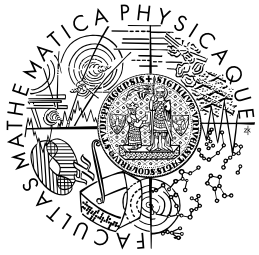


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Mgr. Jiří Švancara

Multi-agent Path Finding

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the doctoral thesis:	prof. RNDr. Roman Barták, Ph.D.
Study programme:	Theoretical Computer Science and Artificial Intelligence

Prague 2020

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date ..... signature of the author

I would like to thank my advisor prof. RNDr. Roman Barták, Ph.D. for his valuable guidance and support during my research and studies. I would also like to thank the numerous reviewers from various conferences for their comments and feedback. Special thanks go to my family and friends who supported me during my studies. The research presented in this thesis was supported by the Czech-Israeli Cooperative Scientific Research Project 8G15027, by Charles University under the SVV project numbers 260 333 and 260 453, by the Czech Science Foundation under the project P103-19-02183S, and by the Charles University Grant Agency under the project 90119.

Název práce: Multi-agentní hledání cest

Autor: Mgr. Jiří Švancara

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí doktorské práce: prof. RNDr. Roman Barták, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Zadáním multi-agentního hledání cest (MAPF z anglického multi-agent path finding) je nalézt nekonfliktní cesty pro neměnnou skupinu agentů, kteří se pohybují ve sdíleném prostředí. Každý z agentů je definován svojí výchozí a cílovou polohou. Tato běžná definice MAPF je velice jednoduchá a často nezohledňuje všechny parametry reálného světa, které je potřeba vyřešit, aby byl problém prakticky aplikovatelný. V této práci se snažíme tento nedostatek odstranit tím, že definici rozšíříme o několik parametrů. Toho dosáhneme v několika krocích. Nejprve představíme přístup k řešení MAPF pomocí převodu na splnitelnost Booleovských formulí. Tento přístup modelujeme v programovacím jazyce Picat, který nám poskytuje snadno upravitelný model, do kterého lze přidávat nové podmínky a omezení. Toho využijeme v dalším kroku, kdy upravíme původní zadání MAPF. Zaprvé povolíme, aby do sdíleného prostředí vstupovali noví agenti během exekuce již nalezeného plánu. Zadruhé relaxujeme požadavek na homogenitu sdíleného prostředí, které je běžně reprezentováno neohodnoceným grafem. V poslední části práce provedeme experimentální studii na skutečných robotech, abychom ověřili, že přidané atributy skutečně modelují situaci lépe než klasická definice.

Klíčová slova: Multi-agentní hledání cest, Splnitelnost, Prohledávání stavového prostoru, Reální roboti

Title: Multi-agent Path Finding

Author: Mgr. Jiří Švancara

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Multi-Agent Path Finding (MAPF) is the task to find efficient collision-free paths for a fixed set of agents. Each agent moves from its initial location to its desired destination in a shared environment represented by a graph. The classical definition of MAPF is very simple and usually does not reflect the real world accurately. In this thesis, we try to add several attributes to the MAPF definition so that we overcome this shortcoming. This is done in several steps. First, we present an approach on how to model and solve MAPF via reduction to Boolean satisfiability using Picat programming language. This provides us with a useful model that can be easily modified to accommodate additional constraints. Secondly, we modify MAPF to portray a more realistic world. Specifically, we allow new agents to enter the shared environment during the execution of the found plan, and we relax the requirement on the homogeneousness of the shared environment. Lastly, we experimentally verify the applicability of the novel models on real robots in comparison with the classical MAPF setting.

Keywords: Multi-agent Path Finding, Satisfiability, State Search, Real Robots

Contents

Introduction

Imagine that an evil wizard has trapped you and a group of your friends in an old labyrinth. He proposes a game – as wizards in riddles usually do. There are exits in the labyrinth, one for each one of you, and everyone also has a map of the labyrinth and a walkie-talkie, so you can communicate. However, to make things worse, the corridors are too narrow to pass each other. The wizard sets a time limit, if you can all reach your assigned exit in time, you all can go home, if not, you will be trapped there forever. Can you escape the labyrinth? Helpful instructions on how to solve this problem can be found in this thesis.

The problem described above is known as multi-agent path finding (MAPF). This problem received a lot of attention in the recent years by the AI community as its applications are numerous and much more practical than in our story – robot navigation, autonomous warehousing, aviation, video games, numerous combinatorial puzzles, and hopefully in the close future systems navigating and coordinating autonomous vehicles.

The theoretical model normally consists of a set of agents that share an environment, usually represented by a graph. An agent is an abstract representation of any entity that is moving in the environment – a car, a train, a plane, a drone, a unit, etc. Each agent is assigned a starting location and a desired destination. The task is to navigate cooperatively all of the agents to their destinations without causing any collisions along their respective paths. The definition of a collision can differ based on the application, however, typically we want to forbid two or more agents to occupy a single physical location. If the application is more theoretical, for example, a video game, we can loosen the restrictions. On the other hand, if we want the movements to be very safe, we can add restrictions to ensure that the agents are keeping extra distance, for example in train transport.

While there are many approaches to solve MAPF, we focus on solving the problem in a centralized way via reduction to Boolean satisfiability (SAT). For this purpose, we create an easily modifiable model that automatically translates constraints into a formula in conjunctive normal form (CNF) that in turn can be solved by any SAT solver. New constraints can be added in a high-level language without any complicated implementation. This is extremely useful as in the latter part of this thesis, we explore new settings and constraints that bring the classical definition of MAPF closer to the real world. Mainly we allow for new agents to appear over time, thus changing the problem into a continuous problem rather than a single snapshot. Another useful change is to the homogeneity of the environment – the under-laying graph is usually expected to be unit-length, we allow for it to have edges of different lengths and capacities of agents that are allowed to move over the edges and/or stay in the vertices. Lastly, to put the theory to test, we perform an experimental study on real robots Ozobot. We observe which constraints are necessary to translate the theoretical movement plan to the robots' action primitives to produce a high-quality plan while also ensuring the safety (ie. no collisions) of the robots.

This thesis consists of the published research results achieved during the PhD study of the author at the Charles University in Prague.

Overview by Chapters

The text is organized as follows (we also highlight parts that are our contribution as opposed to parts that are related work included for better understanding of the overall text).

- **Chapter 1** – the main definitions that are used throughout the thesis are stated. Most of those definitions are well-known in the MAPF community and are used in the latest publications regarding MAPF.
- **Chapter 2** – several approaches to solving the defined problem of MAPF are described. We also try to provide a brief history of the problem as well as a system to categorize the algorithms.
- **Chapter 3** – a closer look into two optimal solvers, search-based and reduction-based, is provided. Our first contribution is creating an easily modifiable reduction-based solver for both makespan and sum of costs optimal solutions. We also provide several models solving the sum of costs optimal version of MAPF. Note that this reduction-based approach is not novel by itself, only the simple implementation and some of the sum of costs optimal models are.

  As our next contribution, we observe that there are instances that are easier to solve using different algorithms. This leads to the idea of combining both mentioned algorithms by splitting the initial instance into smaller subproblems that are in turn solved by different appropriate solvers.
- **Chapter 4** – extensions to the original MAPF problem definition are explored. Namely, we explore situations when new agents can appear over time, and more complex environments modeled by graphs with weighted edges and capacity assigned to the edges. Both of these results are our novel contributions.
- **Chapter 5** – an extensive experimental study on the usefulness of plans when applied to real robots is presented. This is also a novel contribution.
- **Conclusion** – summary of the thesis as well as future work is proposed.
- **Glossary** – for easier understanding, a glossary is included. All of the main definitions from chapter 1 are present as well as notations used solely in any of the previously mentioned chapters.

1. Definitions

In this chapter, we state several formal definitions regarding the multi-agent path finding problem and discuss their implications.

1.1 Instance and Solution

Definition 1. *An* instance of MAPF *is a pair* (G, A)*, where* $G = (V, E)$ *is a graph representing a shared environment and* A *is a set of agents. Each* agent $a_i \in A$ *is represented by a pair* $a_i = (s_i, g_i)$*, where* s_i *represents the start location (sometimes also called the initial location) of agent* a_i*, and* g_i *represents the goal location of agent* a_i*. In both cases, location corresponds to a vertex in the input graph.*

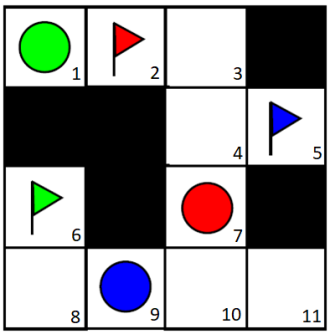


Figure 1.1: An example of MAPF instance. The white cells are vertices of the 4-connected graph, the colored circles denote initial locations of the agents, and the colored flags denote the goal locations of the agents.

A graphical example of an instance of MAPF can be seen in Figure 1.1. In this example, there are three agents (green, red, and blue circles) with three distinct goal locations (green, red, and blue flags). Note that while most of the examples used in publications are grid graphs, just as we used, or at least planar graphs, the MAPF problem itself does not require the underlying graph to have any of these properties. Indeed, the majority of the algorithms presented in this thesis can be used with any type of graph.

Definition 2. *A* plan *for a single agent* i *is a sequence of locations (vertices) where the agent is located at a given timestep. Let* π_i *denote a plan for agent* i*, where* $\pi_i(j) = v$ *represents that agent* i *is located in vertex* v *at timestep* j*. We denote* $|\pi_i|$ *as the length of the plan.*

Definition 3. *A plan* π_i *is said to be a* valid plan *if* $\pi_i(0)$ *and* $\pi_i(|\pi_i|)$ *are the initial and goal locations of* a_i *respectively, and for every* $j = (0, \dots |\pi_i|)$ *either* $\pi_i(j) = \pi_i(j+1)$ *or* $(\pi_i(j), \pi_i(j+1)) \in E$*. This means that at each timestep, the agent is either waiting in the current vertex or moves to a neighboring one.*

From the definitions, it is easy to see that the time is considered to be discrete and at each timestep, every agent either stays in its current location (vertex) or

moves to a new location (vertex connected by an edge). Another assumption is that traversing each edge in the graph takes the same amount of time for all of the agents. This can be understood in such a way that all of the agents are homogeneous in terms of speed and that each of the edges is the same unit length.

So far we have only described plans for a single agent. The core idea and complexity of the multi-agent path finding problem arises from the interactions between agents. Specifically, we have to avoid any conflict between the agents. However, there is more than one way to define what a conflict between two or more agents is.

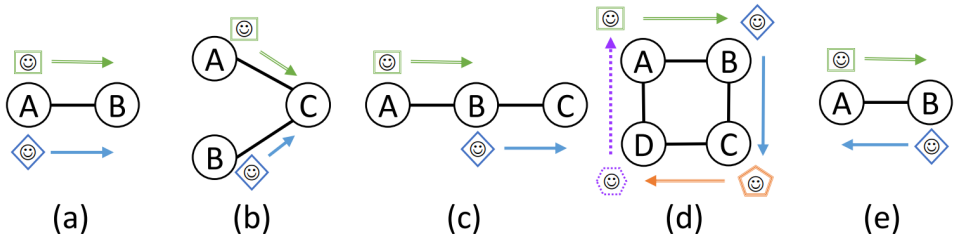


Figure 1.2: All possible MAPF conflict between two or more agents. From left to right *edge conflict*, *vertex conflict*, *following conflict*, *cycle conflict*, *swapping conflict*. Figure taken from [1].

Using a recently proposed MAPF terminology [1], we define five different conflicts that may appear.

Definition 4. *An* edge conflict *(Figure 1.2 (a)) occurs when two agents a_i and a_j move on the same edge in the same direction. I.e. $\pi_i(t) = \pi_j(t) \wedge \pi_i(t+1) = \pi_j(t+1)$ and $\pi_i(t) \neq \pi_i(t+1)$.*

Definition 5. *A* vertex conflict *(Figure 1.2 (b)) occurs when two agents a_i and a_j reside in the same vertex at any timestep. I.e. $\pi_i(t) = \pi_j(t)$.*

Definition 6. *A* following conflict *(Figure 1.2 (c)) occurs when an agent a_i is planned to occupy a vertex that was occupied by another agent a_j in the previous timestep. I.e. $\pi_i(t+1) = \pi_j(t)$.*

Definition 7. *A* cycle conflict *(Figure 1.2 (d)) occurs when a set of more than two agents $a_i, a_{i+1}, \ldots, a_{i+k}$ is planned to move on a fully occupied cycle. I.e. $\pi_i(t+1) = \pi_{i+1}(t) \wedge \pi_{i+1}(t+1) = \pi_{i+2}(t) \wedge \cdots \wedge \pi_{i+k-1}(t+1) = \pi_{i+k}(t)$.*

Definition 8. *A* swapping conflict *(Figure 1.2 (e)) occurs when two agents a_i and a_j move on the same edge in opposite directions. I.e. $\pi_i(t+1) = \pi_j(t) \wedge \pi_i(t) = \pi_j(t+1)$ and $\pi_i(t) \neq \pi_i(t+1)$.*

For most realistic applications, where the moving agents are physical entities (cars, ships, planes, etc.) the edge, vertex, and swapping conflict are forbidden, since in those cases, the agents share the same physical location. As for the following and cycle conflict, it depends on the intended application.

Forbidding some of the conflicts implies that other conflicts can not occur either. If the vertex conflict is not allowed, the edge conflict is impossible as

well. The implication in the other direction is not true. Similarly, if the following conflict is not allowed, both the cycle and swapping conflict are not possible. Again, the implication in the other direction is not true.

In the rest of the thesis, we will work with one of the most common settings of (dis)allowed conflict: edge, vertex, and swapping conflict are forbidden while following and cycle conflicts are allowed. We shall refer to this setting as a *parallel motion*. Another possible and often used setting is so-called *pebble motion* [2] that in addition to parallel motion also forbids following conflicts (and thus also cycle conflicts).

Definition 9. *A* joint plan Π *is a set of valid plans with the same length, one for each agent. We say that* Π *is a* valid joint plan*, if there are no collisions between any of the agents. A valid joint plan is a solution to a MAPF instance. We denote* $|\Pi|$ *as the length of the joint plan.*

This definition of a valid joint plan requires each agent to have the same plan length. This can be achieved simply by prolonging its plan in such a way that the agent stays in its goal location. This is required because once an agent reaches its destination, it does not disappear. Rather it stays in the graph and other agents still need to plan their paths collision-free. Note that once an agent reaches its goal position, it is possible for it to move out and back again in some other time. The whole joint plan ends once all of the agents are located in their respective goal locations.

An example of a valid joint plan for the example in Figure 1.1 can be seen in Table 1.1. Notice that the plan for green and red agents are longer than necessary to compensate for the length of the plan of the blue agent.

Timestep	0	1	2	3	4	5	6	7	8	9
Agent Green	1	2	3	4	7	10	9	8	6	6
Agent Red	7	4	5	5	4	3	2	2	2	2
Agent Blue	9	10	11	11	11	11	10	7	4	5

Table 1.1: An example solution to the MAPF instance defined in Figure 1.1. The numbers correspond to the vertices each agent occupies at that timestep.

A valid joint plan is a solution to an instance of MAPF. It may be obvious, but it is worth noting that there are examples where no solution exists. This depends on the movement constraints used. The most simple instance with no solution is a case where no path from the agent's start to its goal exists (ie. the graph is disconnected). If we are using the pebble motion setting, an instance where all of the vertices are occupied by agents is unsolvable, while it may be solvable under the parallel motion setting. For both of the settings, a simple corridor with two agents that need to pass each other is unsolvable, since the swapping of two agents is forbidden. If we do not enforce any constraint on the conflicts, the problem becomes solvable if there are single-agent paths. However, this setting is not interesting as it reduces the problem to a set of single-agent path finding problems.

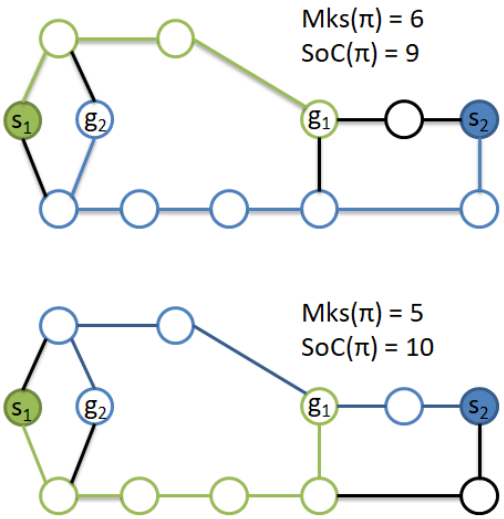


Figure 1.3: MAPF instance with two agents, where optimizing the sum of costs and the makespan objective functions yield different plans. The start and goal locations of both agents are highlighted as well as the found path for each agent. The top plan minimizes the sum of costs. The bottom plan minimizes makespan.

1.2 Cost Functions

In the previous section, we discussed the definitions of an instance and a valid solution. So far we did not care about the quality of the solution and how it impacts the computation complexity.

In the literature, two cost functions are often used to describe the quality of the found solution – *makespan* [3] and *sum of costs* [4]. Briefly said, the makespan is the first time all of the agents are present in their respective goal locations and the sum of costs is the sum of timesteps until all of the agents are in their goal locations and are not required to move again. Specifically, this means that if an agent reaches its goal in timestep t, but is required to move in the future (to allow some other agent to pass) and reaches its goal for the last time in timestep $t+k$, this agents contribution to the sum of costs is $t+k$. We do not subtract the time spent in the goal location before.

To define these cost functions formally, let us denote T_i as the last timestep agent a_i reached its goal location.

Definition 10. *The* makespan *of plan* Π *for agents* A *is*

$$Mks(\Pi) = \max_{a_i \in A} T_i$$

Definition 11. *The* sum of costs *of plan* Π *for agents* A *is*

$$SoC(\Pi) = \sum_{a_i \in A} T_i$$

In addition to finding a valid solution, it may be requested to find a solution that is optimal in one of the defined cost functions. While there are many polynomial-time algorithms that can find a feasible solution, which we will discuss in the next chapter, the task to find either a makespan optimal solution or

a sum of costs optimal solution is an NP-Hard problem [5, 6]. This is true for both pebble motion and parallel motion setting of allowed conflict [7].

In fact, the decision problem if there is a valid solution in T timesteps is NP-Complete. To find an optimal solution, one typically iterates the T until the decision problem returns *yes*. For a more detailed study of the complexity, we refer the reader to [7].

The optimization of either the defined cost functions has a real-life motivation. The minimization of makespan corresponds to minimizing the total time of the execution. This may lead to an unnecessary movement of some agents during the execution, however, the total length of the plan is minimal. On the other hand, minimizing the sum of costs corresponds to minimizing the number of actions each agent performs (including wait actions) and therefore minimizing the fuel consumption of all of the agents combined.

It may not be apparent at first, but these objective functions are not equivalent. Optimizing either the makespan or the sum of costs objective function can yield different solutions [8] – see Figure 1.3 for an example, where minimizing makespan increases the sum of costs and vice versa minimizing the sum of costs increases makespan.

2. Background and Related Work

In the previous chapter, we formally defined the problem of MAPF. In this chapter, we present some interesting points from the history of MAPF as well as algorithms that solve the problem using various techniques. This is just a brief overview of existing solvers without much detail on the inner working and proofs of correctness. Both the history and the list of algorithms are in no way complete. It is meant to provide the reader with some ideas on how the MAPF problem can be solved and how the thinking about the problem evolved. In the next chapter, we will describe two selected solvers in more detail.

2.1 History

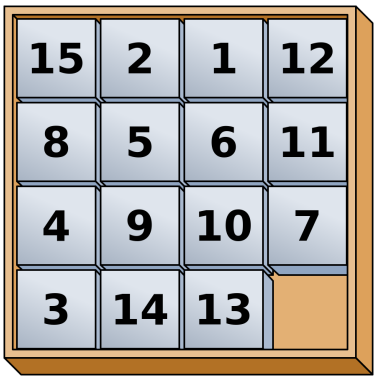


Figure 2.1: 15-puzzle.

One of the first problems that can be seen as a version of MAPF is the famous 15-puzzle dating back to the late 19th century. The puzzle consists of 15 marked tiles in a box of size 4 by 4 (ie. fitting 16 tiles, thus one space is unoccupied, see Figure 2.1). The objective is to slide one tile at a time to reach a desired configuration. It is easy to see how this can be seen as a MAPF problem. This problem has been extensively researched from multiple viewpoints and under many different names (15-puzzle, smaller version 8-puzzle, general version $n \times n$-puzzle, sliding puzzle, etc.). Some studies generalized the problem to any graphs and the tiles to "pebbles", thus providing the pebble motion on graph [2]. Note that the physical movement restriction of the tiles or pebbles in the puzzle exactly corresponds to our definition of pebble motion.

Another area where MAPF is heavily used is in the video game industry. Dating back to the first strategy games in the '90s, the developers faced an interesting challenge. The player is in command of a number of units that need to be moved often. While most players do not think about this much, the task to find a path for each of the units is quite difficult in some cases.

A path for a single unit avoiding static terrain obstacles can be found easily and quite effectively using breadth-first search, Dijkstra's algorithm [9], or A* algorithm [10]. However, when more units need to move at the same time, the other units become moving obstacles that are much harder to deal with. As we

stated earlier, finding an optimal path for a group of agents (units) is computationally hard problem and unsuitable to be computed every time the player issues a new move command. There are some algorithms that compute some solution in polynomial time, however, these solutions may be often much longer and annoying for the player.

The most common problem is when a group of units needs to move through a narrow passage. When the units are moving as a group, the player usually prefers the units to stay together, however, it may be faster for some units to find some path that goes around the narrow passage. Therefore, it is not only required to find a short solution, but also a solution that seems logical to the player.

Figure 2.2: A base in Starcraft with mining units. The mining units do not cause collisions with other units.

Another noteworthy problem is when many units are moving back and forth in a confined space. A nice example of this situation can be seen in strategy game Starcraft, where mining units are moving between base and mineral patches – resources used in the game (see Figure 2.2). The problem is not only in the length of the planned paths but also in great potential for deadlocks. The solution that the developers came up with is quite simple – a harvesting unit does not cause a collision with other units, therefore only a simple single-agent path is needed [11]. Since the release of the first Starcraft game, this has become a feature, however, this "feature" is present solely because of the path finding issues.

Over the years, the game industry became very successful in solving the path finding problem, however, we can still sometimes see some illogical moves that will frustrate us.

