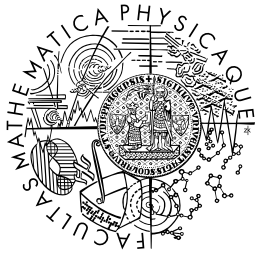

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Samyuktha Ramesh

Reduction-based Solvers for Multi-agent Pathfinding: Comparing Different Models

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Jiří Švancara, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

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 ..... date ....

Author's signature

I would like to express my sincere gratitude to my supervisor RNDr. Jiří Švancara, PhD for his valuable guidance at every step of writing this thesis.

Title: Reduction-based Solvers for Multi-agent Pathfinding: Comparing Different Models

Author: Samyuktha Ramesh

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Švancara, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Multi-agent path finding (MAPF) is the problem of navigating a set of agents from their starting position to their respective goal position without any collisions. In this thesis, we provide an overview of the current approaches to solving MAPF. We implement six different encodings found in the literature using the Python programming language and the Glucose3 SAT solver. We run experiments on maps of different types and sizes to compare the performances of the encodings.

Keywords: multi-agent pathfinding reduction-based solvers SAT makespan

Contents

Introduction **2**

1 Background **4**
1.1 Graphs . . . 4
1.2 Multi-agent Path Finding . . . 4
1.2.1 Agent . . . 5
1.2.2 Types of Conflicts . . . 6
1.2.3 Agent Behaviour at Goal . . . 7
1.2.4 Metrics . . . 8
1.2.5 Complexity . . . 9

2 Approaches to Solving Multi-Agent Path Finding **10**
2.1 Searching a state-space . . . 11
2.2 Constraint-based Search . . . 11
2.3 Reduction to SAT . . . 12
2.3.1 Makespan Optimal Models . . . 13
2.3.2 Sum of Costs Optimal Models . . . 17

3 Methodology **20**
3.1 Implementation . . . 20
3.2 Optimisations . . . 20
3.3 Instances . . . 21

4 Experiments and Results **23**
4.1 Experiments . . . 23
4.2 Results . . . 24
4.3 Limitations . . . 28

5 Conclusion **30**
5.1 Future Work . . . 30

Bibliography **32**

List of Figures **37**

List of Tables **38**

A User Documentation **39**
A.1 Introduction . . . 39
A.2 System Requirements . . . 39
A.3 Getting Started . . . 39
A.4 Arguments . . . 39
A.5 Using the Program . . . 40
A.6 Instances . . . 40

B Attachments **41**

Introduction

Multi-agent path finding (MAPF) is the problem of navigating a set of agents from their starting position to their respective goal position without any collisions. The field of MAPF has seen extensive research due to its applications in numerous real-world scenarios. For example, warehouses and traffic junctions have multiple agents moving along shared paths (Barták and Svancara [2019]). Some of the areas that it can be applied to are autonomous traffic (Gebser et al. [2018]), railway planning (Li et al. [2021]), aviation (Ho et al. [2022], as well as in the video game industry (Botea and Surynek [2015])

Algorithms for finding collision-free paths for multiple agents can be categorized in several ways, but two common ways of categorizing them are by optimality and by the technique used. Optimality refers to the quality of the solution produced by the algorithm. The choice of technique is dependent on the type of problem and its constraints, and they each have their strengths and weaknesses.

Based on optimality, MAPF algorithms can be classified as either optimal or sub-optimal. Optimal solutions guarantee the best possible solution to a given MAPF problem. On the other hand, sub-optimal solutions may prioritize efficiency over optimality. A sub-optimal solution may be useful in situations where finding an optimal solution may be computationally expensive, but a solution is still required.

Based on the technique, MAPF algorithms can be broadly classified as either search-based or reduction-based. Search-based solutions search a graph that represents an MAPF problem and looks for a solution that satisfies all the constraints. Reduction-based solutions convert the MAPF problem into a different problem, such as a constraint satisfaction problem or a Boolean satisfiability problem (SAT), and then try to solve it. Examples of search-based solvers include the A* algorithm and tree search algorithm while examples of reduction-based solvers include constraint satisfaction problems, SAT solvers, inductive logic programming and answer set programming Surynek et al. [2016]. Here, we focus on reduction-based solvers, specifically SAT solvers.

Two of the commonly used objective functions used to measure the optimality of an MAPF solution are makespan and the sum of costs. In this thesis, we will be using the makespan metric to evaluate the solutions.

The reduction from MAPF to a SAT problem involves defining the problem variables and propositional logic formulas. By varying the number and type of variables and propositional rules, we can derive different methods of performing this reduction. In this thesis, we provide an overview of the current research in MAPF, specifically the different methods to reduce an MAPF problem to a SAT problem. We also contribute a comparison of the performances of these methods. We refer to the different methods as encodings. The objective function used in this thesis is the makespan. We conduct experiments on various types and

sizes of maps. The experiments are designed to measure several key parameters, including makespan, time to build clauses, time to solve the SAT formula, and total runtime.

This thesis is structured into four chapters. In Chapter 1, we provide a theoretical background on MAPF. In Chapter 2, we discuss the different approaches to solving MAPF, including search-based and reduction-based methods, as well as various encodings proposed in the literature. Chapter 3 details the methodology used in this project, including the implementation of the different encodings that were discussed in Chapter 2. Finally, in Chapter 4, we present our experimental setup, the selection of benchmark problems and the computational environment setup. We also present the experimental results obtained from running the implemented encodings on various maps and scenarios.

1. Background

In this chapter, we define and provide an overview of some of the concepts that are used throughout this thesis, including graphs and their use in representing the problem space, agents and their possible actions, the different types of conflicts that can arise in MAPF scenarios, and the behaviour of agents when they reach their goal. We also discuss various metrics that can be used to evaluate the performance of an MAPF solution.

1.1 Graphs

A graph $G = (V, E)$ consists of a set V of vertices and a set E of edges. The set of edges E can be defined as the set of two element subsets of V, that is, $E \subseteq V^2$. Two distinct vertices u and v are adjacent if there exists an edge $e \in E(G)$ such that both u and v are endpoints of e. The neighbourhood of a vertex v is the set of all vertices that are adjacent to v.

A graph $G = (V, E)$ is said to be directed if the edges have a direction and are represented by an ordered pair (u, v). An undirected graph $G = (V, E)$ is a graph where the edges are represented by an unordered pair $\{u, v\}$.

In this thesis, we will be using 2D grid graphs where the individual cells are the vertices. Each vertex has four neighbours: the four cardinal directions (north, south, east, west). Since we require agents to be able to wait at a vertex (wait actions will be explained later in this chapter), each vertex has a self-loop, that is, an edge to itself. For simplicity, we consider a vertex a neighbour of itself.

1.2 Multi-agent Path Finding

Most of the definitions in this section are from Stern et al. [2019]. We first explain the general term 'path finding'. Pathfinding is the process of finding a path between two vertices. Multi-agent pathfinding (MAPF) deals with the problem of navigating a set of agents from their initial position to their goal position without any collisions.

