengths referring to the path lengths in the state space. Since we choose for the agents to stay at their target, it is necessary to define if this increases their path lengths. In our setting the path length does not increase for agents which terminally stay at their target. If they are already at their target but then move away from it, we count the stay at the target as wait actions which increase the costs of the single-agent plan. So waiting at the target only increases the cost if the agent plans on leaving it again.

As done in most of the literature regarding the MAPF problem, we limit the input graphs to be four-connected grids. Each cell in these grids represents a vertex and an edge between vertices v and v' exists, if v' is one of the four direct neighbors of v. On these grids a cell can be an *obstacles*, which agents are not allowed to move to. So the vertices which are obstacles and their edges have to be removed from the input graph.

Also, our aim is not to find any valid solution, but rather an *optimal* solution. An optimal solution is a valid solution with minimal cost, where the cost is given by the objective function. This means that for our setting, there exists no valid solution with a sum of costs lower than an optimal solution. Since the cost of a solution is the only criterion whether it is optimal, it is possible for a MAPF instance to have multiple different optimal solutions.

4.2 Group assignment for Meta-agents in CBS

In this section we describe how to build the group assignment to premerge meta-agents for the ICBS algorithm. To internally handle and differentiate the agents, they all are

assigned an identification number, which we refer to as their ID.

Through our approaches, we select pairs of agents which we want to premerge, i.e., before we run the search algorithm. The chosen pairs, which are represented by their IDs, then are transformed into a meta-agent assignment. The shape of such a meta-agent assignment is the following:

$$0,\ 1,\ 2,\ \ldots,\ n{-}2,\ n{-}1$$

with n being the number of agents and the individual numbers representing the IDs of the agents. The above depicted meta-agent assignment represents the case where no agents are merged and thus, there are no meta-agents. To represent a merge of two agents in this notation, the ID of the agent with the higher ID is replaced with the ID of the other agent. The following is an example of such an assignment:

a_0	a_1	a_2	a_3	a_4	a_5
0	1	2	2	4	5

The example above shows an meta-agent assignment for an instance with six agents a_0 to a_5 with IDs also ranging from zero to five, where agents a_2 and a_3 are merged into a meta-agent. The original ID of a_3, which was three, is overwritten with the ID of agent a_2, which is two. This means they are now merged into a meta-agent of size two.

Given the characteristics of this notation it is not possible for an agent to be in two different meta-agents, since it can only have one ID and thus only be a member of one meta-agent. So if we want to merge one specific agent with two other agents, we have to merge all of them into on meta-agent of size three. For this, we consolidate the pairs we want to merge into groups of agents. Consider the following example:

$$[(a_0,\ a_2),\ (a_2,\ a_3)] \longrightarrow (a_0,\ a_2,\ a_3) \longrightarrow$$

a_0	a_1	a_2	a_3	a_4	a_5
0	1	0	0	4	5

In this example the pairs we want to merge are $(a_0,\ a_2)$ and $(a_2,\ a_3)$. Since it is not possible for agent a_2 to be in two different meta-agents, we have to consolidate the pairs into one larger group. The resulting group in the example is of size three and consists of $(a_0,\ a_2,\ a_3)$. Now they can be merged into one single meta-agent of size three by setting the IDs of a_2 and a_3 to the ID of a_0, which is 0. This consolidation process has to be performed for every agent which is part of multiple pairs we want to merge.

CHAPTER 5

Solution Approaches

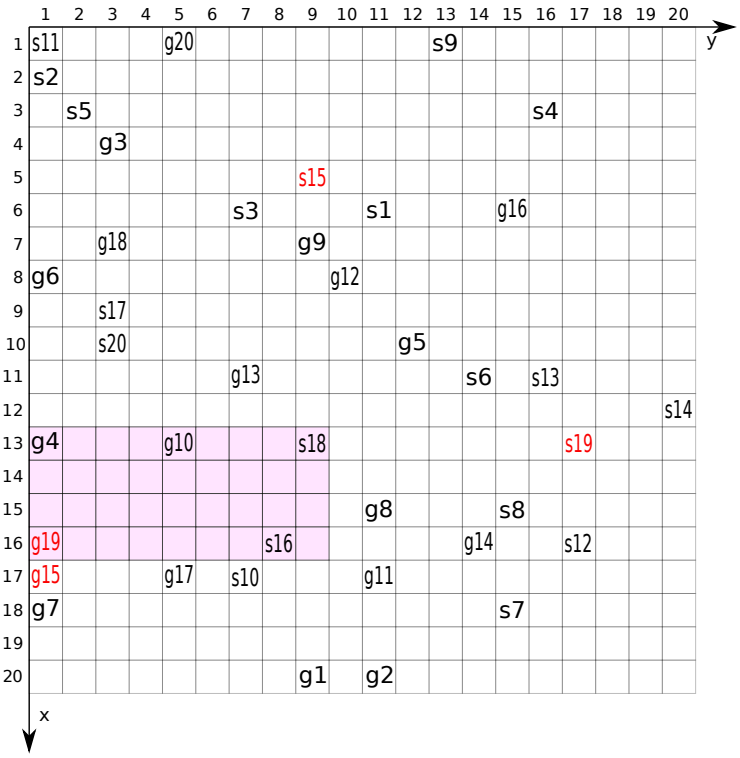
In this chapter we present our solution approaches for finding beneficial meta-agent assignments and for heuristics for CBS which are able to handle meta-agents. First, we propose how pairwise symmetry conflicts can indicate which agents speed up the search substantially when merged. Secondly, we describe how we employ a supervised learning pipeline to train a machine learning model to predict beneficial meta-agent assignments. Lastly, we propose two admissible heuristics for the high level search of CBS which are able to handle meta-agents.

5.1 Merging Agents with Pairwise Symmetries

To increase our understanding of meta-agents, we try to analyze under which circumstances a merge is beneficial. For this, we perform multiple runs on the same instance, but with different preformed meta-agents. We focus on merging only pairs of agents to limit the size of the meta-agents to two. This means that for an instance with n agents, we perform $\binom{n}{2}$ runs, since the order of the agents in the pair is not relevant.

We provide an example in Figure 5.1 of how such an instance could look like and how the runs can differ in the time needed to solve it, with the only difference being the merged pair of agents. Depicted is an instance with 20 agents on a map with no obstacles. To get every possible agent pairing, we generate all $\binom{20}{2}$ pairs, which result in 190 unique agent pairs. Then, we perform the 190 runs with every run having a different pair merged. For this we use ICBS(∞) which does not perform any additional merges during the search.

We track the time needed to solve the instance and also how many nodes the high level search expands. Then we sort the results by the time needed to analyze which merges are beneficial. In the example in Figure 5.1 this is shown on the right side. There, merging the agent pair a_{15} and a_{19} is by far the best option, as the algorithm is able to solve the instance in less than a second with those two agents merged. Also, it only



Pair	Time (s)	Nodes
15-19	0.22	6
1-13	3.24	495
7-19	9.91	3244
7-15	10.19	2794
2-4	11.21	2970
3-7	11.48	2970
6-20	11.60	2771
11-20	11.80	2766
4-7	11.81	2970
3-9	11.92	2970
4-9	11.92	2970
2-12	11.93	2970
1-3	11.95	2970
	⋮	
5-10	120.00	929
1-10	120.00	952
10-11	120.00	1037
10-20	120.00	1443
10-13	120.00	2256

Figure 5.1: An instance with 20 agents on the empty map. On the left side the instance is depicted, with s referring to the start cells of the agents and g to their goals. Highlighted in red are the agents a_{15} and a_{19}, which are in a cardinal rectangle conflict. The area of the rectangle is also highlighted. On the right side we show for a selection of agent pairs the runtimes and high level nodes expanded from runs where this pair is premerged and no further merging is done. For comparison, performing no merge at all results in a runtime of about 18 seconds with 2970 node expansions.

needs to expand six high level nodes. Most other pairs, except for a_1 and a_{13}, result in runs with more than 2000 nodes expanded and runtimes longer than ten seconds. After finding pairs which result in a substantial speed up when merged, we visualize the instance to analyze the paths of the agents. On the left side of Figure 5.1 we show such a visualization. We also highlight the start and goal cells of agents a_{15} and a_{19} in red. It becomes apparent that the agents are in a cardinal rectangle conflict with a rather large rectangle of size 4×9, which is also highlighted. The second best agent pair a_1 and a_{13} is also in a cardinal rectangle conflict, but as the rectangle is only of size 1×3, their merge is not as impactful. Also, most merges involving agent a_{10} result in a timeout after 120 seconds. Presumably, this agent should not be merged into a meta-agent, but analyzing which agents not to merge is beyond the scope of this work.

Analyzing other instances on the empty map leads to similar results. If there is a pair

Table 5.1: Time needed in milliseconds for ICBS(∞) and EPEA* to solve an instance with two agents which are in a cardinal rectangle conflict or corridor conflict of the specified size.

Rectangle Conflict		
Size	ICBS(∞)	EPEA*
6×6	4710	2
7×6	18660	2
7×7	61620	2
8×7	507408	3

Corridor Conflict		
Length	ICBS(∞)	EPEA*
9	1140	3
10	2850	4
11	7260	4
12	20020	5

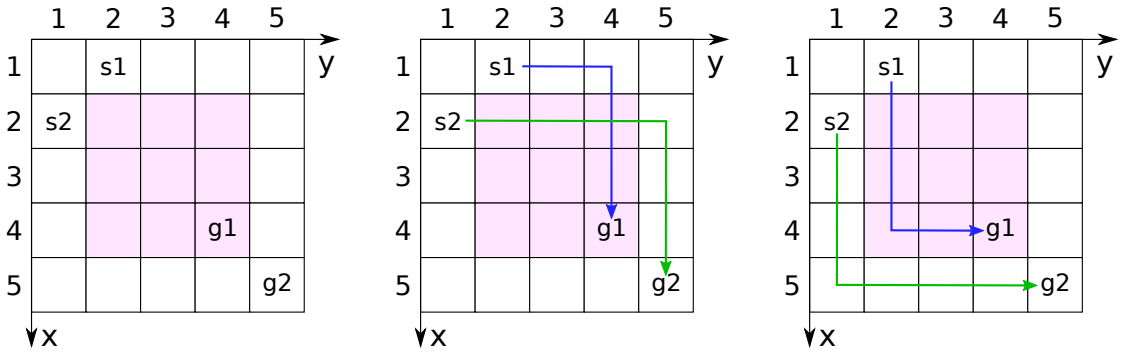
that is by far better than the others, it usually is in a cardinal rectangle conflict. Because of this we assume that the low level solver of ICBS(∞) that is used to find non-conflicting paths for meta-agents is more efficient at handling rectangle conflicts than CBS. To confirm our assumption, we create an instance with two agents which are in a cardinal conflict and let ICBS(∞) and EPEA*, which is the algorithm used as the low level solver for meta-agents in ICBS, solve it and compare their runtimes.

In Table 5.1 these times are shown on the left side in milliseconds for a selection of rectangle sizes. It becomes apparent that EPEA* indeed is able to resolve rectangle conflicts better than ICBS(∞). To solve the instances EPEA* only needs the time one would expect it takes an algorithm to solve an MAPF instance with only two agents. We extend this experiment to also include the other two types of pairwise symmetries described by Li et al. [15], which are corridor and target conflicts. On the right side of Table 5.1 we present the results of experiments on an instance with a corridor of width one. We place two agents on opposite sides of the corridor and set their goals in a way that both have to traverse the corridor in opposite directions at the same time. Thus, they are in a cardinal corridor conflict. Again, we track the runtimes in milliseconds and observe that ICBS(∞) performs poorly on longer corridors, whereas EPEA* resolves these instances almost instantly. The differences in runtimes on instances with target conflicts are not as notable, but EPEA* still outperforms ICBS(∞) in their resolution. Since the ICBS implementation employs EPEA* as the low level solver for meta-agents, we devise merging strategies which focus on these pairwise symmetries.

5.1.1 Detecting Pairwise Symmetries in Initial State

For our case it is not sufficient to use the detection methods proposed by Li et al. in [15]. They state that for rectangle conflicts to occur between two agents, they have to have at least a vertex conflict. This means they only detect rectangle conflicts between two agents if they come into conflict. But since our approach focuses only on the initial state of an instance and thus our information about conflicts is limited to the initial individual paths of the agents, we may miss some semi-cardinal rectangle conflicts.

Conflicts are called semi-cardinal if resolving it increases the cost for one of the conflicting



(a) A semi-cardinal rectangle conflict.

(b) Choice of direction prioritizes right over down.

(c) Choice of direction prioritizes down over right.

Figure 5.2: A 5×5 grid with two agents. The start of a_1 is at (1, 2) and its target at (4, 4) and the start and target cells of a_2 are at (2, 1) and (5, 5) respectively. The colored cells highlight their rectangle.

agents, but not for both. We show an example of a semi-cardinal conflict in Figure 5.2. In Figure 5.2a, a semi-cardinal rectangle conflict between agents a_1 and a_2 is depicted. Both agents have optimal paths leading through the highlighted rectangle, but a_2 also has an optimal path which bypasses this rectangle. Therefore, the conflict can be avoided and the solution cost does not increase. Consequently, the solving algorithm does not have to resolve this conflict at all. However, if a third agent occupies one of the cells which are part of the bypassing path of agent a_2, agent a_2 needs to move through the rectangle. Then, this semi-cardinal rectangle conflict between agents a_1 and a_2 needs to be resolved, which results in substantial computational work.

The state depicted in Figure 5.2a is the initial state of the instance, in which we want to find all cardinal and semi-cardinal pairwise symmetry conflicts. The first step to solve this instance is to invoke the low-level solver to find individual optimal paths for each agent. Since there are multiple optimal paths for both agents, the low-level solver has to decide which path to take. This can be based on a direction priority, which for example could state that if there are multiple optimal options an agent can take, it first has to go up and if that is not possible it has to go right next, and so on. This has implications on the detection of semi-cardinal symmetry conflicts. We show this in Figure 5.2b and Figure 5.2c. In both figures, the agents have to go right and down to get to their goal cells. In Figure 5.2b the direction priority is to first go right and then go down. This results in a collision on (2, 4). In Figure 5.2c the direction priority is reversed and the agents first go down and then to the right. This results in a_2 taking the bypass and no conflict occurring at all. Since we only get our information about conflicts from the initial paths, we have to first check for conflicts in the paths which prioritize horizontal directions and then check for conflicts in the paths which prioritize vertical directions.

To detect pairwise symmetries in the start state of an MAPF instance we employ a modified version of the CBSH2-RTC algorithm introduced by Li et al. [15] and for the

detection of semi-cardinal rectangle conflicts, we modified CBSH2-WDG-R introduced by Li et al. [13]. Instead of letting the algorithms solve the instance, we only let them analyze the start state of the given instance. The first step of the analysis process is to find individual optimal paths for each agent from their start location to their goal location. Then, we check if there are any conflicts present. If there are no conflicts, the current solution is already valid and therefore the instance is solved. In the more likely case that there are conflicts present, we classify them. This process is split into classification by priority and classification by type. The priority of a conflict can be cardinal, semi-cardinal, or non-cardinal. The possible types are target, corridor, rectangle, or standard. The last step is to output information about each of the conflicts so we can further process it. The information about a conflict consists of the two agents which are part of it, its priority, and its type. If it is a rectangle conflict, we also output the size of the rectangle.

We then use this information to choose pairs which we want to merge into meta-agents. The decision on which pairs are picked depends on the chosen strategy. The objective of these strategies is to find a balance between merging all the right agents while also keeping the overall merge count low. We aim to avoid unnecessary merging since each merge increases the complexity of the individual CBS nodes, which can slow down the overall search process.

5.1.2 Strategies

We design and implement the following selection strategies:

rect_all_card. In this strategy, all pairs which are part of cardinal rectangle conflicts are merged. With this strategy we ensure that all cardinal rectangle conflicts are resolved by the meta-agent low-level solver.

rect_all_semi. This strategy is also similar to rect_all_card, but in addition to choosing all pairs which are part of cardinal rectangle conflicts, it also chooses those which are in semi-cardinal rectangle conflicts. Although it is initially not necessary to resolve semi-cardinal rectangle conflicts using the meta-agent low-level solver, it will be beneficial in the case that the search process generates new constraints which make the semi-cardinal conflict cardinal.

rect_largest. For this strategy, we choose exactly one pair which will be merged. This pair has to be in a cardinal rectangle conflict. Since it is likely to have multiple such pairs, we sort them by the size of the area of the rectangle conflict. We then choose the pair with the largest area. We base this choice on the fact that the number of nodes needed to be expanded to resolve a rectangle conflict is exponential in the size of the rectangle. Thus, merging the pair with the largest rectangle should be most beneficial.

rect_size2. Equal to rect_all_card, the focus is on cardinal rectangle conflicts. Every agent pair in these conflicts is a merge candidate but every agent is merged at most once. Thus, the maximum size of meta-agents in this strategy is two. We prioritize merging

pairs in conflicts which have large rectangles, so the agent pair with the largest rectangle gets merged first, than the pair with the second largest rectangle, and so on.

rect_all_card_gr_10. Similar to rect_all_card, all pairs which are part of cardinal rectangle conflicts are merged. The only difference to rect_all_card is that only pairs which have a rectangle with size greater than ten are merged. Not merging pairs with rather small rectangles may prove to be beneficial since their rectangle conflicts can be resolved rather quickly and the individual CBS node complexity will not be increased.

rect_all_semi_gr_15. The aim is again to only merge the necessary pairs. For this, we choose all the pairs which are in cardinal or semi-cardinal rectangle conflicts. The rectangles also have to have sizes larger than 15.

rect_all_semi_gr_40. This strategy is almost the same as rect_all_semi_gr_15, but we aim to reduce the number of potentially unnecessary merges even further. We again merge all pairs in cardinal or semi-cardinal conflicts, but in this strategy, their rectangles' sizes have to be larger than 40.

target_all. In contrast to the previous described strategies, this strategy focuses on target conflicts instead of rectangle conflicts. We merge every pair which is in a cardinal target conflict.

target_random. This strategy also aims to resolve target conflicts. Here, instead of merging all pairs in cardinal target conflicts like it is done in target_all, we randomly choose exactly one pair out of them.

target_size2. Similar to rect_size2, the idea of this strategy is to only create meta-agents of size two. To do this, we find all agents in cardinal target conflicts and randomly choose pairs to merge until there is no further merging possible without building a meta-agent with more than two agents.

corr_all. With this strategy we set our focus on the corridor pairwise symmetries. This is the corridor based equivalent of rect_all_card and target_all, and therefore its aim is to merge all pairs which are in cardinal corridor conflicts.

corr_random. This strategy behaves similar to target_random as it also chooses one pair randomly, but instead of taking a pair in a cardinal target conflict, it chooses a pair present in a cardinal corridor conflict.

corr_size2. This strategy is the corridor based equivalent of target_size2.