In the recent years, more and more real world applications can be seen. These include robotics [12], traffic control [13, 14], and aviation [15].

Over the years, many names have been used to describe the problem of MAPF. The names include for example "motion planning for multiple independent objects", "motion coordination", "multi-robot motion planning", "cooperative pathfinding", etc. In the last years, the term multi-agent path finding (or multi-agent pathfinding) is becoming more common as there are publications that try to synchronize and categorize the research [16, 1]. Also, the community of re-

searches focusing on MAPF is growing which can be seen in workshops dedicated to MAPF and a web page dedicated to MAPF [17], where many researchers share their research, algorithms, and benchmarks.

Not only the naming convention evolved over the years, but, more importantly, the effectiveness of the algorithms evolved as well. While the optimal solvers from the beginning of the 2010s could find a solution for only a few dozens of agents [18], only a few years later, the improved algorithm could deal with almost a hundred agents [19]. Note that the number of agents is not the only parameter of the problem instance that determines the hardness, however, it is usually true that the more agents are present, the harder it is to solve the problem.

Another research paradigm that can be seen in the past years is changing the definition of MAPF to solve a special case of MAPF that can be directly applied to a real world problem. This approach is also a part of our contribution covered in this thesis [20, 21]. Other notable examples include warehousing problem where a stream of tasks are incoming [22]. This is closely related to the task assignment problem – deciding which agent is assigned which task. Another example is the project of autonomous intersection management [14].

2.2 Categorization

The natural way of categorizing the algorithms is in terms of optimality. As mentioned before, the problem of MAPF is NP-Hard if we require an optimal solution with regards to minimal makespan or minimal sum of costs. This leads to the idea of finding a sub-optimal solution while keeping the computational time low.

Sub-optimal solvers A simple approach to solve MAPF is to greedily plan for each agent as if the other agents do not exist and deal with the conflict only when they appear during the execution of the plan. It can be easily seen that this indeed does not guarantee an optimal solution. The local repair A* (LRA*) algorithm [13] follows this idea. Each agent is planned separately and only the agents in the closest proximity (if there are any) are treated as obstacles. If there happens to arise a conflict during execution (i.e. there is another agent present in the location that needs to be traversed), the algorithm needs to be computed again for that blocked agent. Such an approach is often used in the video game industry.

Another idea is to create the whole plan in advance. Again, we start by greedily computing paths for single agents, however, when some agents are already planned, the next agents need to avoid the computed plans. This yields the cooperative A* (CA*) algorithm [13]. Both LRA* and CA* are not guaranteed to find a solution.

A different approach is to take into consideration the type of graph the agents are moving over. Specifically, we mention algorithms that work with grid graphs [23] and bi-connected graphs [24, 25]. These algorithms, that are always guaranteed to find a solution, if a solution exists, use a set of rules to navigate the agents and avoid conflicts.

Some of the Sub-optimal solvers are relaxations of the optimal solver, that we will present next [26].

Optimal solvers Most of the optimal MAPF solvers can be divided into the three following categories.

1. Algorithms based on searching a state-space. These algorithms are often based on the well known A* algorithm [10]. The current position of all of the agents is represented by a state. The initial and goal states are known in advance and the rules of transition from one state to another are also known. A priority queue holds all of the open states, at each timestep, the most promising state is extracted to be explored and all of the possible movements from that state are generated and added to the priority queue. This is repeated until the goal state is found. If the open states are explored in the correct order, the algorithm yields an optimal solution. The limitation of this algorithm is a huge branching factor (number of all possible combinations of movements from the current state). This disadvantage can be countered by not allowing all of the agents to move at the same time, but rather to move them one by one [27]. Additionally, some clever heuristics can be used to steer the A* algorithm in the correct direction [28].

2. Constraint-based search. Rather than searching in a state space, we can search over constraints to each agent's movement. In the literature, the most often used algorithm that falls into this category is Conflict Based Search (CBS) algorithm [18]. The general idea is to find a single-agent path for each agent (this can be done in polynomial time) and check if there are any collisions between the plans. If there are any, we add constraints that disallow this conflict and repeatedly find single-agent plans until no collision is present. Since this is such a popular algorithm, and we are also using it as a comparison to our models, we shall describe it in more detail later.

   Another algorithm that searches over the constraints on single-agent plans is Increasing Cost Tree Search (ICTS) [4]. For each agent a_i, all of the possible plans with length $|\pi_i| = t$ are computed. If there is a combination of plans that form a valid joint plan, this is the solution. If there is no such combination, the restriction on plan length t is increased.

3. Reduction to another problem. As is common in solving NP-Hard problems, we can reduce the problem of MAPF to some other formalism, for which there are heavily optimized solvers. Arguably, the most common approach is a reduction to the problem of the satisfiability of Boolean formulae (SAT) [29, 30, 31]. The basic idea is that propositional variable is introduced for each triple (i, v, t) meaning that agent a_i is located in vertex v at timestep t. The formula consists of constraints over these variables that ensure that the satisfying assignment corresponds to a valid MAPF solution. As this is the approach that we use almost exclusively, we will describe it in much more detail later.

   Another reduction that can be found in the literature include reductions to constraint satisfaction problem (CSP) [32], integer linear programming (ILP) [33], and multi-commodity flow problem [34].

3. Solving MAPF Optimally

In the previous chapter, we briefly described numerous techniques that solve MAPF. In this chapter, we present in much more detail two popular approaches to solve MAPF optimally that we will be using in the rest of the thesis.

3.1 CBS

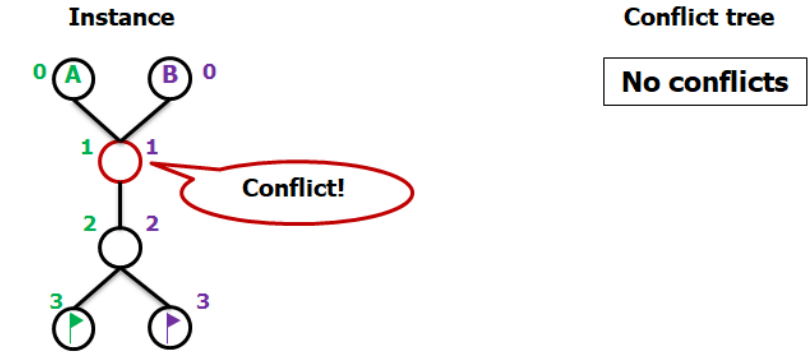
The first optimal MAPF solver that we describe in more detail is Conflict Based Search (CBS) [18]. This solver is considered to be a state-of-the-art algorithm that most papers compare with [35, 29]. CBS was introduced as a way to reduce the number of explored states when using the A* algorithm that explores the state-space. Rather than exploring the possible states, the CBS algorithm finds single-agent paths and performs an exploration over the conflicts caused by these paths.

The algorithm can be seen in two levels – high-level and low-level. The high-level is a search over a binary constraint tree. A node[1] in the constraint tree stores three types of data: the constraints in the format $Cons(a_i, v, t)$ meaning that the agent a_i can not be present in vertex v at timestep t, a set of shortest single-agent paths for each agent that follows the constraints, and a cost of this solution. A best-first search is performed over this tree. At each timestep, a node with the best cost is selected and is explored. The exploration checks if the single-agent paths do not cause any conflicts, and if so, the solution of this node is a valid and optimal solution to the MAPF problem. Otherwise, a conflict is found and two children nodes are created. Let us say that the conflict is that two agents a_i and a_j are present at the same vertex v at the same timestep t. One of the children nodes will add a constraint $Cons(a_i, v, t)$ and the other children node will add a constraint $Cons(a_j, v, t)$. It is easy to see that the valid solution indeed needs to follow one of these constraints, however, we do not know which, so we need to explore both possibilities.

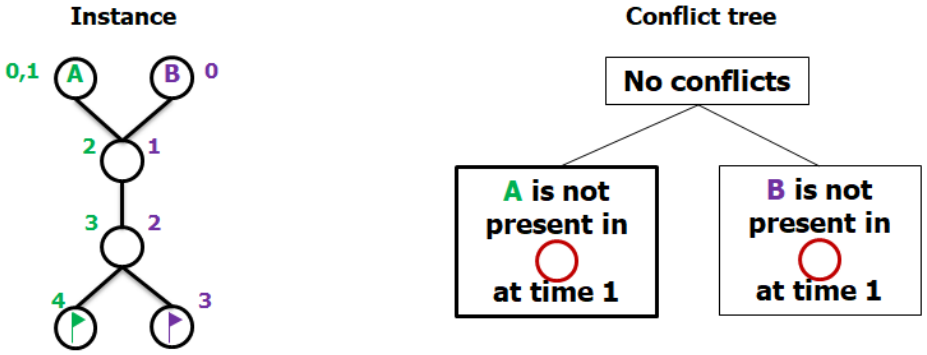
The lower-level consists of finding a single-agent optimal path that follows the constraints for each agent. This path can be found by using any shortest path algorithm, for example, Dijkstra's algorithm [9] or A* algorithm with distance heuristic function.

Some details need to be addressed for the algorithm to work correctly and efficiently. Firstly, each node, with the exception of the root node, needs to store only the new constraint that is added, since all of the other constraints can be found on the way to the root. Similarly, only the shortest path for the agent that is affected by the new constraint needs to be computed as the other paths remain the same. Next, the only conflict that we considered so far is the vertex conflict. If we wish to model the parallel motion, the swapping conflict also needs to be dealt with. To do this, we allow the constraint $Cons(a_i, e, t)$ meaning that an agent a_i is not traversing the edge e in timestep t. Lastly, it is easy to see that

[1]The terms node and vertex are usually interchangeable, however, we will reserve the term "vertex" for the graphs that represent the shared environment in which the agents are moving, while the term "node" will refer to nodes in the binary constraint tree used in the CBS algorithm.



(a) The initial single-agent paths.



(b) The optimal solution.

Figure 3.1: Example of CBS algorithm solving a MAPF instance. Graph on the left is the instance with two agents. The graph on the right is the binary constraint tree.

the cost function that is being optimized can be easily changed by changing how the cost of a solution associated with a node is computed.

An example of the algorithm on a small instance can be seen in Figure 3.1. Two agents – green A and purple B – need to traverse the same corridor to reach their respective goal vertices. At first, the shortest paths are computed with no restrictions. After validating this solution, it is found that there is a conflict. Two new child nodes are created forbidding either A or B presence at the conflicting vertex. Exploring either of these nodes provides a valid solution.

The algorithm can be further improved by applying some rules. For example, it is quite common for a single agent to have a non-unique shortest path. In such a case, the path that causes the least amount of conflicts is chosen. Another

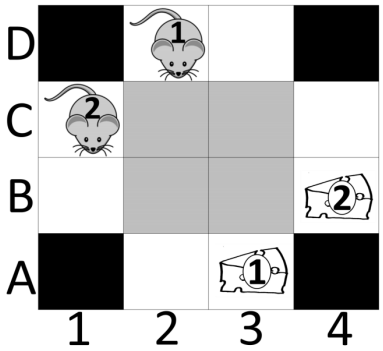


Figure 3.2: Example of a pathological instance for CBS algorithm. Figure taken from [18].

technique is to merge agents that collide often into one meta-agent and solve this group by some other optimal solver [36]. Yet another improvement is to choose carefully which conflicts to resolve first [19].

The experimental results show that the CBS algorithm is indeed successful, especially in cases where the agents do not interact too much. However, the authors of the first paper that describes this algorithm provide a pathological example where the algorithm performs poorly. This example can be seen in Figure 3.2. In this example, the two agents explore all combinations of their possible shortest paths before one of the agents prolongs their path. It is easy to see that this example can be scaled up.

3.2 Reduction to SAT

In this section, we introduce our approach to solving the MAPF problem via reduction to Boolean satisfiability (SAT) both in the makespan and the sum of costs optimal way [37, 35]. Unlike in the CBS algorithm, it is not as easy to change the algorithm to compute makespan or the sum of costs optimal solution. Note that the core idea of reduction to SAT is not novel, however, we introduce our framework and some minor improvements.

3.2.1 Makespan Optimal Model

As the plan length is unknown in advance, reduction-based approaches to solve planning problems use the method of solving the problem with restricted plan length and, in case of failure, increasing the length limit [38]. This makes it natural to look for shortest plans so we shall also start by describing the classical makespan optimal translation of MAPF to SAT.

Let's assume that we are looking for a solution to the MAPF problem with makespan T using the parallel motion restriction on allowed conflicts. We define the following two sets of variables: $\forall v \in V, \forall a_i \in A, \forall t \in \{0, \ldots, T\} : At(v, i, t)$ meaning that agent a_i is at vertex v at timestep t; and $\forall (u, v) \in E, \forall a_i \in A, \forall t \in \{0, \ldots, T-1\} : Pass(u, v, i, t)$ meaning that agent a_i goes through an edge (u, v) at timestep t. More specifically, it starts traversing the edge at timestep t and enters the vertex v at timestep $t + 1$. This is why the variables are not defined

for timestep T. An auxiliary edge (v, v) is added to E, thus $Pass(v, v, i, t)$ means that agent a_i stays at vertex v at timestep t. To model the MAPF problem, we introduce the following constraints:

$$\forall a_i \in A : At(s_i, i, 0) = 1 \tag{3.1}$$

$$\forall a_i \in A : At(g_i, i, T) = 1 \tag{3.2}$$

$$\forall a_i \in A, \forall t \in \{0, \ldots, T\} : \sum_{v \in V} At(v, i, t) \leq 1 \tag{3.3}$$

$$\forall v \in V, \forall t \in \{0, \ldots, T\} : \sum_{a_i \in A} At(v, i, t) \leq 1 \tag{3.4}$$

$$\forall u \in V, \forall a_i \in A, \forall t \in \{0, \ldots, T-1\} : \\ At(u, i, t) \implies \sum_{(u,v) \in E} Pass(u, v, i, t) = 1 \tag{3.5}$$

$$\forall (u, v) \in E, \forall a_i \in A, \forall t \in \{0, \ldots, T-1\} : \\ Pass(u, v, i, t) \implies At(v, i, t+1) \tag{3.6}$$

$$\forall (u, v) \in E : u \neq v, \forall t \in \{0, \ldots, T-1\} : \\ \sum_{a_i \in A} (Pass(u, v, i, t) + Pass(v, u, i, t)) \leq 1 \tag{3.7}$$

The constraints (3.1) and (3.2) ensure that the starting and goal positions of all agents are valid. The constraints (3.3) and (3.4) ensure that each agent occupies at most one vertex and every vertex is occupied by at most one agent. The correct movement in the graph is forced by constraints (3.5) – (3.7). In sequence, they ensure that if an agent is in certain vertex, it needs to leave it by one of the outgoing edges (3.5). If an agent is using an edge, it needs to arrive at the corresponding vertex in the next timestep (3.6). Finally, we forbid two agents to occupy two opposite edges at the same time (forbidding swapping conflict) (3.7).

To find the optimal makespan, we iteratively increase the makespan T until a satisfiable formula is generated. This clearly provides the makespan optimal solution as the iterative approach guarantees that no solution with a smaller makespan exists.

To better visualize the above-described constraints, we can use the notion of a *time-expanded graph*. See Figure 3.3 for reference. We create T copies of the original vertices from graph G and add edges (u_i, v_{i+1}), iff there is an edge $(u, v) \in E$ in the original graph (these edges correspond to move actions), and we also add edges (u_i, u_{i+1}) for each vertex (these edges correspond to wait actions). This creates a time-expanded graph G_T. Agents are moving over this graph in such a way that they start in the 0-th layer and at each timestep they move to the next layer. Constraints (3.3) and (3.4) ensure that agents do not collide in vertices. No swap condition is provided by constraint (3.7).

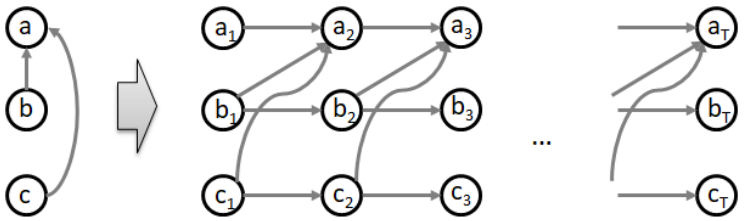


Figure 3.3: Example of a graph on three vertices being transformed to time-expanded graph with T layers

It is possible to speed up the computation by using a better lower bound for the makespan instead of starting with $T = 1$. A straightforward lower bound is to compute for each agent a_i the shortest path SP_i from agent's start location s_i to agent's goal location g_i. The lower bound for T is then the longest of these shortest paths $LB(Mks) = \max_{a_i \in A} SP_i$.

Another way to enhance the computation is to do a pre-processing for the variables $At(v, i, t)$. These variables correspond to an agent being present at some location at a time. However, for some locations, we can determine, that the specific agent can not be present at the specific time, because we know where the agent needs to be present at times 0 and T. Specifically for agent a_i, if vertex v is distance d away from start location s_i, we know that the agent a_i can not be present in vertex v in times $0, \ldots, (d-1)$ simply because it can not travel the whole distance in time. Similarly, if vertex v is distance d away from goal location g_i, agent a_i can not be present in vertex v in times $T - d + 1, \ldots, T$.

As a result, we can also do a pre-processing for the variables $Pass(u, v, i, t)$. If we determined that an agent can not be present at vertex v at time t, we can conclude that neither incoming or outgoing edges of that vertex can be used in time t.

3.2.2 Sum of Costs Optimal Models

Solving the MAPF problem makespan optimally via reduction to SAT is quite straightforward in the sense that once we find the correct number of layers in the time-expanded graph, we are guaranteed that this is the makespan optimal solution. On the other hand, if we use the same approach to finding the sum of costs optimal solution, the first solvable formula is not guaranteed to provide the sum of costs optimal solution. Recall Figure 1.3 from chapter 1, indeed we would first find the makespan optimal solution with $Mks(\Pi) = 5$ and $SoC(\Pi) = 10$. However, by adding another layer to the time-expanded graph, we are able to find a better solution with respect to the sum of costs (which is actually the sum of costs optimal solution in this example). In general, it is not clear how many extra layers need to be added to guarantee that the found solution is optimal. This exact problem is addressed in the following two models.

To design models for sum of costs optimization, we can keep the core model with all of the constraints (3.1) – (3.7) as they describe proper paths for agents and they are not related to any particular objective.

In addition, we assume that we can encode a constraint

$$SoC(\Pi) \leq UB(SoC) \tag{3.8}$$

which ensures that the found plan Π has sum of costs at most $UB(SoC)$, which is some upper bound.

The constraint (3.8) allows us to use a dichotomic branch-and-bound technique to optimize the value $SoC(\Pi)$

$$Minimize\_SoC(LB(SoC), UB(SoC)) \tag{3.9}$$

which, given a lower bound $LB(SoC)$ and an upper bound $UB(SoC)$, ensures that the found plan Π has the minimal sum of costs in the interval $\langle LB(SoC), UB(SoC)\rangle$. This can be achieved by iteratively using the constraint (3.8) and halving the specified interval. Therefore, this technique is just a syntactic shortcut for repetitive usage of constraint (3.8) with more precise bounds.

Model 1

The pioneering SAT-based model to compute the sum of costs optimal solution [35] focuses on bounding both makespan (i.e., the number of layers in the time-expanded graph) and the sum of costs together using constraint (3.8). The bounds are set in such a way that the first solvable formula generated corresponds to the sum of costs optimal solution.

Algorithm 1 Model 1

function MODEL 1
  $\forall a_i \in A : SP_i = shortest\_path(s_i, g_i)$
  $LB(Mks) = \max_{a_i \in A} SP_i$
  $LB(SoC) = \sum_{a_i \in A} SP_i$
  $\delta \leftarrow 0$
  while No Solution **do**
    solve_MAPF($LB(Mks) + \delta, LB(SoC) + \delta$)
    $\delta \leftarrow \delta + 1$
  end while
end function

See Algorithm 1 for the description of the model. We first start by finding the shortest path for each agent while ignoring all other agents. The maximum length of these paths is a valid lower bound for makespan, as was mentioned before. For similar reasons, the sum of the lengths of these paths is a valid lower bound for the sum of costs – each agent is guaranteed to travel at least this distance. The function solve_MAPF(T, C) generates the SAT model with constraints (3.1) – (3.7), makespan T, and using C as the $UB(SoC)$ in constraint (3.8).

Let δ be the extra movement that is allowed to the agents. At first, we start with $\delta = 0$ and try to solve the MAPF problem with the lower bounds. If this is possible then we know it must be the optimal solution, since both values (for makespan and sum of costs) are lower bounds. If there is no solution with these restrictions, we increase δ by 1. Notice that this increment adds an extra layer to the time-expanded graph and also allows some agent to make one extra step.

It has been shown that this is sufficient to find the sum of costs optimal plan and furthermore it will be the first solvable formula [35]. We will now present a different proof of soundness of the approach that opens doors for our novel model.

Theorem 1. *If there exists a solution with the sum of costs $LB(SoC) + \delta$ then this solution can be found in a time-expanded graph with $LB(Mks) + \delta$ layers.*

Proof. Assume that there exists a solution with the sum of costs $LB(SoC) + \delta$, where $LB(SoC) = \sum_{a_i \in A} SP_i$. It means that this plan uses δ extra actions in addition to $LB(SoC)$ actions that are necessary for each agent to cover its shortest path. These extra actions can be used by any of the agents. If agent a_i uses k extra actions then we need a time-expanded graph with $SP_i + k$ layers to model its path. Hence, the largest time-expanded graph is needed if all of the extra actions are used by the agent with the longest of the shortest paths (ie. $\max_{a_i \in A} SP_i$), which is exactly equal to $LB(Mks)$. It means that a time-expanded graph with $LB(Mks) + \delta$ layers is enough to model all paths. □

Model 1 incrementally increases δ both for the sum of costs and for makespan. If it finds a satisfiable model, then according to Theorem 1, this model gives the sum of costs optimal solution (if there was a better SoC-optimal solution then it would be found for a smaller makespan).

Model 2

By observing the behavior of Model 1, we noticed that smaller time-expanded graphs are refuted one by one by using too tight upper bound for the sum of costs. This brought us to the idea of skipping the iterative increase of δ by one and, rather, going directly to the makespan that guarantees the existence of the sum of costs optimal solution, even at the expense of over-estimating the number of layers of the time-expanded graph required. The new model is called Model 2.