We can define an MAPF problem with k agents as a tuple $< G, s, g, A >$ where $G = (V, E)$ is a graph that can be either directed or undirected. Here, all our graphs are undirected so the agents can move both forwards and backward. The set of all agents is denoted by A. The injective functions $s : [1, ..., k] \rightarrow V$ and $g : [1, ..k] \rightarrow V$ are mappings from an agent to a source vertex and an agent to a target vertex respectively. In this thesis, we will refer to the source as the "start" and to the target as the "goal". Time is discretized and is measured in time steps.

Figure 1.1 shows an example map where the black squares are obstacles, that is, positions where the agent cannot be located. The circles represent the start positions of the agents and the triangles represent their goal positions.

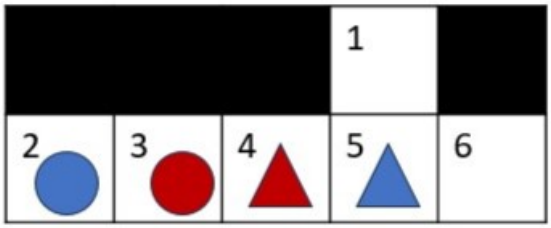

Figure 1.1: Example of an MAPF map where each cell is a vertex

1.2.1 Agent

Every agent has a start position and a goal position. Here, these positions are assumed to be unique and therefore no two agents start at the same position nor do they aim for the same goal position. Further, all agents are present on the map at the start (timestep 0).

At every time step, the agent is present at a vertex and can perform one *action*. An action is a function $action : V \to V$ such that $action(u) \to v$ means that if an agent at vertex u performs that action, then in the next time step it will be at vertex v. An agent can perform any of the two following actions:

1. *move*: an agent can move from its current vertex to a neighbouring vertex. It is of the form $action(u) \to v$, where $(u, v) \in E$.

2. *wait*: an agent can stay at its current vertex. It is of the form $action(v) \to v$, where $v \in V$.

For a sequence of actions $\pi_i(action_1, ..., action_n)$ and an agent i, the location of the agent after the first x actions in π_i, starting from the agent's start position $s(i)$ is denoted by $\pi_i[x]$. A sequence of actions π_i is a *single agent plan* for an agent if and only if executing all the actions in π_i results in the agent being at its goal position. We denote the length of the plan as $|\pi_i|$. A *solution* to MAPF is a set of k single agent plans, each corresponding to a unique agent. An example solution can be seen in the table below.

Timestep	0	1	2	3	4	5
Blue Agent	2	3	4	5	1	5
Red Agent	3	4	5	6	5	4

Table 1.1: Example solution to the MAPF instance in Figure 1.1. The numbers correspond to the vertex each agent occupies at that timestep.

We assume that it takes the same amount of time for each agent to traverse each edge. This can be inferred as all edges having the same unit length and all agents having the same speed.

1.2.2 Types of Conflicts

An MAPF solution is *valid* if and only if there is no conflict between any two single-agent paths. Listed below are some common conflicts. Let π_i and π_j be any two single-agent plans.

- **Vertex Conflict**: A vertex conflict is said to be present between two plans π_i and π_j if and only if there exists a time step x such that $\pi_i[x] = \pi_j[x]$. We can state this alternatively as this conflict is present if the two agents i and j are planned to be at the same vertex at the same time step.
- **Edge Conflict**: An edge conflict is said to be present between two plans π_i and π_j if and only if there exists a time step x such that $\pi_i[x] = \pi_j[x]$ and $\pi_i[x+1] = \pi_j[x+1]$. We can state this alternatively as being present if the two agents i and j are planned to traverse the same edge, in the same direction, at the same time step.
- **Swapping Conflict**: A swapping conflict is said to be present between two plans π_i and π_j if and only if there exists a time step x such that $\pi_i[x] = \pi_j[x+1]$ and $\pi_i[x+1] = \pi_j[x]$. We can state this alternatively as being present if the two agents are planned to traverse the same edge, in opposite directions, at the same time step i.e. they swap position.
- **Following Conflict**: A following conflict is said to be present between two plans π_i and π_j if and only if there exists a time step x such that $\pi_i[x+1] = \pi_j[x]$. We can state this alternatively as being present when one agent plans to occupy a vertex that was occupied by another agent in the previous time step.
- **Cycle Conflict**: A cycle conflict is said to be present between a set of plans $\pi_i, \pi_{i+1}, ..., \pi_j$ if and only if there exists a time step x such that $\pi_i[x+1] = \pi_{i+1}[x]$ and $\pi_{i+1}[x+1] = \pi_{i+2}[x]$ and ... and $\pi_{j-1}[x+1] = \pi_j[x]$ and $\pi_j[x+1] = \pi_i[x]$. We can state this alternatively as being present if and only if every agent moves into a vertex occupied by a different agent in the previous time step, thus forming a cycle.

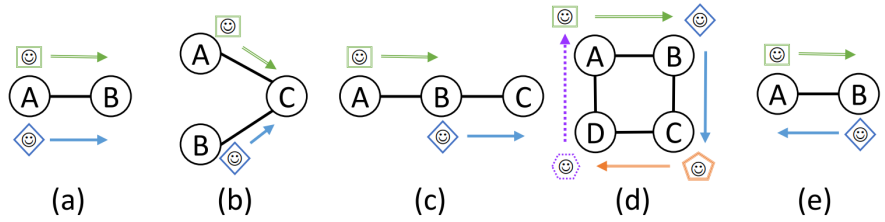

Figure 1.2: Types of MAPF Conflicts (a) *edge conflict* (b) *vertex conflict* (c) *following conflict* (d) *cycle conflict* (e) *swapping conflict*. Figure taken from Stern et al. [2019]

From the definitions, we can see that some of the conflicts are related to each other:

1. forbidding vertex conflicts implies that edge conflicts are automatically forbidden

2. forbidding following conflicts implies that both cycle conflicts and swapping conflicts are forbidden

3. forbidding cycle conflicts implies that swapping conflicts are also forbidden (a swapping conflict is a specific type of cycle conflict where the cycle is of size 2)

These forward implications don't necessarily mean the revere implications are also true. This list is not exhaustive and there are other possible conflicts that can occur. Further, not all of the defined conflicts need to be forbidden in an MAPF problem. Conflicts can be allowed or forbidden depending on the situation. Most real-life situations involving physical agents such as warehouses or cars at the very least disallow vertex, edge and swapping conflicts.

In this thesis, we will forbid vertex and swapping conflicts (therefore also forbidding edge conflicts). We will allow following and cycle conflicts. This setting is referred to as *parallel motion* (Svancara et al. [2022]). Another commonly used setting is called *pebble motion* (Kornhauser et al. [1984]). Pebble motion differs from parallel motion in that it not only forbids all the conflicts that are forbidden in parallel motion but also specifically forbids the following conflict (and by extension forbids cycle conflicts).