For target and corridor conflicts we focus only on those that are cardinal as these types of symmetries affect the runtime of CBS not as severely as rectangle conflicts do.

The last step is to let the ICBS algorithm [1] solve the instance with the predefined meta-agent assignment. The corresponding implementation by the authors is able to form meta-agents using our assignment before the search process starts. Because we build our preferred meta-agents before starting the search we disable the algorithms capabilities of forming new meta-agents while solving. We do this by setting its B-value to ∞. Since

no pair of agents will be able to have a count of conflicts between them that is higher then the B-value, no additional agents will be merged. The ICBS implementation uses an A*-based low-level solver for the single agents and an EPEA*-based low-level solver for the meta-agents.

Since this approach uses two different CBS implementations to solve the instances, parts of the computational work the conflict classification algorithm performs are lost, as we only extract information bout the symmetry conflicts. If the implementations were combined, the solving part could utilize the initial paths and information about all the conflicts, which the classification part already found.

5.2 Machine Learning Model Enhanced Merging

In this section we describe how we employ machine learning to form meta-agent assignments before starting the CBS search process. Again, as with the pairwise symmetry approach, our aim is to form meta-agents which reduce the computational work needed to solve an instance of the MAPF problem. In this approach, instead of devising strategies to select agents for merges manually, we let a machine learning model predict which merges decrease the runtime of the algorithm. The idea is that the process of gathering data to learn on and the learning process itself are time-consuming but only performed once, and the online prediction is reusable and fast. Thus, we train a model on a set of training instances which then can be used to preform meta-agents for new instances. The goal is to have a model which correctly predicts whether merging two agents is beneficial or detrimental to the runtime. For this, we train a machine learning model to categorize agent pairs into these two classes. Then, we merge the agent pair which has the highest probability of belonging to the beneficial class. The process of this approach is as follows:

1. First, we prepare training data which contains the input values and the measured label for numerous agent pairs.
2. Secondly, we use this training data to fit a classification model. After the learning process has finished, we can use the trained model on instances it did not encounter while learning.
3. Again, we calculate the features for each agent pair of the MAPF instance we aim to solve. This process has to be done equally as in the generation of the training data.
4. Instead of finding the output value ourselves, we let the model predict for each agent pair the probability of it belonging to the class of beneficial merges.
5. Then we pick the agent pair which has the highest probability and combine its agents into a meta-agent.
6. The last step is to let ICBS(∞) solve the instance with this preformed meta-agent.

Steps 1 and 2 have to be performed only once, whereas steps 3 to 6 are required for each new instance encountered.

5.2.1 Training Data

For the supervised learning process we prepare training data, which is a list of examples. Each entry consists of input values, the features, and an output value, the label, and each entry corresponds to one agent pair. The vector of input values consists of features we choose and the desired output value is either a *yes* or a *no* to the question, whether merging the pair decreases or not.

To create this training data, we again generate all unique pairs of agents, as explained in Section 5.1. There are $\binom{n}{2}$ unique pairs in an MAPF instance with n agents, which means that an instance contributes $\binom{n}{2}$ entries to our training data. The next step is to find suitable features for the vector of input values. For this, we choose three different types of features, which are features of the instance, the agent pair, and both individual agents. We give an overview of our selected features in Table 5.2.

Instance Features. The features about the instance have their focus on the topology and how packed with agents the instance is. The first feature we choose is the overall number of agents in this instance. Since this feature in itself does not correctly represent if an instance is crowded or not, we choose another feature, which is the map size divided by the number of agents. The next feature is even further refined in this aspect and is the number of non-obstacle cells of the map divided by the number of agents. This feature represents how much space is available for the agents to maneuver around the map.

As mentioned by Sharon et al. [20], the benefit of merging agents depends on the conflict rate. On maps densely filled with obstacles the conflict rate is higher, as there are only few valid paths for the agents to take, which they have to share. Because of this, we also use the percentage of obstacle cells of the map as a feature. The problem with this feature is that it does not correctly reflect if the obstacles are placed in positions where they interfere with the agents' path or if they are clumped up along the walls. To get a better understanding of how the obstacles are placed, we analyze the individual optimal paths of the agents. We count the number of obstacles and the number of non-obstacle cells which are next to each cell of the agents' paths. This process is depicted in Figure 5.3. In this example, we only look at the four direct neighbors of each cell. We do this for every agent and sum up the number of obstacles and divide it by the sum of all non-obstacle cells. With this feature we get a more clear idea if the environment, which the agents have to navigate through, rather consists of narrow corridors or open spaces. We use three versions of this feature, which only differ in the number of neighboring cells we look at. For the *small* version we only consider the direct neighbors, for *medium* we look at a 3×3 rectangle placed around each cell of the paths, and for *large* we use a 5×5 rectangle.

Table 5.2: The features we use for our supervised machine learning process. The features are based on the instance, the individual agent, and each combination of two agents. Some of the features have three versions which differ in the size of the area which is relevant for the calculation. These features are marked with *(small, medium, large)*. *Small* means that we only look at the four direct neighbors of the current cell, which is shown in Figure 5.3. For *medium*, we use a 3×3 rectangle which has its center at the current cell as the area which we observe. This is the same for *large*, except for the size of the rectangle, which is 5×5.

Feature	Description
Instance	Number of agents
	Map size to number of agents
	Number of non-obstacle cells to number of agents
	Obstacles to map size
	Obstacles to non-obstacles along individual paths (small, medium, large)
Pair	Number of conflicts
	Closeness of individual optimal paths (small, medium, large)
	Distance of start cells
	Distance of goal cells
	Distance of cells of individual optimal paths every step
	Sum of path lengths
	Difference of path lengths
	Sum of obstacles along individual optimal paths (small, medium, large)
Single	Obstacles around start
	Obstacles around goal
	Obstacles along individual optimal path (small, medium, large)
	Length of individual optimal path

Agent Pair Features. These features describe how two agents are related to each other. The first feature we choose is the number of conflicts their individual optimal paths already have. The second pair based feature is about how close the individual paths are to each other. We do this similar as to how we count the obstacles along the paths. For each cell of the path of one of the agents in the pair, we look at all neighboring cells and count how many of them are part of the path of the other agent. Again we implement three versions of this feature, which differ in the number of neighboring agents looked at per cell. The next feature is the distance between their start cells. We use the Manhattan distance between these two cells to get a value for our feature. This metric does not factor in obstacles and thus may underestimate the real distance significantly on maps with obstacles. We refrain from using a more sophisticated distance calculation method, as these values need to be calculated for the model to make predictions as well. Thus, this calculation process also affects the runtime of the resulting algorithm.

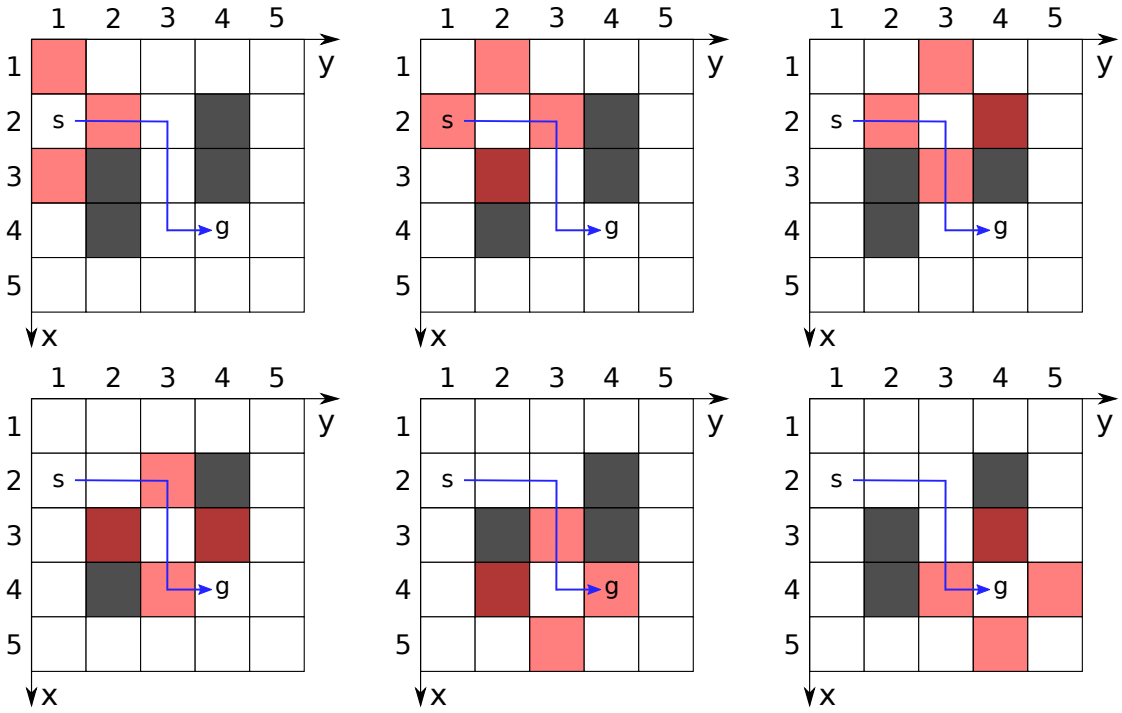


Figure 5.3: An example of how the number of obstacles along the individual optimal path of an agent is calculated. The agent starts at (2, 1) and moves to (4, 4) and its shortest path is indicated by the blue arrow. The four grey cells are obstacles. The agent occupies six different cells along its path. For each location we look at the direct neighbors, highlighted in red, and count how many cells are obstacles and how many are non-obstacles. In the starting position, the agent is located at the border of the map. Since it is not possible for the agent to move to the left, we consider it an obstacle. Overall, there are seven obstacle and 17 non-obstacle cells along its path.

Similar to the previous feature, we calculate the Manhattan distance between the goal cells of the agents. Another agent pair feature is how far they are away from each other at every step of their individual paths. As we already have the distance between their start cells, we do not need to recalculate it. Then, we find the distance between the cells to which the agents move at the next timestep. We do this for the whole paths and sum up the distances. If the paths are not of the same length, we calculate the distance between the goal cell of the shorter path and the cells of the other path, if the timestep is higher than the length of the shorter path. Then, to get our final value for this feature, we divide the sum of the distances by the length of the longer path. Other features are the sum of the path lengths of both agents and also the difference of their lengths. The last pair feature is based on the number of obstacles along their individual paths. For this we sum up the values, which we already calculated for the obstacles to non-obstacles instance feature, for the two agents of the pair. Again, there are three different versions of this feature.

Single Agent Features. As each entry in the training data is for an agent pair, each entry contains two values for each single agent feature. For the first feature we count the number of obstacles and the number of non-obstacles around the starting cell of the agent. We analyze the 3×3 area around the cell and then we divide the number of obstacles by the number of non-obstacles. The second feature is equally to the first one, but focuses on the goal cell of the agent. The third feature is the number of obstacles along the individual optimal path. This is already calculated for the instance feature where we do this for all agents. The last feature is the length of the agent's path.

To construct these features for a given instance we modify the CBSH2-WDG algorithm introduced by Li et al. [13]. The modified algorithm does not start the search process after the setup. Instead, it calculates the input values, writes them into a file, and subsequently terminates.

Output. The selected features are the input values for the learning process. But for this process to work, we also need a desired output value for each entry of the training data. This output value tells the model if the merge of an agent pair is beneficial or not. To calculate this value, we use a similar approach as described in the beginning of Section 5.1. There, we generate all $\binom{n}{2}$ pairs of an instance and then let ICBS(∞) solve it repeatedly, but with a different pair premerged each time. The result is a list of runtimes where we can compare the agent pairs. To get an output value for the training data, we add an ICBS(∞) run with no meta-agents formed at all to the comparison. Then, we can clearly see which merges result in a shorter runtime and thus are beneficial, and which are detrimental to the search process and should not be done. So the desired output value is *yes* if the runtime of the pair is shorter than the runtime of ICBS(∞) with no merged agents, and *no* otherwise.

5.2.2 Classification

Before we fit a model on the training data, we scale each feature column so all the values are between zero and one, where zero represents the lowest value of this column and one the highest. We use the same instance of this scaling operator when we classify agent pairs of unseen instances, so that the scaling of the features is consistent. This means that the feature values of new instances most likely will not have zeros and ones, as these correspond to the lowest and highest values seen by the model in the training process.

We choose two different types of machine learning models for our approaches. The first model is a support vector machine (SVM) and the second model is a multi-layer perceptron classifier (MLP).

In the case that there is a disproportionate large amount of one class of entries in the training data, the model would simply learn to always predict this class, as it is the right prediction in most cases. To avoid this, we balance our training data by randomly selecting an equal amount of training samples from both classes. For the learning process

of the SVM model, we take 20% of the balanced training data and use it as test data to evaluate the model after learning.

We also do this for the MLP model, but additionally we split off 25% of the remaining training data, which we use for the validation of the model. The training process of the MLP model involves repeatedly training a model from start, but each time with different hyper-parameters. Between each run, we evaluate the model on the validation data. Then, when we are satisfied with the hyper-parameters, we let it learn one last time on the training data and we evaluate it on the test data. Evaluating the model on the validation data would result in a biased outcome, as we tuned the parameters specifically for this data. Because of this, we evaluate it on the test data. This way we can correctly evaluate the model on data it has not seen during learning and for which the results are not biased.

After fitting the models we use them in our algorithm. As described earlier, to use the models to make predictions for arbitrary MAPF instances, we need to calculate the input values for each agent pair of this instance. Our modified CBSH2-WDG algorithm handles this. To get values that are consistent with the data the models learn on, we need to apply the same scaling process as we do in the training data generation. Then we load one of our trained models and let it predict for each agent pair the probability of belonging to the beneficial class. The next step is to pick the agent pair which is most likely to result in a beneficial merge. We then let ICBS(∞) solve the instance with this pair premerged.

Another strategy is to only perform this merge if the probability of it belonging to the beneficial class is above a certain threshold. We do this to avoid merging in instances where no merging at all is the best strategy. We choose a suitable value for our threshold by looking at the precision and recall values of our models. In our case, it is important that the positives we have are actually true positives, whereas misidentifying true positives as negatives is not as detrimental. This is because we aim to keep the number of merged agents low, as large meta-agents increase the complexity of the search quite substantially. Consequently, our focus is not on finding all positives. Choosing an adequate threshold is a trade off between precision and recall. Since we want to avoid merging agent pairs which are actually false positives, we prefer thresholds which result in high precision. After we find a suitable threshold, we use it to decide if merges should be performed or not. Again we choose the pair with the highest probability of belonging into the class which is beneficial if merged. But before we merge it, we check if the probability is above our threshold. Only if it is, we merge it. Otherwise, we perform no merge at all.

Similar as in Section 5.1, we again employ two different CBS implementations to solve the MAPF instances. In this case, the information about the instance already gathered by the feature measuring algorithm is lost, as we only extract the values of the feature. Combining them into a singular implementation would eliminate the need to perform the setup process of the CBS algorithm twice.

5.3 Heuristics for Meta-Agent Conflict Based Search

Merging agents into meta-agents is not the only enhancement proposed for CBS. One of these enhancements is the use of heuristics to guide the high-level search. Heuristics are applied to CBS in a similar manner as they are used in the A* algorithm. These heuristics give an informed estimate of how often the cost of a node will increase before reaching a goal node, if we were to resolve all of its conflicts.

This value is called the h-value of the node. The heuristics need to be admissible to guarantee that CBS will find an optimal solution. When this h-value is added to the current cost of the node, which is the g-value, we can calculate its f-value. With this f-value, we can compare the high-level nodes to each other in terms of their possible minimal goal cost. A smaller f-value allows for a smaller goal cost. Using this value, the best-first search of the high-level tree can effectively prioritize more promising nodes by skipping nodes with an f-value larger than the cost of an optimal solution and choosing the node with the smallest f-value.

Felner et al. [6] propose such a heuristic. The heuristic focuses on cardinal conflicts and uses the fact that the presence of a cardinal conflict in a solution implies that its cost has to increase by at least one to be a conflict-free solution. Then, using these cardinal conflicts, a conflict graph is built. Every agent present in a cardinal conflict is represented as a vertex and they share an edge if they are in the same cardinal conflict. With this conflict graph, an admissible h-value can be calculated. This is done by finding the number of disjoint pairs in cardinal conflicts, as these correspond to disjoint cardinal conflicts. This number is an admissible h-value for this high-level node, since resolving a cardinal conflict can not implicitly resolve another disjoint cardinal conflict, as only the paths for agents present in the original conflict are replanned. What follows is that the size of a minimum vertex cover of the conflict graph is an admissible h-value.

In Figure 5.4 a conflict graph we encountered in one of our experiments is shown. It consists of nine agents which are in cardinal conflicts. Each node holds the identifier of the agent it represents. If the node represents a meta-agent, it holds a list of identifiers. Also, the nodes highlighted in green represent a minimum vertex cover of this conflict graph.