Algorithm 2 Model 2

function MODEL 2
  $\forall a_i \in A : SP_i = shortest\_path(s_i, g_i)$
  $LB(Mks) = \max_{a_i \in A} SP_i$
  $LB(SoC) = \sum_{a_i \in A} SP_i$
  $\gamma \leftarrow 0$
  while No Solution **do**
    $SoC \leftarrow$ opt_MAPF($LB(Mks) + \gamma$, $LB(SoC), |A| * LB(Mks) + \gamma$)
    $\gamma \leftarrow \gamma + 1$
  end while
  $\delta \leftarrow SoC - LB(SoC)$
  opt_MAPF($LB(Mks) + \delta, LB(SoC), SoC$)
end function

See Algorithm 2 for reference. Again, we start by finding the shortest paths for each agent and computing the lower bounds for both makespan and sum of costs. We then find the makespan optimal solution for the problem just as it was described in the previous section. In this step, there is no restriction on the sum

of costs. This solution gives us some sum of costs, that we will use as an upper bound.

Computing δ in this algorithm gives information on how many extra steps all of the agents used in the found solution. Following the idea of Theorem 1, $LB(Mks)+\delta$ is the number of layers in the time-expanded graph that guarantees to find the sum of costs optimal solution. Finding this optimal solution is the last step in the algorithm.

The function opt_MAPF(T, L, U) generates the SAT model with constraints (3.1) – (3.7), makespan T, and using L and U as $LB(SoC)$ and $UB(SoC)$ respectively for the dichotomic branch-and-bound search using (3.9). If the model is satisfiable, the function returns the minimal value of SoC within the interval $\langle L, U\rangle$.

In particular, when finding the makespan optimal solution, we let the interval be $\langle LB(SoC), |A| * LB(Mks) + \gamma\rangle$. The upper bound allows each agent from A perform as many actions as possible in the currently given makespan. This means that there is no restriction on how many steps each agent can take.

When we are finding the makespan optimal solution to get an upper bound on the sum of costs, there is no need to find the optimal solution. In fact, any solution provided by some polynomial sub-optimal solver would suffice. However, these solutions tend to overestimate the solution quite a bit and, therefore, produce much higher δ, resulting in a bigger time-expanded graph. The trade-off is so big that in the algorithm, we find not only a makespan optimal solution, but from all of the makespan optimal solutions, we select the one with the minimal sum of costs.

This approach can be summed in a sequence of three optimizations – first, we optimize the makespan, then, for that makespan, we optimize the sum of costs, and lastly, from these upper and lower bounds, we create a sufficiently large time-expanded graph, on which we once again optimize the sum of costs, which is guaranteed to be the globally optimal sum of costs of that problem.

Reduction of Used Variables

An improvement that can be applied to both of the described models focuses on decreasing the number of variables that enter the SAT solver. Once again recall the time-expanded graph used to solve the MAPF problem makespan optimally. A possible way to look at the visualization is that the agents are not moving over one time-expanded graph, but rather, each agent has its own time-expanded graph. These graphs are then interconnected by the constraints that prohibit collisions.

When computing makespan optimal solutions, we need for all agents to have the time-expanded graph of the same size (same number of layers), since we do not know how much of movement each agent needs to perform. On the other hand, when optimizing the sum of costs, we do not necessarily need the same number of layers for each agent.

Assume that there are agents a_i and a_j with their respective shortest paths SP_i and SP_j and furthermore SP_i is the longest of all of the shortest paths, while SP_j is much shorter. Now recall Model 1 and Model 2, where optimizing the sum of costs means, that even agent a_j is allowed to move in the time-expanded graph with $SP_i + \delta$ layers, however, there is allowed only δ extra movement for

all of the agents combined. Even if all of that movement was used up by agent a_j, there would still be $(SP_i - SP_j) > 0$ extra layers in the time-expanded graph that could not be used. This creates superfluous variables that the SAT solver needs to work with.

The improvement applied to Model 1 and Model 2 then works as follows. When computing function *solve_MAPF*, we create for each agent a_i a separate time-expanded graph TEG_i with $SP_i + \delta$ layers. The time-expanded graphs are interconnected by the constraints (3.1) – (3.7) as usual. The last thing to solve is the goal location g_i of an agent with a smaller SP_i. When an agent reaches its goal, it needs to stay in that goal (i.e. it does no disappear) and all the other agents need to avoid it. We can simply achieve this by forbidding vertex g_i for all other agents in timesteps greater than $SP_i + \delta$.

Note specifically in Model 2 that we use this enhancement only when finding the sum of costs optimal solution (i.e. the second call to solving MAPF in Algorithm 2). We do not want to use this enhancement while finding makespan optimal solution, because it does not guarantee to find one.

Using this enhancement on Model 1 and Model 2, we create Model 1+ and Model 2+.

3.2.3 Experiments

Implementation

Each of the described models was implemented using the Picat language (Picat language and compiler version 2.2#3), which is a logic-based programming language similar to Prolog. The main advantage, and the reason this tool was used, is that the necessary constraints are easily represented and then automatically translated to a propositional formula. See Figure 3.4 for an example of code in the Picat language. Notice the similarity between constraints (3.1) – (3.7) and the code itself.

In addition, the language provides tools to solve the generated formula and to add the constraint (3.8) while applying the branch-and-bound technique (3.9). Moreover, it has been shown that the solver using Picat language is comparable with the state-of-the-art SAT-based MAPF solver [29].

Instances

To test the described models, we generated pseudo-random instances. The instances are 4-connected grid maps with increasing sizes from 8×8 to 16×16 with an increment of 2. To introduce some influence between the agents, 20% of the cells in the grid were marked as an impassable obstacle. Representation of one of the maps can be seen in Figure 3.5.

An increasing number of agents were placed into the created grids. If the grid was of size $W \times W$ then the number of agents was in a range of W to $2W$ with an increment of 2. The start and goal locations were chosen randomly in such a fashion, that no two agents are to start in the same location or to end in the same location. Each of such setting was created five times. Together, this yields 175 unique instances.

```
import sat.

path_for_delta(N,E,As,K,M) =>
    ME = M - 1,

    B = new_array(M,K,N),
    C = new_array(ME,K,E),

    % Initialize the first and last states
    foreach(A in 1..K)
        (V,FV) = As[A],
        B[1,A,V] = 1,
        B[M,A,FV] = 1
    end,

    B :: 0..1,
    C :: 0..1,

    % Each agent occupies up to one vertex at each time.
    foreach (T in 1..M, A in 1..K)
        sum([B[T,A,V] : V in 1..N]) #=< 1
    end,

    % No two agents occupy the same vertex at any time.
    foreach(T in 1..M, V in 1..N)
        sum([B[T,A,V] : A in 1..K]) #=< 1
    end,

    % if an edge is used in one direction
    % it can not be used in the other direction (no swap)
    foreach(T in 1..ME, EID in 1..E)
     oposit_edges(EID, E, EList),
     sum([C[T,A,W] : A in 1..K, W in EList]) #=< 1
    end,

    % if an agent is in a node
    % it needs to move through one of the edges from that node
    foreach(T in 1..ME, A in 1..K, V in 1..N)
     out_edges(V,E,EList),
        B[T,A,V] #=> sum([C[T,A,W] : W in EList]) #= 1
    end,

    % if agent is using an edge
    % it must arrive to the connected node in next timestep
    foreach(T in 1..ME, A in 1..K, EID in 1..E)
        edge(EID,_,V),
        C[T,A,EID] #=> B[T+1,A,V] #= 1
    end,

    solve(B).
```

Figure 3.4: Example of a Picat code solving MAPF in the makespan optimal way.

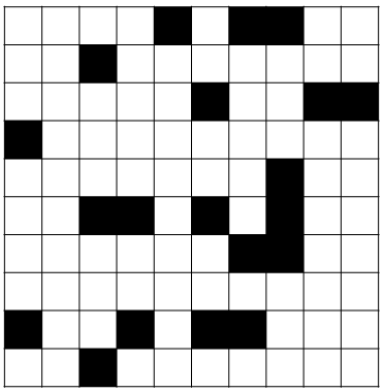


Figure 3.5: Example of a 10×10 grid graph with obstacles used in the experiments.

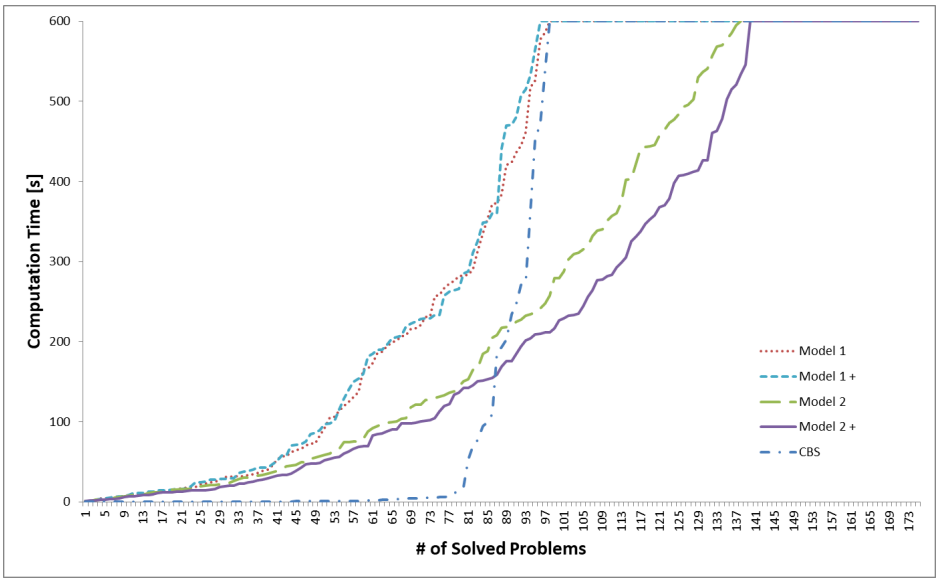


Figure 3.6: Measured results of the experiments. The y-axis is measured runtime. The x-axis describes the number of solved problems within a given time

Results

All of the generated instances were solved by each of the described models. In addition, we used Conflict Based Search (CBS) [18] solver as a state-of-the-art sum of costs optimal solver. The timeout for each instance was set to 600 seconds. Since all of the solvers compute the same problem and the quality of the solution is the same, the main property we are interested in is the time it takes each model to compute it. This result can be seen in Figure 3.6.

The instances are ordered based on how long it took to complete them, therefore, the lower the line in the results graph, the better. We can see that CBS is hands-down fastest for about 90 instances, however, then there is a rapid increase in computation time and it becomes the slowest solver. The four reduction-based models seem to be ordered quite well with model Model 2+ being the fastest, closely followed by model Model 2 and Model 1 and Model 1+ being the slowest.

A more detailed analysis of the performance can be seen in Table 3.1. The most instances solved were by Model 2 and Model 2+, 137 and 139 respectively.

	M. 1	M. 2	M. 1+	M. 2+	CBS
# of solved	97	137	95	**139**	97
# of fastest	0	4	3	46	**88**
# of fastest (without CBS)	9	8	6	**118**	–
IPC score	16.11	44.56	16.76	57.07	**92.50**
IPC score (without CBS)	57.19	110.54	53.93	**134.29**	–

Table 3.1: Number of solved instances, number of times the specified solver was fastest, and IPC scores (both in total and when comparing only reduction-based solvers). "Model" is abbreviated to "M."

All of the other solvers solved a similar number of instances – 95 to 97. If we disregard CBS for the moment, we can see that Model 2+ was fastest on most of the instances, while the other reduction-based solvers were fastest on only a few instances.

To see the difference between the computation times, we compute IPC score[2] for each instance. Solver gets a score of 0 if it did not manage to solve the instance in a given time limit. Otherwise, the score is computed as

$$\frac{min.\ time}{solver\ time},$$

where *min. time* is the time it took the fastest solver and *solver time* is the time it took the solver in question. This produces a score in the range from 0 to 1, where the bigger the number the better. The scores of all instances were summed and the result is presented in Table 3.1 as IPC score. We can see that (if we disregard CBS again) the Model 2+ is indeed the most successful closely followed by Model 2. Model 1 and Model 1+ reached similar results.

If we take into consideration CBS as well, we can see a trend similar to the one seen in the graph in Figure 3.6. If CBS is able to solve the instance in the given time limit, it can solve it faster than the reduction-based solvers. This is apparent from the similar number of solved instances by CBS and instances on which CBS is the fastest.

On the other hand, the reduction-based solvers were able to solve more (or the same) number of instances than CBS. In this sense, Model 2 and Model 2+ are the most successful.

An interesting thing to note is that the enhancement "+" added to Model 1 and Model 2 seems to be useful only for Model 2, while applied on Model 1 it gives no advantage.

3.3 Combining the Two Approaches

As can be seen from the description of the CBS algorithm and the reduction-based algorithm, both are exponential in some value. In this section, we will identify

[2]The evaluation score was introduced in recent International Planing Competitions, hence the name.

the type of instances that are hard for each of the two algorithms respectively. We will use this knowledge to craft a solver that combines both of the algorithms and hopefully has a better runtime than either of the two [39].

The satisfiability problem is exponential in the number of variables of the propositional formula. The reduction of MAPF to SAT produces one variable for each triplet (agent, vertex, time). If we increase the number of agents, the number of variables increases linearly. On the other hand, if we increase the size of the graph and assume that the starting and goal locations of the agents are randomly chosen, the length of the shortest path (path of an agent if no other agents are present in the graph) for each agent increases as well. This is important because the longest of the optimal paths is a lower bound for the makespan T. This means that increasing the graph size can make the problem harder than increasing the number of agents.

Of course, this does not hold true in every case since some formulas are harder for SAT solvers than others. However, it can be generally said that the reduction of a MAPF problem to a satisfiability problem is effective on smaller graphs, even with a high density of present agents.

On the other hand, CBS is exponential in the number of conflicts between agents in the planned paths. While it is fast (polynomial-time) to find a path for a single agent that follows the constraints defined by the constraint tree, the binary constraint tree itself is exponential in the number of explored conflicts. This generally means that the fewer conflicts there are on the optimal path of each agent, the faster CBS is.

Increasing the graph size, while fixing the number of agents (again assuming that the initial and goal locations of the agents are randomly chosen), decreases the likelihood of conflict between the optimal path of each agent. Conversely, increasing the number of agents, while fixing the size of the graph, increases the probability of conflict. This means that the CBS algorithm is more effective on graphs with a low density of present agents.

Combining these two observations, we see that there are instances where reduction to SAT outperforms CBS and vice versa. If we want to only choose the appropriate algorithm for the input instance, it is possible to just start both algorithms in parallel and see which one terminates and yields result faster.

However, we conjure that there are hidden structures in the input instance that can be separated and each can be solved by an appropriate algorithm. Indeed, assume that the set of agents A can be split into two disjoint sets A_1 and A_2, where any optimal solution for A_1 does not collide with any optimal solution for A_2. Furthermore, A_1 forms a dense MAPF instance, while A_2 forms a sparse instance. Then it is more effective to find paths for the agents in A_1 by reduction to SAT and for the agents from A_2 by CBS.

In general, the input instance may be split into many disjoint sets of agents and each set can be solved by a different algorithm. If the found optimal solutions are independent (i.e., there are no collisions between these solutions), we can merge them to form a valid optimal solution for the whole instance [27]. This way, we are able to find the solution faster than by running either of the algorithms (reduction to SAT or CBS) on the initial input instance.

3.3.1 Automatic Decomposition

It is clear that an automatic way of splitting the agents into independent subsets is needed in order to create a useful algorithm. We propose such a method, based on the Independence Detection (ID) algorithm [27]. ID is a MAPF framework that attempts to split a MAPF instance into independent subproblems such that combining their solutions yields a valid solution for the whole instance. In addition, the ID algorithm maintains that if the solutions for the independent subproblems are optimal then the combined solution for the whole instance is also optimal [27].

Algorithm 3 Independence Detection

```
1: assign each agent to a group
2: compute plan for each group
3: while there is a conflict in plans do
4:     G1, G2 ← conflicting groups
5:     if G1 and G2 conflicted before then
6:         Merge G1 and G2 into new group G
7:         Find plan for G
8:         Continue
9:     else if can replan G1 and avoid G2 then
10:        Replan G1 and avoid G2
11:        Continue
12:    else if can replan G2 and avoid G1 then
13:        Replan G2 and avoid G1
14:        Continue
15:    else
16:        Merge G1 and G2 into new group G
17:        Find plan for G
18:    end if
19: end while
20: solution ← path of all groups combined
```

ID works as shown in Algorithm 3. We start by assigning each agent into its own group and finding an optimal solution for each group while ignoring all other groups. In the beginning, this is a single-agent path for each group. We then repeatedly check the solutions of all groups for conflicts. If there is no conflict, and each group has an optimal plan, these plans can be combined into an optimal plan for the whole instance.

Assume that there are two conflicting groups G_1 and G_2. If we can replan one of the groups while avoiding conflict with the other, and keeping the same cost of the found plan, we will keep that solution (lines 9 – 14). If no such replanning is possible, we have to merge G_1 and G_2 into a single new group and find a new optimal plan (line 16).

By changing the plan of one group, we can create a new conflict with some other group. This can lead to infinite cycles. To fix this problem, we have to keep track of groups that were already checked for conflicts. If we visit such a pair again, we are possibly in a cycle, therefore, we merge the groups immediately into a new group and find optimal paths for that group (line 5).

There exist some enhancements for the ID algorithm [40]. These enhancements allow returning suboptimal solutions, and include a way how to prioritize the replanning of the conflicting groups, what conflict to resolve first, and how to choose the initial path for single-agent groups (if there are more possible paths).

We design our novel hybrid SAT-CBS algorithm on top of ID, as follows. It runs ID, but whenever ID requires to compute a solution for a group, we run both algorithms (reduction to SAT and CBS) simultaneously. Since both algorithms return optimal solutions, we can safely halt both algorithms whenever one of them finds a solution, and use that solution. Implementing our hybrid algorithm also requires adapting each underlying solver (reduction to SAT or CBS) such that it can search for a solution that avoids a previously planned solution of another group. This is needed to support ID's replanning step (lines 9 – 14 in Algorithm 3).

We hypothesize that for small (and thus not dense) groups, the CBS solver will find a solution first, while for larger groups that are packed in a small area (indicating that there are many conflicts among the agents), the reduction-based solver will find a solution first. We verify this hypothesis experimentally.

3.3.2 Experiments

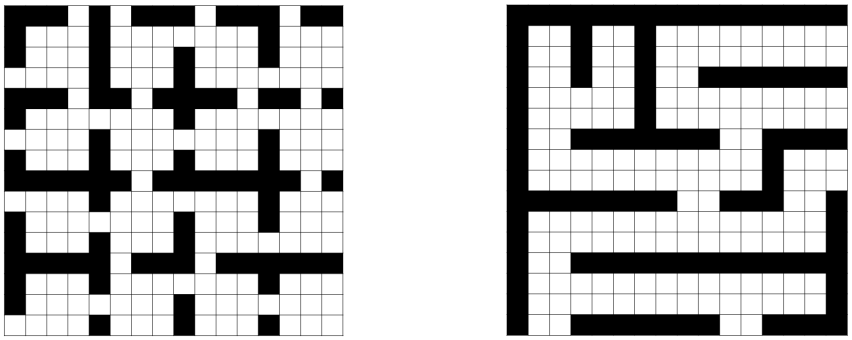


Figure 3.7: The two underlying grid maps used in the experiments.

To test the proposed algorithm, we created several test instances inspired by the movingai.com [1] benchmarks. The two underlying maps are 16 by 16 4-connected grid maps shown in Figure 3.7. In these maps, an increasing number of agents were placed, starting from 1 and increasing until no solver was able to solve the given instance. This means that if a solver is able to solve the instance in a given time limit, we add a new agent to the existing set of agents and run the instance again from scratch. If the solver is unable to solve the instance with the given number of agents, it will not be able to solve the instance with more agents and thus we can end.

The initial and goal locations were selected in two ways. One set of instances has completely random start a goal positions for each agent. The other set of instances is biased in such a way that for half of the agents both start and goal locations are placed in one section of the map. The other half of the agents are again placed completely randomly. We create these biased instances to study the impact of the increased interaction between agents on the proposed algorithm. Each of these instance settings was created 2 times. This gives a total of 8

Agents	CBS	Picat	Hybrid
1 – 13	8	8	8
14	5	8	8
15	5	8	8
16	5	8	8
17	4	6	8
18	3	6	8
19	2	5	8
20	1	5	8
21	1	4	7
22	1	2	7
23	0	1	2
24	0	0	2
25	0	0	0

Figure 3.8: Number of solved instances for each algorithm based on number of agents present in the instance.

instances per number of agents. We created only solvable instances as solvability can be checked by some complete polynomial-time algorithm.

Implementation Details

We used the implementation of the SAT-reduction solver in the Picat language and compiler (version 2.2#3) [41]. We refer to our reduction-based MAPF solver as *Picat*, for briefness.

In the following experiments, we compare CBS, Picat, and our proposed hybrid algorithm that uses CBS and Picat as the underlying solvers. For briefness, we refer to this hybrid algorithm as simply *Hybrid*. Note that for our CBS and Picat implementations, we also used ID, as it proved to be beneficial, but both CBS and Picat use as an underlying solver always only CBS or Picat, respectively. By contrast, our Hybrid algorithm runs both CBS and Picat in parallel and takes the result of the faster one.

The used cost function is makespan, however, it is easy to see that the experiments can be done with the sum of costs as well. All experiments were conducted on a PC with an Intel® Core™ i7-2600K processor running at 3.40 GHz with 8 GB of RAM.

Results

We ran the three solvers – *CBS*, *Picat*, and *Hybrid* – on all of the created instances with a time limit of 600 seconds. Since all of the solvers compute makespan optimal solution, we do not compare the quality of the solution, but rather the computational time.

Figure 3.8 shows how many of the 8 instances for each number of agents each solver managed to solve in time. The largest numbers of agents each solver could solve are 24 for Hybrid, 23 for Picat, and 22 for CBS. The difference is more apparent when we look at the number of agents present when each solver did not solve at least one instance - 14 for CBS, 17 for Picat and 21 for Hybrid. We can clearly see that the most successful solver is Hybrid.