1.2.3 Agent Behaviour at Goal

It is necessary for us to define the agent's behaviour after it has reached its goal position. Each agent will reach its goal position at different time steps and therefore, its behaviour at the goal position will influence the other agents' plans. There are two common assumptions for the types of behaviour exhibited:

- **Disappear at goal**: Once the agent reaches its goal position, it disappears. This implies that the number of agents present on the map decreases over time. This is usually the case when the agents have some physical location that they can occupy, for example, a parking space (Svancara et al. [2019]). Alternatively, if agents are assigned to specific tasks, they can be removed from the map once the task is completed. Further, once an agent has disappeared, its plan cannot conflict with any other agent's plan.

- **Stay at goal**: If the agent is not set to disappear at its goal position, it has two further options:

  - Once an agent reaches its goal position, it stays at that vertex until all other agents have reached their goal positions. This implies that no other agent can pass through that vertex as it will result in a vertex conflict. Formally, under this assumption, plans π_i and π_j will have a vertex conflict if there exists a time step $t > |\pi_i|$ such that $\pi_i[|\pi_i|] = \pi_j[t]$.

– Once an agent reaches its goal position, it may move away from its goal temporarily to make way for another agent. This is the setting that will be used in this thesis.

1.2.4 Metrics

The quality of MAPF solutions can be evaluated by an objective function. Two common metrics often used in MAPF are *Makespan* and *Sum of Costs*. Makespan is the distance between the time the first agent leaves its start position to the time when the last agent has reached its goal position. The sum of costs is the sum of the lengths of all the plans. Let us denote the last time step at which agent a_i reaches its goal position as T_i. Then, for a plan π for a set of agents A, we can formally define the makespan metric as $Mks(\pi) = \max_{a_i \in A} T_i$ and the Sum of Costs metric as $SoC(\pi) = \sum_{a_i \in A} T_i$.

It is important to define how the agent's behaviour at the goal position will affect the objective function. If the agent's behaviour is to stay at the goal position but can temporarily move away to make way for other agents, and the chosen objective function is the sum of costs, then one needs to specify how this would impact the sum of costs. For instance, it may be defined that once an agent reaches its goal position for the first time, it no longer incurs additional costs for subsequent returns to that goal. This feature can be particularly useful in delivery scenarios, where only the initial delivery to the goal matters. Alternatively, if the agents have limited fuel resources that are used during move actions, it may be defined that all movements, regardless of whether or not a goal is reached, contribute to the sum of costs. This type of cost function is often referred to as the "sum of fuel" in the literature.

Optimizing makespan and the sum of costs objective functions yields different plans. This can be seen in Figure 1.3 where the plan on the top minimizes the sum of costs while the plan on the bottom minimizes makespan. Further, optimizing the makespan might increase the sum of costs and vice versa.

The choice of which objective function to optimize depends on the situation being modelled. Makespan may be optimized in situations where the total time may need to be minimal. This, however, can lead to unnecessary movements by some agents. On the other hand, optimizing the sum of costs can be useful in situations where agent actions are expensive and may need to be minimal. An example of this is warehousing where agents use fuel for their actions.

We also note that for any solution S with makespan μ and sum of costs ε, we have that $\mu \leq \varepsilon$. In other words, if there is only one single-agent plan, then the makespan equals the sum of costs. Otherwise, it is always smaller than the sum of costs.

In this thesis, the plans utilize the makespan metric. Once the agent arrives at its goal position, it may either wait there for the other agents to also reach their goal, or it may move to make way for other agents and return to its goal at

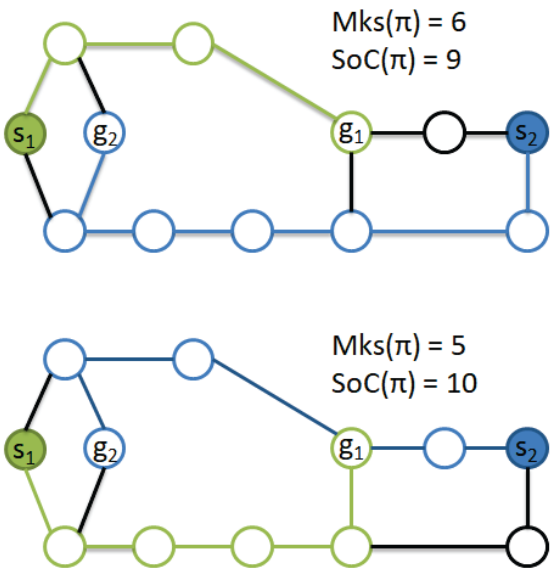

Figure 1.3: Sum of Costs vs Makespan. Figure taken from Barták and Svancara [2019]

a later time step. This means the length of all the plans $|\pi_i|$ is the same.

1.2.5 Complexity

Many sub-optimal polynomial-time algorithms can be used to solve MAPF. A brief overview of these will be given in the next chapter. Deciding if an MAPF problem is solvable in a given time in NP-Complete since it is a generalisation of the sliding tile puzzle which is NP-Complete Sharon et al. [2012]). Therefore, finding the optimal solution, either with respect to the makespan Surynek [2010] or the sum of costs Yu and LaValle [2013], is NP-Hard.

2. Approaches to Solving Multi-Agent Path Finding

In this chapter, we discuss the different approaches that are currently used to solve an MAPF problem. We then provide a brief overview of optimal MAPF solvers before providing a more detailed section on solving MAPF through reduction.

Due to the computational complexity of finding optimal solutions for the multi-agent path finding (MAPF) problem, which is NP-hard, one possible approach is to instead aim for sub-optimal solutions. This means that instead of minimizing the makespan or the sum of costs, we prioritize finding a feasible solution that meets the constraints of the problem. Another possible approach is to aim for an optimal solution. In an optimal solution, we aim to minimize the makespan or the sum of costs while still satisfying all of the constraints of the problem. Figure 2.1 shows an example illustrating the difference between a sub-optimal MAPF solution and an optimal MAPF solution.

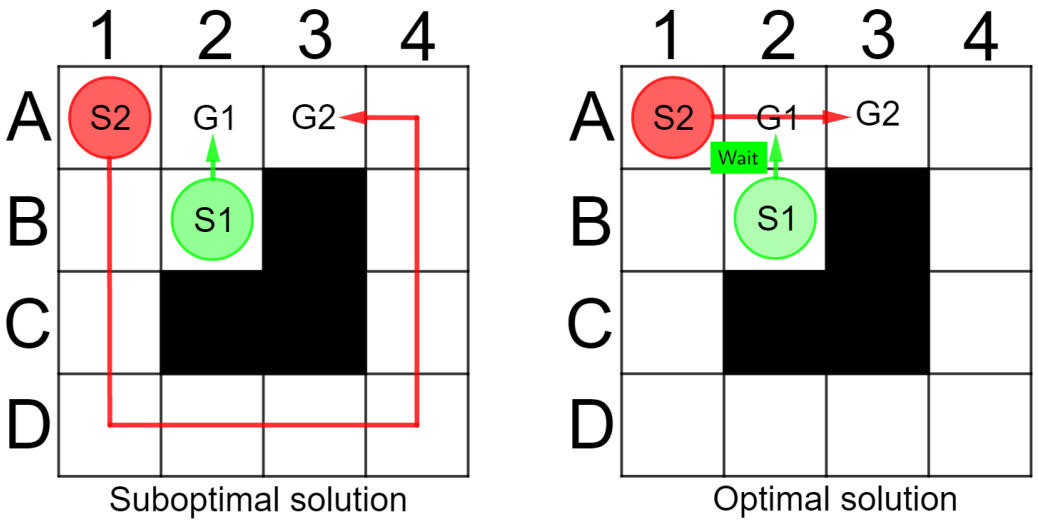

Figure 2.1: An example of a sub-optimal vs optimal MAPF solution (Ivanashev et al. [2022])