Because finding a minimum vertex cover is also an NP-hard problem, Felner et al. use another property of CBS to reduce the computational time needed to find this h-value. The fact that in CBS only one of the paths is replanned while generating a child-node means that in its conflict graph also only edges incident to one vertex can be added or removed. Thus, the size of the minimum vertex cover of a child-node can only be one smaller, exactly the same, or one larger than the minimum vertex cover of its parent. This reduces the problem of finding a minimum vertex cover to the problem of checking if a graph has a vertex cover of given size, which is an easier to solve problem. The time needed to solve this problem is determined by the size of the vertex cover k and the number of nodes n in the graph. Chen et al. [4] propose algorithm which is able to determine whether a vertex cover of size k exists in a graph in time $O(1.2738^k + (k \cdot n))$. But

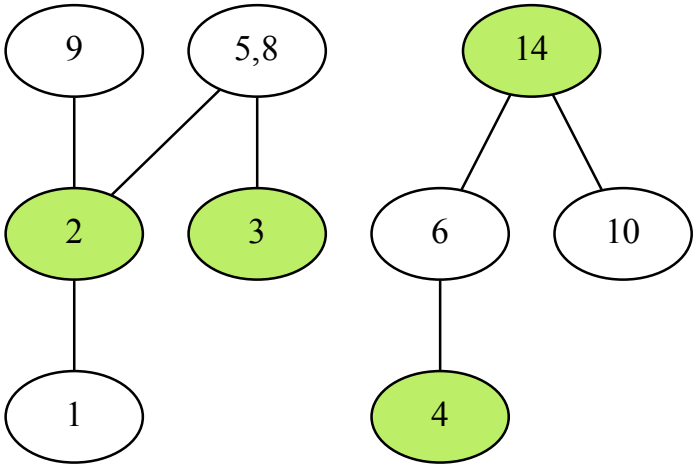


Figure 5.4: A conflict graph consisting of nine (meta-)agents. The nodes highlighted in green are an example of a minimum vertex cover of this graph.

since we deal with rather small graphs and thus also small vertex cover sizes, algorithms which are not as sophisticated are sufficient. For the implementation of the heuristic, the algorithm proposed in [5] which solves the problem in time $O(2^k \cdot n)$, is used.

For CBS without meta-agents the classification of conflicts is done using Multi-Valued Decision Diagrams (MDD) as proposed by Sharon et al. [19]. These MDDs are directed acyclic graphs and used in CBS to represent all the possible paths an agent can take to reach its goal in a specific time. Each node in the MDD represents a location and an arc represents a move from one location to another. The level of the node corresponds to the timestep at which the agent occupies the location. The MDDs disregard the position of other agents but do factor in previously generated constraints for the agent. To check if a conflict between two agents is cardinal, their MDDs are analyzed. If both MDDs are of width one at the level which corresponds to the timestep of the conflict, the conflict must be cardinal, since there is no way for them to avoid it with these exact path costs.

To use this heuristic in ICBS with meta-agents present the conflict classification technique needs to be adapted, since the MDD approach is only designed for single agents. For this, we propose two different techniques. The first technique is based on the look ahead concept [1] and the second technique is an adaption of the classification using MDDs. These techniques are used to classify conflicts where meta-agents are involved. Since meta-agents are treated the same as singular agents by the high-level search, the conflict graph and the minimum vertex cover parts of the heuristic need not be adapted.

5.3.1 Look Ahead

The idea of prioritizing cardinal conflicts in the high-level search is introduced by Boyarski et al. [1]. They achieve this by classifying conflicts with MDDs. Since this is not possible for conflicts where meta-agents are present, they skip those conflicts at first and revisit them only if no cardinal conflict without meta-agents is found. Then, they simulate the expansion of the current node on this conflict, a 1-lookahead. If the solution cost of both of the children increases, the conflict is cardinal.

We make use of this lookahead to derive heuristics for meta-agents, which is based on the idea of conflict classification. If a conflict involving a meta-agent is encountered in the conflict graph building phase of the heuristic, our adaption of this look ahead technique is invoked. The same way as the CBS algorithm would handle a conflict, we resolve it by generating the two corresponding constraints. The first constraint is added to the left child and we invoke the low-level solver to find a new path for the newly constrained agent, which may even be a meta-agent. A change in the agents' path may also entail a change in their path costs. If we observe no increase in cost, we know for certain that the conflict cannot be cardinal and we go on to classify the next conflict. Otherwise, if there is an increase in cost, we build the right child and again invoke the low-level solver to replan for it. Now we again check if the cost increased. Same as before, if the cost stayed the same, the conflict can not be cardinal and we move on to the next conflict. On the other hand, if there is an increase, we found a cardinal conflict, since resolving the conflict increased the path costs in both children. Thus, the agents present in the conflict can be added to the conflict graph.

5.3.2 MAMDD

Our second approach is to classify the conflicts using MDDs which are designed for meta-agents of size two, called MAMDDs. We restrict our implementation to meta-agents of size two, since larger agents would generate MAMDDs of drastically larger size. Like MDDs for single agents, MAMDDs consist of levels which correspond to the timestep and these levels contain nodes which represent locations. The difference between MDDs and MAMDDs lies in the nodes. In the MAMDD, the nodes contain locations for both agents, since the meta-agent occupies two vertices at once. This means that the MAMDD contains for each timestep all the possible combinations of locations the two agents can occupy, while still reaching their targets in a specified cost. We refer to this cost as the *target cost* and it is the sum of the individual current path costs of the two agents.

Figure 5.5 depicts an example of an MAPF instance with an MAMDD. The MAMDD is constructed for agent one and agent two with a combined target cost of three. Agent one starts at field A1 and its goal is at B1. Agent two moves from A2 to B3. In the root node at level zero, both agents are at their starting position. At level one, there are two different nodes, because agent two also has two different optimal paths to its goal. In the left node, agent one moves to B1 and agent two moves to A3. In the right node, agent one again moves to B1, since it is its only viable option, and agent two moves to B2.

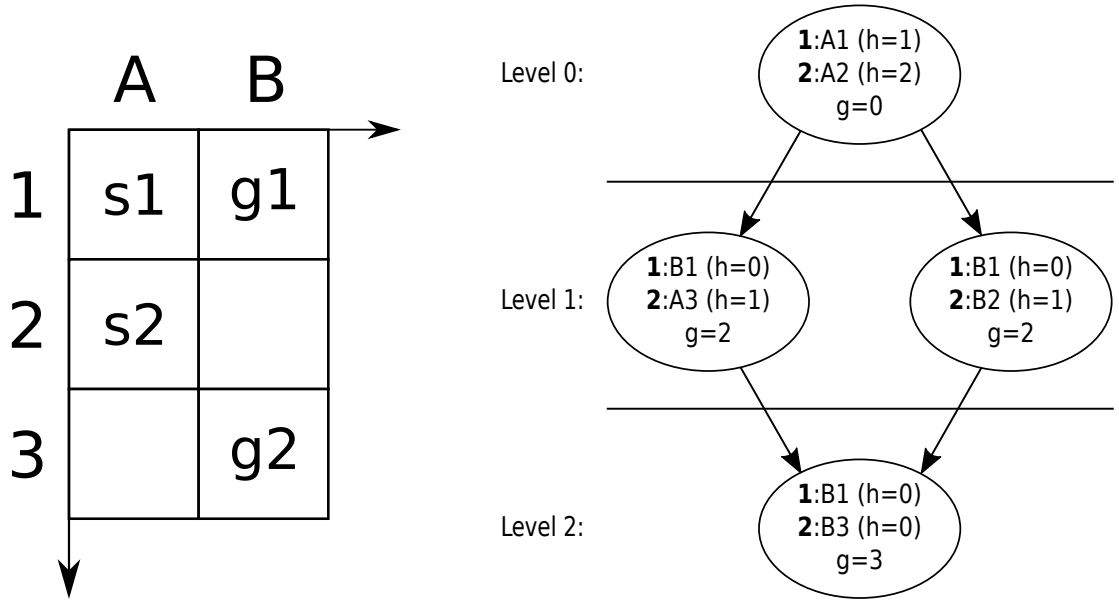


Figure 5.5: A small MAPF instance with an MAMDD built for the agents one and two.

The last level again only contains one node, which is also the goal node. Both agents move to or wait at their respective goals. The relevance and the calculation process of the heuristic values will be explained later.

We treat this meta-agent again exactly as single agents, which means that we focus only on the combined path costs of the agents while we build the MAMDD. It is important that we add no restrictions to the costs of the individual paths, they can be as long or as short as needed, as long as their combined cost is exactly the combined target cost. The only restrictions to the paths are previously generated constraints, obstacles, or collisions with other agents that are part of this meta-agent. After determining the target cost, we need to collect the already existing constraints for the two agents. Then, we can build the MAMDD for the meta-agent using the starts and goals of the two agents, their combined target cost, and their constraints.

Construction The MAMDD construction process is described in Algorithm 5.1. We start by building the root node of the MAMDD. This node consists of the starts of the two agents, since, at timestep zero, they have not yet moved. We add the root node to the first level and we enter the main loop, where we iteratively fill the remaining levels. The number of levels is known in advance and can be at most equal to the target cost. This is the case if one agent starts at its target location and stays there terminally, which results in an individual path cost of zero. Thus, the other agent needs to move exactly as many steps as the target cost. Therefore, the number of levels can be set to the target cost of this MAMDD. If not all levels are needed for the full MAMDD, the remaining levels are filled with nodes where both agents stay at their goals. The induced

Algorithm 5.1: Construction process of an MAMDD.

```
1  root ← newNode(StartA, StartB)
2  levels[0].add(root)
3  for i ← 0 to levels.count do
4      seenNodes ← new HashSet()
5      foreach node ∈ levels[i] do
6          children ← getAllChildren(node)
7          if children.count == 0 then // no feasible path
8              toDelete.add(node)
9          foreach child ∈ children do
10             if not seenNodes.contains(child) then
11                 levels[i + 1].add(child)
12                 seenNodes.add(child)
13         end
14     end
15     foreach delNode ∈ toDelete do
16         levels[i].remove(delNode)
17         deleteIfChildless(delNode.Parent, i − 1)
18     end
19 end
```

computational overhead of filling the remaining levels is negligible.

Then we iterate over all the nodes of the current level and generate their children. At the first level there is only the root node, since their only possible locations can be their start points. To generate the children of a node, we look at all combinations of moves the agents can make which also are subject to certain criteria. This child generation process will be described in more detail below. If a node does not have any viable children, it will be put into a separate list which keeps track of all the nodes that need to be deleted. Childless nodes need to be deleted because there is no feasible path for them to reach a goal node anymore. For a node to be a goal node, both of its moves need to have the individual goals of the agents as targets and its cost has to be the same as the target cost. We delete the childless nodes in a separate loop later.

On the other hand, if a node does indeed have viable children, all the children are added into the level below, since the children are one timestep ahead in the path of the meta-agent and thus belong into the next level. Then, after iterating over all nodes in this level, we delete the nodes which are childless. This is done by removing it from the level and decreasing the count of children of its parent by one. Additionally, we need to recursively check if its ancestors need to be deleted as well. This is the case if the deletion of a node results in a childless parent, since this now removed child was the only possibility for the parent to reach a goal node.

Algorithm 5.2: Recursively deletes nodes that have no valid path to any goal node.

```
1 Function deleteIfChildless(node, level):
2     if node.numOfChildren == 0 then
3         levels[level].remove(delNode)
4         deleteIfChildless(node.parent, level − 1)
```

Node Deletion To handle this recursive deletion process, we added the function *deleteIfChildless*, which is described in Algorithm 5.2. The function takes as input a node and a level, which will be the parent of the just deleted child and the child's level minus one. Then the function checks if the received node is childless. If not, the recursion stops, since this node may still be able to reach a goal node and not deleting this node results in no change in the number of children of its ancestors. Otherwise, if it is childless, it is removed from the level. Afterwards, *deleteIfChildless* is again called with its parent and the level minus one.

Algorithm 5.3: Returns all next move combinations of the node fitting certain criteria.

```
   Input: MAMDDNode node
   Output: List<MAMDDNode> children
1  Function getAllChildren(node):
2      children ← newList()
3      foreach moveA, moveB ∈ node.nextMoves() do // 25 possibilities
           // obstacles
4          if not (moveA.isValid and moveB.isValid) then
5              continue
           // constraints
6          if not (moveA.isAllowed and moveB.isAllowed) then
7              continue
8          if moveA.collides(moveB) then
9              continue
10         child ← newNode(moveA, moveB)
11         if child.f > targetCost then
12             continue
13         child.Parent ← node
14         children.add(child)
15     end
16     return children
```

Children Generation For the generation of the children of a node, we designed the function *getAllChildren*. Algorithm 5.3 shows a description of the function which takes

as an input a node and returns a list of all child-nodes which fit certain criteria. The function iterates over all possible children and only adds them into the list it returns if they pass all checks. In our MAPF setting, each agent has five possible moves at each timestep. It can move north, east, south, west, or stay at its position. Since we have two agents, there are a combined 25 possibilities of moves. Simply adding all the children to the next level is not feasible since the overall number of nodes to expand would increase exponentially in the depth of the MAMDD, as every child again would generate 25 new possible children. To counteract this increase, we perform some sanity checks. With these checks, we remove children which do not fit certain criteria. In line 4 the first check is done. We require both moves to be a valid move, with means they are not allowed to collide with obstacles of the map. The next check in line 6 verifies if the agents are allowed to perform these moves. There could be constraints imposed by the high-level search on the paths of the agents which prohibit them from performing certain moves. In line 8 we check if the two moves collide with themselves. Then, we construct a new node from the two moves and perform a last check. With this last check in line 11 we remove moves which have no way of reaching a goal node in the target cost.

We do this by calculating the f-value of the node and only keep the node if this value is smaller then the previous calculated target cost. This f-value is the sum of the g-value of the node and the individual h-values of both locations. The h-values are calculated in a preparatory step before the whole high-level CBS search even starts. A breadth-first search is performed for every agent. The search starts at the agent's goal location and searches the shortest path to every other location on the map. The length of the paths are saved for every location and agent. These path lengths are then used as a suitable heuristic for the children generation process.

The g-value is inherited from its ancestor but needs to be increased by a certain amount. The root node starts with a g-value of zero, since no moves have been made at the first level. Then, the children of the root node have g-values of two, since we have to account for both the moves of both agents. This pattern repeats until at least one agent reaches its goal. In our chosen setting, an agent which terminally resides at its goal counts as finished and the wait move after reaching its goat does not contribute to the sum-of-costs. This means that the g-value of a node actually increases by the number of agents not yet finished. This is illustrated in Figure 5.5. The g-value of the root node is zero, but the h-values of the two moves are one and two. In level one, both nodes have a g-value of two, as both agents moved. Since agent one is at its goal, it has an h-value of zero. Also, because agent one is already waiting at its goal, the g-value of the final node in level two only increases by one.

We also have to take into account that an agent may need to leave its goal again after reaching it earlier to make way for other agents. This is likely if the agents goal is in a narrow corridor. Since the path cost increases until the agent terminally stops at the goal, we have to consider this case while calculating the g-value. To do this, we add counters to keep track of how many timesteps each agent already stayed at its goal. Then, if a move leads to an agent leaving its goal, the wait time at the goal we previously assumed

to have no impact on the sum-of-costs, should have actually been added to it. Because of this, we add this counter to the current g-value and reset the counter to zero. The h-value of an agent's move is the length of the shortest path of the move's coordinates to the agent's goal. These path lengths are precomputed for every agent and every location of the map using a breath-first search implementation. If all checks are passed, we add the node to the list of children to be returned.

Algorithm 5.4: Determines if two meta-agents are in a cardinal conflict at given timestep using their MAMDDs.

```
1  Function isCardinal(firstMAMDD, secondMAMDD, t):
2      foreach firstA, firstB ∈ firstMAMDD[t] do
3          foreach secondA, secondB ∈ secondMAMDD[t] do
4              if firstA.collides(secondA)
5              or firstA.collides(secondB)
6              or firstB.collides(secondA)
7              or firstB.collides(secondB) then
8                  continue
9              return false
10         end
11     end
12     return true
```

Collision Check With the MAMDDs we can now determine the cardinality of conflicts involving meta-agents. We have to differentiate between two possibilities regarding the conflicting agent. This can be either also a meta-agent or a single agent. If both conflicting agents are meta-agents, we have to build a second MAMDD. Otherwise, since the other agent is a single agent, is is enough to build an MDD for it. Then, the conflict classification is similar in both cases. In Algorithm 5.4 the process of classifying a conflict between two meta-agents is described. The function takes the MAMDDs of both agents and also the timestep at which the conflict occurs, which indicates the level of the MAMDDs we need to look at.

We form node pairs by combining every node from the selected level of the first MAMDD with every node from the same level of the second MAMDD. Then, we check each node pair for collisions. Collisions happen if any move of a node collides with any move of the other node. This can be either a vertex conflict, where the agents occupy the same vertex at the same time, or an edge conflict, where they move along the same edge at the same time. If there exists such a node pair that does not collide, we know that the conflict is not cardinal, since this set of moves is a valid option for all agents and therefore, there exist paths with the same costs which avoid the conflict we want to classify. Otherwise, if there is a collision, we check the next pair. If we find collisions in every pair, the conflict must be cardinal, as there are no possibilities with these path costs for the agents to move without colliding with each other. The only difference between this cardinality

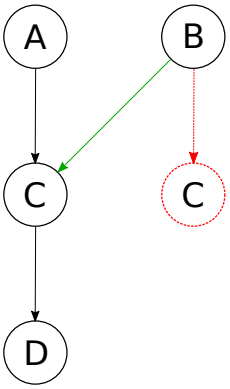


Figure 5.6: An abstraction of an MAMDD building process. Node A and node B generate nodes that are equal, which we refer to as node C. Node C generates one child, which is node D. Because node C was already encountered when generated by node B, it is not added to the MAMDD a second time, which is highlighted in red. To keep an upwards link for the recursive deletion process, we add the node B to the list of additional parents of node C, which is represented by the green arrow.

check when using an MAMDD and an MDD lies in the number of collision checks we need to perform. This is because a node of an MDD only contains one move. So for each node pair we only check if the single agent move collides with either the first or the second move of the meta-agent.