For a more detailed comparison of the runtimes, we present Figure 3.9. It shows the number of solved instances (x-axis) in a given time (y-axis). The lower

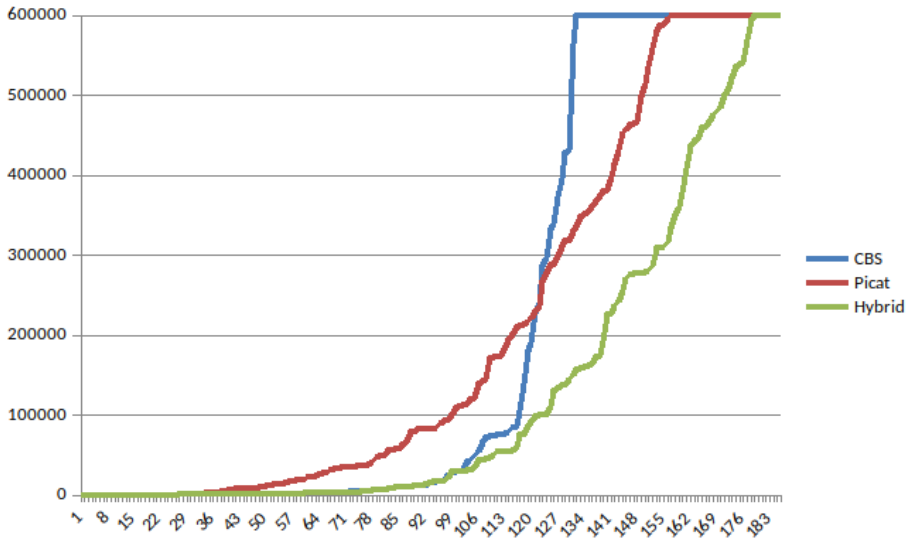


Figure 3.9: Comparison of algorithms CBS, Picat, and Hybrid sorted by the runtime of each instance.

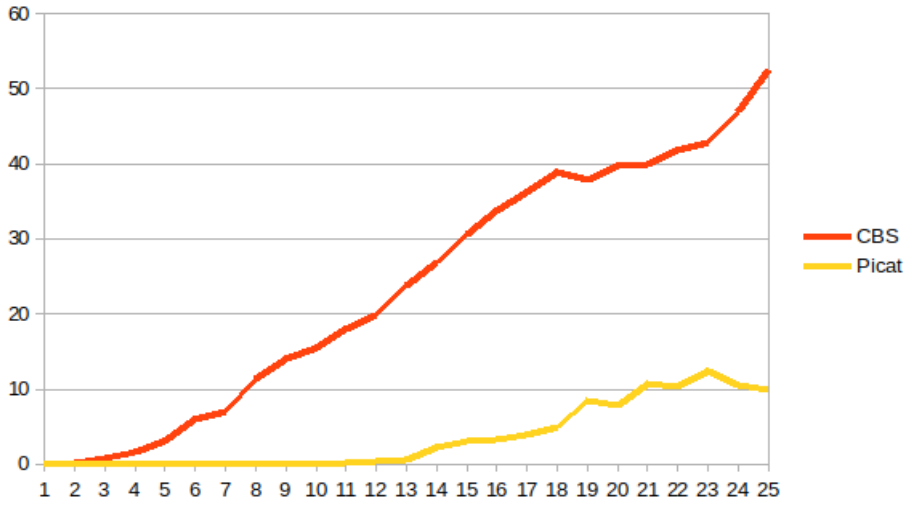


Figure 3.10: Comparison of how many times each underlying solver (CBS and Picat) is used during the computation of Hybrid algorithm. The x-axis is the number of calls to each solver in instances with the corresponding number of agents (y-axis).

the curve, the better. We can see that for some instances, CBS is much better than Picat, however, then there is a sudden increase in complexity. This trend indicates, that when an instance can be solved by CBS, it can be solved very fast. Picat, on the other hand, is much slower even on easier instances but is able to solve a few more instances.

The Hybrid algorithm takes the best of each solver (quite literally). It uses the computation of CBS when the instances are small and not too crowded. As the instances increase in difficulty, it uses the Picat computation to solve the part that is hard for CBS, thus resulting in a better result than either of the algorithm.

Note that it is not guaranteed for the Hybrid algorithm to always have runtime at most equal to the faster solver. It is possible that during the computation we used a solution for some groups from each algorithm and therefore we have different collisions between groups than by using only one of the algorithms (there exist several optimal plans). These different collisions can result in an increased number of computations in the independence detection algorithms and therefore in increased overall computation time. However, this is, in our experience, an extremely rare situation and the increased runtime is not that extreme.

The number of successful calls to each underlying solver during the computation of the Hybrid algorithm is showed in Figure 3.10. A successful call to a solver means that the computation of that solver finished first. We can see that as the number of agents in the graph increases (and thus increases the hardness of the problem), the number of calls increases for both algorithms. Notice that Picat is rarely faster than CBS for instances with less than 13 agents.

The number of times Picat computation is used is significantly lower than that for CBS, yet the line for Hybrid in Figure 3.9 seems to be copying Picat, rather than CBS. We conjure that this is caused by some smaller subproblem of the instance that is very hard for CBS, but the rest of the instance is still preferable to be solved by CBS.

Our experiments support the conjecture that different structures of MAPF instances are hard for different solvers. Indeed, we observed that splitting the instance into subproblems and solving these subproblems by appropriate solver yields better results than using either of the solvers for the subproblems.

4. Bringing MAPF Closer to Reality

In this chapter, we try to expand the definition of the MAPF problem from chapter 1. More specifically, we address the fact that in the classical definition the set of agents is fixed and does not change during the execution of the computed plan. Another extension to the original definition is that we allow the underlying graph to be non-unit. Both of these extensions are meant to bring the problem of MAPF closer to the real world.

4.1 Online MAPF

Most work on MAPF has focused on finding a plan for all the agents before the agents start to move. The agents are then expected to follow that plan until eventually, each agent reaches its designated goal. We refer to this a priori planning problem as *offline MAPF*. In this section, we present our contribution, an *online MAPF*, where new agents may appear while the other agents are following a previously generated plan [20].

Online MAPF is a natural generalization of offline MAPF with applications in controlling fleets of vehicles or teams of robots. For example, consider the autonomous intersection management project [14]. An autonomous agent controls an intersection, and driver agents that wish to pass through the intersection must follow the intersection manager's directions. Once an agent passes the intersection, it leaves the system, while new agents may enter the intersection. The intersection manager, thus, is actually solving an online MAPF problem. The general approach they used is a first-come first-serve approach, which is clearly not optimal.

Online MAPF is related to the *lifelong MAPF* setting [22]. In lifelong MAPF the set of agents does not change, but over time the problem solver receives a sequence of navigation tasks that they need to perform (e.g., get from one location to another). Thus, task assignment plays an important role in lifelong MAPF. In our setting, we focus on the intersection problem, where new agents can appear, and there is no need for task assignment as every new agent is associated with a specific start and goal.

4.1.1 Problem Definition

An instance of an *online* MAPF problem starts with an offline MAPF problem, which an online MAPF solver must first solve. Then, while the agents execute the generated plan, a sequence of *new agents* appears on the map. Each of the new agents a_i^+ has assigned its starting location, goal location, and time t_i at which it appears.

Importantly, this sequence of new agents is given to the problem solver *online*, such that only at time t_i it is revealed where the new agent wishes to start and end. Thus, this part of the problem input – the new agents – is referred to as the *online* input of an online MAPF problem, while the graph and the initial set of

agents are referred to as the *offline* input. Formally, the instance and solution can be defined similarly to the definitions of classical (offline) MAPF from chapter 1.

Definition 12. *An* instance of online MAPF *is a triplet* (G, A, A^+)*, where* G *and* A *correspond to the offline MAPF instance.* A^+ *is a set of agents that are to appear. Each* new agent $a_i^+ \in A^+$ *is represented by a triple* $a_i = (s_i, g_i, t_i)$*, where* s_i *represents the start location (sometimes also called the initial location) of agent* a_i*, and* g_i *represents the goal location of agent* a_i*. In both cases, location corresponds to a vertex in the input graph.* t_i *is the timestep in which the agent appears.*

Definition 13. *A* solution to an online MAPF problem *is a sequence of valid plans* $\Pi = \langle \pi^0, \pi^1, \ldots, \pi^m \rangle$*, where* m *is the number of times a new agent appears,* π^0 *is the plan created for the offline input, and* π^i *for* $i > 0$ *is the plan created when the* i^{th} *new agent appears at time* t_i*.*

Definition 14. *A* partial plan $\pi^i[x : y]$ *is the part of the plan* π^i *that is planned for timesteps* $x, x + 1, \ldots, y$*.*

We assume that the agents follow the most recently generated plan, i.e., after the i^{th} agent appears, all agents follow plan π^i until the $(i+1)^{th}$ agent appears, after which the agents follow plan π^{i+1}. Thus, the plan the agents end up executing, denoted $Ex[\Pi]$, can be written as $Ex[\Pi] = \pi^0[0 : t_1] \circ \pi^1[t_1 + 1 : t_2] \circ \ldots \circ \pi^m[t_m + 1 : \infty]$, where $\circ$ represents concatenation of partial plans. We refer to $Ex[\Pi]$ as the *execution* of Π. A solution to an online MAPF problem is a sequence of plans Π such that $Ex[\Pi]$ forms a valid plan (i.e., without collisions).

Online MAPF Variants

One can imagine many variants of online MAPF, in particular with respect to (1) what happens when an agent reaches the goal and (2) what happens when a new agent appears in its initial location.

Consider first the question of what happens when an agent reaches its goal. One option is that when an agent reaches its goal it stays there. This results in a setting similar to the above mentioned lifelong MAPF [22]. A different option is that an agent disappears when reaching its goal. Such an assumption makes sense when the goal is associated with some location that the agent can actually enter and stay there without interfering with others, e.g., a private parking space.

Orthogonal to the decision of what happens to an agent when it reaches the goal is the decision of what happens when a new agent appears. One option is to assume that a new agent immediately appears in its initial location. This can cause unavoidable collisions, as another agent may already be in that location when the new agent appears. A different assumption regarding new agents is that when a new agent appears it needs to perform a move action in order to enter its start location, and it can also wait as long as it wishes before doing so. This assumption also corresponds to a private parking space scenario, where the agent can wait, e.g., if it sees that its initial location in the graph is already occupied. We refer to this private place as the agent's *garage*.

Consider the assumption that an agent disappears at its goal and the assumption that the agent appears in its garage. These correspond to a scenario where

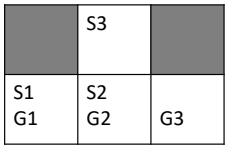


Figure 4.1: An example of an instance that is solvable by the offline optimal solver, but cannot be solved by any online MAPF solver.

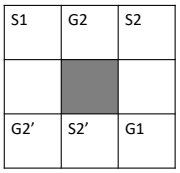


Figure 4.2: An example of an instance in which no online MAPF solver can return the optimal solution.

there is a private part of the world that is not managed in a centralized way, and the agents start from and wish to go to such private locations. To do so, the agents must pass through a public area that is controlled by some autonomous agent, e.g., an autonomous driving scenario where only the traffic in the city centers is fully automatized [14].

Next, we analyze the online MAPF problem under each of these four combinations of assumptions – staying at the goal, disappearing and appearing in the grid, or in the garage.

4.1.2 Problem Analysis

A common way to analyze online problems and algorithms is to consider the behavior of an offline optimal solver for this problem. An offline optimal solver is one that accepts all the inputs to the online problem upfront. In our case, an offline optimal solver for online MAPF knows in advance when new agents will arrive and what will be their start and goal locations. One can easily modify any offline MAPF solver to serve as an offline optimal solver. Clearly, an offline optimal solver cannot be used in practice, but it is useful for analysis as clearly no online MAPF solver can do any better than the offline optimal solver.

Observation 2. *If agents do not disappear when they reach their goals, then there are problem instances that are solvable by an offline optimal solver but cannot be solved by any complete online MAPF solver.*

Proof. Consider an online MAPF problem with 3 agents, situated in the grid given in Figure 4.1. $S1$, $S2$, and $S3$ are the start locations of agents 1, 2, and 3, respectively. Similarly, $G1$, $G2$, and $G3$ are the agents' goal locations. Now, assume that agent 1 appears first, then agent 2, and finally agent 3. An offline optimal solver will have agent 2 wait in its private space until agent 3 appears and reaches its goal. By contrast, any complete online MAPF solver will have to eventually decide to move agent 2 to its goal. We can construct an online MAPF problem that will have agent 3 appear only after this has occurred. When this occurs, the problem becomes unsolvable, as agent 3 can only reach its goal if agent 2 has not entered its starting location. □

Proposition 3. *An online MAPF problem where agents disappear at the goal and where new agents may wait before entering their initial location is solvable iff the offline part of the problem is solvable, assuming there exists a path for each agent from its initial location to its goal location.*

Appears	Stay at goal	Disappear
In grid	Collisions & Incomplete (Obs. 2)	Collisions
In garage	Incomplete (Obs. 2)	Complete (Prop. 3)

Table 4.1: Summary of theoretical results for online MAPF.

Proof. If the offline part is solvable, we can then just let the new agents wait in their garages until all of the other agents disappear. The problem is then reduced to a single-agent shortest path. The other implication is trivial. □

Since assuming that an agent appears immediately on the grid may cause unavoidable collisions, and assuming that the agent stays at the goal may lead to instances that no online solver can solve (Observation 2), we decided to focus the rest of this section on online MAPF where the new agents appear in their garage and agents that reach their goal disappear. Note that waiting in a garage to enter the graph counts as an action in the plan. Table 4.1 summarizes our observations for the four combination of assumptions.

4.1.3 Objective Functions

Now we ask the question of how to evaluate a solution Π for an online MAPF problem. Porting the sum of costs measure from offline MAPF to online MAPF is straightforward: the sum of costs of an online MAPF solution $\Pi = \langle \pi^0, \pi^1, \ldots, \pi^m \rangle$ is defined as the sum of costs of the executed plan $Ex[\Pi]$, i.e., $f_1(\Pi) = \sum_{i \in A} |Ex[\Pi]_i|$, where $|Ex[\Pi]_i|$ is the number of steps the i^{th} agents took until it reached its goal.

Porting the makespan measure, however, is problematic, since makespan is defined as the arrival time of the last agent to its goal position. Due to the online nature of online MAPF, agents may keep emerging infinitely, resulting in Makespan= ∞, which is clearly undesirable. We now propose several possible objective functions intended to fill this gap. Let A be the set of agents, *NotAtGoal(t)* be the number of agents that are not at their goals at timestep t yet, and SP_i be the length of the optimal (shortest) path for the i^{th} agent when no other agent is present. For a plan Π and its execution $Ex[\Pi]$, $|Ex[\pi]_i|$ is the number of steps the i^{th} agent took until it reached its goal.

- **Sum of agents not at goal over time.**

$$f_2(\Pi) = \sum_{t=1}^{\infty} NotAtGoal(t)$$

- **Sum of times over individual cost.**

$$f_3(\Pi) = \sum_{i \in A} |Ex[\Pi]_i| - SP_i$$

Definition 15. *A pair of objective functions f_x and f_y are* equivalent *if for every online MAPF problem and every pair of solutions Π and Π' for that problem, we have*

$$f_x(\Pi) \leq f_x(\Pi') \Leftrightarrow f_y(\Pi) \leq f_y(\Pi')$$

Proposition 4. *All the objective functions listed above are equivalent to the sum of costs.*

Proof. "$f_1 \Leftrightarrow f_2$" Let χ_i^t be 1 if the i^{th} agent is not yet at its goal location at time t, otherwise 0. Then $f_1(\Pi) = \sum_{i \in A} \sum_{t=1}^{\infty} \chi_i^t$ and $f_2(\Pi) = \sum_{t=1}^{\infty} \sum_{i \in A} \chi_i^t$. Hence, by swapping the sums, $f_1(\Pi) = f_2(\Pi)$ for arbitrary Π.

"$f_1 \Leftrightarrow f_3$" Since $f_1(\Pi) = \sum_{i \in A} |Ex[\Pi]_i| = \sum_{i \in A} |Ex[\Pi]_i| - SP_i + \sum_{i \in A} SP_i = f_3(\Pi) + \sum_{i \in A} SP_i$, and because $\sum_{i \in A} SP_i$ is a constant depending just on the problem and not on the solution, it holds that $f_1(\Pi_x) \leq f_1(\Pi_y) \Leftrightarrow f_3(\Pi_x) \leq f_3(\Pi_y)$ for arbitrary plans Π_x, Π_y. □

Thus, we only discuss the sum of costs objective function (f_1) for online MAPF and refer to a solution as optimal if it minimizes the sum of costs.

Observation 5. *There is no complete online MAPF solver that can guarantee to return a solution with a cost equal to the offline optimal MAPF solver.*

Proof. Consider an online MAPF problem with 2 agents, situated in the grid given in Figure 4.2. $S1$ and $G1$ are the start and goal of agent 1, which is the first agent to appear. The agent has two shortest paths to reach its goal: going right and then down, or going down and then right. After the agent chooses one of these options, the second agent appears, either in S2 or in S2'. An offline optimal solver will know upfront where the second agent will appear and choose appropriately if the first agent will go down and right (avoiding S2) or right and down (avoiding S2'). An online MAPF solver cannot know this in advance and is thus bound to yield a suboptimal solution. □

4.1.4 Online MAPF Algorithms

While Observation 5 states that guaranteeing an optimal solution is not possible with a complete solver, there is still a need to solve online MAPF problems in some principled way. In this section, we propose several online MAPF algorithms and discuss their properties.

The offline part of the problem, i.e., planning for the initial set of agents, can be done by a different algorithm than the one used for the online part. We focus our discussion on the online part and assume that all algorithms we propose start with an optimal solution to the initial offline problem. Hence, in what follows we only describe the replan function, which is called when new agents appear. The input to such replan functions is always the set of current agents A, the set of new agents A^+, and the ongoing plan π_A, which is the plan the agents in A are currently following. Note that more than one new agent may appear at the same time, and thus A^+ may contain multiple agents.

Replan Single and Replan Single Grouped

The first two algorithms we describe serve as a baseline. The Replan Single (RS) algorithm searches for an optimal path for each new agent, one at a time, while avoiding all other (already planned) agents. The Replan Single Grouped (RSG) algorithm searches for optimal paths for all new agents at once.

To formally describe RS and RSG, we introduce the following helping notation. Let $\psi_A^{\pi_B}$ be an optimal plan for a group of agents A while avoiding some plan π_B for a group of agents B. This assumes that the groups A and B are disjoint. In particular, $\psi_A^{\emptyset}$ means an optimal plan for the agents in a group A without considering any agent that is not in A.

Algorithms 4 and 5 list the pseudo-codes for RS and RSG using our helping notation ψ_A appropriately. Note that when only one new agent appears, RS and RSG behave in the same way.

Algorithm 4 Replan Single

function RS(agents A, new agents A^+, ongoing plan π_A)
  for each $a \in A^+$ **do**
    $\pi \leftarrow \pi_A \cup \psi_a^{\pi_A}$
    $A \leftarrow A \cup \{a\}$
  end for
end function

Algorithm 5 Replan Single Grouped

function RS(agents A, new agents A^+, ongoing plan π_A)
  $\pi \leftarrow \pi_A \cup \psi_{A^+}^{\pi_A}$
  $A \leftarrow A \cup A^+$
end function

RS can be solved in polynomial time since it runs one single-agent path finding search. Therefore, in our implementation of the rest of the algorithms described in this work, we first run RS to obtain a baseline solution, and then try to improve on it in the rest of the runtime.

RSG, on the other hand, may require more runtime, depending on the number of new agents.

However, what can be said about the solution quality? Since both RS and RSG do not allow changing the plans of the other agents, then using them may lead to solutions of poor quality. Next, we propose a solution quality criteria that online MAPF algorithms can aim for in an effort to achieve better overall solution quality.

Snapshot Optimality

Definition 16. *A* snapshot optimal *plan in an online MAPF setting is a plan for all agents that is optimal in terms of the sum of costs assuming no new agent will appear in the future.*

There is no guarantee that always producing a snapshot optimal solution will result in a minimal sum of cost solution for the online MAPF problem. In fact, we know from Observation 5 that such minimality guarantee is not possible in online MAPF. Nonetheless, demanding that an online MAPF algorithm will return snapshot optimal

solutions may bias it towards an overall low sum of costs in practice. Indeed, we observe this in our experimental results. Thus, we now propose several algorithms that provide such a guarantee.

Replan All

The simplest way, conceptually, to return snapshot optimal solutions is to plan optimally for all agents from their current positions whenever new agent appears. We call this algorithm Replan All (RA), and list its (simple) pseudocode in Alg. 6.

Algorithm 6 Replan All

function RA(agents A, new agents A^+, ongoing plan π_A)
  $A \leftarrow A \cup A^+$
  $\pi \leftarrow \psi_A^{\emptyset}$
end function

RA is the extreme opposite of RS: it solves a much harder problem – offline MAPF for all current agents without considering the already computed ongoing plan (π_A) – but is expected to return a high-quality solution, i.e., a plan with a small sum of costs since it computes the optimal solution for all of the agents currently present and the new agents.

Online Independence Detection

RA plans optimally for all current agents from their current positions, fully ignoring the plan the existing agents were following. This can be wasteful in terms of runtime. Moreover, changing the route of an agent that is already moving, which we refer to as re-routing the agent, may be undesirable, as it requires communication with that agent and modifying the agent's plan may incur some overhead.

Next, we propose an algorithm that returns snapshot optimal solutions but also attempts to minimize the number of re-routes. We call this algorithm Online Independence Detection (OID) for online MAPF since it is based on Standley's Independence Detection (ID) algorithm [27].

The main idea of Standley's ID is to plan for each agent separately while ignoring the other agents. If there is a conflict between the generated plans then the conflicting agents are merged into a group and replanned together. This process continues iteratively until there are no conflicts anymore.

The main idea of OID is very similar: allow the new agents to plan while ignoring the other agents. If there is a conflict with the plan of an already planned agent, then we merge the groups of conflicting agents and plan for them altogether (again disregarding the other agents). This is iterated until there are no conflicts anymore.

This adaptation of ID to online MAPF, however, may return plans that are not snapshot optimal. The reason is that an online MAPF algorithm is called multiple times, whenever new agents appear. Consequently, when a new agent a_i^+ conflicts with an already existing agent a_j, it is not sufficient to replan for a_i^+ and a_j together since a_j may have been grouped with other agent, e.g., a_k in the previous call. There, perhaps agent a_k chose a longer path in the previous call

to allow a_j use a shorter path. Now, when a_j is replanning due to a conflict with the new agent a_i^+, perhaps it frees up locations that a_k can use to have a shorter path for itself.