Sub-Optimal Solvers. In a sub-optimal solution approach, we create a plan for each agent without considering the other agents. If a conflict is found during execution, we compute the plans again. From this, we can see where the sub-optimality comes from. Some sub-optimal solvers prioritize quickly finding a solution, as opposed to ensuring it is as close as possible to the optimal solution. These solvers are useful when the number of agents is high, but may not be complete. Such algorithms are referred to as *any-solution* MAPF solvers, several of which have been proposed by Ryan [2010], Cohen et al. [2015], Silver [2005], Botea and Surynek [2015]and Sajid et al. [2012], including a polynomial time any-solution MAPF solver by Luna and Bekris [2011] and de Wilde et al.

[2014]. Another type of sub-optimal solution is what is known as *bounded sub-optimal*. Solvers producing bounded sub-optimal solutions provide a guarantee of the quality of the solution: the cost of the solution is $\leq (1 + \epsilon) \times c_{opt}$, where c_{opt} is the cost of the optimal solution and ϵ is a parameter, also called error, that sets the level of sub-optimality that is desired.

Optimal Solvers. We can divide MAPF optimal solvers into three different categories: searching a state-space, constraint-based search, and reduction to another problem. We will first briefly describe algorithms that involve searching a state-space and then delve into the other two types of algorithms.

2.1 Searching a state-space

Algorithms involving searching a state-space are usually based on the A* algorithm. The A* algorithm, often seen as an extension of Dijkstra's algorithm, was first published in the paper 'A formal basis for the heuristic determination of minimum cost paths' by Hart et al. [1968]. However, while a common variant of Dijkstra's algorithm constructs the shortest path tree by finding the shortest paths between the start position to all possible goal positions, the A* algorithm finds the shortest path from the start position to the specified goal position, by using heuristic estimates to guide its search. As described in the paper, heuristics used by the A* algorithm are admissible and consistent. This means that they will never overestimate the actual cost of reaching the goal position and hence guarantee the optimality of A*.

The current positions of all agents are represented by states. The algorithm is given the start state and the goal state, as well as the transition rules for moving an agent from one state to another. All the open states, at each timestep, are stored in a priority queue. At every timestep, the state which promises to yield the best result is chosen to be explored. All possible states from the chosen state are then added to the queue. This process is repeated until the goal state has been reached.

This algorithm yields an optimal solution if the open states are explored in the correct order. However, the disadvantage of this algorithm is its large branching factor. One way to counter this disadvantage is to prevent all agents from moving at once and instead move each agent one by one (Standley [2010]).

2.2 Constraint-based Search

Instead of searching all states, as in the A* algorithm mentioned above, we can instead search over constraints on agents' movements. The most often used algorithm that achieves this is the Conflict Based Search (CBS) algorithm, which is a complete and optimal algorithm (Sharon et al. [2012]). This algorithm was introduced with the purpose of reducing the state-space over which the A* algorithm searches. The CBS algorithm finds single-agent paths and performs an

exploration of the conflicts caused by these paths.

CBS is a two-level algorithm: it has a higher level and a lower level. The higher-level search is performed over a binary constraint tree. The nodes of the tree are constraints that are in the format $Cons(a_i, v, t)$, which means that agent i cannot be at position v at time t. At best-first search is performed and at each timestep, a node with the best cost is selected and explored. It checks that no conflicts are present in the single-agent paths, and if this is true, then the solution is deemed valid and optimal. However, if a conflict is found, two child nodes are created and are subsequently explored. The lower-level searches for a single-agent optimal path that follows the constraints for all of the agents. This can be found through any shortest pathfinding algorithm such as the A* algorithm or Dijkstra's algorithm.

Two improvements to the CBS algorithm were proposed by Sharon et al. [2012] and Boyarski et al. [2015a]. The first one, called Meta-agent CBS merges small groups of agents into meta-agents when it is favourable. This proved to reduce the runtime. The second was an improvement on the Meta-agent CBS, called the Bypass improvement. Instead of arbitrarily choosing paths, as in Meta-agent CBS, the Bypass improvement tries to find an alternative path for an agent that has a conflict. This proved to have a better runtime compared to the previous version. A paper by Boyarski et al. [2015b] offered a further improvement on CBS, called the Improved Conflict-Based Search (ICBS), that added to the two previous improvements.

Another such algorithm is the Increasing Cost Tree Search (ICTS). The ICTS algorithm runs on two levels, where the higher level performs a search over a search tree called an increasing cost tree, and the lower level performs a goal test on each of the nodes of the tree (Sharon et al. [2013]). Each node in the increasing cost tree is a k-vector $[C_1, C_2, ..., C_k]$. This k-vector represents all the possible solutions to the MAPF problem where for each agent a_i, the cost of its individual path is exactly equal to C_i. Experimental results show that in many cases, the ICTS algorithm performs better than the A* algorithm by up to three orders of magnitude (Sharon et al. [2013]).

2.3 Reduction to SAT

In the reduction to SAT approach, we construct a propositional formula $\varphi(T)$ such that it is satisfiable if and only if there exists a solution to the given MAPF problem with makespan T. $\varphi(T)$ is a complete propositional model of MAPF. Further, $\varphi(T)$ exactly represents the MAPF instance, and if it is satisfiable, the solution can be reconstructed from a satisfying assignment of the formula.

We can define the formula formally as follows: Propositional formula $\varphi(T)$ is a complete propositional model of an MAPF problem if the following condition holds: $\varphi(T)$ is satisfiable, if and only if there exists a solution with makespan T.

2.3.1 Makespan Optimal Models

Since we don't know the length of the plan in advance, we start with a restriction on the length. In case the algorithm fails to find a solution in the given plan length, we increment the length (Kautz and Selman [1992]). In the context of finding the optimal makespan, we iteratively increase the makespan T until a valid solution is found. This guarantees that no solution with a lower makespan exists and hence gives us the optimal makespan.

Let us suppose that we are looking for a solution to an MAPF problem with makespan T. We also impose restrictions on vertex conflicts, edge conflicts and swapping conflicts. As mentioned in the background chapter, forbidding vertex conflicts implies that edge conflicts are also forbidden. Therefore, we only check for vertex conflicts and swapping conflicts. In this section, we define six different encodings found in the literature relating to solving MAPF problems via reduction to SAT.