Enhancements We also propose two enhancements to our MAMDD heuristic. The first enhancement is a rather common method of reducing computational work by caching previous results. Instead of building the MAMDDs new every time there is a conflict, we first check if we already build an MAMDD with the same properties. The important properties, which we use to decide if we can reuse an MAMDD, consist of the indices of both agents, the combined target cost, and the set of constraints that are imposed on either of the two agents. We maintain a dictionary with all the previously built MAMDDs where the keys are tuples consisting of the four mentioned properties.

Our second enhancement targets the building process of the MAMDDs. When constructing an MAMDD, it is possible to come across duplicated nodes. These duplicated nodes are defined by having identical moves, parents' moves, and g-values. This enhancement is described by the red lines in Algorithm 5.1. For each level we visit, we maintain a set of nodes which we already encountered once on this level. In line 10, instead of adding all children to the next level, we first look for this node in our set of seen nodes. If the set already contains this node, we do not add it to the next level, as it would only lead to duplicated computational work. Not only the number of nodes in the next level is reduced, but also its children in future levels are never generated. Otherwise, if we have not seen this node at this level, we add it to the next level and also to our set of seen nodes. When using this technique, the deletion mechanism needs to be adapted.

Previously, upon node deletion, we recursively check its ancestors. But with this enhancement, there can be nodes which have valid children but none of them are kept, since they are already seen nodes. This is illustrated in Figure 5.6. Node B has a child C that is equal to the child of node A. Because this node is already part of the MAMDD, we will skip it this time. Node C also has a child, which is node D. If we later decide that it is necessary to remove node D, we recursively delete nodes D, C, and A, but since we have no upwards link to node B, it will not get deleted. To counteract this, each node maintains a list of additional parents. If we expand a node and one of its child is not added because it is already seen, we retrieve its duplicate from the set and add the current node to its list of additional parents. This is illustrated by the green arrow connecting node B to the original C node. Then, if a child is deleted, we not only check its original ancestors, but also call the *deleteIfChildless* function on all additional parents. Therefore, node B gets correctly deleted as well.

CHAPTER 6

Computational Study

In this chapter, we evaluate our proposed solution approaches on a selection of maps. First, we describe the maps we choose for the study. Secondly, we perform experiments with our pairwise symmetry merging strategies and analyze the results. We then evaluate our merging approach based on machine learning. Lastly, we also study the effects of our proposed heuristics.

We conduct this computational study on the cluster of the Algorithms & Complexity Group at the TU Wien. We use Intel Xeon E5540 processors with a memory limit of eight GB. To compare our solutions to their base version and state of the art algorithms we use ICBS [1], CBSH2-WDG-R [13], and CBSH2-RTC [15]. ICBS[1], which is able to handle meta-agents, is implemented in C#, whereas CBSH2-WDG-R[2] and CBSH2-RTC[3] are implemented in C++. Our approaches are implemented in Python 3.9.1. For the pairwise symmetry detection we use the CBSH2-RTC implementation except for semi-cardinal rectangle conflicts, which we detect with the CBSH2-WDG-R implementation. To solve the instances with preformed meta-agents, we use the ICBS implementation.

As introduced by Sharon et al. [20], we will also use the notation ICBS(B) to represent the ICBS algorithm with a conflict threshold of B. This means, ICBS will merge two agents into a meta-agent once they conflicted B times. Consequently, ICBS(∞) corresponds to the non-meta-agent version of ICBS. If not stated otherwise, we use both the *merge and restart* and the *prioritize conflicts* improvements introduced by Boyarski et al. [1]. In particular, we use ICBS(25) as the dynamic merging algorithm for our comparisons. We use 25 as the conflict threshold as this is also the value used by Boyarski et al. [1] for their final comparison.

[1] https://github.com/eli-b/mapf
[2] https://github.com/Jiaoyang-Li/CBSH2
[3] https://github.com/Jiaoyang-Li/CBSH2-RTC

In most related works regarding CBS based algorithms, the runtime of the algorithms and the number of high level expanded nodes are used for comparison. The expanded nodes evaluation criterion should be taken with a grain of salt in the studies where algorithms with different merging strategies or different heuristics are compared, because the amount of high level nodes expanded can differ greatly based on the number of agents merged, likewise the effort to calculate the heuristics. This is because meta-agents can substantially increase the complexity a single high-level node can have. Therefore, computational work is shifted from the search in the high-level tree to the low-level solver, which results in a lower number of expanded nodes. The extreme case would be to merge all agents into a single meta-agent. Then, the search would finish with exactly one node generated. This is because this meta-agent represents effectively the whole MAPF instance and thus, a valid path assignment for this meta-agent also represents a valid solution for the whole instance.

To create maps and instances we use the CBSH2-WDG code base. This implementation is capable of generating new MAPF maps and instances. We create two maps, both being 20 by 20 4-connected grids. The first map *empty* is a plain grid without any obstacles. The second map *dense* in contrast contains obstacles, with 30% of its 400 cells being obstacles. These obstacles are placed randomly while also guaranteeing connectedness, i.e., that no placed obstacles cuts off a non-obstacle cell. Every non-obstacle cell has to be able to reach every other non-obstacle cell. We also use the CBSH2-WDG code base to generate instances for these maps. For both maps we chose different numbers of agents. For the dense map, we generated instances with 10, 15, 16, 18, 20, 22, and 24 agents. For the empty map, we generated instances with 10, 20, 30, 40, 50, 60, 70, 75, 80, 85, 90, and 95 agents. We generated 50 instances for each map and each number of agents.

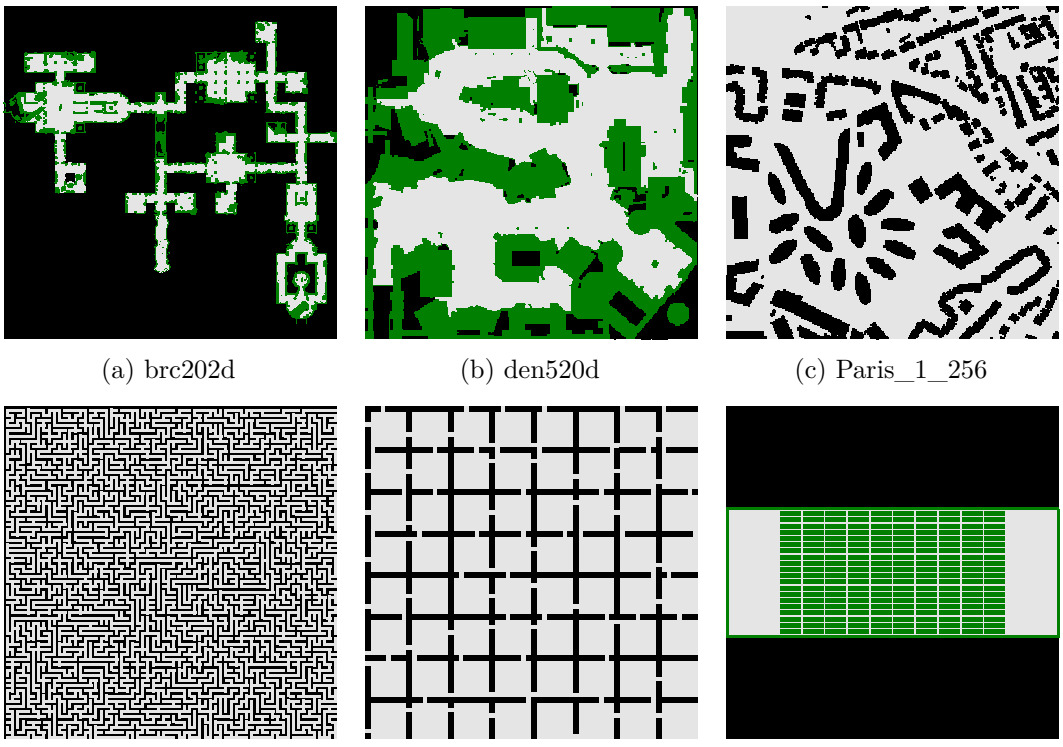
We also use instances which are based on real world scenarios. These instances are not created by us but rather taken from the online repository for benchmark instances for MAPF[4], as recommended in [22]. Our aim is to choose maps which differ greatly in their topology, as we want to test our solutions on a variety of maps. Also, to make future comparisons more convenient, we use the same selection of maps and instances as in [15]. Therefore, additionally to the *empty* and *dense* grids, we select *brc202d*, *den520d*, *Paris_1_256*, *maze-128-128-1*, *room-64-64-8*, and *warehouse-10-20-10-2-1*. We will refer to these maps as empty, dense, brc202d, den520d, paris, maze, room, and warehouse. These real world scenario maps are depicted in Figure 6.1. We also use the same six different numbers of agent per map as in [15]. These numbers are shown in Table 6.1. For these maps, we use 25 instances for each agent number. The number of agents is chosen in a way so most of the instances with the lowest count of agents are solved comfortably by the algorithms in our specified time limit. In contrast, only a small amount of the instances with the highest number of agents should get solved in time.

brc202d is a large map but most of it is covered with obstacles. It represents rather small rooms with corridors connecting them. den520d is also rather large and consists of many

[4]https://movingai.com/benchmarks/mapf.html

Table 6.1: Information about empty, dense, and the six chosen maps which are based on real world scenarios. Each real world map has six different number of agents and 25 instances for each combination of map and agent number.

Map	Agents	Size	Obstacles
empty	10 - 95	20×20	0 %
dense	10 - 24	20×20	30 %
brc202d	20 - 70	530×481	83 %
den520d	40 - 140	256×257	57 %
Paris_1_256	30 - 180	256×256	28 %
maze-128-128-1	3 - 18	128×128	50 %
room-64-64-8	15 - 40	64×64	21 %
warehouse-10-20-10-2-1	30 - 130	161×63	44 %



(a) brc202d (b) den520d (c) Paris_1_256

(d) maze-128-128-1 (e) room-64-64-8 (f) warehouse-10-20-10-2-1

Figure 6.1: A depiction of the six maps based on real world scenarios taken from the repository for MAPF instances as described in [22].

open spaces. Paris is similar to den520d and is mostly open space, but with obstacles scattered throughout the map. Maze in contrast only consists of corridors which only allow for one agent at a time to pass through and thus has no open spaces. Room has many small rooms, which are open spaces, but the passages which connect them are only one space wide. The last map, warehouse, is designed to resemble the layout of an actual warehouse, with narrow corridors separated by blocks of obstacles.

We use the runtimes of our selected algorithms on these instances to calculate their success rates. The success rate is the percentage of instances solved in a given time frame for each category, where a category is defined by the map and the number of agents.

6.1 Evaluation of Pairwise Symmetries Based Merging Technique

In this section, we will study the impact merging agent pairs in pairwise symmetry conflict has on the performance of ICBS. We will look at different strategies for pair selection and we will also test our algorithms on different types of maps. This section is split into three subsections, which correspond to the three types of pairwise symmetries found by Li et al. [15]. Consequently, the first subsection investigates the benefits of merging pairs in rectangle conflicts, the second subsection focuses on target conflicts, and in the third subsection we will form meta-agents from pairs in corridor conflicts.

6.1.1 Rectangle Conflicts

We will start by comparing our different selection strategies on the empty map. In Figure 6.2 our selection strategies are compared to each other and in Figure 6.3 they are compared to ICBS(∞), ICBS(25), CBSH2-WDG-R, and CBSH2-RTC. The comparison uses the success rates of the algorithms on the empty map and we use a time limit of two minutes.

We show the success rates on the empty map of our pair selection strategies which are based on rectangle conflicts in Figure 6.2. Since they are all able to solve all instances with ten and 20 agents in the given time limit, there are no differences in their success rates. This low variance in the success rates on this small instances is expected, since a low number of agents also mean a low number of conflicts. Thus, rectangle conflicts are also less likely to occur. If there is neither a cardinal nor a semi cardinal rectangle conflict, our selection strategies are unable to pick a pair to merge. Therefore, the instance is solved without merging, which reduces our algorithms to an ICBS(∞) algorithm. But as more agents are added, more rectangle conflicts occur, which lets our selection strategies take effect.

The performances of rect_largest and rect_all_semi_gr_40 clearly fall off early and do not recover with higher agent numbers compared to the others. This could be attributed to the fact that both have rather limited merging options, with rect_largest merging at most one pair and rect_all_semi_gr_40 merging only pairs in conflicts with rather large

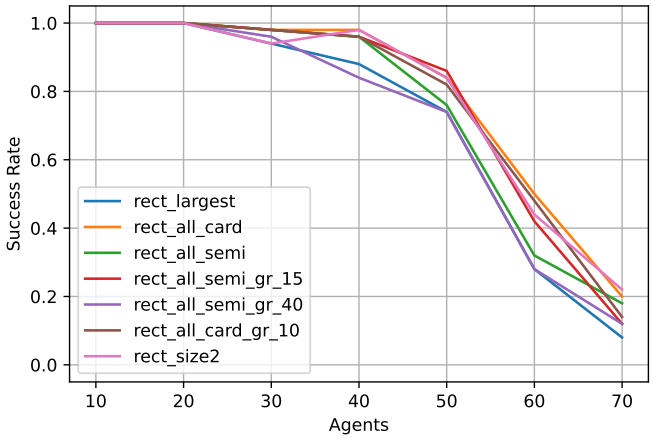
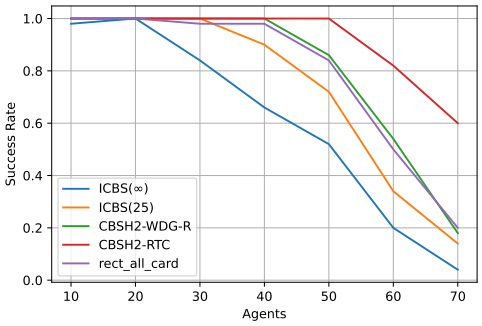
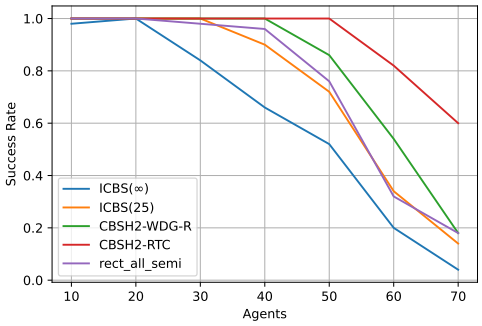


Figure 6.2: Success rates of the pair selection strategies based on rectangle conflicts on the empty map.



(a) Success rate of rect_all_card compared to ICBS and CBSH2 algorithms.



(b) Success rate of rect_all_semi compared to ICBS and CBSH2 algorithms.

Figure 6.3: Success rates of ICBS, CBSH2, and our pair selection strategies on the empty map.

rectangles. Therefore, some rectangle conflicts, which are detrimental to the solving time, go unhandled. rect_all_card, rect_all_card_gr_10, and rect_all_semi_gr_15, seem to generally outperform the others, with rect_all_card being the strongest by a small margin. Although rect_size2 has some problems with the 30 agents instances, it is still one of the top performing strategies on the instances with more than 30 agents.

From the comparison to our reference algorithms in Figure 6.3 we can see that there is one instance with ten agents which only ICBS(∞) is unable solve, while all other algorithms manage a 100% success rate. Since it is unusual for an advanced algorithm like ICBS(∞) to not be able to solve this rather trivial instance, we investigate it. With the help of the pairwise symmetry detection of CBSH2-RTC, we learn that there are two cardinal rectangle conflicts present in this instance. One of these conflicts has a

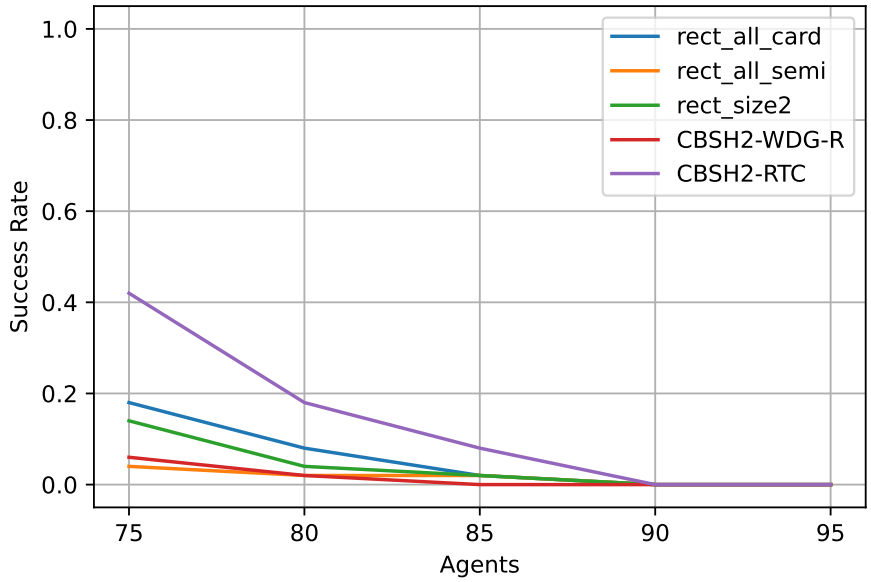


Figure 6.4: Success rates of rect_all_card, rect_all_semi, rect_size2, CBSH2-WDG-R, and CBSH2-RTC on the empty map with high agent density.

rather small rectangle area with a size of 22 while the other conflict's rectangle is of size 56. To solve this instance in less than two minutes it is necessary to handle the larger rectangle conflict by either using constraints like in CBSH2-WDG-R and CBSH2-RTC, or by merging the two conflicting agents into a meta-agent. ICBS(25) does this after they conflicted 25 times, but ICBS(∞) does not and thus, is unable to solve it in time.