To correct this, we modify OID so that it keeps track of the groups used to create the incumbent plan. Algorithm 7 lists the pseudo-code for OID. First, the algorithm requires that the set of already planned agents A consists of mutually disjoint and collectively exhaustive sets of agents $g_1, g_2, ..., g_m$, and the ongoing plan π_{g_i} for every g_i, such that π_{g_i} is the lowest cost plan for the agents in g_i. When new agents A^+ appear, each of them is placed into a new group and an optimal plan for each group is found, disregarding all the other agents (lines 2 – 6). Then the algorithm iteratively resolves the conflicts until there are no more conflicting groups. Assume there is a conflict of plans between groups g_i and g_j. Then the algorithm tries to find such a plan for g_i that avoids conflicts with the agents from a group g_j while not deteriorating the plan in terms of the sum of costs (line 13). If it does not succeed, it tries analogically to replan for g_j while avoiding g_i (line 15). If it still did not succeed, it merges the conflicting sets of agents and replans for them together, while, again, disregarding all other agents (lines 17 – 20). However, this way the algorithm could get stuck in an infinite loop, that is why it is necessary to first check whether the two conflicting groups were already in conflict together before, and thus merge and replan them straight away (lines 9 – 12).

Algorithm 7 Online Independence Detection

1: **function** OID(agents $A = \dot{\bigcup}_{i=1}^{m} g_i$, new agents A^+, ongoing plan π_A)
Require: π_{g_i} for each group g_i
2: $k \leftarrow m + 1$
3: **for each** $a \in A^+$ **do**
4: $g_k \leftarrow a$
5: $\pi_{g_k} \leftarrow \psi_a^{\emptyset}$
6: $k \leftarrow k + 1$
7: **end for**
8: **while** g_i and g_j conflict **do**
9: **if** g_i, g_j conflicted before **then**
10: $g_i \leftarrow g_i \cup g_j$
11: $g_j \leftarrow \emptyset$
12: $\pi_{g_i} \leftarrow \psi_{g_i}^{\emptyset}$
13: **else if** $\psi_{g_i}^{\pi_{g_j}}$ is as good as π_{g_i} **then**
14: $\pi_{g_i} \leftarrow \psi_{g_i}^{\pi_{g_j}}$
15: **else if** $\psi_{g_j}^{\pi_{g_i}}$ is as good as π_{g_j} **then**
16: $\pi_{g_j} \leftarrow \psi_{g_j}^{\pi_{g_i}}$
17: **else**
18: $g_i \leftarrow g_i \cup g_j$
19: $g_j \leftarrow \emptyset$
20: $\pi_{g_i} \leftarrow \psi_{g_i}^{\emptyset}$
21: **end if**
22: **end while**
23: **end function**

Theorem 6. *OID returns a snapshot optimal solution if it is given a disjoint partition of agents to groups $g_1, \dots g_m$ such that for every group g_i the cost of its current plan π_{g_i} is equal to the cost of $\psi^{\emptyset}_{g_i}$.*

Proof. Consider the output of OID after being called by a replan function. OID outputs a new partition of agents to groups $g'_1, \dots g'_{m'}$ that includes the new agents along with a plan to each of these groups. Since OID never breaks a group, then every group in the original set of groups $g_1, \dots g_m$ must be equal to or a subset of one of the new groups $g'_1, \dots g'_{m'}$. Since OID returns an optimal plan for each of the new groups, then it cannot miss an optimal plan for any agent. □

Corollary 7. *If the ongoing plan was snapshot optimal and OID is used to replan when new agents appear then OID is guaranteed to always return snapshot optimal solutions.*

Corollary 7 follows by induction due to Theorem 6 and the fact the initial plan is snapshot optimal. Since OID attempts to minimize the number of existing agents it replans for, it has the potential to save runtime and require fewer re-routes than RA.

However, OID has one clear disadvantage compared to RA. If during an online MAPF execution RA is not used to replan for some agents, e.g., due to runtime limitations, RA may still return snapshot optimal solutions if it is used later to replan. By contrast, OID depends on previous calls to replan to return a snapshot optimal solution, as well as a partition of groups and a plan for each group such that each group is solved optimally. If OID does not accept this as input, then it cannot guarantee snapshot optimality. For example, if in one call to replan RS is used instead of OID, the OID loses its snapshot optimality property.

Suboptimal Independence Detection

As noted earlier, returning a snapshot optimal plan does not guarantee an optimal solution to the online MAPF problem. Therefore, we propose to change OID by allowing it to return plans whose sum of costs is at most D times more than the optimal sum of costs but allowing it to further reduce the number of agents that have to deviate from the ongoing plan. We call this algorithm Suboptimal OID (SubID).

In details, when OID replans for the group of agents g_i while avoiding g_j and ignoring all other agents (line 13 and symmetrically line 15), it only accepts plans that have exactly the same sum of costs as that of the optimal plan for g_i while ignoring all other agents (the sum of costs value being $f_1(\psi^{\pi_{g_j}}_{g_i})$). It is likely that such a solution does not exist. SubID allows the new plans to have higher cost, namely any cost in the range $[f_1(\psi^{\pi_{g_j}}_{g_i}), D \cdot f_1(\psi^{\pi_{g_j}}_{g_i})]$. This is expected to increase the likelihood that such a plan can be found.

Theorem 8. *Given a disjoint partition of agents to groups $g_1, \dots g_m$ such that for every group g_i the cost of its current plan π_{g_i} is at most w_1 times the cost of $\psi^{\emptyset}_{g_i}$, then SubID with parameter $D = w_2$ will return a solution whose cost is at most $w_1 \cdot w_2$.*

Proof. The proof is similar to that of Theorem 6: every group g_i is contained in exactly one group from the solution of SubID. The added suboptimality is a

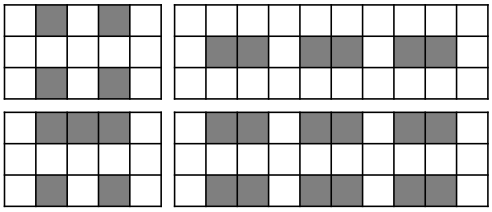


Figure 4.3
Small grids.

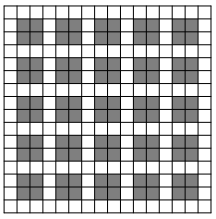


Figure 4.4
Large grids.

factor of at most w_2 and the suboptimality of the existing plan is at most w_1 thus the overall suboptimality is $w_1 \cdot w_2$. □

A direct corollary of Theorem 8 is that if SubID is always used to replan when new agents appear then after the m replan calls SubID will output a plan whose cost is at most D^m times the cost of a snapshot optimal plan.

4.1.5 Experiments

We implemented all the algorithms and evaluated their performance on a set of randomly generated problems. All experiments were conducted on Dell PC with an Intel® Core™ i7-2600K processor running at 3.40 GHz with 8 GB of RAM.

Instances

We created two datasets of online MAPF problems based on 4-connected grids designed to simulate intersections for autonomous vehicles. The first dataset is on small and dense grids, chosen from the 4 types of small grids depicted in Figure 4.3. Each problem started with no agents present at the outset. Then we are adding new agents, the total number of which is in the range {10, 12, 15, 17, 20, 22, 25}. The starting time of each agent is set uniformly at random from the interval $[1, 30]$, so a number of agents may appear at the same time. The start location is a random location from a randomly picked margin of the map and the goal is randomly picked from the other margins. We generated 5 problems for each configuration. Altogether this dataset contained 140 problems.

The second dataset is on a larger grid depicted in Figure 4.4. The starting time of each agent is set uniformly at random from the interval $[1, 100]$. The number of new agents to appear over these timesteps increments from 60 to 70. The agents are moving from a randomly picked margin to another randomly picked margin. No agent is present at the outset. We generated 5 problems for each configuration. Altogether we generated 30 testing instances.

Implementation Details

All the proposed algorithms require an optimal offline MAPF solver, adapted to our assumptions that a new agent appears in its garage and agents disappear at their goals. We created two such modified optimal offline MAPF solvers: one based on a reduction to Boolean satisfiability (SAT) via the Picat language and compiler (version 2.2#3) [29], and the other based on the Conflict Based Search (CBS) algorithm [18]. For the first dataset (small grids) we used the Picat-based

solver and for the second dataset (large grid) we used the CBS-based solver. This is because Picat-based solvers perform better on small and dense grids and CBS-based solver performs better on large grids [29].

Dataset #1: Small Grids

We ran the following algorithms on the small grids dataset: Replan Single (RS), Replan Single Grouped (RSG), Replan All (RA), Online Independence Detection (OID), and Suboptimal ID (SubID) with D chosen to be 1.1. For every instance, we always run first RS and then the algorithm in question. Whenever the runtime for a newly appeared agent exceeds the given time limit, which was set to 30 seconds, the output from RS is taken into results. The reason is that RS is done extremely quickly and it is always good to have some solution rather than no solution. The number of instances, where the time limit was reached and thus the result from RS was taken, can be seen in Table 4.2.

#agents	RSG	SubID	OID	RA
10	0.00	1.70	1.70	0.55
12	0.00	2.55	2.20	1.05
15	0.05	5.90	4.25	2.40
17	0.35	6.55	7.10	3.40
20	1.20	9.55	9.80	5.65
22	1.70	10.85	10.55	9.40
25	3.50	11.65	11.65	10.00

Table 4.2: The average number of timeouts per instance of each algorithm.

#agents	SubID	OID	RA
10	2.65	3.30	5.15
12	2.05	2.80	3.80
15	1.50	3.45	6.45
17	3.80	3.85	10.05
20	2.85	2.65	10.15
22	2.95	4.25	6.80
25	1.90	2.55	6.40

Table 4.3: Avg. # of re-routes for smaller grids using Picat.

#agents	RSG	SubID	OID	RA
10	1.01	1.06	1.06	1.10
12	1.02	1.08	1.09	1.14
15	1.03	1.04	1.11	1.17
17	1.05	1.14	1.13	1.22
20	1.04	1.04	1.04	1.19
22	1.06	1.06	1.10	1.15
25	1.05	1.07	1.08	1.15

Table 4.4: Avg. gain in SOC over RS for smaller grids using Picat.

The behavior of the algorithms with respect to the number of times an agent had to change its plan (number of re-routes) is shown in Table 4.3. The number of re-routes for RS and RSG is always zero since it never replans for the other agents, so we do not report it in Table 4.3. The results clearly show that OID requires fewer re-routes than RA, and SubID requires even fewer re-routes than OID. For example, with 15 agents, SubID required on average 1.5 re-routes, while OID required 3.45 and RA 6.45. The slight decrease in the number of re-routes for 20 agents is caused by the increase in the number of timeouts and therefore more RS calculations that avoid re-routes are used.

The relative gain in terms of the sum of costs over RS is shown in Table 4.4. The expectations that RA will always be the best w.r.t. the sum of costs have been confirmed. It is worth noticing that SubID, which does not return an optimal solution in every replan, brings more or less the same gains in terms of the sum of costs as OID. Thus, SubID shows promising results, with a comparable sum of costs to OID but significantly fewer changes.

Dataset #2: Large Grids

#agents	RSG	SubID	OID	RA
60	0.0	1.0	0.4	0.8
62	0.0	0.0	0.2	0.6
64	0.0	1.0	0.8	0.4
66	0.0	0.0	0.2	0.6
68	0.0	0.0	0.0	0.4
70	0.0	0.6	1.0	0.4

Table 4.5: Avg. # of timeouts on the larger grid using CBS.

#agents	SubID	OID	RA
60	5.6	13.4	30.0
62	4.6	11.4	26.0
64	4.4	11.6	25.0
66	5.0	16.6	32.0
68	6.4	12.4	27.4
70	5.8	12.0	24.6

Table 4.6: Avg. number of re-routes for larger grid using CBS.

#agents	RSG	SubID	OID	RA
60	0.997	1.008	1.015	1.015
62	0.994	1.008	1.012	1.012
64	1.000	1.009	1.012	1.013
66	1.000	1.008	1.009	1.009
68	0.996	1.007	1.009	1.009
70	0.998	1.009	1.010	1.011

Table 4.7: Avg. gain in SOC over RS on larger grid using CBS.

The experiments on large grids dataset were carried out in the same way. The number of instances where time limit was reached is shown in Table 4.5. The number of re-routes is shown in Table 4.6, and the relative gain w.r.t. the sum of costs is shown in Table 4.7. Briefly speaking, the results are in concordance with those of small grids.

While we can again see a clear advantage in terms of the sum of costs for RA, OID, and SubID compared to RS, the differences are significantly smaller. This is because larger grids are sparser, and thus it is easier for RS to find a higher quality solution. The differences in the sum of costs between RA, OID, and SubID are negligible, but SubID still shows a significant advantage in terms of the number of re-routes. For example, with 66 agents SubID required on average only 5 re-routes while RA required 32, and their sum of costs was virtually the same.

In conclusion, SubID confirms to have brought the best trade-off between the quality of the resulting plans and the computational efficiency.

4.1.6 Related Work

The term online planning, in general, refers to planning that is done while executing a plan, in contrast to offline planning where all the planning is done upfront. Many papers on online planning defer planning to execution to minimize the computational effort of creating a complete plan for every contingency offline. This includes the seminal work of Korf on Real-Time A* and its many successors [42]. This is different from our setting, where some of the planning is done online because the agents do not know in advance how many and where the new agents will appear.

Online MAPF can be viewed as an instance of ad-hoc teamwork [43], which exactly addresses cases where the agents constantly need to coordinate with new agents. However, the form of interactions between the agents in online MAPF is very specific – they cannot collide with each other, while ad-hoc teamwork usually involves deeper forms of collaboration. In the motion planning literature, there is work on single robot online planning that replans for newly observed obstacles by using pre-defined planning patterns [44]. In addition, some prior works on multi-robot motion planning that are fundamentally forms of prioritized planning [45] and can be adapted to the online case, resulting in a behavior similar to RS. Other motion planning techniques [46, 47] are designed for offline, continuous spaces and adapting them to a discrete and online MAPF is non-trivial.

4.2 MAPF with Weighted and Capacitated Edges

The state of the art algorithms solving the MAPF problem, as well as our definition of MAPF, assume that all of the edge lengths of the underlying graph are identical and that each vertex and edge can be occupied by at most one agent at any time. These limitations on the solved problem do not correspond to reality in many cases. For example, some roads have a larger capacity than others. In this section, we add new attributes to the problem specification that bring it closer

to the real world. To create this extension to the original MAPF problem we add two properties to the definition, more specifically to the underlying graph.

4.2.1 Weighted Edges

The first property we add is in the form of edges' weights or lengths. Each directed edge $e = (u, v) \in E$ has assigned weight $w(e)$. Therefore, we are now working with a weighted graph $G = (V, E, w)$. Note that so far, we did not specify if the graph over which we solve MAPF is directed or undirected. Indeed, all of the algorithms would not change when using either. However, when we specify a weight for each directed edge, it allows us to model situations where traversing one direction takes longer than the other. For example, a road going downhill has a shorter travel time than the same road going in the opposite direction (this is also the reason why we usually use a more general term weight rather than length because the value does not necessarily mean physical distance).

This change indeed models real world properties that may be desirable in some applications and can not be modeled by changing the original graph. For example, assume that we try to simulate the length of the edges by simply splitting the edge $e = \{u, v\}$ with $w(e) = n$ into n edges $e_1, \ldots, e_n$ with unit length. This leads to a possible solution (that would be valid under these conditions) where an agent moves into the edge e from vertex u and then, while traversing the edge e, turns and returns to vertex u. This is, of course, undesirable for example on highways where turning is not permitted. An effort to prevent this can be made by orienting the edges in only one direction, thus creating edges $e = (u, v)$ and $e' = (v, u)$ and splitting them in the same way as before. This can again lead to undesirable behavior such as the agent stopping while traversing the edge e.

Before proposing a change to a solver to deal with this extension, it is noteworthy to mention how does this extension change the definition of MAPF introduced in Chapter 1. The definition of a plan for a single agent (Definition 2) remains the same as before, $\pi_i(j) = v$ still represents agent a_i being present in vertex v at timestep j, however, this function is now not defined for each timestep as the agent can be moving over an edge and therefore is not present in any vertex. A valid plan for a single agent (Definition 3) is changed in such a way that an agent does not have to be present in the neighboring vertex in the next timestep but rather after the time, it takes to traverse the edge. Specifically, moving over an edge means $\exists e \in E : e = (\pi_i(j), \pi_i(j + w(e)))$, while $\pi_i(j+1), \ldots, \pi_i(j+w(e)-1)$ are not defined if $w(e) \geq 2$.

We show how to change the constraint to the makespan optimal solver implemented in Picat programming language introduced in Chapter 3.2. The change to the sum of costs optimal model would be the same. To introduce the edge weight, we need to change the arrival timestep in constraint (3.6). It needs to be set only for timesteps that make sense in accordance with the length of the edge. Therefore we get the following constraint:

$$\forall (u, v) \in E, \forall a_i \in A, \forall t \in \{0, \ldots, T - w((u, v)) - 1\} : \\ Pass(u, v, i, t) \implies At(v, i, t + w((u, v))) \quad (4.1)$$

The constraint (4.1) means that if an agent a_i starts traversing edge (u, v) at time t, it will arrive to vertex v at timestep $t + w((u, v))$ which is exactly after the time it takes to traverse the edge (u, v).

To ensure that there is no edge or swapping conflict, we need to forbid more than one agent to move along the edge. When there are only unit-length edges the edge conflict is forbidden implicitly by forbidding vertex conflict, as was discussed earlier. The swapping conflict in the makespan optimal model from Chapter 3.2 was forbidden by checking if two agents start to move over the opposite edges at the same time (constraint (3.7)). Now, however, we need to check all $w((u, v))$ times when an agent can start to move over the edge (in either direction) since the previous agent may still be using that edge. We get the following constraint:

$$\forall (u, v) \in E, \forall t \in \{0, \dots T - w((u, v))\} : \sum_{\substack{a \in A \\ t' \in t, \dots, t+w((u,v))-1}} Pass(u, v, a, t') + Pass(v, u, a, t') \leq 1 \quad (4.2)$$

4.2.2 Capacitated Edges

Another property we can add is a maximal capacity of an edge. This number represents how many agents can traverse the same edge at the same time. Each undirected edge $e = \{u, v\} \in E$ has assigned capacity $c(e)$. Therefore, we are now working with a graph $G = (V, E, c)$. Note that for capacity we prefer to work with undirected edges as opposed to the situation with the weighted edges. The motivation will be apparent when we analyze the problem. To formally stay true to the directed graph, we can redefine the capacity to be shared for the two opposite edges as $c(e, e')$, where $e = (u, v)$ and $e' = (v, u)$.

If the graph is in form $G = (V, E, c)$, ie. each edge remains unit-length, only two possible capacities are reasonable – 1 and 2. If the capacity of the edge is 1, it is the same as the classical MAPF setting. If the capacity is 2, swapping of two agents along the edge is allowed. If the capacity is 0, the edge can never be used by an agent and it is equivalent to the edge not being present in the graph. On the other hand, if the capacity is 3 or more, we allow more agents to be on the edge than can ever reach it at the same time since vertex conflict is still forbidden.

The capacities become more interesting and useful when combined with the weighted edges. From now on, we are working with a graph $G = (V, E, w, c)$. In this situation, there can indeed be more than 2 agents traversing one edge at the same time, see Figure 4.5 for an example. On the other hand, if we use only weighted edges and not increased capacity of the edges, there can be only one agent on a very long edge, which may also be undesirable. Therefore, both of these changes should go hand in hand.

Adding capacities, however, changes some of the conflicts defined in chapter 1, specifically edge conflict and swapping conflict. The two conflicts occur only if more than $c(e)$ agents are moving over the edge e in either direction at the same time.

To introduce edge capacity into the makespan optimal model we only need to change the constraint (4.2) in the following minor way:

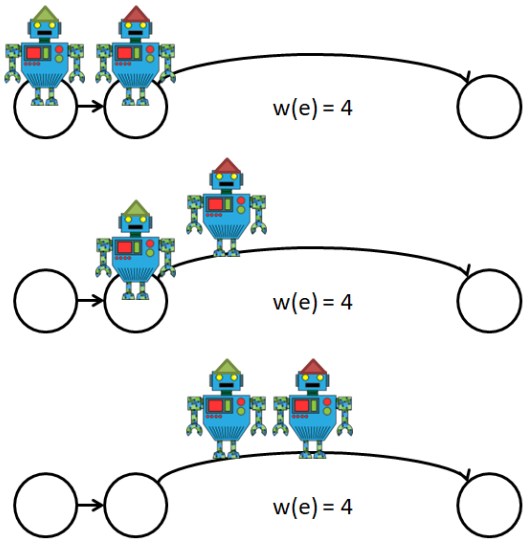


Figure 4.5: An example where two or more agents can move on an edge in the same direction at the same time. Since the edge has weight 4, more agents can enter the edge before the previous agents manage to leave the edge.

$$\forall (u,v) \in E, \forall t \in \{0, \dots T - w((u,v))\} : \sum_{\substack{a \in A \\ t' \in t, \dots, t+w((u,v))-1}} Pass(u,v,a,t') + Pass(v,u,a,t') \leq c(\{u,v\}) \quad (4.3)$$

This constraint is needed to ensure that the number of agents that are moving over one edge in the same direction or over two opposite edges simultaneously does not exceed its capacity.

4.2.3 Experiments

We added both of the changes described above into our Picat model from chapter 3.2. To measure the impact of the changes we tested the new model on a set of randomly generated problems. All experiments were conducted on a PC with an Intel® Core™ i7-2600K processor running at 3.40 GHz with 8 GB of RAM and Picat compiler used was version 2.7b7 with the timeout of 100 seconds.

Instances

The problem instances are created over strongly biconnected undirected graphs. These types of graphs ensure that the instance is always solvable as long as there are at least 2 unoccupied vertices [24]. To create a different complexity of the instances, we incrementally increase the number of vertices in the graph (from 20 to 40 vertices with the increment of 5) as well as the number of agents in the graph (from 2 to 9 agents with the increment of 1). Both the origin and destination locations of agents are randomly placed in the graph.