We use the idea of a time-expanded graph (TEG). The TEG consists of T copies of the vertices in graph G. Each layer of the TEG corresponds to a specific timestep and indicates the positions of the agents at that time. We add edges (u_i, v_{i+1}), where u is in the i-th layer of the TEG and v is in the i+1-th layer, if and only if there is an edge (u, v) in the original graph G. These edges represent an agent moving from one vertex to another vertex. We also add the edges (u_i, u_{i+1}) for every vertex u. These edges correspond to agents waiting at a vertex. In this graph, the agents start at the 0-th layer and move to the next layer at each timestep.

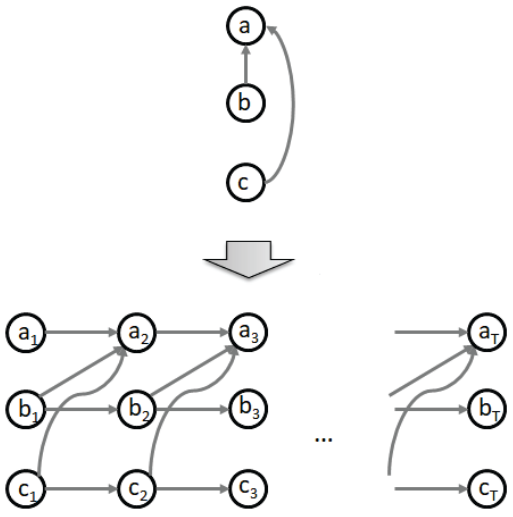

Figure 2.2: Example of a 3-vertex graph being transformed to a T-layered time-expanded graph (Barták and Svancara [2019])

Instead of starting the computation of the optimal makespan with $T = 1$, we can speed up the computation by starting with a different lower bound. For each agent a_i, we find the shortest path P_i from its start position s_i and its goal

position g_i. We can then use the longest of these shortest paths as the lower bound for computing the makespan: $LB(Mks) = max_{a_i \in A}|P_i|$.

We present below six encodings that are found in the current literature. In the encodings that utilize edges of the graph in the variable definition, an auxiliary edge (v, v) is added to E for all vertices $v \in V$. This is done to enable the agents to wait at a vertex.

Encoding 1

This encoding is inspired by two papers (Barták et al. [2017];Surynek [2021]). The paper by Surynek [2021] does not explicitly forbid swapping conflicts. Instead, it uses pebble motion for the agents. This forbids following conflicts, which implies that swapping conflicts are also prohibited. However, we use parallel motion and therefore, to remain consistent with the other encodings, we forbid swapping conflicts in this version of Encoding 1.

We define the variable At as follows: $\forall t \in \{0, ..., T\}, \forall a \in A, \forall v \in V$, $At(t, a, v)$ means that agent a is at vertex v at timestep t.

In order to reduce to MAPF problem to a SAT problem, we introduce the following constraints:

$$\forall a \in A : At(0, a, s(a)) \tag{2.1}$$

$$\forall a \in A : At(T, a, g(a)) \tag{2.2}$$

$$\forall t \in \{0, ..., T\}, \forall a \in A, \forall u, v \in V : \neg At(t, a, u) \vee \neg At(t, a, v) \tag{2.3}$$

$$\forall t \in \{0, ..., T\}, \forall v \in V, \forall a_i, a_2 \in A : \neg At(t, a_1, v) \vee \neg At(t, a_2, v) \tag{2.4}$$

$$\forall t \in \{0, ..., T-1\}, \forall u \in V, \forall a \in A : At(t, a, u) \implies \bigvee_{(u,v) \in E} At(t+1, a, v) \tag{2.5}$$

$$\forall t \in \{0, ..., T-1\}, \forall (u, v) \in E, \forall a_1, a_2 \in A : \neg At(t, a_1, u) \vee \neg At(t+1, a_1, v) \vee \neg At(t, a_2, v) \vee \neg At(t+1, a_2, u) \tag{2.6}$$

We ensure that all agents are at their start positions $s(a)$ at the time 0 and at their goal positions $g(a)$ at the time T through constraints (2.1) and (2.2). Constraint (2.3) states that no agent can be present at more than one vertex at any timestep and constraint (2.4) defines the vertex conflict. The movements of the agent are explained in constraint (2.5) which states that if an agent is at a certain vertex at a particular timestep, then it is in one of the neighbours in the next timestep. Lastly, in constraint (2.6), we define the swapping conflict.

Encoding 2

This encoding was introduced in a paper by Barták and Svancara [2019].

In Encoding 2, we use a combination of two variables to model the MAPF problem. The first variable is the At variable, as defined in Encoding 1. We define the second variable as follows: $\forall t \in \{0, ..., T-1\}, \forall a \in A, \forall (u, v) \in E$, $Move(t, a, (u, v))$ means that agent a goes through edge (u, v) at timestep t. Note that, the timesteps for this variable are defined between 0 and $T-1$, unlike the timesteps for the At variable. This is because the agents need to be at their goal position at timestep T and not travelling through an edge.

In order to reduce to MAPF problem to a SAT problem, we introduce the following constraints:

$$\forall a \in A : At(0, a, s(a)) \tag{2.7}$$

$$\forall a \in A : At(T, a, g(a)) \tag{2.8}$$

$$\forall t \in \{0, ..., T\}, \forall a \in A, \forall u, v \in V : \neg At(t, a, u) \vee \neg At(t, a, v) \tag{2.9}$$

$$\forall t \in \{0, ..., T\}, \forall v \in V, \forall a_i, a_2 \in A : \neg At(t, a_1, v) \vee \neg At(t, a_2, v) \tag{2.10}$$

$$\forall t \in \{0, ..., T-1\}, \forall (u, v) \in E, \forall a_1, a_2 \in A : \neg Move(t, a_1, (u, v)) \vee \neg Move(t, a_2, (v, u)) \tag{2.11}$$

$$\forall t \in \{0, ..., T-1\}, \forall (u, v) \in E, \forall a \in A : Move(t, a, (u, v)) \implies At(t+1, a, v) \tag{2.12}$$

$$\forall t \in \{0, ..., T-1\}, \forall u \in V, \forall a \in A : At(t, a, u) \implies \bigvee_{(u,v) \in E} Move(t+1, a, (u, v)) \tag{2.13}$$

We ensure that all agents are at their start positions $s(a)$ at the time 0 and at their goal positions $g(a)$ at the time T through constraints (2.7) and (2.8). Constraint (2.9) states that no agent can be present at more than one vertex at any timestep and constraint (2.10) defines the vertex conflict. In (2.11) we define the swapping conflict. Lastly, we explain the movements of the agents. Constraint (2.12) states that if an agent moves through an edge (u, v) in timestep t, it will be present at vertex v in the next timestep. Constraint (2.13) states that if an agent is present at a vertex at a particular timestep, then it moves via one of the vertex's outgoing edges in the next timestep. This includes the self-loop, i.e. the agent can wait at that vertex.

Encoding 3

This encoding was introduced in a paper by Achá et al. [2021].

In Encoding 3, we use a combination of the At variable, as defined in Encoding 1, and the $Shift$ variable. We define the $Shift$ variable as follows: $\forall t \in \{0, ..., T-1\}, \forall (u, v) \in E$, $Shift(t, (u, v))$ means that agent a is goes through edge (u, v) at timestep t. Like in Encoding 2, we note that the timesteps are between 0 and $T - 1$, and we add the auxiliary edge (v, v) for all vertices v to E so that self-loops are included in the list of edges.