In Figure 6.3a we show the success rate of the rect_all_card pair selection strategy compared to the success rates of our reference algorithms. Clearly, rect_all_card outperforms both ICBS variants overall. This strategy also yields similar results as CBSH2-WDG-R. A better success rate at the instances with 70 agents might hint at a overall better handling of high agent number instances. However, it is still not able to reach the same success rates as CBSH2-RTC, which also takes target and corridor symmetry conflicts into account. Figure 6.3b also depicts the success rates of our reference algorithms, but this time with the addition of our rect_all_semi selection strategy. The success rate of rect_all_semi falls of earlier than CBSH2-WDG-R and is more similar to the success rate of ICBS(25). However, similar to rect_all_card, rect_all_semi also performed quite well on the instances with 70 agents.

We further investigate the capabilities of rect_all_card, rect_all_semi, and rect_size2 to handle instances with high agent densities. For this, we use the instances we created with 75, 80, 85, 90, and 95 agents. Again, for each agent number we let the agents solve 50 instances. Since these instances are more complex and thus harder to solve, we set a time limit of five minutes. In Figure 6.4 the success rates of our three selection strategies rect_all_card, rect_all_semi, and rect_size2 are compared to those of CBSH2-WDG-R

Table 6.2: For the empty map, we show for each agent number how many of the initial states of the instances have at least one cardinal rectangle conflict, and also in how many either a cardinal or a semi-cardinal rectangle conflict is present.

Agents	10	20	30	40	50	60	70
Cardinal	6/50	14/50	25/50	39/50	47/50	48/50	47/50
Cardinal/Semi-Cardinal	8/50	32/50	45/50	48/50	50/50	50/50	50/50

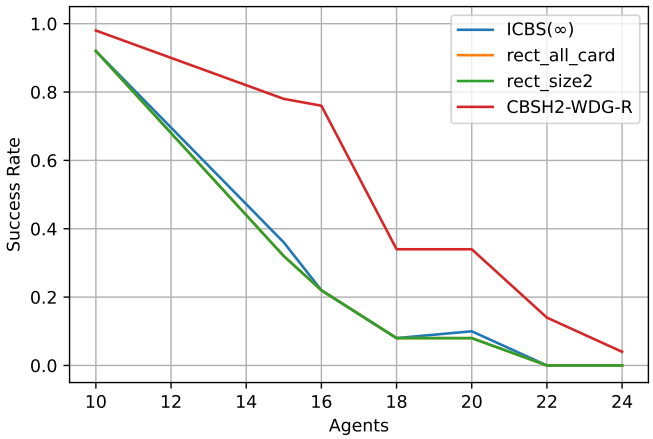


Figure 6.5: Success rates of ICBS(∞), rect_all_card, rect_size2, and CBSH2-WDG-R on the dense map. The success rate of rect_all_card is hidden behind rect_size2, as they performed equally.

and CBSH2-RTC. rect_all_card is able to solve a few more instances compared to CBSH2-WDG-R, but is still outperformed by CBSH2-RTC. rect_all_semi on the other hand is not able to substantially perform better than CBSH2-WDG-R. The performance of rect_size2 is similar to rect_all_card but with a few instance solved less. While the success rate on the instances with 85 agents is already low for all four of the algorithms, they are not even able to solve any of the 90 and 95 agents instances in the given time limit.

To find out in how many instances our merge strategies find suitable candidates, we analyze the initial states of our instances for the empty map. The results are shown in Table 6.2. In the first row of the table we show for each number of agents how many of the instances already start with a cardinal rectangle conflict present. As the number of agents per instance grows, it also gets more likely for a cardinal rectangle conflict to occur. Because of this, most of the instances with at least 50 agents have a cardinal rectangle conflict in their initial state. As expected, there are even more instances which have at least one cardinal or at least one semi-cardinal rectangle conflict. Almost every instance of the empty map with at least 30 agents meets this criteria.

On the dense map the merge strategies based on rectangle conflicts do not provide as

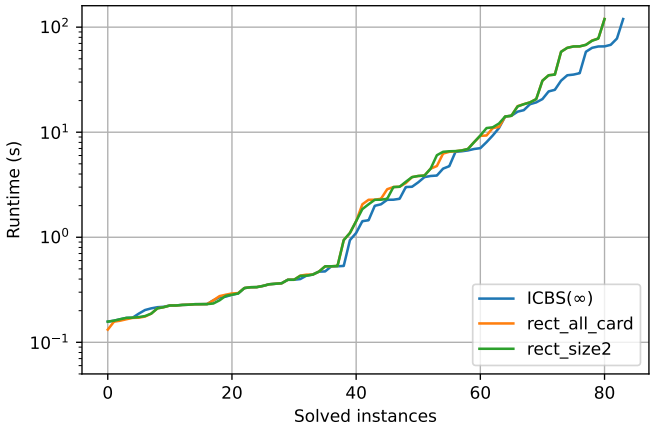


Figure 6.6: Runtime distributions of ICBS(∞), rect_all_card, and rect_size2, for the dense instances, where the x-axis represents the number of solved instances and the y-axis the time needed to solve this amount of instances.

much improvement as on the empty map. This can be seen in Figure 6.5, where we show the success rates of ICBS(∞), rect_all_card, rect_size2, and CBSH2-WDG-R for the dense map. CBSH2-WDG-R performs by far better than ICBS(∞) or any of our merging strategy, since it also uses the strong WDG heuristic. Both merging strategies performed equally in terms of success rates and thus only the line for rect_size2 is visible. Not only is there no difference between the two merging strategies, there is also also little difference compared to ICBS(∞), with ICBS(∞) even performing better in some categories.

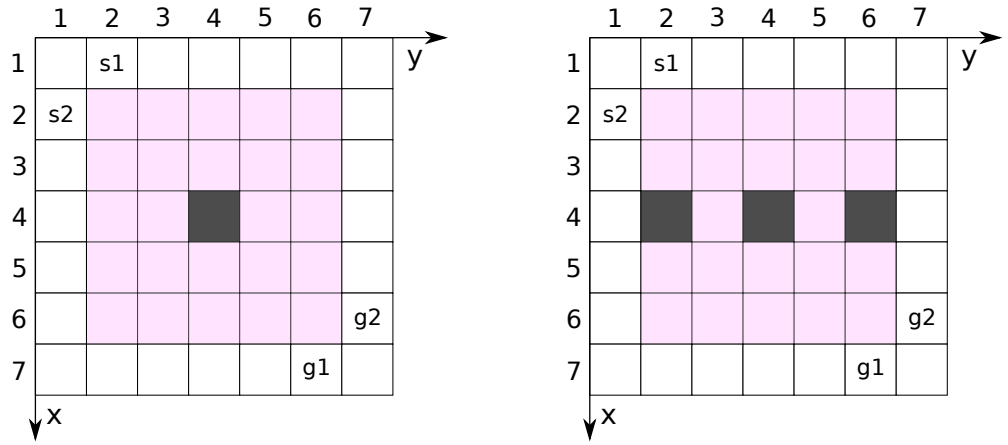
Similar results can be observed in Figure 6.6, where we show the runtime distribution of ICBS(∞), rect_all_card, and rect_size2 for the dense instances. Also visible is that the merging strategies not only solve less instances, but also provide no noticeable speed up in most instances they manage to solve. This can be attributed to two factors, which are the low number of agents and the relative high number of obstacles.

As mentioned earlier, with a low number of agents on the map, it is also less likely for agents to be in cardinal rectangle conflicts. As instances on the dense map are harder to solve than the instances on the empty map, we only test on instances with at most 24 agents. In Table 6.3 we show the number of merges performed by the rect_all_card strategy summed up for each number of agents for both maps. Since the maximum number of agents for the dense map is 24, the number of merges performed by our strategy is accordingly low overall. Thus, rect_all_card behaves equally to ICBS(∞) in a sizable portion of all dense instances.

To show why the rectangle merge strategies are not as beneficial on the dense map as they are on the empty map, we provide two examples of cardinal rectangle conflicts. In Figure 6.7a two agents are in a cardinal rectangle conflict. Right in the middle of the rectangle at (4, 4) is an obstacle cell, which the agents are not allowed to pass through. Without

Table 6.3: The sum of the merges performed by rect_all_card for each number of agents for the empty and dense maps with 50 instances each.

Empty		Dense	
Agents	Merges	Agents	Merges
10	7	10	6
20	16	15	19
30	44	16	11
40	75	18	38
50	105	20	40
60	175	22	35
70	209	24	46

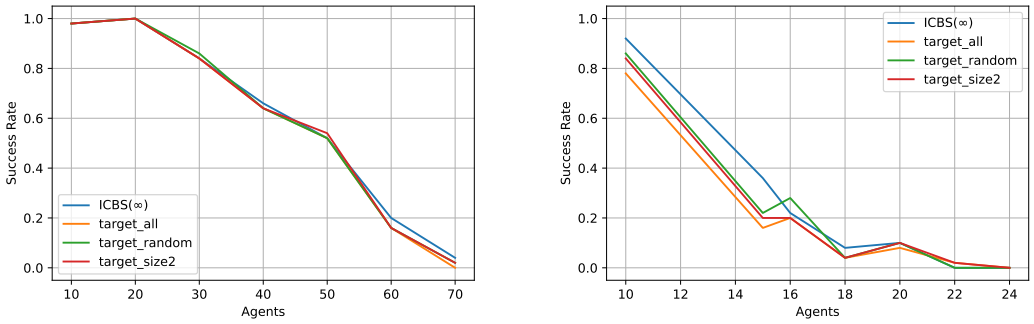


(a) One obstacle at (4, 4). (b) Three obstacles at (4, 2), (4, 4), and (4, 6).

Figure 6.7: The two agents a_1 and a_2 are in a cardinal rectangle conflict with a rectangle of size 5×5, which corresponds to the colored cells. Agent a_1 starts at (1, 2) and moves to (6, 7) and agent a_2 starts at (2, 1) and moves to (7, 6).

this obstacle, ICBS(∞) needs to expand 1252 high level nodes before terminating with the optimal solution. In contrast, with a single obstacle in the middle of the rectangle as depicted in Figure 6.7a, ICBS(∞) only needs to expand 395 nodes. With three obstacles in the rectangle like shown in Figure 6.7b, the difference is even more apparent. In this instance, only 84 high level nodes need to be expanded by ICBS(∞). As the rectangle conflicts get easier to solve with an increasing number of obstacles, the benefit of merging agents in such conflicts diminishes. Therefore, the rectangle merging strategies are more effective on maps with large open spaces.

On the instances for the empty and the dense map, the preparation time, which includes finding pairwise symmetries and building the meta-agent assignment, is almost negligible. It ranges from 20 milliseconds on the instances with the least amount of agents to 150



(a) Empty map.

(b) Dense map.

Figure 6.8: Success rates of ICBS(∞), target_all, target_random, and target_size2.

milliseconds on the instance with a higher agent density. These preparation times are similar for all our rectangle merging strategies.

6.1.2 Target Conflicts

In the next study we evaluate our three proposed merging strategies based on target symmetries, which are target_all, target_random, and target_size2. We compare them to ICBS(∞) on the empty and dense maps. We show their success rates for the empty map in Figure 6.8a, where virtually no differences can be observed. In contrast, on the dense map the success rates can be distinguished from each other with a small margin, at least on the instances with a lower number of agents. This can be seen in Figure 6.8b. Also, on the dense map our merging strategies perform worse than no merging at all in most cases.

The fact that there is a noticeable difference between the success rates on the dense map but not on the empty map can be explained by looking at the number of instances where at least one merge is performed. We provide this information in Table 6.4. For the empty map, approximately a third of all instances have a cardinal target conflict in the initial state, so in two thirds of the instances all four algorithms behave equally. In contrast, on the dense map there is in most of the instances at least one candidate pair, which explains the difference in the success rates. Additionally, the target conflicts are generally not as expensive for ICBS(∞) to solve as the other pairwise symmetries and thus, there is little benefit in merging them.

On the dense map, target_all performs the worst of all our target merging strategies. Since it merges all pairs which are in cardinal target conflicts, the meta-agents tend to get rather large. This is because if there is an agent which has its goal in a bottleneck, which there are many of on the dense map, and only a short path to it, many agents will collide with it. This can result in a sizable number of agents which are in a target conflict with this agent. Thus, they all get merged with this agent that stays in the

Table 6.4: For each map and agent number we show how many instances have at least one cardinal target conflict in the initial state.

Empty		Dense	
Agents	Card. Target	Agents	Card. Target
10	1/50	10	33/50
20	4/50	15	47/50
30	11/50	16	46/50
40	20/50	18	49/50
50	24/50	20	47/50
60	26/50	22	50/50
70	35/50	24	50/50
Overall:	121/350		322/350

Table 6.5: For the dense map we present for each number of agents the average size of the largest meta-agent and the percentage of how many of the overall number of agents are in the largest meta-agent.

Agents	10	15	16	18	20	22	24
Largest Meta-Agent	2.2	4.3	3.4	4.9	5.4	6.3	7.5
Largest as Percentage	22%	29%	21%	27%	27%	29%	31%

bottleneck and consequently form a meta-agent consisting of many agents. To analyze this further, we calculate for each of the dense instances the size of the largest meta-agent. Then, we calculate the mean for each category of instances, which corresponds to the agent numbers. The data is displayed in Table 6.5. For the instances with more agents the largest meta-agent consists on average of six to seven agents, which is substantially higher than for target_random and target_size2, as they only allow for meta-agents of size two. We also show the fraction of overall agents which are contained in the largest meta-agent. This value is around 20% to 30% for each category.

Such high numbers of agents in a single meta-agent are not beneficial in most cases. Having nodes with a large meta-agent is inefficient as a single conflict with one of the agents of this meta-agent means that the paths for the whole meta-agent need to be replanned. This is a reason why target_random and target_size2 outperform target_all on the dense map.

Similar as for the rectangle merging strategies, the preparation time for these strategies is also rather inconsiderable, with an average of 170 milliseconds.

6.1.3 Corridor Conflicts

In this study we focus on our three merging strategies based on corridor conflicts. We compare the success rates of ICBS(∞), corr_all, corr_random, and corr_size2 on the

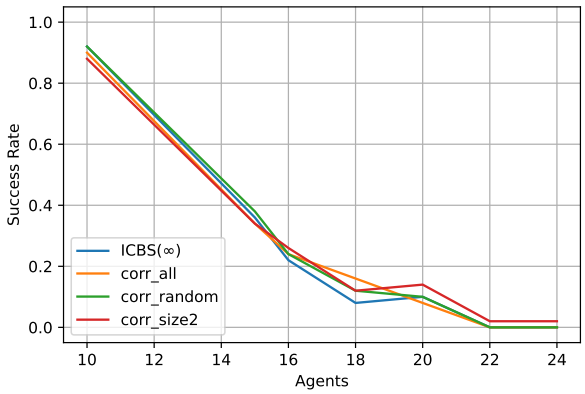


Figure 6.9: Success rates of ICBS(∞), corr_all, corr_random, and corr_size2 on the dense map.

Table 6.6: For the dense map we present for each number of agents the number of instances which have at least one, at least two, and at least three cardinal corridor conflicts in the initial state.

Agents	10	15	16	18	20	22	24	Overall
$\geq$ 1 Conflicts	23/50	40/50	38/50	47/50	46/50	47/50	48/50	289/350
$\geq$ 2 Conflicts	4/50	21/50	22/50	38/50	33/50	37/50	45/50	200/350
$\geq$ 3 Conflicts	2/50	12/50	14/50	25/50	25/50	29/50	36/50	143/350

dense map. We do not evaluate them on the empty map as it does not contain any corridors and therefore all algorithms would act equally. The success rates for the dense map can be seen in Figure 6.9. There are only slight differences noticeable, but the impact of the merging strategies is not negligible. On the instances with 16 agents or more they perform better or equal to ICBS(∞). Also notable, corr_size2 outperforms the other algorithms on instances with at least 20 agents and is even able to solve an instance with 22 and one with 24 agents. ICBS(∞), corr_all, and corr_random all have a success rate of 0 on these instances.

We also analyze how often cardinal corridor conflicts are present in our dense instances. This data is presented in Table 6.6. We show how many of the instances in each category contain at least one, at least two, and at least three cardinal corridor conflicts in the initial state. For 289 of the 350 instances there is at least one agent pair in such a conflict in the initial state. As with other types of conflicts, having more agents on the same map also increases the chance of encountering a cardinal corridor conflict.

The low variance of the success rates can be explained with the data provided in Table 6.6. We can see that in 150 of the 350 instances there are less than two conflicts present. In the case that there are no conflicts, the heuristics find no agents to merge and thus

Table 6.7: The number of positive and negative training samples for the empty, dense, and medium maps. Only agent pairs which result in a beneficial merge regarding the runtime are positive samples.

Map	Positives	Negatives	Overall
empty	10784	75796	86580
dense	1242	4760	6002
medium	3064	18686	21750
Overall	15090	99242	114332

all behave like ICBS(∞). If there is only one merge candidate pair, then there is also no difference between our three merging strategies. This is because every strategy will pick this one pair to merge. Corr_all merges all available pairs, which is this one pair. Corr_random chooses a pair randomly, but since there is only one pair to choose from, it also merges this pair. Similar to corr_all, corr_size2 also merges all available pairs, if the resulting meta-agent is not of size greater than two. Consequently, it also merges this pair. Therefore, our three merging strategies behave equally in 150 of the 350 instances, which is around 43%. Of the remaining 200 instances, all four algorithms are not able to solve 178 of them in the two minute time limit. With the small amount of instances where the merging strategies differ and which are solvable for at least one of them, the rather small difference in success rates can be explained.

As with the strategies based on the other two pairwise symmetries, the strategies based on corridor conflicts also have short preparation times. The average preparation time over all instances is only 75 milliseconds.