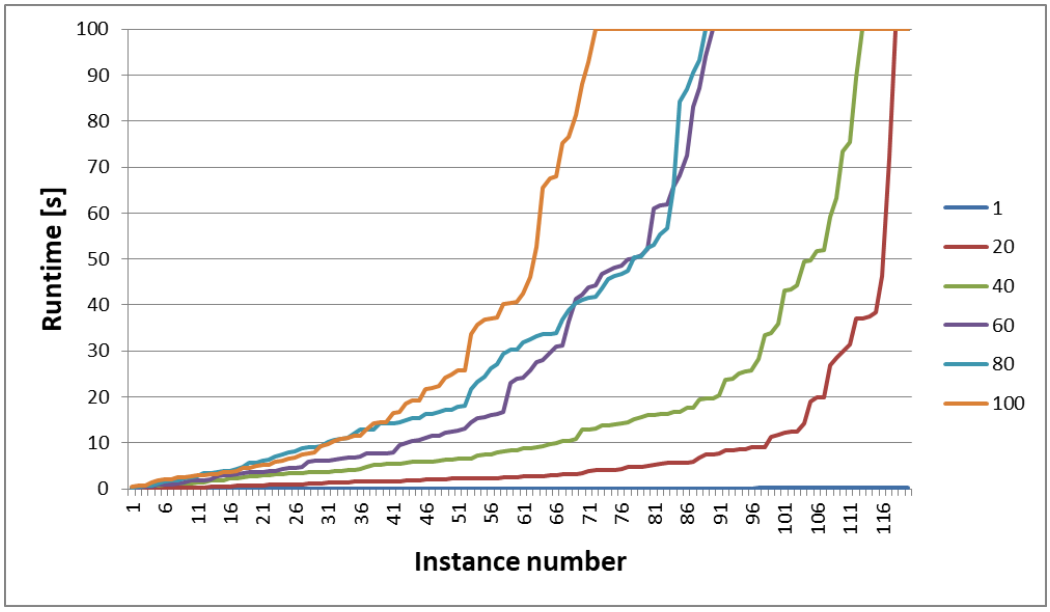


Figure 4.6: Comparison of a runtime based on the maximal length of the edges.

Further, we added lengths to the edges. The length of each edge is chosen uniformly at random from the range $[1, W]$, where $W \in \{1, 20, 40, 60, 80, 100\}$. Instead of assigning a capacity to each edge individually, we assigned the same capacity to all of the edges. For each instance created, we fixed it (the graph with weighted edges and a set of agents A) and created several versions with different capacities. The capacities ranged from 1 to the number of agents with the increment of 2. This has been done to observe the effect of capacities more easily. Also, note that creating an instance with a capacity greater than the number of agents is meaningless since there can never be that many agents traversing the edge. Altogether we generated 720 instances.

Results

We try to determine how each of the individual changes affected the reduction-based model's complexity. First, let us look at the instances divided by the maximal length of the edges. In Figure 4.6 we can see a comparison of the 6 different types of instances. The y-axis is the runtime in seconds, while the x-axis shows how many instances were solved under the given runtime. The lower the line the better the solver performed. From the results, we can clearly see that the classical setting with unit edges performed the best with all of the instances solved in the given time limit. In fact, all of the instances were solved under one second. When we increase the maximal weights of the edges, the model performs progressively worse.

The reason for this increase in complexity is due to the increase in the optimal makespan of each instance. See Figure 4.8 for the average makespan of the solutions for a given maximal edge weight. The increase in makespan corresponds to the increase in the number of layers in the time-expanded graph and thus the number of variables that enter the SAT solver.

Analyzing the impact of the different capacities is not as straightforward as with the weights. First of all, there are not the same number of instances for each

max. weight	1	20	40	60	80	100
avg. makespan	5.8	64.7	119.4	170.8	226.5	261.5

Table 4.8: Average makespan of a solution in comparison with the maximal weight of the edges.

of the capacity (there is no reason the test larger capacity than there are agents present). Secondly, three properties affect the complexity in both a positive and a negative way.

1. When the capacity is increased, it allows for a more flexible solution that may end up being shorter in terms of makespan.

2. While the changes may seem straightforward when written as equations, it is not very natural for SAT to work with numbers other than ones and zeros (the Picat solver transforms the above arithmetic constraints to clauses of a Boolean formula automatically). Thus, increasing the capacity parameter may create larger formulas.

3. The largest capacities were used for the instances with the most agents which are naturally the more difficult instances.

Rather than comparing instance by instance, we report the percentage of instances with given capacities solved in the set time limit (see Table 4.9). From the results, it seems that for the instances we created, the sweet spot for the capacity parameter is 3 with 1, 5, and 7 being a close second.

capacity	1	3	5	7	9
% solved instances	80%	96%	83%	78%	30%

Table 4.9: The percentage of solved instances based on the maximal capacity of the edges.

In conclusion, adding the attributes of weights and capacities to the edges is beneficial to model some real world properties, however, there is a significant increase in complexity as the weights of the edges increase. From the results, we can see that our reduction-based solver is useful when the maximal weight remains small – up to a few dozen. There exists a scheduling-based approach to solve MAPF that is more prominent when dealing with instances with large weight and capacities but falls behind the state-of-the-art classical MAPF solvers. This solver is studied more in-depth in a paper by a colleague [21].

5. MAPF on Real Robots

In the previous chapters, we defined the problem of MAPF, showed several techniques that solve this problem, and introduced extensions to the classical problem. In this chapter, we focus on answering two questions: how to execute abstract plans obtained from existing MAPF algorithms and models on real robots, and how the quality of the abstract plans is reflected in the quality of the executed plans [48]. The goal is to verify if the abstract plans are practically relevant and, if the answer is no (as expected), to provide feedback to improve abstract models to be closer to reality. We use a fleet of Ozobot Evo robots to perform the plans. These robots provide motion primitives, for example, they can turn left/right, follow a line, and recognize line junction, so it is not necessary to solve classical robotics tasks such as localization. Though the robots have proximity sensors, the plans are executed blindly based on the MAPF setting as the plans should already be collision-free.

Specifically, we explore the very classical MAPF setting as described above, the k-robust setting [49], where a gap is required between the robots to compensate possible delays during execution, and finally a model that directly encodes turning operations (the classical setting does not assume direction of movement). The abstract plans are then translated to motion primitives, which consist of moving forward, turning left/right, and waiting. We explore different durations of these primitives to see their effect on robot synchronization.

Though the plans obtained by different MAPF solvers might be different, the optimal plans are frequently similar and tight (no superfluous steps are used). As solving MAPF is not the topic of this chapter (we focus on evaluating the practical relevance of obtained plans), any optimal MAPF solver shown earlier can be used. We decided on the reduction-based solver implemented in the Picat programming language that uses translation to SAT. This solver was also described in the previous chapters.

5.1 Motivation

The abstract plan outputted by MAPF solvers is, as defined, a sequence of locations that the agents visit. However, a physical agent has to translate these locations to a series of actions that the agent can perform. We assume that the agent can turn left and right and move forward. By concatenating these actions, the agent can perform all the required steps from the abstract plan (for simplification, we will be working with grid worlds). This translates to five possible actions at each timestep - (1) wait, (2) move forward, (3,4) turn left/right and move, and (5) turn back and move. As the mobile robot cannot move backward directly, turning back is implemented as two turns right (or left). For example, an agent with starting location in v_1 and goal location in v_7 in Figure 5.1 has an abstract plan of seven locations. However, the physical agent has to perform four additional turning actions that the classical MAPF solvers do not take into consideration.

As the abstract steps may have duration different from the physical steps, the abstract plans, which are perfectly synchronized, may desynchronize when being

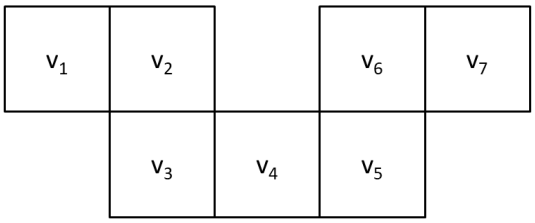


Figure 5.1: Example of graph where an agent has to perform turning actions.

executed, which may further lead to collisions. This is even more probable in dense and optimal plans, where agents often move close to each other.

The intuition says that such desynchronization will indeed happen. We will empirically verify this hypothesis and we will explore several abstract models for MAPF and the output transformations to robot actions. These models not only try to keep the agent synchronous during the execution of the plan but also to avoid collisions caused by some small unforeseen flaws in the execution. We then compare and evaluate these models on an example grid using real robots. Note that the real robots only blindly follow the computed plan and cannot intervene if, for example, an obstacle is detected.

5.2 Models

In this section, we describe several variants of abstract MAPF models and possible transformations of abstract plans to executable sequences of physical actions. Let t_t be the time needed by the robot to turn by 90 degrees to either side and t_f be the time to move forward to the neighboring vertex in the grid. Both t_t and t_f are nonzero. The time spend while the agent is performing the wait operation t_w will depend on each model.

5.2.1 Classic Model

The first and most straightforward model is a direct translation of the abstract plan to the action sequence. We shall call this a *classic* model. At the end of each timestep, an agent is facing in the same direction it entered the vertex. Based on the next location, the agent picks one of the five actions described above and performs it. This means that all move actions consist of possible turning and then going forward. There are no independent turning moves.

The waiting time t_w can be set arbitrarily, but in this case, we choose it to be $t_f + 1/2 * t_t$ reasoning that the two most common actions are (2) and (3,4) and taking the average duration of them.

Note that this abstract model is the same as the typical definition of MAPF, and the solution (sequence of vertices for each agent) is translated into physical actions. Furthermore, we let the agents perform the actions without any delay (each moving action is performed as fast as possible).

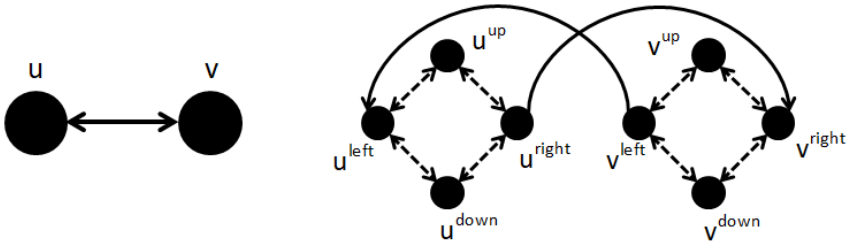


Figure 5.2: Example of how two horizontally connected vertices (left) are split into new vertices (right) describing possible agent's orientations. The dotted edges correspond to turning actions.

5.2.2 Classic Model with Padding

One can easily see that the classic model can be prone to desynchronization, as turning adds time over agents that just move forward (different move actions have different durations depending on if turning is needed). Recall Figure 5.1 and suppose there is another agent with the same number of steps, but all of the actions are moving forward. This agent will reach its goal $4 * t_t$ time units sooner than the agent from the example. This is not consistent with the abstract model, where all of the agents visit the vertices at the same time.

To fix this synchronization issue, we introduce a *classic+wait* model. The basic idea is that each abstract action takes the same time, which is realized by adding some wait time to "fast" actions as a padding. The longest action is (5), therefore each action now takes $2 * t_t + t_f$ including the waiting action ($t_w = 2 * t_t + t_f$). The consequence is that plan execution takes a longer time, which may not be desirable.

The abstract model for *classic* and *classic+wait* models are the same, only the durations of the obtained physical actions differ.

5.2.3 Split Actions Model

One may desire to represent the executable actions directly in the abstract model. In particular, the need to turn can be represented by an abstract turning action. In the reduction-based solvers, this can be done by splitting each vertex v_i from the original graph G into four new vertices $v_i^{up}, v_i^{right}, v_i^{down}, v_i^{left}$ indicating directions where the agent is facing to. The new edges now represent the turn actions, while the original edges correspond to move only actions, see Figure 5.2. Note that when an agent leaves a vertex facing some direction, it will arrive to the neighboring vertex also facing that direction. This change to the input graph also requires a change in the MAPF solver (constraints), because the split vertices need to be treated as one to avoid vertex collisions. This means that at any time there can be at most one agent in those four vertices representing a given location. The abstract plan is then translated to an executable plan in a direct way as the agent is given a sequence of individual actions: wait, turn left/right, and move forward. In this case, the waiting time t_w is set as the bigger time of the remaining actions: $t_w = \max(t_t, t_f)$. We shall call this a *split* model.

5.2.4 Split Model with Padding

A synchronization issue is still present in the *split* model, if the times t_t and t_f are not the same. Recall that the solvers assume an equal duration of all actions. One way to fix this is to use the trick with padding the "fast" actions with some extra waiting time.

This creates a *split+wait* model, where all physical actions take the same time. Specifically we set all actions to take $t_w = \max(t_t, t_f)$. Note that this model may save some time over *classic+wait* since we add much less padding as a result of splitting the turn and move actions into two actions.

5.2.5 Weighted-Edges Model

Another way to solve the possible synchronization issue in the *split* model is to use a weighted MAPF [21]. Each edge in the graph is assigned an integer value that denotes its length. The weighted MAPF solver finds a plan that takes these lengths into account. Formally this can cause gaps in the plan of an agent as the agent may not be present in any vertex in the next step because the agent is still moving over an edge. This indeed does not break our definitions of MAPF and the time is still discrete, only more finely divided. Also, note that it is needed to use a modified solver that can work with edges with non-unit length. Simply splitting the longer edges into several unit edges would allow the agents to turn or wait in the middle of the original edge, which is not allowed. This was discussed in the previous chapter.

The lengths of turning edges are assigned a length of t_t and the other edges are assigned a length of t_f (or its scaled value to integers). The waiting time t_w is set as the greatest common divisor of the remaining actions: $t_w = gcd(t_t, t_f)$. We choose the greatest common divisor so that an agent can wait for exactly the length of any other action, while not granulating the actions too much. We shall call this a *weighted-split* model or *w-split* for short.

Note that for the previous models (*classic+wait* and *split+wait*), synchronization means that all agents leave and enter vertices in the original graph at the same time. This is not necessarily true for the *w-split* model. Let us assume, for example, that there are agents with $t_t = 1$, $t_f = 2$, and $t_w = 1$. In this scenario, it is possible for two agents to have planed the following sequence of actions: {move forward, move forward} and {turn right, move forward, wait}. This scenario is shown in Figure 5.3. Each plan has a duration of 4. While the first agent arrives to some vertices at times 2 and 4, the other arrives to a vertex at time 3 (at the time, the first agent is traversing some edge). This means that *w-split* is not synchronized in the sense that agents are arriving to vertices at the same time, but it is synchronized in the sense that the actions that are planned to happen at the same time are indeed happening at the same time.

5.2.6 Weighted-Edges Model with Padding

While it was possible and useful in the two previous models to pad actions with waiting time so that they take the same time, it is not meaningful to do so with the *w-split* model. In this case, there are still actions that take a different amount of time, however, these different times are incorporated in the theoretical model

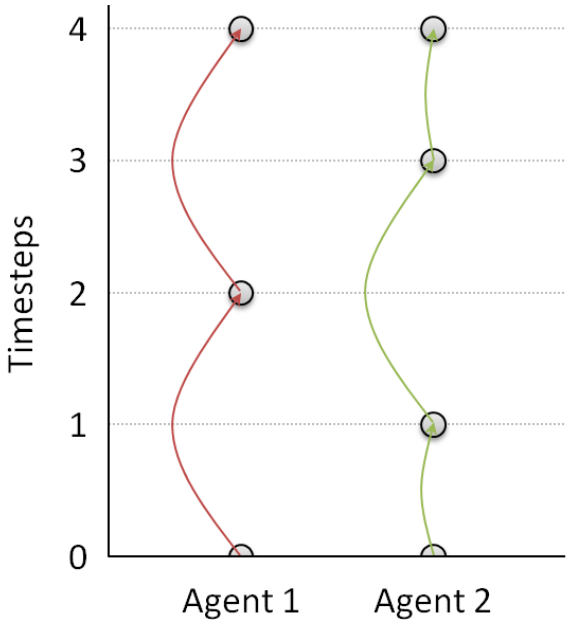


Figure 5.3: Example of a plan for two agents computed by *w-split* that do not arrive to some vertices at the same time. The red agent has planned a sequence of actions {move forward, move forward}, while the green agent has planned {turn right, move forward, wait}.

itself. For this reason, there is no need to create a new model with padding actions for *w-split* model.

5.2.7 Robustness

Some of the previously described models and translation to physical actions (theoretically) guarantee a perfect synchronization of the physical agents when performing the plan. However, as the agents are real robots moving in an imperfect real world, there still might be some desynchronization introduced during the execution. This desynchronization is not caused by the plan itself, but rather by some attributes of the environment. This may include imprecise speed of one of the robots, a wheel slipping, a roughness of the terrain, a desynchronized start, and many more.

To minimize these effects, we may require the abstract plan to keep some space between the moving agents. We will use the notation from k-robust MAPF [49]. The k-robust plan is a valid MAPF plan that in addition requires each vertex of the graph to be unoccupied for at least k timesteps before another agent can enter it. Note that this is a change to the abstract model itself and needs to be performed by the solver, and it is not added during the translation of the abstract plan to the real actions. This enhancement can be added to all of the previously described models and can be combined with the padding translation to real actions. All that is left to do is to choose a proper k for each model.

For the *classic* type models, we choose k to be 1. We presume that this is a good balance between keeping the agents from colliding with each other while

not prolonging the plan too much. For the *split* type models, however, it is not enough to use 1-robustness, as the plan is split into more timesteps. Instead, we use $\max(t_t, t_f)$-robustness.

5.2.8 Overview of Models

In this section, we defined several models that can be seen in Table 5.1 for a quick overview. We defined three different approaches on how to encode the MAPF problem (Classic, Split, Weighted-Edges) with a possible enhancement to the desired plan (Robustness). This creates six abstract models. The abstract models can then be translated to the real actions performed by the physical agents (these actions depend on the model). The translation can then be done in one of two ways – performing the action as fast as possible one after another with no padding, or adding padding to actions that take a shorter amount of time, so all actions take the same time. Note that for the *w-split* model this padding is equivalent with no padding, therefore we omit these two models. Together we defined ten models that can be experimentally tested.

	Classic Model	Split Model	Weighted-Edges Model
no padding	*classic*	*split*	*w-split*
padding	*classic+wait*	*split+wait*	–
no padding + robustness	*classic+ robustness*	*split+ robustness*	*w-split+robustness robustness*
padding + robustness	*classic+wait+ robustness*	*split+wait+ robustness*	–

Table 5.1: Overview of all of the defined models.

5.3 Experiments

The proposed models for MAPF were empirically evaluated on real robots and in this section we will present the obtained results. We shall first give some details on the robots, that we used, on the problem instances, and on a system, that was used to create these instances.

5.3.1 Ozobots

The robots used were Ozobot Evo from the company Evollve [50]. These are small robots (about 3cm in diameter) shown in Figure 5.4. We have chosen them because their built-in actions are close to actions needed in the MAPF problems so there is no need to do low-level robotic programming. The robots are programmable through a programming language Ozoblockly [51], which is primarily meant as a teaching tool for children. This can be seen in the simplicity of the drag-and-drop design of the language, see Figure 5.5. The program is uploaded to the robot and then the robot executes it. Most importantly, the robots have sensors underneath that allow the robot to follow a line and to detect intersection. An intersection is defined as at least two lines crossing each other.

Figure 5.4: Ozobot Evo from Evollve used for the experiments. Picture is taken from [50].

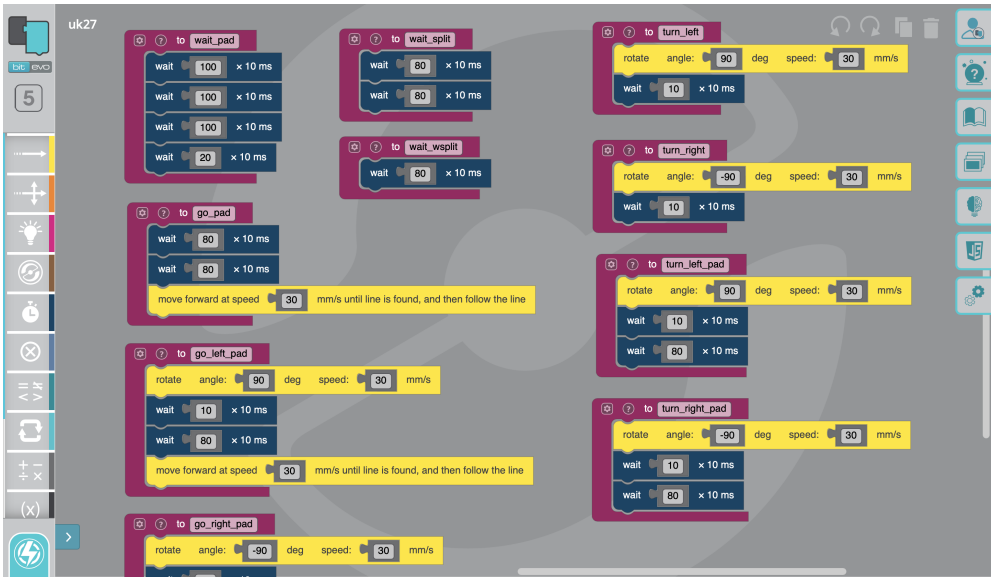


Figure 5.5: Example program coded in Ozoblockly language. The displayed procedures encode actions used by Ozobots for execution of MAPF plans.

The robots also have forward and backward-facing proximity sensors allowing them to detect obstacles. We used them to synchronize the start of robots (see further), but we did not exploit sensors further during plan execution. In addition, the robots have LEDs and speakers that act as the output of the robot. We use them to indicate some states of the robot such as a finished plan. The moving speed and turning speed can be adjusted up to a speed limit of the robot.

There are some drawbacks in the simplicity of the robots. The main one is that there is currently no communication between multiple robots and therefore starting an instance of MAPF for all of the present robots at the same time is difficult. To solve this problem, we used the proximity sensors and forbid the agents to start performing the computed plan if an obstacle is present in front of them. An obstacle was placed in front of all of the agents and once all of them were ready to start executing the plan, all of the obstacles were removed. This ensured that the start time was identical and any desynchronization at the end of the plan was caused during the execution and not at the start.