In order to reduce to MAPF problem to a SAT problem, we introduce the following constraints:

$$\forall a \in A : At(0, a, s(a)) \tag{2.14}$$

$$\forall a \in A : At(T, a, g(a)) \tag{2.15}$$

$$\forall t \in \{0, ..., T\}, \forall a \in A \forall u, v \in V : \neg At(t, a, u) \vee \neg At(t, a, v) \tag{2.16}$$

$$\forall t \in \{0, ..., T\}, \forall v \in V, \forall a_i, a_2 \in A : \neg At(t, a_1, v) \vee \neg At(t, a_2, v) \tag{2.17}$$

$$\forall t \in \{0, ..., T-1\}, \forall (u, v) \in E : \neg Shift(t, (u, v)) \vee \neg Shift(t, (v, u)) \tag{2.18}$$

$$\forall t \in \{0, ..., T-1\}, \forall (u, v) \in E, \forall a \in A : At(t, a, u) \wedge Shift(t, (u, v)) \implies At(t+1, a, v) \tag{2.19}$$

$$\forall t \in \{0, ..., T-1\}, \forall (u, v) \in E, \forall a \in A : At(t, a, u) \wedge At(t+1, a, v) \implies Shift(t, (u, v)) \tag{2.20}$$

$$\forall t \in \{0, ..., T-1\}, \forall v \text{ such that } (u, v) \in E, \forall a \in A : At(t+1, a, v) \implies \sum_{u:(u,v)\in E} At(t, a, u) \tag{2.21}$$

We ensure that all agents are at their start positions $s(a)$ at the time 0 and at their goal positions $g(a)$ at the time T through constraints (2.14) and (2.15). Constraint (2.16) states that no agent can be present at more than one vertex at any timestep and constraint (2.17) defines the vertex conflict. In constraint (2.18), we define the swapping conflict. Lastly, we define the agents' movements.

Constraint(2.19) can also be referred to as *positive effects axiom* (Reiter [2001]). This states that if an agent a is at a vertex u at timestep t, and a shift occurs at the same timestep from u to v, then agent a is at vertex v at the next timestep $t+1$. Constraint (2.20) states that if an agent a is present at vertex u at timestep t and at vertex v at timestep $t+1$, then such a change can be explained by a shift at time t between u and v. Constraint (2.21) states that if an agent a is at a vertex u at timestep $t+1$, then in the previous time t, it must have been at a predecessor of u.

Encodings 4, 5 and 6

In the previous three encodings, we included constraints to forbid all conflicts at once initially. For the next three encodings, namely encoding 4, encoding 5, and encoding 6, a slightly different approach is taken. The idea behind the encodings in this subsection is taken from Surynek [2019].

In these encodings, we do not forbid any constraints at the beginning. Instead, we incrementally extend the propositional model when new conflicts are detected. The hypothesis of this idea is that in many scenarios, we do not need to add all constraints to form a propositional model and still obtain a conflict-free solution. Examples of these scenarios would be large sparse environments with few agents.

Encoding 4, 5 and 6 use this approach while using the base of encoding 1, 2 and 3 respectively. Certain constraints are present from the beginning of the program. These include ensuring that all agents are at their start position at timestep 0 (2.1), and all agents are at their goal position at timestep T (2.2). Further, an agent cannot be at two vertices at the same timestep (2.3) and can only move along edges present in the map (2.5). Constraints that forbid the vertex conflict (2.4) and the swapping conflict (2.6) are added when necessary.

The benefit of such an approach is saving time on sparsely populated maps as building and solving an incomplete propositional model takes less time. We define an incomplete propositional formula as follows. Let $\psi(T)$ be an incomplete propositional formula of an MAPF problem. Then, $\psi(T)$ is satisfiable $\Leftarrow \exists$ solution with makespan T.

See Algorithm 1 for a brief description of Encoding 4. Encoding 5 and Encoding 6 are similar to this, but are based on variable combinations defined in Encoding 2, and Encoding 3, respectively.

2.3.2 Sum of Costs Optimal Models

In the case of makespan optimal models, once we find the correct number of layers in the time-expanded graph, we are guaranteed that it is the makespan optimal solution. However, this guarantee does not extend to the sum of costs optimal models. A good example of this can be seen in Figure 1.3 from the Background chapter. We can see that when the Mks(Π) = 5, we don't have the optimal SOC. However, when we add another layer to the TEG, we get the optimal sum of costs

Algorithm 1 Encoding 4

```
function ENCODING 4
    T = lowerbound
    conflicts_dict ← {vertex_conflict : False, swapping_conflict : False}
    while True do
        At ← createDict()
        addConstraint(all agents at start position at timestep 0)
        addConstraint(all agents at goal position at timestep T)
        addConstraint(agents can't be at two different vertices at same timestep)
        addConstraint(agents moves to vertex neighbour in next timestep)
        if conflicts_dict[vertex_conflict] then
            addConstraint(forbid vertex conflict)
        end if
        if conflicts_dict[swapping_conflict] then
            addConstraint(forbid swapping conflict)
        end if
        solvable, res ← trySolve()
        if solvable then
            if no conflicts then
                results ← res
                return results
            else
                conflicts_dict[vertex_conflict] = True
                conflicts_dict[swapping_conflict] = True
                continue
            end if
        end if
        T+ = 1
    end while
end function
```

SOC(Π) = 9. It is unclear how many additional TEG layers are required for the solution to be guaranteed to be optimal.

The following two models take this into consideration. To address this issue, we add some additional constraints to the original constraints that were discussed in the previous subsection.

The first model (Surynek et al. [2016]) finds a sum of costs optimal solution by adding a constraint to bound both the makespan and the sum of costs. The bounds are set to ensure that the first solvable formula yields the sum of costs optimal solution. The model finds the shortest path for each agent. All the other agents are ignored while this is being done. A valid lower bound for the makespan is the maximum of the lengths of the paths and a valid lower bound for the sum of costs is the sum of the lengths of all the paths. The algorithm then incrementally allows extra movement for the agents. It starts with 0 extra movements and tries to solve the problem using the chosen lower bounds. If a solution exists, then we know that it must be a sum of costs optimal solution since the values chosen

for both the makespan and the sum of costs were the lower bounds. The extra movement adds a layer to the TEG and allows some agent to take one extra step.

The second model starts in a similar way to the first model. It finds the shortest path for each agent while ignoring all of the other agents on the map, and sets the lower bounds for the makespan and the sum of costs. It then finds a solution with an optimal makespan. The model sets the upper bound as the sum of costs of this makespan-optimal solution. In this model (Barták and Svancara [2019]), instead of iteratively increasing the extra allowed movement one by one, it uses the lower bound and upper bound of the sum of costs to find an optimal sum of costs.