6.2 Evaluation of Machine Learning Model Enhanced Merging

In this section, we evaluate our machine learning based approaches. As described in Section 5.2, we use a modified version of the CSBH2-WDG algorithm to retrieve the features of MAPF instances. For the learning part and the models we use the Python module *scikit-learn* (0.24.2) introduced by Pedregosa et al. [16]. We again use ICBS(∞) to solve the instances with the preformed meta-agent.

6.2.1 Learning

For the creation of the training data we use numerous instances which we created for the empty and dense maps, as described in the beginning of this chapter. We use 117 of the empty instances and 64 of the dense instances. Additionally, we use 50 instances of a medium map which we create for this study. The medium map is similar to the empty and dense maps. It is a 20×20 grid where 15% of the cells are obstacles. From each of the selected instances we get $\binom{n}{2}$ training samples, as each entry in the training data

is for a pair of agents. We show the number of resulting samples in Table 6.7. Overall, we have 114332 entries for our training data. Of these 114332 entries, only 15090 of the pairs are beneficial to the runtime when merged, which is only around 13%. To decrease the impact small fluctuations in the runtime have on our learning process, we only keep training samples where the difference in runtime between the merged pair and our reference ICBS(∞) run is more than 10%, based on preliminary tuning runs. With this measure, we mostly avoid mislabeled training data entries, but reduce the number of positive entries to 7528 and the overall entries to 89677.

We balance our training data by selecting all the positive entries and randomly the same amount of negative entries, which results in data set of 15056 samples. Then, we split our data into 80% training and 20% test data. For the training of the MLP classification model, we further split off 25% from the remaining 80% training data for validation, which results in a 60-20-20 split. Before we fit our selected machine learning models on the data, we scale it so each value of our features is between zero and one. We save this instance of our scaling function so we can use it later again when we solve new MAPF instances, as the feature values need to be scaled consistently. For the SVM model we choose the *C-Support Vector Classification* implementation, which is called SVC. We use a value of ten for the regularization hyperparameter C as we want to avoid misclassified agent pairs. Otherwise, we use the default values provided by the implementation. For our MLP approach we use the *MLPClassifier* implementation. We repeatedly train the MLP model with different hyperparameters each time and evaluate them by calculating the mean accuracy of the model on the validation set. We achieved the best performing model by selecting 0.001 for the weight decay hyperparameter, a batch size of 200, two hidden layers of size 20, a maximum of 1000 iterations, and otherwise the default hyperparameters given by the implementation of the MLPClassifier. We show the final selected values for the hyperparameters of both machine learning models in Table 6.8.

We put the model in pipelines and before both of them we place the scaling function. This way, before the models are fitted, the data is first transformed as described above. When we save the pipeline and later load it to make prediction, the same transformation process will be applied to the data.

6.2.2 Classification

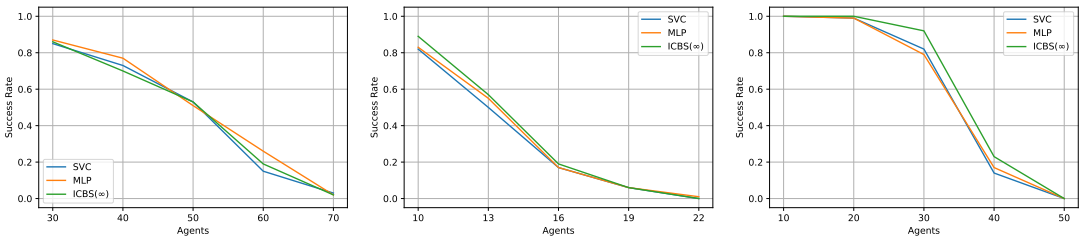
After the learning process is done, we incorporate the models into our algorithm. To reiterate, the finished algorithm consists of three steps: First, we use a modified version of CBSH2-WDG to create the features of each pair of the MAPF instance. Secondly, we let one of our models predict which agent pair is most likely to result in an advantageous merge. We do this by letting the MLP model output the probabilities for the agent pairs to belong into the class of beneficial merges. For the SVC model, we select the pair which is on the positive side and furthest from the hyperplane that separates the two classes of pairs. Lastly, we let ICBS(∞) solve the instance with this pair premerged. We will refer to this whole algorithm as SVC if it uses the SVM model for prediction, and as MLP if it uses the MLP model.

Table 6.8: The selected values for the hyperparameters of the SVC model and the MLPClassifier model.

Hyperparameter	**Value**
SVC	
Regularization Parameter C	10
Decision Function Shape	One-vs-Rest
Gamma	1
Kernel Type	Radial Basis Function
Iterations Limit	No Limit
Shrinking Heuristic	$True$
Tolerance	0.001
MLP	
Activation Function	Rectified Linear Unit Function
Alpha	0.001
Batch Size	200
Beta_1	0.9
Beta_2	0.999
Early Stopping	$False$
Epsilon	1e-08
Hidden Layers	(20, 20)
Learning Rate	Constant (0.001)
Maximum Iterations	1000
Number of Iterations with no Change	10
Shuffle	$True$
Solver	Adam
Tolerance	0.0001

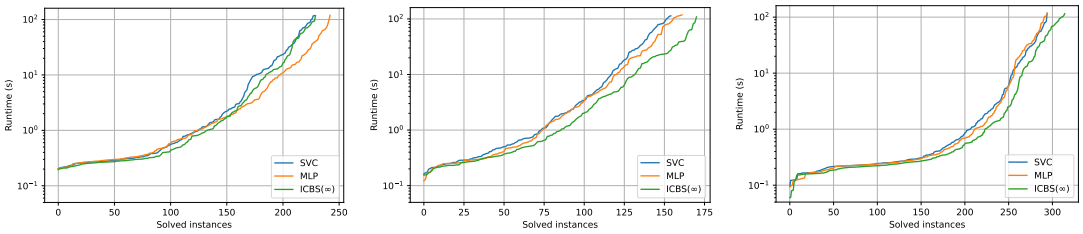
Table 6.9: Additionally to the empty and dense maps we already described, we test on a medium map with 15% obstacles. For each the empty and dense map we create 500 new instances which have not been seen by the machine learning model in the training process. We also create 500 instances for the medium map.

Map	Agents	Size	Obstacles
empty	30, 40, 50, 60, 70	20×20	0 %
dense	10, 13, 16, 19, 22	20×20	30 %
medium	10, 20, 30, 40, 50	20×20	15 %



(a) Empty map. (b) Dense map. (c) Medium map.

Figure 6.10: The success rates of SVC, MLP, and ICBS(∞) for the empty, dense, and medium maps.



(a) Empty map. (b) Dense map. (c) Medium map.

Figure 6.11: The runtime distributions of SVC, MLP, and ICBS(∞) for the empty, dense, and medium maps, where the x-axis represents the number of solved instances and the y-axis the time needed to solve the individual instances.

As we use some of the instances for the empty, dense, and medium maps for the training data, we generate new instances for these maps. This is important as we want to test how our models perform on new instances which they have not seen during the learning process. For each map we choose five different number of agents, and for each map and number of agents we create 100 instances. We show the chosen number of agents in Table 6.9.

We test SVC and MLP on the newly generated instances and compare them to ICBS(∞) based on their success rates, which we show in Figure 6.10. Again, we set a time limit of two minutes. In Figure 6.10a the success rates for the empty map are depicted. SVC, MLP, and ICBS(∞) perform similar but MLP is marginally more successful on most categories. On the dense map in Figure 6.10b the only noticeable difference between the runtimes is on the instances with a small amount of agents. There, ICBS(∞) performs better then to others, although MLP manages to solve an equal amount of instances with 13 agents. The success rates for the medium map in Figure 6.10c are more distinguishable. The success rates of our two machine learning approaches lie below the reference algorithm for almost all number of agents.

We also show the runtime distributions of the four algorithms for the empty, dense, and medium maps in Figure 6.11. On the empty map in Figure 6.11a, the use of the MLP

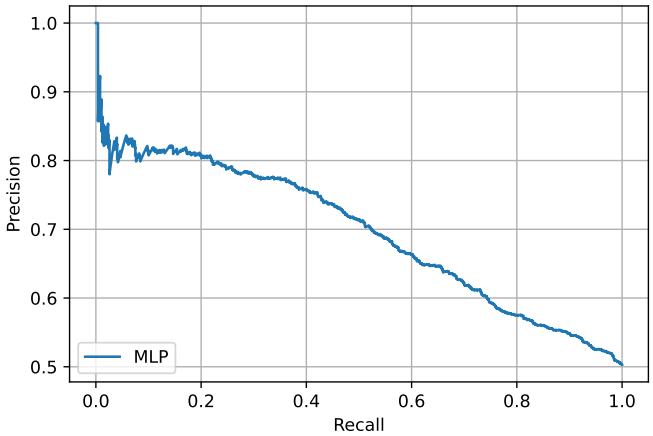


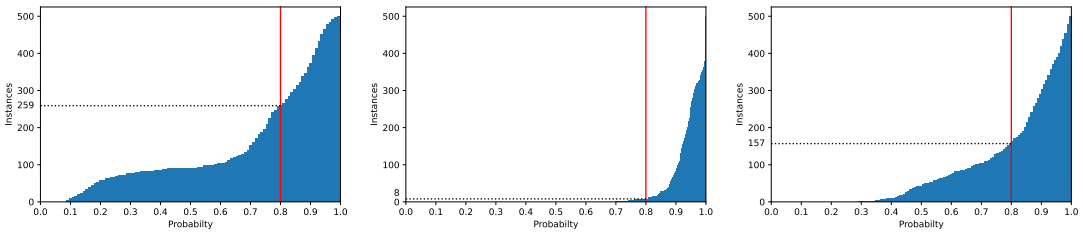
Figure 6.12: The precision-recall curve of the MLP model calculated on the test set.

model is advantageous compared to the SVC model. Also on the empty map, MLP is able overall to solve more instances faster than ICBS(∞). On the dense and the medium map there is no significant difference between the two machine learning based approaches, but ICBS(∞) manages to solve some instances in less time than the other algorithms.

Threshold The fact that the algorithms with our learned models perform worse than ICBS(∞) in some cases could be a shortcoming of our strategy. Because we always pick the pair which has the highest probability of all pairs to be beneficial, we also perform merges which are more likely to be detrimental than advantageous. Even if our model predicts for all agent pairs to be bad merges with a high probability, we still merge one of them. Because of this, we use a threshold for our next approach. In this approach, we again merge the pair which is most likely to be beneficial, but only if the predicted probability is above a certain threshold.

As described in Section 5.2.2, we choose this value by analyzing the precision and recall of our model for numerous different thresholds. Our approach works best if we pick a threshold which represents a high precision value. Again, a high precision value means that predicted positives are likely true positives. This is important as we aim to avoid performing counterproductive merges.

We calculate the precision and recall for numerous thresholds using the output values and the predictions of our trained MLP model for the input values of the test data. Since the MLP model outperformed the SVC model overall on the previous study, we focus on the MLP model for this approach. In Figure 6.12 we show the precision-recall curve which we calculate for our model on the test set. For different thresholds, the corresponding precision and recall values are found. As depicted, an increase in recall results in a decrease in precision, so it is a tradeoff between those two. As our priority is to have a high precision value, we choose a precision of about 82% and a recall of about 15%. We then find the respective threshold, which is approximately 80%. Therefore, we



(a) Empty map. (b) Dense map. (c) Medium map.

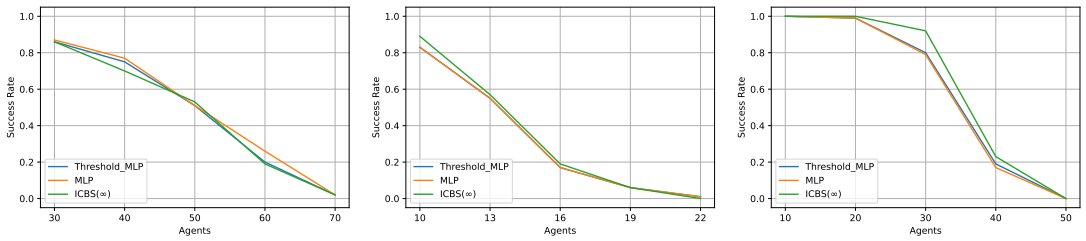
Figure 6.13: For the three maps we show cumulative histograms of the highest probability predicted by our MLP classifier for each instance. Additionally we show our chosen threshold of 80% as a red line.

only perform merges in instances with an agent pair with a probability equal or higher than 80% of belonging into the class which result in advantageous merges. Otherwise, if there are no suitable merge candidates, we merge no pair and thus the algorithm behaves equally as ICBS(∞). We refer to this algorithm as *Threshold_MLP*. In essence, in instances with an agent pair with a high predicted probability, Threshold_MLP behaves equally to MLP, in the other cases it behaves equally to ICBS(∞).

To better understand how the chosen threshold affects our algorithm, we present information about the highest probabilities for each instance in Figure 6.13. For the empty, dense, and medium map we show in the form of histograms the cumulative values of the highest probability for each instance. We also show the threshold as a red line. Therefore, it becomes visible in how many instances Threshold_MLP actually merges. In Figure 6.13a this information for the empty map is depicted. On the empty map no merges are performed in 259 instances, which is more than half of all instances. There are also many instances which have fairly low values. In contrast, on the dense map almost all instances have a pair which has a higher probability than 80%, which is depicted in Figure 6.13b. 492 of the 500 instances above the threshold with the lowest value still being about 70%. In Figure 6.13c the information for the medium map is displayed. With 157 instances below the threshold, it is a mixture between empty and dense.

In Figure 6.14 we show the success rates of the above described algorithm Threshold, MLP, and ICBS(∞). On the empty map in Figure 6.14a Threshold_MLP performs worse than MLP and more similar as ICBS(∞). This can be explained by the fact that Threshold_MLP only merges in 241 of the 500 instances, and thus behaves more like ICBS(∞). In Figure 6.14b the success rates for the dense map can be seen. As expected, Threshold_MLP and MLP have identical success rates, as they behave equally in 492 instances. On the medium map in Figure 6.14c there is also no notable difference between Threshold_MLP and MLP. This is because it does not merge in 157 of the 500 instances, and as the success rate of ICBS(∞) shows, this is the better suited strategy for this map.

Preparation Time In this part of our study we analyze how long it takes until the actual solving of the MAPF instance starts, which we refer to as the preparation phase.



(a) Empty map. (b) Dense map. (c) Medium map.

Figure 6.14: The success rates of Threshold_MLP, MLP, and ICBS(∞) for the empty, dense, and medium map. On the dense map, the line for Threshold_MLP is hidden behind the line of MLP.

This preparation includes the feature generation, the process of building the feature values into the required input vector, using the model for predictions, and building the group assignment.

As mentioned earlier, since we use a different CBS code basis for the feature measurement as we use for the actual solving, some computational work is duplicated. This includes finding initial paths for the agents and checking for collisions, which we need to do to measure our features. Fortunately, this is done rather quickly on the instances we use in this study. The whole process of getting the feature values and transforming them into the required input vector format takes less than a second for almost all instances and is therefore negligible.

The time it takes for our models to make a prediction for every pair is on average also less than one second. As expected, this time scales with the number of agents, as more agents result in more pairs. For an instance with n agents, adding another agent adds n new possible pairs. Lastly, forming the actual group assignment for the ICBS algorithm is a simple process, as we only merge one pair of agents. As shown in Chapter 4, this only requires us to change one entry of the default group assignment, which can be done in an instant.

Feature Importance To calculate the importance of our chosen features, we use a random forest classifier model. This model consists of numerous decision tree classifiers which are all trained on parts of the training data. Then, the predictions of each individual decision tree are averaged to get a final prediction.

We use this model to calculate the importance of our features. First, we fit the model, which consists of 100 individual decision trees, on the training data. Then, we calculate its accuracy on the test set, which is calculated by dividing the number of correct predictions by the number of total predictions performed. This accuracy is around 0.76 and serves as a baseline for the following calculations. The metric with which we analyze the feature importance is called permutation importance. The idea is to calculate the importance of each feature by permuting its column in the test set and then again calculating the

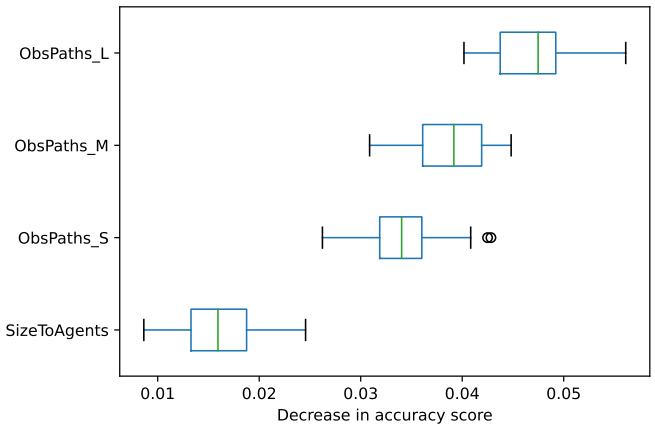


Figure 6.15: Boxplots of the 40 calculated accuracy differences for the four most important features ObsPath_L, ObsPath_M, ObsPath_S, and SizeToAgents.

accuracy of the model. Then, the difference of this new accuracy compared to the baseline accuracy is used as the value which describes the importance of this feature for the model. The bigger the difference, the more important the feature is.

We repeat this process 40 times, so we have 40 values for each feature. In Figure 6.15 we show these values in boxplots for the four most important features. The features omitted have a mean decrease in accuracy which is lower than 0.01, which makes them unimportant to the model. The three most important features ObsPath_L, ObsPath_M, and ObsPath_S are the three variants of the instance based feature for which we calculate the ratio of nearby obstacles to non-obstacles for all agents' paths. This feature aims to analyze the topology of the parts of the map which the agents traverse, e.g., if they move through narrow corridors or open spaces. The variant ObsPath_L analyzes the biggest area around the paths, ObsPath_M a smaller area, and ObsPath_S the smallest. The feature SizeToAgents, which is based on the ratio of the map size to the number of agents, is also rather important.