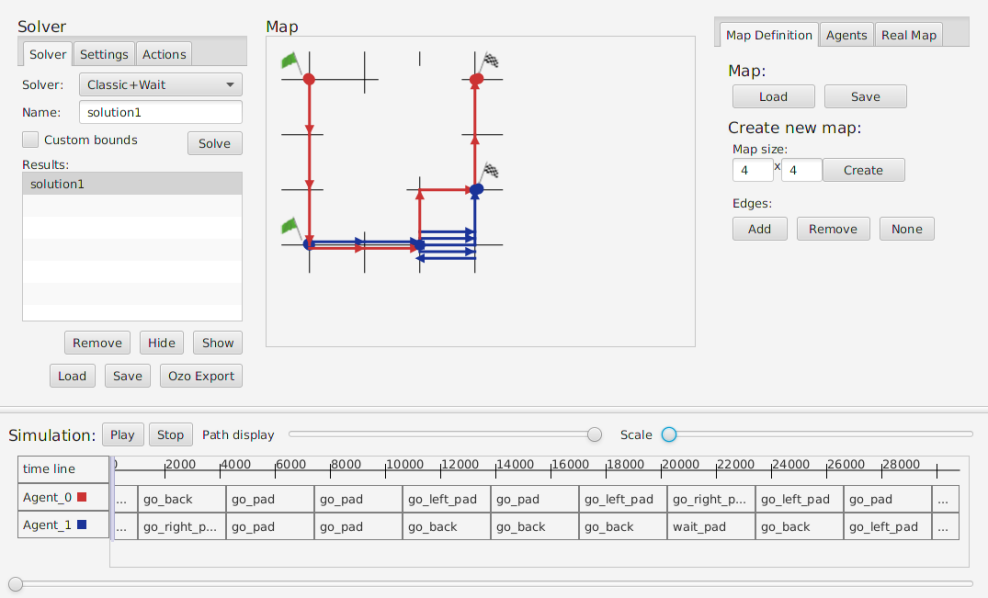


Figure 5.6: User interface of a system that lets the user define and solve MAPF instances. The picture shows an instance on a 4 by 4 grid map with obstacles and two agents (red and blue). The solution shown was computed by the model *classic+wait*.

5.3.2 MAPF Scenario Software

To simplify the process of creating and solving MAPF instances, a software MAPF Scenario [52] was designed that lets its user define grid maps, place agents, and solve this instance with any of the models described above. We will describe some of the features of the system now. The user interface can be seen in Figure 5.6.

First, a user needs to define a grid map over which the instance will be built or to load a previously created map. The user can define the dimensions of the grid, then obstacles can be introduced into the map by removing some of the vertices and edges of the graph. This map can be also printed on a paper, in which case, the user will be asked to define the length of the edges. A set of agents can be placed on this map. For each agent, the user will be asked to specify its color, starting position, and goal position. The map and all agents specify the MAPF instance, which is displayed in the middle part of the user interface.

To solve the defined instance, the user can choose from ten different solvers, which correspond to the defined models in the previous section. Once a solution is found, the actions for each agent are displayed at the bottom part of the user interface. Note that each action has a defined duration. This lets us observe the total time of the plan on the timeline and the synchronization of the plan.

For even better visualization, it is possible to simulate the found plan on the displayed map. In such a case, circles representing the agents will appear and move in the map based on the actions shown in the bottom part.

Lastly, the user can choose to export the found plan to the Ozoblockly language that can be uploaded to the Ozobot robots. Running the plan on real robots rather than in simulation can show some further flaws in the plan caused by the dimensions of the robots and imperfections of the real world.

5.3.3 Problem Instance

Using the MAPF system described above, several instances were crafted to test the defined models. These instances are shown in Figures 5.7 – 5.11.

As opposed to the usual representation, where agents reside in the cells in between lines, here the agents follow the line and a vertex is represented as the crossing of two lines. These maps were printed on a paper in two scales. In the first scale, each edge is 5cm long and the line is 5mm thick as per Ozobots' recommended specification. The edge length was chosen to allow two robots to safely stay in neighboring vertices and to observe even minor desynchronization due to turning. For the second scale, we doubled the length of the edges to 10cm while keeping the thickness at 5mm. This second scale was chosen to observe the behavior when robots are not as close to each other and thus allowing for bigger slack in the synchronization. We shall further refer to these sets of maps as smaller and larger maps.

In the Figures 5.7 – 5.11, the initial (s_i) and goal locations (g_i) are indicated. These circles were not printed, they are added as a notation for the reader. The robots are placed on the indicated initial location facing upwards (north). After reaching the goal location, it is not required to be facing any specific direction, only the presence at the specified intersection is required.

The speed of the robots was set in such a way that moving along 5cm of a line takes 1600ms (3200ms for 10cm lane) and turning takes 800ms. This means that $t_f = 1600$ for smaller maps, $t_f = 3200$ for larger maps, and $t_t = 800$. However, since all the numbers are divisible by 800, we can simplify the times for the MAPF solver to $t_f = 2$ for smaller maps, $t_f = 4$ for larger maps, and $t_t = 1$. This then gives us all the required times for the models as described in the previous section.

Each instance was designed to test some property of the theoretical models. Instance *Bottleneck* (Figure 5.7) is the largest map, that forces four agents to pass through a narrow pass at roughly the same time, thus creating a bottleneck, where agents need to wait for others.

Instance *Switch* (Figure 5.8) requires two agents to switch places. This map is very small and any desynchronization and close proximity can cause the agents to hit each other.

In instance *Basket* (Figure 5.9) the two agents do not interact with each other, however, each agent has to travel a different distance. One will use the bottom path, while the other will use the top path.

In instance *Riddle* (Figure 5.10), three agents need to move along a cycle. This is again a very small map with big interaction between the agents.

Lastly, instance *Spiral* (Figure 5.11) requires three agents to follow the spiral structure of the map. This requires the agents to turn many times, however, each agent turns a different number of times.

5.3.4 Results

We generated plans using each MAPF model for all of the problem instances described above and then we executed the plans five times in total for each model. Several properties were measured with results shown in Tables 5.2 – 5.6. The tables show results for both smaller maps (left columns) and larger maps (right

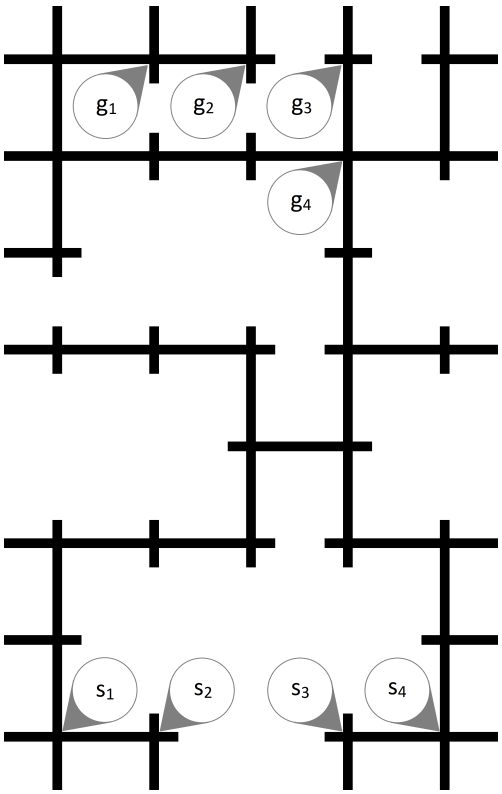


Figure 5.7: Instance map for Ozobots called *Bottleneck*.

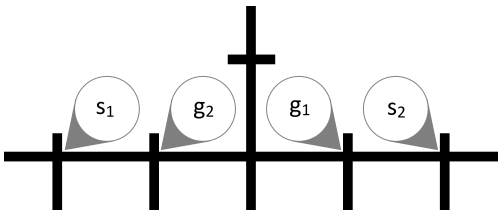


Figure 5.8: Instance map for Ozobots called *Switch*.

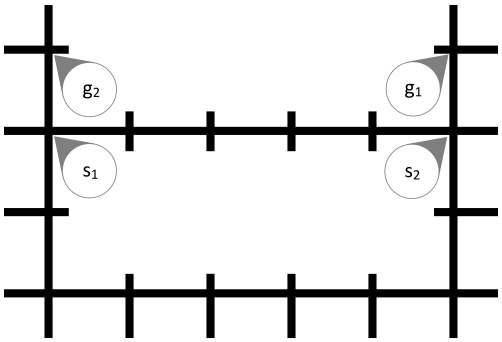


Figure 5.9: Instance map for Ozobots called *Basket*.

columns). Due to the size of the tables, they are placed at the very end of this chapter.

Computed makespan is the makespan of the plan returned by the MAPF solver. It is measured by the (weighted) number of abstract actions. Note that the *split* models have larger makespan than the rest because the *split* models use a finer resolution of actions, namely turning actions are included in the makespan

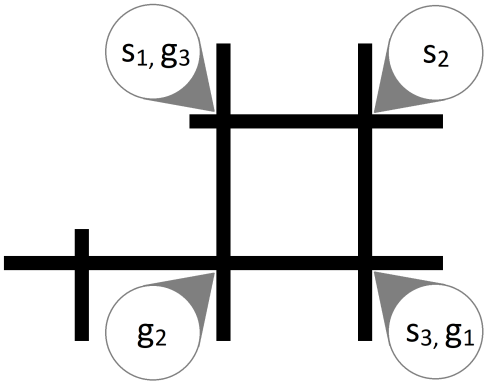


Figure 5.10: Instance map for Ozobots called *Riddle*.

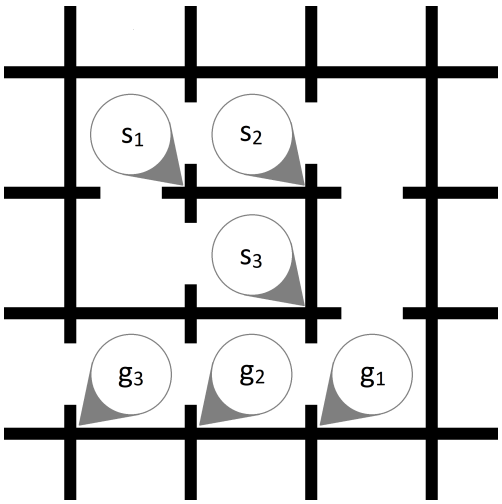


Figure 5.11: Instance map for Ozobots called *Spiral*.

calculation. This is even more noticeable with *w-split* and *w-split+robustness*, where the moving-forward action has a duration (weight) of two for smaller maps, and four for larger maps. Also note that for all models with the exception to *w-split* and *w-split+robustness*, the makespan for smaller and larger maps are identical. This is because there are no changes to the theoretical model if longer edges are used. Only the translation of the robot's actions is different. On the other hand, *w-split* and *w-split+robustness* compute the solution using the length of the edges.

The number of failed runs is also shown. A model that had most often problems finishing the run is the *classic* model while the rest (with the exception of *split* and *split+robustness* in *Bottleneck* shown in Table 5.2) managed to finish all of the runs. A run fails if there is a collision that throws any of the robots off the track so the plan cannot be finished. One such collision can be seen in Figure 5.12. The average number of collisions per run shows how many collisions that did not ruin the plan occurred. These collisions can range from a small one, where the robots only touched each other (such as in Figure 5.13) and did not affect the execution of the plan, to big collisions, where the agent was slightly delayed in its individual plan, but still managed to finish the plan. In the case that the execution fails, we present the number of collisions occurring before the major collision that stopped the plan.

Figure 5.12: Example of collision that caused the plan to fail on instance *Riddle* solved by the model *Classic*. After the collision, the middle robot was unable to follow the line anymore.

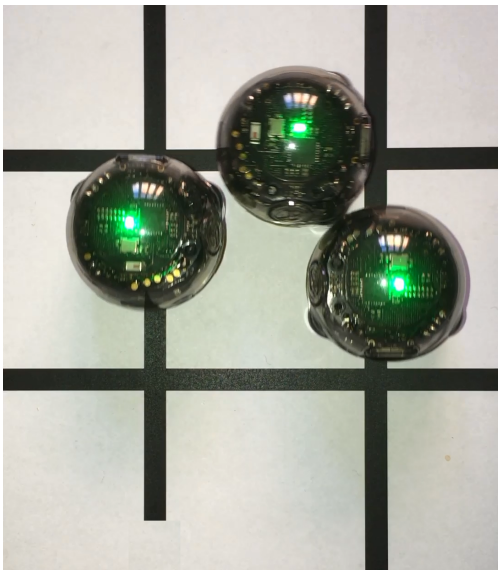


Figure 5.13: Example of collision that did not cause the plan to fail on instance *Riddle* solved by the model *Classic+wait*.

Since we are using the makespan objective function, all of the plans can have their length equal to the longest plan without worsening the objective function. Even if the agents reached their destination sooner, their plan was prolonged by waiting actions to match the length of the longest plan. To visually observe this, we used the LEDs on the robots. The LEDs were turned on during the whole plan (including wait actions) and turned off once the plan was finished. This helped us measure the overall time of the plan execution as the time from start to the last robot turning LEDs off. For the models that did not finish any of the five runs, there is no total time to show.

Each agent was let to execute the plan without interference with other agents to measure the difference between the fastest and slowest agent as Max Δ time. If the agents are perfectly synchronized then this Δ should be zero. All of the times are rounded to one-tenth of a second because the measurements were conducted by hand.

From the number of collisions, the total times, and the Max Δ times, we can conclude some properties of the models. Indeed, models that have added *+wait* and *w-split* models keep the agents synchronous, while the other models do not (there is a gap between finishing the plans by different agents). From all models, the *classic+wait+robustness* model is the slowest one to perform the plan. This is expected as this model uses the longest durations of actions and the robustness may add some extra steps to perform. We can see that by just splitting the actions in *split* models, we save some time even when we use the padding in models *split+wait* and *split+wait+robustness*.

Further, we can see that even if the agents are synchronous, some collisions may appear, since the agents have a nonzero diameter and are often moving close

to each other. This issue is solved by making a k-robust plan, however, if the agents are not synchronous, the agents can still collide.

Also note that on the larger maps, where the edges are longer and thus the agents are not moving close to each other, there are fewer collisions. This type of map makes the robustness less useful since even the non-robust plans do not usually collide. On the other hand, adding the padding in *+wait* models is counterproductive because it prolongs the plans too much.

Desynchronization is still an issue even in the larger maps, even though it is not as much visible on any map with the exception of *Bottleneck* shown in Table 5.2. On the other maps, the plans are too short for the desynchronization to cause any collision, however, with a longer plan we can see that the difference in individual plan lengths is large for *split* and *split+robustness* models.

Recall that adding *+wait* is just a change in translation of the plan to the actions, while adding *+robustness* requires to find a different plan. This can create a phenomenon seen in the results for instance *Basket* shown in Table 5.4. Here the plans for *classic* and *split* have the same makespan as *classic+robustness* and *split+robustness* respectively, while the Max Δ times are different. This is caused by the solver finding a different plan with the same length for each model. By a chance, the *robustness* models found a plan of the same length (each agent performed the same amount of turning and moving actions), while the *classic* and *split* models found a plan where the agents performed different actions.

To compare various models across different maps, we use a *quality index* (a similar index is used, for example, in International Planning Competitions). This index equals one for the best value of a given parameter and progressively decreases to zero for worse values. Formally, let *best* be the minimum value of a given parameter across all models and a given map (we assume that the minimum value is the best value, which is the case of all evaluation parameters that we measured). Let *value* be the value of the parameter for a given model and a given map. Then the quality index for that model and parameter equals $\frac{best}{value}$. For parameters, where the minimum value can be zero (for example, the number of collisions), we used a modified version of the quality index $\frac{best+1}{value+1}$. For non-finished runs, we used a big number 6000 for the total time. Quality indexes are then summed across all the maps. The larger the sum is, the better the model performed in that parameter. Table 5.7 shows the accumulated indexes for all five maps.

In general, we can see that simply translating the sequence of vertices to the actions of robots is not enough to create a good executable plan. This can be solved by padding all of the movements with some waiting time so that the agents remain synchronous. However, this costs extra time during the execution. Splitting the actions to turning actions and moving actions provides us with finer granularity and thus a faster plan. This plan is still not synchronous, so padding is required to ensure the agents stay synchronous. The overall best solution is to take the action lengths into account while creating the plan. This allows the agents to stay synchronous without any extra waiting time, thus having the fastest execution time. Adding robustness furthermore betters the plan if the agents are moving close to each other. Table 5.7 confirms that robust plans are without collisions, but their execution takes more time.

	Computed Makespan		Failed Runs		Number of Collisions		Total Time [s]		Max Δ time [s]	
classic	**17**	**17**	5	**0**	3	**0**	NA	**63.1**	4	2.7
classic+wait	**17**	**17**	**0**	**0**	6	**0**	53.8	78.9	**0**	**0**
classic+robustness	19	19	**0**	**0**	**0**	**0**	40.4	68.7	1.2	1.1
classic+wait+robustness	19	19	**0**	**0**	**0**	**0**	59.9	88.2	**0**	**0**
split	27	27	**0**	5	4	1	37.6	NA	2	7.1
split+wait	27	27	**0**	**0**	3	**0**	44.2	85.2	**0**	**0**
split+robustness	28	28	5	5	2	1	NA	NA	4.4	14.3
split+wait+robustness	28	28	**0**	**0**	**0**	**0**	46.1	88.2	**0**	**0**
w-split	44	80	**0**	**0**	1	**0**	**37**	65	**0**	**0**
w-split+robustness	44	80	**0**	**0**	**0**	**0**	**37**	65	**0**	**0**

Table 5.2: Measured performance of Ozobots on map *Bottleneck* (Figure 5.7) using each proposed model. The left columns are for 5cm edges, the right columns are for 10cm edges.

	Computed Makespan		Failed Runs		Number of Collisions		Total Time [s]		Max Δ time [s]	
classic	**6**	**6**	1	**0**	1	**0**	**14.3**	**22**	2.6	2.3
classic+wait	**6**	**6**	**0**	**0**	1	**0**	18.2	26	**0**	**0**
classic+robustness	8	8	**0**	**0**	**0**	**0**	18.1	29.1	2.5	2.2
classic+wait+robustness	8	8	**0**	**0**	**0**	**0**	24.5	36.5	**0**	**0**
split	11	11	**0**	**0**	1	1	15	**22**	0.8	0.3
split+wait	11	11	**0**	**0**	**0**	**0**	18.1	34	**0**	**0**
split+robustness	11	11	**0**	**0**	**0**	**0**	**14.3**	**22**	**0**	**0**
split+wait+robustness	11	11	**0**	**0**	**0**	**0**	18.3	34	**0**	**0**
w-split	16	26	**0**	**0**	**0**	**0**	**14.3**	**22**	**0**	**0**
w-split+robustness	16	26	**0**	**0**	**0**	**0**	14.5	**22**	**0**	**0**

Table 5.3: Measured performance of Ozobots on map *Switch* (Figure 5.8) using each proposed model. The left columns are for 5cm edges, the right columns are for 10cm edges.

	Computed Makespan		Failed Runs		Number of Collisions		Total Time [s]		Max Δ time [s]	
classic	**11**	**11**	**0**	**0**	**0**	**0**	23.9	39.5	2.5	2.3
classic+wait	**11**	**11**	**0**	**0**	**0**	**0**	34.1	50	**0**	**0**
classic+robustness	**11**	**11**	**0**	**0**	**0**	**0**	21.3	**37.1**	**0**	**0**
classic+wait+robustness	**11**	**11**	**0**	**0**	**0**	**0**	34.1	50	**0**	**0**
split	15	15	**0**	**0**	**0**	**0**	22.4	39.4	1	2.2
split+wait	15	15	**0**	**0**	**0**	**0**	24.6	46.7	**0**	**0**
split+robustness	15	15	**0**	**0**	**0**	**0**	**21.2**	**37.1**	**0**	**0**
split+wait+robustness	15	15	**0**	**0**	**0**	**0**	24.6	46.7	**0**	**0**
w-split	25	45	**0**	**0**	**0**	**0**	21.4	**37.1**	**0**	**0**
w-split+robustness	25	45	**0**	**0**	**0**	**0**	21.4	**37.1**	**0**	**0**

Table 5.4: Measured performance of Ozobots on map *Basket* (Figure 5.9) using each proposed model. The left columns are for 5cm edges, the right columns are for 10cm edges.

	Computed Makespan		Failed Runs		Number of Collisions		Total Time [s]		Max Δ time [s]	
classic	**3**	**3**	5	**0**	1	**0**	NA	11	1.7	1.9
classic+wait	**3**	**3**	**0**	**0**	2	**0**	8.6	11.5	**0**	**0**
classic+robustness	7	7	**0**	**0**	**0**	**0**	15.7	25.1	0.9	0.8
classic+wait+robustness	7	7	**0**	**0**	**0**	**0**	21.3	30.6	**0**	**0**
split	6	6	**0**	**0**	**0**	**0**	8.5	13.1	0.9	2.2
split+wait	6	6	**0**	**0**	2	**0**	10	18	**0**	**0**
split+robustness	8	8	**0**	**0**	**0**	**0**	11.2	17.1	1	2.2
split+wait+robustness	8	8	**0**	**0**	**0**	**0**	13.4	24.3	**0**	**0**
w-split	8	12	**0**	**0**	2	**0**	**7.8**	**10.9**	**0**	**0**
w-split+robustness	9	13	**0**	**0**	**0**	**0**	8.6	11.6	**0**	**0**

Table 5.5: Measured performance of Ozobots on map *Riddle* (Figure 5.10) using each proposed model. The left columns are for 5cm edges, the right columns are for 10cm edges.

	Computed Makespan		Failed Runs		Number of Collisions		Total Time [s]		Max Δ time [s]	
classic	**14**	**14**	5	**0**	1	**0**	NA	**49.2**	1.6	1.6
classic+wait	**14**	**14**	**0**	**0**	6	**0**	43.8	64.3	**0**	**0**
classic+robustness	16	16	**0**	**0**	**0**	**0**	32.7	56.3	1.7	1.5
classic+wait+robustness	16	16	**0**	**0**	**0**	**0**	50.1	74	**0**	**0**
split	22	22	**0**	**0**	**0**	**0**	30.3	52.3	1.3	2.3
split+wait	22	22	**0**	**0**	6	**0**	36.1	69.1	**0**	**0**
split+robustness	23	23	**0**	**0**	**0**	**0**	31.2	53.1	1.2	2.2
split+wait+robustness	23	23	**0**	**0**	**0**	**0**	37.5	72.2	**0**	**0**
w-split	36	66	**0**	**0**	**0**	**0**	**30.2**	54	**0**	**0**
w-split+robustness	36	66	**0**	**0**	**0**	**0**	**30.2**	54.1	**0**	**0**

Table 5.6: Measured performance of Ozobots on map *Spiral* (Figure 5.11) using each proposed model. The left columns are for 5cm edges, the right columns are for 10cm edges.