3. Methodology

In this chapter, we describe the methodology used to implement the six encodings from the previous chapter. First, we discuss the implementation process for the algorithms, including the tools and data structures used. Then we state the preprocessing done on the data, to optimise the data that is available to the SAT solver. Next, we provide a detailed overview of the instances that were used for testing and to conduct the experiments which will be highlighted in the next chapter.

3.1 Implementation

We implemented each of the encodings described in the previous chapter using the Python language. Python is an extremely popular programming language due to its easy-to-read syntax and extensive library support. In the context of MAPF, the language's efficient data structures provide easy implementation of algorithms. In our implementation, we construct CNF clauses for each encoding and then use PySAT Glucose3 (Ignatiev et al. [2018]). PySAT Glucose3 is a Python library that uses the Glucose3 SAT solver to solve propositional satisfiability problems.

Python has four collection data types: lists, sets, tuples and dictionaries. Python dictionaries are useful data structures that enable efficient storage. Further, it has fast retrieval of key-value pairs with a time complexity of $O(1)$ (Cormen et al. [2009]). This is useful in solving complex problems efficiently. In the context of using the Glucose3 SAT solver, we store the variables as the dictionary keys and their values are the respective integer values that can then be accepted by the solver. This way, we can work with the variable names, instead of the integer values, making the code more readable.

3.2 Optimisations

One of the ways that the computation has been enhanced is through the notion of "feasible" variables. In the case of the At variable, $At(t, a_i, v)$ corresponds to an agent a_i being present at vertex v at timestep t. However, because we know that the agent needs to be at its start position s_i at timestep 0 and at its goal position g_i at timestep T, we can deduce the agent cannot be present at certain positions at certain timesteps. If a vertex v is at distance d away from start position s_i, we know that the agent a_i cannot be present at v at any timestep from $\{0, ..., d-1\}$. Likewise, if a vertex v is distance d away from goal position g_i, agent a_i cannot be present at v at timesteps $\{T-d+1, ..., T\}$. This is because in the first case, the agent will have insufficient time to reach vertex v from its start position s_i, and in the second case, the agent will have insufficient time to reach its goal position g_i from vertex v. These variables are referred to as unfeasible. If an agent can be present at a specific vertex at a specific time, the variable representing that state is referred to as "feasible".

Similarly, we can determine feasible variables for $Move(t, a, (u, v))$. An agent can move through an edge (u, v) at timestep t if and only if the agent can be present at the vertex u at timestep t and at vertex v at timestep $t + 1$. Formally, $Move(t, a, (u, v))$is feasible if and only if both $At(t, a, u)$ and $At(t + 1, a, v)$ are feasible.

There is no pre-processing for the $Shift$ variable. Since the $Shift$ variable does not refer to any specific agent, the vertex shift occurs irrespective of the presence or absence of an agent there.

3.3 Instances

The instances used to test the encodings are from Moving AI Lab. A more detailed overview of these benchmarks can be found in Stern et al. [2019]. There are multiple maps from real-life cities, video games, open grids both with and without obstacles, maze-like grids, and room-like grids. In Figure 3.1, we can see examples of each type of map that is available.

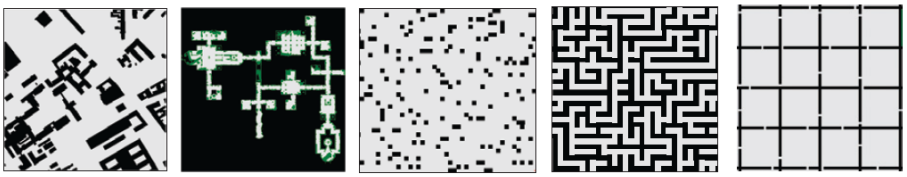

Figure 3.1: Types of maps available at Moving AI Lab

Every map has 25 scenarios, each is a $*.scen$ file. The first line of the scenario file specifies its version number. Each line of the file after that has a single agent and is of the format '* *.map-file x-size-map y-size-map x-start y-start x-goal y-goal *'. The program contains a scenario loader that reads each line of the $*.scen$ file and creates an agent object with a start position and a goal position. The intended way to use these files is, for each map and scenario, to try and solve as many agents as possible within the given time limit, adding them consecutively in order. We first create an instance with one agent by choosing the first agent from the scenario file and solve it with the encoding of choice. If it is solvable within the given time limit, then create an instance with two agents, by choosing the first two agents from the scenario file, and try to solve it. Repeat this until an instance cannot be solved within the given time limit. Note that the time limit is reset when a new instance is created.

Each scenario was created by randomly pairing all points in the largest reachable region of the map and then selecting the first 1000 of these. Therefore, any

subset of these pairs of start and goal positions can be chosen to create a scenario.

From the scenario file, we can get the file path of the map it corresponds to. Each map is represented by a $*.map$ file. Figure 3.2 shows an example of an 8x8 map file. All map files begin with these lines:

type octile
height y
width x
map

where x and y are the respective width and height of the map. The contents of the map are in the form of an ASCII grid where '.' represents a space where an agent can be present. All other symbols represent obstacles. An agent cannot be present at a vertex marked as an obstacle at any given time.

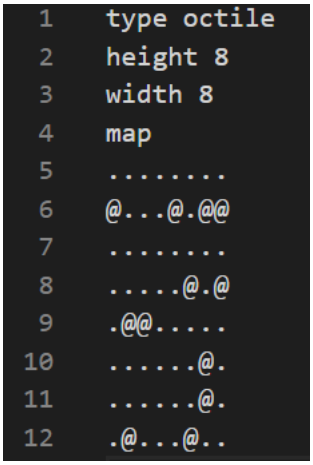

Figure 3.2: Example of an 8x8 map file

The program contains a Maploader which reads each map file and creates a map object with its associated height, width, file path, and the map itself. The map is stored as a 2D array filled with 0s and -1s. The 0s represent spaces an agent can occupy while the -1s represent obstacles.

4. Experiments and Results

In this chapter, we describe the experiments we conducted, along with the metrics that were measured, as well as the challenges encountered and how we addressed them. We present the results obtained from the experiments and an analysis of them.

4.1 Experiments

We conducted experiments to evaluate the performance of the six encodings and evaluate if there was any significant difference between them. The experiments were run on three different types of maps: empty maps, random maps, and room-like maps. Further, to see the effects of size on the performance of the encodings, we chose maps of three different sizes: 8x8, 16x16, and 32x32, resulting in a total of nine maps. Figure 4.1 shows visualisations of some of the maps that were used. Some combinations of size and type were not available in the benchmark set so we created our own maps and scenarios that follow the general structure of the existing maps. For each map, we created three different scenarios to test the performance of the algorithms on each scenario. The scenarios corresponding to the 8x8 maps had thirty-two agents (certain 8x8 could not accommodate more than thirty-two agents), while the 16x16 maps and the 32x32 maps had a hundred agents each.

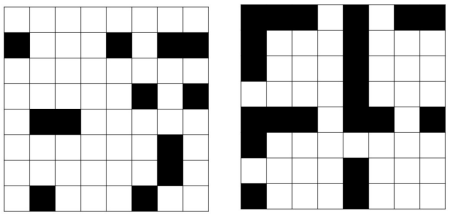

Figure 4.1: (a) 8x8 random map (b) 8x8 room map

The time limit for each instance was sixty seconds. For each scenario, we executed the program using the six different encodings. For each run, we logged the makespan and its lower bound, the time taken to build the clauses, the time taken to solve the SAT problem, and the total run time into a CSV file.