These four features differ greatly when comparing the empty and the dense maps, as the number of obstacles along the paths of the agents is on average much smaller on the empty map, and the dense map is of the same size as the empty map but contains less agents on average. They provide a bias regarding the general tendency to merge—to select pairs, the agent features have to come into play, but as the results have shown seem not be sufficiently strong.

Other features which also may be important but have not made it into our study are features based on pairwise symmetries. For example, a feature could be whether the agent pair is in a rectangle conflict or not. Then, the size of the area of the rectangle is another interesting feature. Also, aspects of agents in target and corridor conflicts, which we missed in our proposed merging strategies in Section 5.1, maybe are considered by our other features and therefore lead to a more successful merging of these agents.

Table 6.10: The runtimes and expanded high level nodes for the instances on the empty map. We calculate the average values for each number of agents over the instances solved by all three algorithms.

Empty							
Agents	Solved	ICBS(25)	Lookahead	MAMDD	ICBS(25)	Lookahead	MAMDD
		Runtime (ms)			Nodes		
10	50/50	202	200	197	3.0	3.0	3.0
20	50/50	448	407	425	8.2	7.6	7.6
30	50/50	1103	988	897	33.0	31.0	31.0
40	44/50	2927	3471	2681	67.2	58.6	58.6
50	33/50	2167	4020	3098	107.2	105.3	105.1

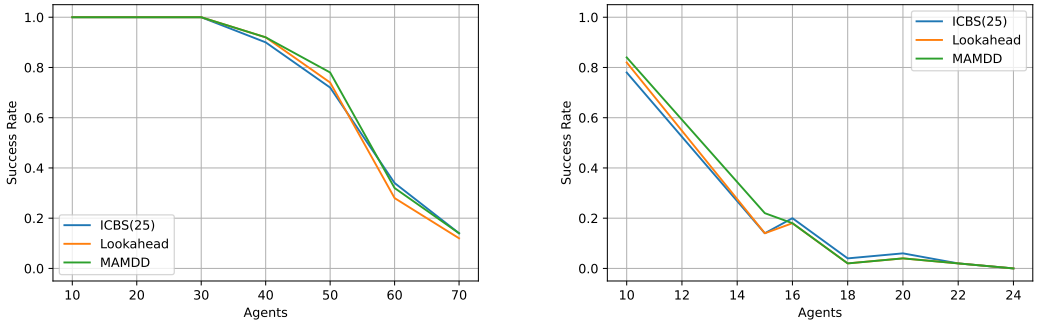
Another thing that could be done in future work is to evaluate the random forest classifier models, which we trained for the feature importance, the same way we evaluate our SVC and MLP models.

6.3 Evaluation of Heuristics for Meta-Agent Conflict Based Search

We evaluate the heuristics proposed in Section 5.3 on our chosen scenarios, which are described in the beginning of this Chapter. We will refer to the heuristic based on the look ahead technique as *Lookahead* and the other heuristic which uses MAMDDs as *MAMDD*. First, we compare ICBS(25) with no heuristics added to ICBS(25) which employs one of our heuristics to guide the high level search. The comparison will be based on the number of high level nodes expanded, the overall time needed to solve the instance, and also their success rates. Secondly, we will perform a similar analysis on a premerge algorithm in Section 5.1, as they also require heuristics which are able to handle meta-agents. Thirdly, we will conduct an analysis of the built MAMDDs throughout these runs. The analysis will focus on aspects of the MAMDDs and how they relate to the aspects of the maps which they were constructed on.

6.3.1 Heuristics for ICBS(25)

We investigate how the addition of our two proposed heuristics affects the runtime and the high level nodes expanded by ICBS(25). The expected outcome of adding a heuristic function to guide the high level search is a reduction in expanded nodes. A more informed search process can avoid expanding unpromising nodes which results in less computational work for the low level solver. Although this would mean a reduction in the runtime as well, we must not forget about the computational overhead the heuristic functions add to the search.



(a) Empty map with no obstacles.

(b) Dense map with 30% obstacles.

Figure 6.16: Success rates of ICBS(25) with no heuristic, with the Lookahead heuristic, and with the MAMDD heuristic.

Empty and Dense

The results of our first experiments are shown in Table 6.10. The tests are performed on the empty map which has no obstacles. For the comparison we only use instances which were solved by all three algorithms within the time limit, which is 120 seconds. We track the time it takes to solve the instance in milliseconds (ms) and the number of nodes expanded by the high level search. As expected, both heuristics performed similar in terms of expanded nodes, which both show a small reduction compared to the base algorithm, except for the smallest instances with only ten agents. Since the difference in expanded nodes is rather small, the time saved is also rather insignificant. This results in almost no time saved overall and even a worse runtime for both heuristics for the instances with 50 agents. On this type of map, there is no clear benefit from adding the proposed heuristics to ICBS(25). If we compare the heuristics to each other, we can see that MAMDD is faster than Lookahead in most cases while expanding the same amount of nodes. This means that the MAMDD heuristic finds similar h-values as the Lookahead heuristic, but its computational overhead is less.

On the dense map, ICBS(25) generally performs poorly. Although it manages to solve most of the instances with ten agents, is not able to solve most instances with more agents. Of the 50 instances with ten agents 39 are solved by all three algorithms. For these instances, ICBS(25) expands on average 29 nodes, where both heuristics only expand 22. The average runtime is 929 ms for ICBS(25), 2685 ms for Lookahead, and 1081 ms for MAMDD. The sample size here is rather small, but same as on the empty map, the MAMDD heuristic outperformed the Lookahead heuristic again, but both slow down the search process overall.

In Figure 6.16a and Figure 6.16b the success rates of the algorithms for the empty and the dense maps respectively are shown. On the empty map, the success rates are similar and fall off steep on the instances with more than 50 agents. There are only small differences in the success rates between the three algorithms and no clear winner can be selected. As

Table 6.11: The percentage of the overall solving time spent by the algorithms in the calculation of the h-values of the high level nodes. The numbers are the mean values taken over all 50 instances for each map and agent number.

Empty		
Agents	Lookahead	MAMDD
30	2.9 %	3.7 %
40	13.2 %	5.6 %
50	27.2 %	4.1 %
60	53.8 %	2.4 %

Dense		
Agents	Lookahead	MAMDD
10	18.5 %	8.4 %
15	79.6 %	12.0 %
16	75.1 %	9.6 %
18	85.7 %	7.0 %

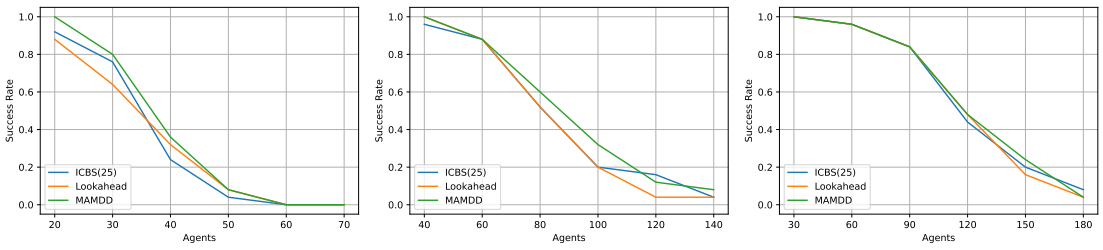
mentioned earlier ICBS(25) is not able to solve many instances on the dense map. The success rates start off high at the ten agents instances but drop off sharply. Although the MAMDD heuristic enables ICBS(25) to solve more of the smaller instances, its addition does not increase the success rate for the instances with more agents.

To further analyze the effects of adding the two heuristics to ICBS(25), we track the time they spend in the calculation process of the f-values for the high level nodes. Since the differences in the results of the heuristic functions are negligible, the only criteria to compare them on is the time they need to come to their results. In Table 6.11 we show the time the algorithms spend in the heuristic function as a percentage of the overall search time. The table shows the mean values averaged over all 50 instances of the corresponding maps and number of agents. The time the Lookahead heuristic needs clearly scales with the difficulty of the instance. As the instances get more crowded with agents, more collisions happen, which in turn results in more constraints on the individual paths. The immense difference in the time needed by the heuristic on the more difficult instances can also be attributed to the fact that the MAMDD heuristic is only limited to meta-agents of size two. Thus, it skips the classification of conflicts where meta-agents of size greater than two are present, which means that such meta-agents are never added to the conflict graph in the MAMDD heuristic.

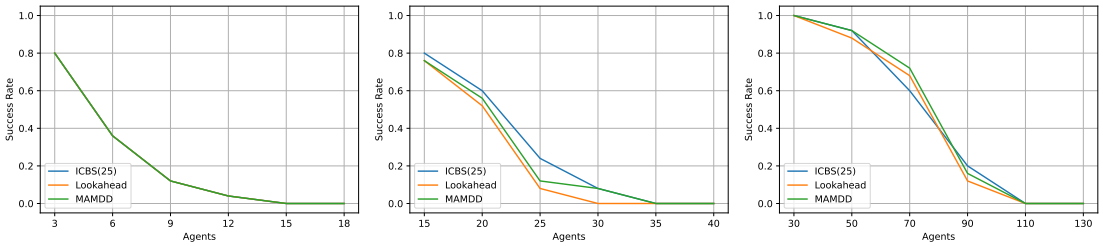
In the instances with more agents, it is more likely for meta-agents to grow larger. Simulating the expansion of nodes on conflicts with larger meta-agents is costly, which explains why the algorithm with the Lookahead heuristic spends a sizable portion of its solving time in the heuristic function. As the MAMDD heuristics skips conflicts with larger meta-agents, it also spends less time in the calculation of the f-values as the instances get more crowded. Since the MAMDD heuristic performed better than the Lookahead heuristic in terms of success rates, skipping the classification of conflicts with bigger meta-agents may be more beneficial to the solving time than classifying them.

Real World Maps

We also run experiments on the six maps which are based on real world scenarios. The success rates of the algorithms are depicted in Figure 6.17. On brc202d, the MAMDD



(a) brc202d (b) den520d (c) Paris_1_256



(d) maze-128-128-1 (e) room-64-64-8 (f) warehouse-10-20-10-2-1

Figure 6.17: The success rates of the ICBS(25) algorithm without heuristics, with the Lookahead heuristic, and with the MAMDD heuristic for our the maps based on real world scenarios are shown.

heuristic clearly outperforms the base algorithm without any heuristics and also manages to solve more than the Lookahead heuristic. Adding the Lookahead heuristic to ICBS(25) actually decreases the success rate on the instances with a low amount of agents. It only starts to become beneficial as the number of agents grows in the instances.

On den520d and paris again the overhead the Lookahead heuristic adds does not outweigh the time saved in the high level search. Thus, its success rates are similar or even worse then those of the base algorithm. The MAMDD heuristic performs slightly better on the den520d map than the other algorithms.

The narrow corridors of the maze map allow only for one agent to pass through them at the same time. As a result, a collision between agents usually entails multiple future collisions. ICBS(25) merges agents after they have collided 25 times, which is a rather low threshold. Because of this, it is likely for agents to be merged if they traverse the same corridor in different directions. For the maze map we also chose instances with a very low number of agents, since they are already hard to solve with only six agents per instance. The low number of agents combined with the rather high merge rate means that the conflict graphs cannot grow large, which in turn only allows for small h-values. For example, for instances with only three agents, the maximum h-value a node can have is two, as the conflict graph can consist of at most three nodes. The combination of the low number of agents and the high conflict to merge rate, which leads to an even lower number of possible conflict graph nodes, reduces the impact of the heuristics as far as

			Expanded Nodes		Solving Time (s)	
Map	Agents	Solved	ICBS(25)	MAMDD	ICBS(25)	MAMDD
brc202d	20	23/25	32.3	18.7	15.41	10.40
	30	18/25	73.8	28.8	38.89	17.60
den520d	40	24/25	14.8	12.0	6.57	5.97
	60	22/25	42.4	27.5	23.62	17.95
	80	13/25	69.3	49.9	42.71	39.32
Paris	30	25/25	6.1	5.7	2.79	2.08
	60	24/25	18.6	12.3	8.14	5.92
	90	21/25	44.8	29.9	22.30	15.46
Maze	3	20/25	11.7	10.4	3.45	2.49
	6	9/25	17.0	16.4	3.06	2.58
Room	15	18/25	53.7	33.2	2.71	1.42
	20	13/25	112.5	86.6	9.06	6.65
Warehouse	30	25/25	15.0	14.1	1.48	1.24
	50	23/25	28.8	28.3	6.59	5.80
	70	15/25	55.6	54.3	19.81	17.79

Table 6.12: A selection of results for the chosen maps. We average the results over all instances solved by both algorithms for each number of agents and each map. Thus, we focus on the data of instances with lower agent numbers since they provide a higher sample size than instances with a higher agent count.

that there is no difference in the success rates of the three algorithms.

On the room map there is no number of agents where the use of our proposed heuristics is beneficial to the success rate of ICBS(25), and on the warehouse map there is no clear winner overall. They all perform similar, for some number of agents the heuristics are a bit better, but for some they add too much computational overhead.

Comparing the success rates of the two heuristics shows that the MAMDD heuristic outperforms the Lookahead heuristic in most map types and agent numbers. There are only minimal differences in the classification results between the two heuristics, but the MAMDD heuristic uses a more efficient calculation process.

To further analyze the effects of the heuristics we compare them to ICBS(25) based on the number of expanded high level nodes and also the time in seconds spent solving the instances. We present this data in Table 6.12. Again we only compare them on instances they both are able to solve in two minutes. As a result, there is a small sample size on the more crowded instances, as they are generally harder to solve, which can also be seen in the success rates. This is why the focus of this particular study is set on the instances with amounts of agents which we consider to be low for the respective map type. Also,

Table 6.13: Information about the MAMDDs built in our experiments. For each map, we present the mean values over all MAMDDs which are generated by the algorithms using the MAMDD heuristic. The sample size is the number of MAMDDs generated for the map. We also stored for each individual MAMDD the width of the level with the most node. Final nodes is the number of nodes that are still present in the finished MAMDD and the final to size column presents the final number of nodes divided by the number of non-obstacle cells in the map.

Map Type	Widest Level	Final Nodes	Non-Obstacle Cells	Final to Size
Empty	159	1517	400	3.80
Dense	119	1104	280	3.94
brc202d	1423	126017	43151	2.92
den520d	4063	317275	28178	11.26
Paris	2447	141812	47240	3.00
Maze	55	7591	8191	0.93
Room	248	3186	3232	0.99
Warehouse	599	19609	5699	3.44

since the MAMDD heuristic outperforms the Lookahead heuristic regarding their success rates, we focus on the more promising MAMDD heuristic.

On these instances a difference in expanded high levels nodes between ICBS(25) with no heuristic and ICBS(25) with the MAMDD heuristic can be observed. As expected, the algorithm with the MAMDD heuristic is able to solve the instances with a lower number of node expansions. The difference is especially noticeable on brc202d, den520d, paris, and room. On the warehouse and maze types of maps the difference is almost negligible. Consequently, the average solving time is also lower for the MAMDD algorithm. The biggest time reduction is achieved on the brc202d map. On the maps where the heuristic has only little impact, the decrease in solving time is accordingly small. However, on these maps the heuristic also adds only insignificant amounts of computational overhead and is thus not detrimental to the search.

6.3.2 Analysis of MAMDDs

We also gather data about the MAMDDs built during the search of algorithms which use the MAMDD heuristic. We analyze the MAMDDs with regards to their size and how they differ on our selected map types.

The information about the MAMDDs is presented in Table 6.13. For each map type, we calculate the mean value of the widest levels and the number of final nodes over all MAMDDs. The width of the largest level of an MAMDD gives a reasonable overview of its overall width. This is an important metric as the computational work of the collision check of the conflict classification for the MAMDD heuristic scales directly with the width of the levels which are to be checked. Final nodes refers to the number of nodes

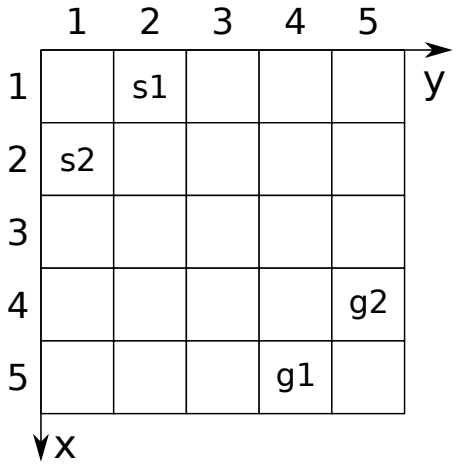


Figure 6.18: A 5×5 grid with two agents. Agent a_1 starts at (1, 2) and moves to (4, 5) and agent a_2 starts at (2, 1) and its goal is at (5, 4). Both have multiple individual optimal paths of length six, but as they are in a cardinal rectangle conflict, one of them has to wait once and the optimal solution thus has a cost of 13.

the MAMDD consists of after its generation process is finished. Again, we take the mean over all MAMDDs for each map type. Clearly, bigger maps will also lead to wider and overall larger MAMDDs, as the paths the agents take can be much longer. To get a more clear comparison on how the topology of the maps affects the size of the MAMDDs, we divide the number of final nodes by the number of non-obstacles cells of the maps.

The width of the largest level of the MAMDDs also depends on the size of the map, but not as much as the final number of nodes. Regarding the width of an MAMDD, the number and the distribution of obstacles are deciding factors. Because of this, the maze map has rather thin MAMDDs. The narrow corridors of the map do not allow for a lot of variety in the paths the agents can take. Thus, there are also not many different possibilities of paths for MAMDDs. This is similar on the room map, as the narrow doors between the rooms are bottlenecks which most paths have to pass through, which in turn again limits the possibilities for valid children in the MAMDD generation process. On open maps with almost no bottlenecks, like den520d and Paris, and on the empty map which has no obstacles, the MAMDDs are much wider if we consider the map sizes.