	Computed Makespan		Failed Runs		Number of Collisions		Total Time		Max Δ time	
classic	**5.00**	**5.00**	2.00	**5.00**	2.75	**5.00**	1.90	**4.93**	1.52	1.61
classic+wait	**5.00**	**5.00**	**5.00**	**5.00**	2.12	**5.00**	3.69	4.10	**5.00**	**5.00**
classic+robustness	3.95	3.95	**5.00**	**5.00**	**5.00**	**5.00**	4.12	3.98	2.64	2.74
classic+wait+robustness	3.95	3.95	**5.00**	**5.00**	**5.00**	**5.00**	2.79	3.08	**5.00**	**5.00**
split	3.04	3.04	**5.00**	4.17	3.70	4.00	4.80	3.72	2.35	1.82
split+wait	3.04	3.04	**5.00**	**5.00**	2.73	**5.00**	4.11	3.50	**5.00**	**5.00**
split+robustness	2.87	2.87	4.17	4.17	4.33	4.50	3.67	3.57	3.14	2.69
split+wait+robustness	2.87	2.87	**5.00**	**5.00**	**5.00**	**5.00**	3.83	3.29	**5.00**	**5.00**
w-split	1.97	1.15	**5.00**	**5.00**	3.83	**5.00**	**4.99**	4.88	**5.00**	**5.00**
w-split+robustness	1.92	1.13	**5.00**	**5.00**	**5.00**	**5.00**	4.88	4.82	**5.00**	**5.00**

Table 5.7: Summary performance of Ozobots using each proposed model (quality indexes, larger value is better). The left columns are for 5cm edges, the right columns are for 10cm edges.

Conclusion

In this thesis, we investigated the multi-agent path finding (MAPF) problem. First, we defined the problem and discussed the possible settings. This mainly includes the allowed movements – pebble motion and parallel motion. The other specification is that we are interested in optimal solutions with regards to a cost function – makespan or sum of costs. It is known that finding an optimal solution with regards to either of these two functions is NP-Hard.

To solve MAPF, we introduced a solver that reduces the MAPF problem to the Boolean satisfiability decision problem (SAT). This approach is not novel by itself, however, our solver is an implementation in Picat programming language which is simple, yet powerful logic-based language that allows the user to specify constraints that are in turn transferred into a CNF formula. This means that the code is quite simple, readable, and most importantly easy to change to accommodate additional constraints. This property is important, as this reduction-based solver is the basis for all experiments presented in this thesis.

The first contribution is modeling the sum of costs optimal MAPF using our reduction-based solver. We compared two approaches on how to find an appropriate number of layers in a time-expanded graph needed to solve the problem optimally. An additional enhancement that tries to minimize the number of variables in the CNF formula was used. In our experiments, we found that it is usually more advantageous to create more layers in one step (at the expense of overshooting the minimal required number) than to increase the number of layers one by one.

Next, we observed that there are types of instances that are easier to solve using different algorithms. Namely, we differentiate instances with large graphs and small density of agents, and smaller graphs with a high density of agents. The former is usually preferably solved by the CBS algorithm, while the latter is preferably solved by the reduction-based solver. This leads to the idea of combining both algorithms using an independence detection algorithm that splits the initial instance into smaller subproblems that are in turn solved by different appropriate solvers.

Another contribution is exploring an extension of the original MAPF definition where new agents can appear over time, while agents that reach their goal destination disappear. This setting corresponds to a restricted space that needs to coordinate agents that are moving through it, for example, an intersection, a harbor, or airport airspace. It is also orthogonally related to a setting where the number of agents is fixed, however, the goal locations are changing over time (such setting corresponds to a warehouse management). In our study, we compared techniques that try to minimize the distance traveled by each agent, minimize the number of interruptions to an already existing plan, or a combination of both.

The last contribution is in the form of an extensive experimental study on the usefulness of plans when applied to real robots. The findings suggest that the straight-forward translation of plans to robot motion can cause desynchronization or can be costly in execution time overhead. A more sophisticated approach deals with the turning actions of the robot in the planning phase.

Future Work

There are many open problems that are related to MAPF. We shall mention just a few that are closely related to the topics discussed in this thesis.

The hybrid algorithm that combines the CBS solver and the reduction-based solver can combine any number of different optimal MAPF solver. If we are able to find an algorithm that excels in some other type of instances, it would be worth incorporating this solver into the hybrid solver. Note that the hybrid solver is stronger than just a portfolio of solvers and also some changes to the underlying solvers are needed.

So far we tested only the classical setting of MAPF on the Ozobot robots. An interesting experimental study is to test the two mentioned dynamic settings – online MAPF (intersection model) and life-long MAPF (warehouse model). The challenge of this experiment is of technical nature. As of yet, there is no system that provides us with controlled starting of a large number of Ozobots, and that allows for a plan update during the execution phase.

Bibliography

[1] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Roman Barták, and Eli Boyarski. Multi-agent pathfinding: Definitions, variants, and benchmarks. In Pavel Surynek and William Yeoh, editors, *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, pages 151–159. AAAI Press, 2019.

[2] Daniel Kornhauser, Gary L. Miller, and Paul G. Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 241–250, 1984.

[3] Pavel Surynek. Compact representations of cooperative path-finding as SAT based on matchings in bipartite graphs. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*, pages 875–882, 2014.

[4] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 662–667, 2011.

[5] Daniel Ratner and Manfred K. Warmuth. Nxn puzzle and related relocation problem. *J. Symb. Comput.*, 10(2):111–138, 1990.

[6] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA.*, 2013.

[7] Pavel Surynek. On the complexity of optimal parallel cooperative path-finding. *Fundam. Inform.*, 137(4):517–548, 2015.

[8] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016.*, pages 145–147, 2016.

[9] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[11] Patrick Wyatt. The StarCraft path-finding hack, 2013. `https://www.codeofhonor.com/blog/the-starcraft-path-finding-hack`.

[12] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics Auton. Syst.*, 41(2-3):89–99, 2002.

[13] David Silver. Cooperative pathfinding. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*, pages 117–122, 2005.

[14] Kurt M. Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res.*, 31:591–656, 2008.

[15] Lucia Pallottino, Vincenzo Giovanni Scordio, Antonio Bicchi, and Emilio Frazzoli. Decentralized cooperative policy for conflict resolution in multivehicle systems. *IEEE Trans. Robotics*, 23(6):1170–1183, 2007.

[16] Hang Ma and Sven Koenig. AI buzzwords explained: multi-agent path finding (MAPF). *AI Matters*, 3(3):15–19, 2017.

[17] Nathan Sturtevant Sven Koenig and Ariel Felner. mapf.info, 2019. `http://mapf.info/`.

[18] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent path finding. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[19] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. ICBS: improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 740–746, 2015.

[20] Jiří Švancara, Marek Vlk, Roni Stern, Dor Atzmon, and Roman Barták. Online multi-agent pathfinding. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 7732–7739, 2019.

[21] Roman Barták, Jiri Svancara, and Marek Vlk. A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 748–756, 2018.

[22] Hang Ma, Jiaoyang Li, TK Kumar, and Sven Koenig. Lifelong multi-agent path finding for online pickup and delivery tasks. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 837–845. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[23] Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*, 2012.

[24] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *2009 IEEE International Conference on Robotics and Automation, ICRA 2009, Kobe, Japan, May 12-17, 2009*, pages 3613–3619, 2009.

[25] Pavel Surynek. Solving abstract cooperative path-finding in densely populated environments. *Comput. Intell.*, 30(2):402–450, 2014.

[26] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*, 2014.

[27] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[28] Jiří Švancara and Pavel Surynek. New flow-based heuristic for search algorithms solving multi-agent path finding. In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence, ICAART 2017, Volume 2, Porto, Portugal, February 24-26, 2017*, pages 451–458, 2017.

[29] Roman Barták, Neng-Fa Zhou, Roni Stern, Eli Boyarski, and Pavel Surynek. Modeling and solving the multi-agent pathfinding problem in picat. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 959–966, 2017.

[30] Pavel Surynek. Makespan optimal solving of cooperative path-finding via reductions to propositional satisfiability. *CoRR*, abs/1610.05452, 2016.

[31] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Boolean satisfiability approach to optimal multi-agent path finding under the sum of costs objective: (extended abstract). In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore, May 9-13, 2016*, pages 1435–1436, 2016.

[32] Malcolm Ryan. Constraint-based multi-robot path planning. In *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*, pages 922–928, 2010.

[33] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. In *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013*, pages 3612–3617, 2013.

[34] Jingjin Yu and Steven M. LaValle. Multi-agent path planning and network flow. In *Algorithmic Foundations of Robotics X - Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics, WAFR 2012, MIT, Cambridge, Massachusetts, USA, June 13-15 2012*, pages 157–173, 2012.

[35] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, pages 810–818, 2016.

[36] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Meta-agent conflict-based search for optimal multi-agent path finding. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*, 2012.

[37] Roman Barták and Jiří Švancara. On sat-based approaches for multi-agent path finding with the sum-of-costs objective. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, pages 10–17, 2019.

[38] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[39] Jiří Švancara and Roman Barták. Combining strengths of optimal multi-agent path finding algorithms. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 1, Prague, Czech Republic, February 19-21, 2019*, pages 226–231, 2019.

[40] Trevor Scott Standley. Independence detection for multi-agent pathfinding problems. In *Multiagent Pathfinding, Papers from the 2012 AAAI Workshop, MAPF@AAAI 2012, Toronto, Ontario, Canada, July 22, 2012.*, 2012.

[41] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer, 2015.

[42] Richard E. Korf. Real-time heuristic search. *Artif. Intell.*, 42(2-3):189–211, 1990.

[43] Peter Stone, Gal A Kaminka, Sarit Kraus, Jeffrey S Rosenschein, et al. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *AAAI*, 2010.

[44] Anirudha Majumdar and Russ Tedrake. Robust online motion planning with regions of finite time invariance. In *Algorithmic Foundations of Robotics X*, pages 543–558. Springer, 2013.

[45] Jur P Van Den Berg and Mark H Overmars. Prioritized motion planning for multiple robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 430–435, 2005.

[46] Andrew Dobson, Kiril Solovey, Rahul Shome, Dan Halperin, and Kostas E Bekris. Scalable asymptotically-optimal multi-robot motion planning. In *International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 120–127. IEEE, 2017.

[47] Julio E Godoy, Ioannis Karamouzas, Stephen J Guy, and Maria Gini. Adaptive learning for multi-agent navigation. In *the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1577–1585, 2015.

[48] Roman Barták, Jiří Švancara, Věra Škopková, David Nohejl, and Ivan Krasičenko. Multi-agent path finding on real robots. *AI Commun.*, 32(3):175–189, 2019.

[49] Dor Atzmon, Ariel Felner, Roni Stern, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. k-robust multi-agent path finding. In *Proceedings of the Tenth International Symposium on Combinatorial Search, Edited by Alex Fukunaga and Akihiro Kishimoto, 16-17 June 2017, Pittsburgh, Pennsylvania, USA.*, pages 157–158, 2017.

[50] Ozobot & Evollve, Inc. Ozobot — Robots to code, create, and connect with, 2018. `https://ozobot.com/`.

[51] Evollve, Inc., Ozobot & OzoBlockly. Welcome to OzoBlockly, 2015. `https://ozoblockly.com/`.

[52] Ivan Krasičenko. *Multi-agent Path Finding on Ozobots.* Bachelor's Thesis, Charles University, Faculty of Mathematics and Physics, 2018.

Glossary

This addendum contains a list of used definitions, abbreviations, and terms that are specific for the covered topic. We do not include terms that were used only once and are not referenced in the text again.

4-connected grid graph – A graph that is created from a grid. The neighbors are only the four orthogonally adjacent tiles, not the diagonally adjacent tiles.

A – A set of agents.

a_i – Agent i. It is characterized by its start and goal location.

$At(v, i, t)$ – In the context of reduction-based solver: Propositional variable indicating that agent a_i is at vertex v at timestep t.

Capacity – In the context of weighted MAPF: Maximal number of agents that can traverse an edge at the same time.

CBS – Conflict Based Search algorithm.

Classic model – In the context of experiments on Ozobots: A model that directly translates a sequence of vertices to robot actions.

CNF – Conjunctive normal form of a formula.

Constraint tree – A binary tree over which the top-level search is performed in CBS algorithm.

$Cons(a_i, v, t)$ – Constraint used in CBS algorithm that represents agent a_i is not present in vertex v in timestep t.

Cycle conflict – When at least three agents move along a fully occupied cycle.

Edge conflict – When two or more agents traverse the same edge in the same direction at the same time.

Execution – The carrying out of a found plan. The steps of the plan that were actually performed. The future steps of a plan can be changed while executing.

Following conflict – When one agent is planned to occupy a vertex that was occupied by another agent in the previous timestep.

g_i – Goal location of agent a_i.

Hybrid – Hybrid MAPF algorithm that combines CBS and reduction-based solvers.

ID – Independence detection algorithm.

Instance of MAPF – The input for MAPF. Consists of a graph and a set of agents.

k-robustness – A MAPF solution that forces a gap of k steps in between the agents.

$LB(X)$ – Lower bound estimate of cost function X. Specifically $LB(Mks)$ and $LB(SoC)$.

Lifelong MAPF – MAPF setting where agents may be assigned new goal locations after reaching their original goal location.

Makespan – A cost function. It is defined as the minimal timestep all of the agents are present in their respective goal locations.

MAPF – Multi-agent path finding.

MAPF Scenario – A program that lets its user design, solve, and simulate MAPF problems. In addition, it is possible to generate codes for Ozobots.

Mks – See Makespan.

$Mks(\Pi)$ – Makespan of plan Π.

Offline MAPF – In the context of online MAPF: The classical MAPF setting where the set of the agents is unchanging.

Offline optimal solver – In the context of online MAPF: A MAPF solver that knows all of the agents that will appear and plans optimally with this knowledge. No offline optimal solver can exist in practice.

OID – In the context of online MAPF: Online independence detection.

Online MAPF – MAPF setting where new agents may appear over time.

Ozobot Evo – Robots used in the experiments performing MAPF plans.

$\psi_A^{\pi_B}$ – An optimal plan for a group of agents A while avoiding some plan π_B for a group of agents B.

$|\Pi|$ – The length of a plan Π.

Padding – In the context of experiments on Ozobots: Added wait time to synchronize robots during execution. The desynchronization is caused by actions of different lengths.

Parallel motion – The setting of MAPF where edge, vertex, and swapping conflicts are forbidden; following and cycle conflicts are allowed.

$Pass(u, v, i, t)$ – In the context of reduction-based solver: Propositional variable indicating that agent a_i is traversing edge (v, u) at timestep t.

Pebble motion – The setting of MAPF where edge, vertex, following, cycle, and swapping conflicts are forbidden. None of the defined conflicts are allowed.

Picat – Logic-based programming language used to create our reduction-based MAPF solver.

Plan – Sequence of locations a given agent visits. If we are interested in only one agent, we talk about a single-agent plan. If there are more agents involved, we talk about a joint plan.

RA – In the context of online MAPF: Replan all.

Re-route – In the context of online MAPF: A change to already found plan due to the appearance of a new agent.

+robust – see k-robustness.

RS – In the context of online MAPF: Replan single.

RSG – In the context of online MAPF: Replan single grouped.

s_i – Start location of agent a_i.

SAT – The problem of the satisfiability of Boolean formulae.

Snapshot Optimal plan – In the context of online MAPF: A plan that is optimal until a new agent appears.

SoC – See Sum of costs.

$SoC(\Pi)$ – Sum of costs of plan Π.

SP_i – Shortest path of agent a_i when ignoring other agents. This corresponds to the shortest path between vertices s_i and g_i.

Split model – In the context of experiments on Ozobots: A model that takes the agent's orientation into consideration.

SubID – In the context of online MAPF: Suboptimal independence detection.

Sum of costs – A cost function. It is defined as the sum of lengths of individual paths.

Swapping conflict – When two agents traverse the same edge in opposite directions at the same time.

t_f – In the context of experiments on Ozobots: Time it takes the robot to move forward to a neighboring vertex.

t_t – In the context of experiments on Ozobots: Time it takes the robot to turn 90°.

t_w – In the context of experiments on Ozobots: Time spend performing wait operation.

Time-expanded graph – A graph that is created by copying the original graph t times. Edges are connected to the neighboring layers.

$UB(X)$ – Upper bound estimate of cost function X. Specifically $UB(Mks)$ and $UB(SoC)$.

Vertex conflict – When two or more agents are located in the same vertex at the same time.

+wait – see padding.

Weighted-split model – In the context of experiments on Ozobots: A model that takes both the agent's orientation and the different t_t and t_f into consideration.

Weighted MAPF – MAPF variant where the agents are moving over a graph with a non-unit length of edges.

List of Figures

List of Tables

Full List of Publications

2020

Roman Barták, Jiří Švancara [35%], and Ivan Krasicenko. MAPF scenario: Software for evaluating MAPF plans on real robots. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13602–13603. AAAI Press, 2020.

Jindrich Vodrázka, Roman Barták, and Jiří Švancara [30%]. On modelling multi-agent path finding as a classical planning problem. In Daniel Harabor and Mauro Vallati, editors, *Proceedings of the Thirteenth International Symposium on Combinatorial Search, SOCS 2020, Online Conference [Vienna, Austria], 26-28 May 2020*, pages 141–142. AAAI Press, 2020.

Věra Škopková, Roman Barták, and Jiří Švancara [15%]. What does multi-agent path-finding tell us about intelligent intersections. In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence, ICAART 2020, Volume 1, Valletta, Malta, February 22-24, 2020*, pages 250–257, 2020.

2019

Roman Barták, Ivan Krasičenko, and Jiří Švancara [35%]. Multi-agent path finding on ozobots. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6491–6493, 2019.

Roman Barták, Ivan Krasičenko, and Jiří Švancara [35%]. Multi-agent path finding on real robots. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019*, pages 2333–2335, 2019.

Roman Barták, Jiří Švancara [40%], Věra Škopková, David Nohejl, and Ivan Krasičenko. Multi-agent path finding on real robots. *AI Commun.*, 32(3):175–189, 2019.

Roman Barták and Jiří Švancara [50%]. On sat-based approaches for multi-agent path finding with the sum-of-costs objective. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, pages 10–17, 2019.

Jiří Švancara [70%], Marek Vlk, Roni Stern, Dor Atzmon, and Roman Barták. Online multi-agent pathfinding. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 7732–7739, 2019.

Jiří Švancara [80%] and Roman Barták. Combining strengths of optimal multi-agent path finding algorithms. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 1, Prague, Czech Republic, February 19-21, 2019*, pages 226–231, 2019.

2018

Roman Barták, Jiří Švancara [20%], and Marek Vlk. A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 748–756, 2018.

Roman Barták, Jiří Švancara [40%], Věra Škopková, and David Nohejl. Multi-agent path finding on real robots: First experience with ozobots. In *Advances in Artificial Intelligence - IBERAMIA 2018 - 16th Ibero-American Conference on AI, Trujillo, Peru, November 13-16, 2018, Proceedings*, pages 290–301, 2018.

Jiří Švancara [100%]. Bringing multi-agent path finding closer to reality. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 1784–1785, 2018.

Jiří Švancara [100%]. Bringing multi-agent path finding closer to reality. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 5787–5788, 2018.

2017

Roman Barták, Jiří Švancara [20%], and Marek Vlk. Scheduling models for multi-agent path finding. In *Proceedings of the Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA)*, pages 189–200, 2017.

Pavel Surynek, Jiří Švancara [20%], Ariel Felner, and Eli Boyarski. Independence detection in sat-based multi-agent path finding. In *Proceedings of the 31st Annual Conference of The Japanese Society for Artificial Intelligence, JSAI 2017, Nagoya, Japan*. The Japanese Society for Artificial Intelligence, 2017.

Pavel Surynek, Jiří Švancara [20%], Ariel Felner, and Eli Boyarski. Integration of independence detection into sat-based optimal multi-agent path finding - A novel sat-based optimal MAPF solver. In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence, ICAART 2017, Volume 2, Porto, Portugal, February 24-26, 2017*, pages 85–95, 2017.

Pavel Surynek, Jiří Švancara [20%], Ariel Felner, and Eli Boyarski. Variants of independence detection in sat-based optimal multi-agent path finding. In *Agents and Artificial Intelligence - 9th International Conference, ICAART 2017, Porto, Portugal, February 24-26, 2017, Revised Selected Papers*, pages 116–136, 2017.

Jiří Švancara [80%] and Pavel Surynek. New flow-based heuristic for search algorithms solving multi-agent path finding. In *Proceedings of the 9th International Conference on Agents and Artificial Intelligence, ICAART 2017, Volume 2, Porto, Portugal, February 24-26, 2017*, pages 451–458, 2017.

A. Attachments

This thesis includes an attached medium containing source codes and input files to all of the experiments described throughout the text. Following is a brief description of each folder.

- `01Picat_SoC` – Experiments from chapter 3.2. All 4 used models as well as a binary of CBS are present. We do not include the source files to CBS as it is not our contribution.

- `02Hybrid_MAPF` – Experiments form chapter 3.3. A project that combines the reduction-based solver and CBS is present. It is suited for more algorithms to be added to the project. Again, we are including only an executable form of the CBS algorithm.

  A generator that can generate problems described in chapter 3.3 is also included so that the user can change some parameters of the experiment if desired.

- `03Online_MAPF` – Experiments from chapter 4.1. Two projects are included – one using the reduction-based solver, the other using only CBS. For both of these projects, instance generators are included as well.

- `04Weighted_MAPF` – Experiments from chapter 4.2. A model that takes as an input a graph with specified weights (lengths) of edges as well as a maximal capacity of the edges. A generator that can create similar instances to those described in chapter 4.2 is also present should the reader desire to change some of the parameters.

- `05Ozobot_Experiments` – Description of the physical experiments from chapter 5. All of the experiments were measured by hand and the relevant results are described in tables in chapter 5. In the attachment, we provide the description of the maps used in the experiments, the ozocodes that can be uploaded to Ozobots, and solutions that can be opened in MAPF Scenario [52].

- `06Picat_Current` – In addition to the experiments, we also include the current version of the reduction-based MAPF solver written in Picat language. This folder also contains a README file explaining the input and usage of the models.

  There are 3 models provided – `mks` solving the makespan optimal version of parallel motion MAPF, `mks_pebble` solving the makespan optimal version of pebble motion MAPF, and `soc` solving the sum of costs optimal version of parallel motion MAPF. These models are our most developed implementations that are currently available.