The number of final nodes correlates mostly to the widest level of an MAMDD and, more importantly, it also correlates with the size of the map. This can be seen in the last column, which shows the mean final size divided by the number of available cells of the map. In this metric, three map types have a rather different value than the others. First, den520d seemingly has extremely large MAMDDs. This can be attributed to its enormous spaces which are almost free from obstacles, which allow for a large amount of different paths from the start to the goal cells of the agents. Also, due to its layout, the paths can get rather long. The other two maps which are out of the norm are maze and room. Both have small MAMDDs in relation to their map size. As stated earlier,

both maps are defined by their numerous amount of bottlenecks of width one. Thus, the MAMDDs are also narrow which results in small numbers of final nodes.

Taking a closer look at some of the largest MAMDDs gives an explanation of how they become so big. Similar to CBS, the MAMDD generation also does not handle rectangle conflicts efficiently. In Figure 6.18 a cardinal rectangle conflict is depicted. Since the conflict is cardinal, one agent will have to wait one timestep while the other can move along one of its optimal paths. Not only are there many individual optimal paths for both agents, which result in an even greater number of path combinations the MAMDD has to contain, but there is also the one wait action which can be done at numerous different occasions. Because of this wait action, there are even more possible paths to consider.

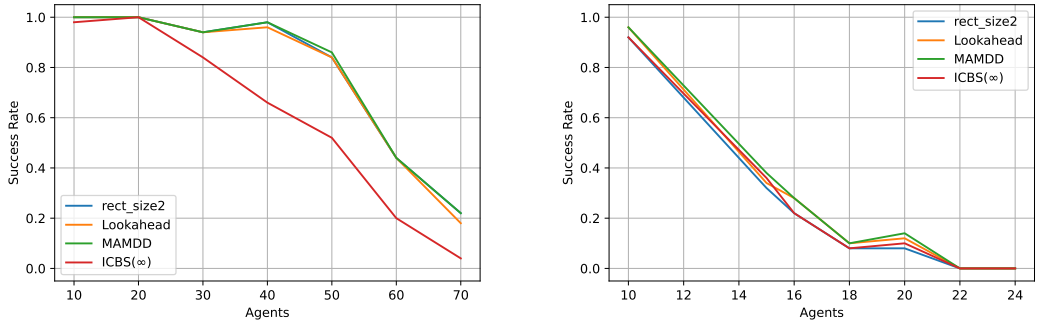
Having a look at the first three levels of an MAMDD for agents a_1 and a_2 illustrates the node explosion the heuristic has to deal with. The root node of an MAMDD for agents a_1 and a_2 on this instance would consist of both starting locations. So for the first level, we have only one node. Of the 25 children of the root node, seven are feasible. Three where both of them move, and two each where one of them waits. At the third level there are already 38 nodes and it reaches it maximum width at level four with 68 viable nodes. Overall, the MAMDD of this simple example with a cardinal rectangle conflict of size 3×3 consists of 243 nodes. With a rectangle of size 5×5 the widest level is already of size 230 and the MAMDD consists of 1115 nodes, and a rectangle size of 10×10 results in a MAMDD which widest level is 1055 nodes large.

This is unfortunate for the ICBS(25) algorithm, as agents in large rectangle conflicts are likely to get merged, since they are sure to collide repeatedly. The same problem occurs in our approach of merging agents in pairwise symmetries, as rectangle conflicts are part of the pairwise symmetries. In the end, the return of the reduced node expansions can become somewhat diminished by the explosion of the MDDs and a proper balance has to be found, which is the rationale behind allowing only meta-agent pairs.

6.3.3 Heuristics for Pairwise Symmetry Based Merging

In this study we add our heuristics to one of our proposed strategy for merging agents in pairwise symmetry conflicts as discussed in Section 5.1. For this we choose the best performing strategy, which is rect_all_card. The strategy finds all agents which are part of cardinal rectangle conflicts and merges them. To increase the synergy between the merge strategy and our MAMDD heuristic, we modify rect_all_card to only merge agents into meta-agents of size two. This allows the heuristic to handle every conflict classification case, as it is designed to classify conflicts between agents and meta-agents, but only if the meta-agents are of size two.

We call this modified strategy rect_size2. Same as for rect_all_card, we first find all agents which are part of a cardinal rectangle conflict in the initial state of the instance. Then, we calculate the size of the rectangles of the rectangle conflicts. We use this information to give the rectangle conflicts different priorities. Merging agent pairs with



(a) Empty map.

(b) Dense map.

Figure 6.19: Success rates of rect_size2 with no heuristic, with the Lookahead heuristic, and with the MAMDD heuristic.

larger rectangles is more important than merging agents in rectangle conflicts of small size, as their complexity scales exponentially with the size. We sort the agent pairs which are part of these conflicts by the size of the rectangle and then repeatedly pick the pair with the largest rectangle to merge into meta-agents and remove them as candidates for further merges. Since we only want meta-agents of size two, we only merge agents if they are not already merged with another agent.

For comparison purposes, we only focus on the actual runtime of the solving component and disregard the time needed to analyze the initial state to find the pairwise symmetries. We do this as all of the work done by the symmetry detection mechanism is discarded before the solving process starts, which is the result of employing two different implementations. Therefore, this loss of information, like initial paths and already detected and classified conflicts, is a problem of our implementation design and not of the approach itself. The time spent in the symmetry detection phase ranges from 0.02 seconds on the smallest instances to 30 seconds on the largest instances of the real world maps, but most of the time in this phase is spent preparing and filling data structures. These data structures are filled to speed up the low-level path planning process later, but due to the design of our algorithm, they remain mostly unused as the actual solving is performed by another CBS algorithm.

Empty and Dense

In Figures 6.19a and 6.19b the success rates of the algorithms for the empty and the dense maps are shown respectively. We again use a time limit of two minutes. We compare the rect_size2 algorithm with no heuristic to the same algorithm but with our heuristics added. Also, we add ICBS(∞) to the comparison to see if the addition of our merging strategy and our heuristics provide an improvement. We choose ICBS(∞) since it is closest to the other algorithms in their behavior, as neither of them merge after the search process has started.

Table 6.14: The runtimes and expanded high level nodes for the instances on the empty map. We calculate the average values for each number of agents over the instances solved by all three algorithms and where merge candidates are present. On instances with no merge candidates, which means that in the initial state no agents are in a cardinal rectangle conflict, our algorithm behaves exactly like ICBS(∞) and our heuristics like the single agent version introduced by Felner et al. [6].

Empty							
Agents	Instances	rect_size2	Lookahead	MAMDD	rect_size2	Lookahead	MAMDD
		Runtime (ms)			Nodes		
30	22	431.17	432.84	432.12	14.3	13.5	13.5
40	34	7064.00	9183.49	6505.26	579.6	389.2	385.4
50	35	6223.77	8548.50	6358.32	360.6	286.1	286.3
60	20	7171.54	8113.10	6171.33	474.3	382.8	382.7

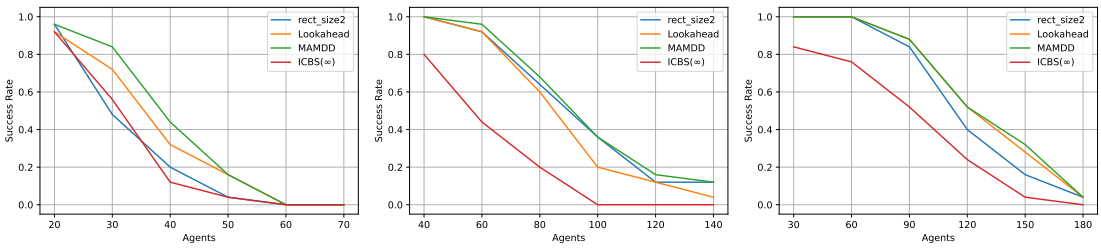
On the empty map, merging the agent pairs which are part of cardinal rectangle conflicts is beneficial. This is reflected in the superior success rates of the rect_size2 algorithms compared to the success rate of ICBS(∞). The addition of our heuristics however does not increase the success rate and in the case of the Lookahead heuristic, it even decreases the number of instances it is able to solve in the time limit.

This is different on the dense map. There, only merging agents in cardinal rectangle conflicts is not sufficient and it even worsens the success rate, which can be seen when comparing the success rates of rect_size2 and ICBS(∞). In contrast to the merging strategy, the heuristics do improve the success rate in every category where the algorithms are able to solve at least one instance.

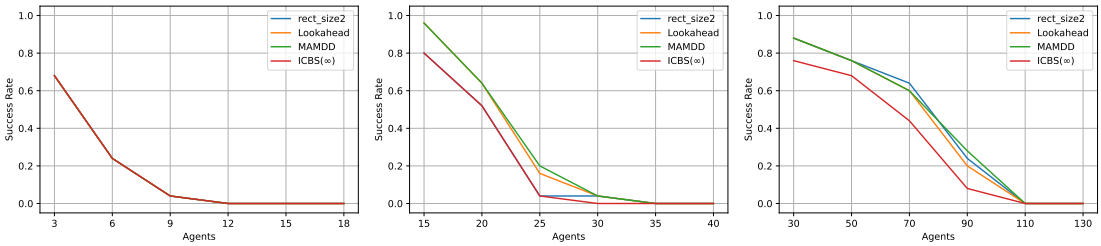
Again, the MAMDD heuristic outperforms the Lookahead heuristic. This can be seen more clearly in Table 6.14, which shows data about the algorithms on the empty map. Regarding the runtime, MAMDD is substantially faster than Lookahead, while providing similar h-values. This is reflected in the number of high level nodes expanded. The difference in expanded nodes of Lookahead and MAMDD is negligible. Compared to rect_size2 with no heuristic, Lookahead adds an increase in time needed to solve the instances, whereas MAMDD is mostly faster or as fast as rect_size2. In terms of expanded nodes, both heuristics provide a sizable reduction.

Real World Maps

We again test our algorithms on our selection of maps based on real world scenarios. In Figure 6.20 we present the success rates of rect_size2 with no heuristics, with the Lookahead heuristic, and with the MAMDD heuristic. To compare their performance to the base algorithm, we also add the success rate of ICBS(∞) to the plots. As expected after analyzing rect_size2 on the empty and dense maps, it performs best on the maps with large open spaces. Open spaces are likely to cause cardinal rectangle conflicts with large rectangles which are expensive to resolve for ICBS(∞). Because of this, rect_size2



(a) brc202d (b) den520d (c) Paris_1_256



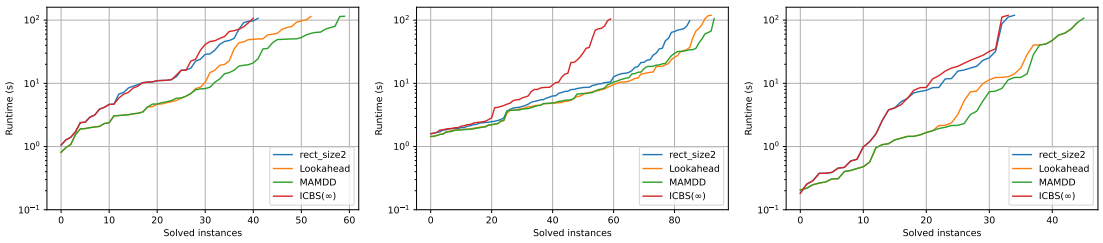
(d) maze-128-128-1 (e) room-64-64-8 (f) warehouse-10-20-10-2-1

Figure 6.20: The success rates of the rect_size2 algorithm without heuristics, with the Lookahead heuristic, and with the MAMDD heuristic for our the maps based on real world scenarios. Additionally, the success rate of the base algorithm ICBS(∞) is also shown.

outperforms ICBS(∞) on the more open maps den520d and paris. Also, the addition of the heuristics increase the success rates even further. On brc202d and room, the rect_size2 merge strategy is no clear improvement to no merging at all.

Although the merging strategy alone is not beneficial on these maps, the heuristics do improve the success rates by a sizable amount. In contrast, on the warehouse type of map the merging strategy is better than ICBS(∞) but the heuristics show no clear improvement at any number of agents. On the maze map, our merging strategy does not find any viable candidates and thus, rect_size2 behaves equally as ICBS(∞). The heuristics also provide no improvement on this map, which we also observed when we tested the ICBS(25) algorithm with our heuristics. Again, this can be attributed to the low number of agents in the instances we use for the maze map.

Success rates only provide a limited amount of information about the decrease in runtimes provided by the heuristics. To further compare our algorithms, we analyze how much time they need to solve the instances. This is shown in Figure 6.21. On the brc202d map the heuristic functions speed up the search noticeably, which also leads to more solved instances overall by them. Also, the MAMDD heuristic works better than Lookahead on this map type for instances which take more than ten seconds to solve. The difference in runtimes is not as apparent on the paris map, but still both heuristics are faster. The gap between the algorithms with heuristics and the algorithm with no heuristic is most



(a) brc202d (b) Paris_1_256 (c) room-64-64-8

Figure 6.21: The runtime distributions of ICBS(∞), rect_size2 with no heuristic, with the Lookahead heuristic, and with the MAMDD heuristic. The x-axis represents the number of solved instances and the y-axis the time taken by the algorithms.

noticeable on the room type map. Both heuristics provide a substantial speed up for the search process. Even though the MAMDD heuristic seems to be superior in the middle, the Lookahead heuristic is able to catch up to it in the end.

CHAPTER 7

Conclusion and Future Work

In this work we designed and implemented improvements for meta-agent based solvers to the MAPF problem. The focus was on reducing the runtime of ICBS by providing beneficial meta-agent assignments upfront given only the initial state of a MAPF instance and making existing admissible heuristics able to handle multi-agents.

Our first proposed approach is to merge agents which are part of pairwise symmetries, which are rectangle, target, and corridor conflicts, with our focus set primarily on rectangle conflicts. The second approach is to use a machine learning model to predict which merges reduce the runtime of the algorithm. For this, we trained support vector machine and multi-layer perceptron classifier models and used them to predict beneficial merges. Additionally, we introduced two admissible heuristics for meta-agents, based on lookahead and MAMDD, which aim to reduce the number of high-level nodes expansions in MA-CBS.

With our merging strategy based on rectangle conflicts, we were able to decrease the runtime of ICBS substantially for instances with no obstacles, which can be seen in the success rates. Although it performed better with this merging strategy, it still was outperformed by the current best CBS version CBSH2-RTC designed by Li et al. [15]. The strategies based on target and corridor conflicts did not provide a notable improvement.

Similarly, the approaches using our machine learning model also did not increase the success rates substantially, while the runtimes on empty maps could be slightly improved. Regarding our two proposed heuristics, MAMDD ourperformed Lookahead on most instances. Although they provide almost identical heuristic values, the calculation process of MAMMD is faster and it is therefore superior. Adding the MAMDD heuristic to ICBS(25) decreased the number of expanded nodes necessary to reach a valid solution on the maps based on real world scenarios substantially. Also, the instances which ICBS(25) was able to solve within a time limit, it was able to solve faster on average with the heuristic added.

Since our heuristic functions are based on the design of Felner et al. [6], adapting them to function similar to the better performing heuristics based on weighted dependency graphs, proposed by Li et al. [13], could also improve their performance. To improve our machine learning approaches, symmetry conflict based features could be used. Since merging agents in rectangle conflicts lead to a better performance, in particular incorporating rectangle conflicts as features seems promising. These features could be about the size of the rectangle, how many other agents are present in the rectangle, or how many pass through it.

Other pair selection strategies could also be employed. Instead of only taking one pair, a tournament selection strategy could be implemented. With such a strategy, multiple pairs could be merged. Then, it seems helpful to extend the training data to also contain entries where more than one pair is merged. Additionally, regarding the labels, extending the classification approach of having the two labels *yes* and *no* to a regression approach with labels showing exactly how much faster or slower merging this pair is, could also yield interesting results. Lastly, merging all our approaches to combine their strengths could result in an algorithm which is superior to all its single components.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Eyal Shimony. Icbs: improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 740–746, 2015.

[2] Eli Boyarski, Shao-Hung Chan, Dor Atzmon, Ariel Felner, and Sven Koenig. On merging agents in multi-agent pathfinding algorithms. In *Symposium on Combinatorial Search (SoCS)*, pages 11–19, 2022.

[3] Eli Boyarsky, Ariel Felner, Guni Sharon, and Roni Stern. Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 47–51, 2015.

[4] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved parameterized upper bounds for vertex cover. In *In Proc. 31st MFCS, volume 4162 of LNCS*, pages 238–249. Springer, 2006.

[5] Rodney G. Downey and Michael R. Fellows. Parameterized computational feasibility. In *Feasible Mathematics II*, pages 219–244. Birkhäuser Boston, 1995.

[6] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, TK Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28, 2018.

[7] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert C Holte, and Jonathan Schaeffer. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research*, 50:141–187, 2014.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Peter E. Hart, Nils J. 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.

[10] Taoan Huang, Sven Koenig, and Bistra Dilkina. Learning to resolve conflicts for multi-agent path finding with conflict-based search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11246–11253, 2021.

[11] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(85)90084-0.

[12] Richard E. Korf. Recent progress in the design and analysis of admissible heuristic functions. In *Abstraction, Reformulation, and Approximation*, pages 45–55, 2000.

[13] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 442–449. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[14] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Symmetry-breaking constraints for grid-based multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 6087–6095, 2019.

[15] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, Graeme Gange, and Sven Koenig. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 301:103574, 2021.

[16] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[17] Stuart Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Pearson, 4 edition, 2021.

[18] David Sculley and Gordon V. Cormack. Filtering email spam in the presence of noisy user feedback. In *CEAS*, 2008.

[19] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013.

[20] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[21] David Silver. Cooperative pathfinding. AIIDE'05, page 117–122. AAAI Press, 2005.

[22] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. *Symposium on Combinatorial Search (SoCS)*, pages 151–158, 2019.

[23] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, 2015.