The experiments were conducted on a laptop with an AMD Ryzen 7 PRO processor and 16 GB RAM. Python 3.9.5 was used as the programing language.

Overall, these experiments aimed to provide an evaluation of the MAPF encodings' performance and identify the most effective encoding for different map types and scenarios. The results of these experiments are presented and analyzed in the next section.

4.2 Results

An MAPF instance is considered solved if the algorithm found a collision-free plan for every agent in that instance within the given time limit of sixty seconds. We are interested to see the amount of time each encoding takes to find a solution for a specific map type or size. Tables with the average times are provided below. "Encoding" is abbreviated as "E" in the column names. Further, all data is given to two decimal places.

However, it is important to note that the encodings that did not manage to solve many instances often have lower average runtimes compared to the encodings that solved more instances. Therefore, trends in the data can be more easily viewed in a graphical format. We sorted the instances in ascending order based on their total runtime and plotted them for each encoding. Lower lines on the graph indicate lower runtime for the instances, and therefore the lower the line on the graph, the better the performance of the encoding.

Number of Solved Instances

In Table 4.1, we have the number of solved instances for each map type and for each map size, as well as the total number of solved instances by each encoding.

	E1	E2	E3	E4	E5	E6
8x8	267	263	**270**	269	263	**270**
16x16	471	352	**545**	426	272	462
32x32	95	36	**107**	61	30	74
Empty	284	225	**316**	262	187	273
Random	290	227	**312**	260	202	266
Room	259	199	**294**	234	176	267
Total Number of Solved Instances	833	651	**922**	756	565	806

Table 4.1: Number of solved instances by each encoding on different sizes and types of maps.

Encoding 3 performed better than the rest of the encodings of every size and type of map. It solved a total of 966 instances while Encoding 5 solved 565 instances which was the least number of instances. Encoding 3 and Encoding 1 significantly outperformed Encoding 2. A similar trend can be seen among Encoding 4, 5, and 6, where Encoding 4 and Encoding 6 outperformed Encoding 5. This is not surprising given that Encoding 5 is modelled with the same variables as Encoding 2.

However, what is surprising is that the first three encodings performed better compared to their counterparts where the constraints were added ad hoc. We conjecture that this is because while the first few instances with a small number of agents were solvable without encountering any conflicts, the vast majority of instances required the conflict clauses to be added. This resulted in an extra iteration of clause-building for each instance.

Average clause-building Time

	E1	E2	E3	E4	E5	E6
8x8	0.83	0.94	**0.34**	0.94	1.26	0.43
16x16	16.25	19.81	**9.52**	19.16	20.80	11.45
32x32	23.18	27.82	27.08	21.54	25.52	**20.62**
Empty	11.80	13.51	**8.48**	13.29	12.28	9.06
Random	11.75	11.82	7.83	12.56	11.30	**7.38**
Room	12.82	12.56	10.39	12.73	12.36	**9.34**
Average clause-building Time	12.10	12.63	8.87	12.867	11.95	**8.60**

Table 4.2: Average time (seconds) to build clauses for each encoding on different sizes and types of maps.

Table 4.2 shows the average time that each instance took to build the clauses that model the required constraints. Encoding 3 performed better than Encoding 1 and Encoding 2, both of which took similar amounts of time to build their clauses. A similar trend is noticeable among Encoding 4, Encoding 5, and Encoding 6.

Encoding 3 and Encoding 6 have similar performances for most maps, although a significant difference is present for 32x32 maps where Encoding 6 took lesser time to build the clauses. This might be due to the fact that, compared to smaller maps, larger maps with fewer agents tend to have fewer conflicts. Therefore, adding conflict clauses ad hoc saves clause-building time for more instances on the 32x32 maps compared to the other map sizes.

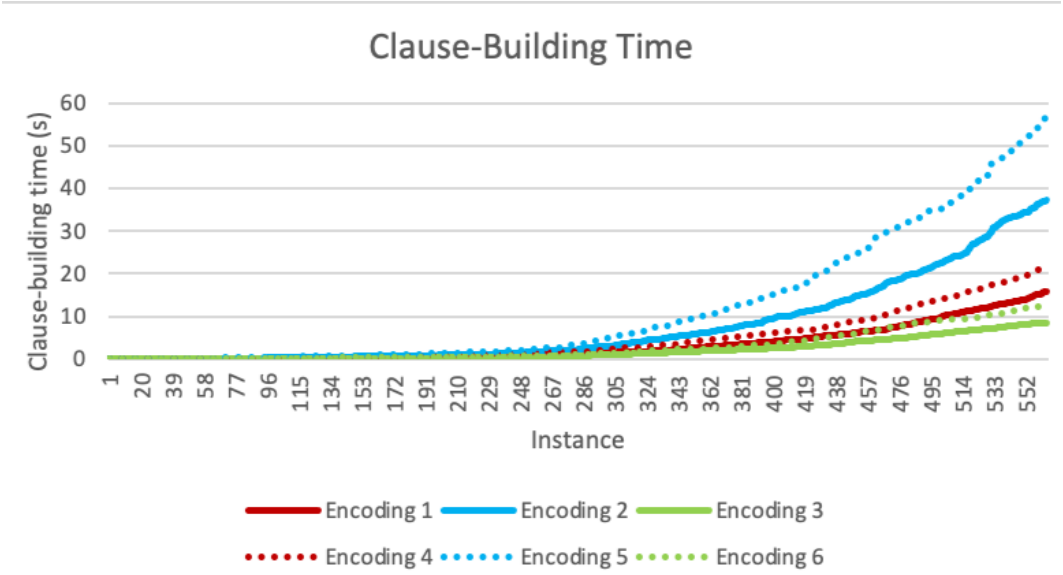

Figure 4.2: Average time to build clauses.

This trend is confirmed by the graph 4.2 where the solid green line (encoding 3) is the lowest, and therefore has the best performance, and the dotted blue line (Encoding 5) is the highest and therefore has the worst performance.

Average Solving Time

	E1	E2	E3	E4	E5	E6
8x8	1.04	**0.66**	1.30	1.23	0.71	1.20
16x16	1.36	1.27	10.11	2.01	**1.01**	9.30
32x32	0.33	0.075	1.54	0.15	**0.07**	0.66
Empty	0.65	0.57	8.03	0.97	**0.43**	5.80
Random	0.93	0.78	6.66	1.33	**0.64**	5.54
Room	1.91	1.61	4.79	2.56	**1.45**	6.05
Average Solving Time	1.14	0.96	6.53	1.58	**0.82**	5.80

Table 4.3: Average time (seconds) for the SAT solver to try to find a satisfying assignment for each encoding on different sizes and types of maps.

Table 4.3 shows the average time that the SAT solver took to find a satisfying assignment for the MAPF problem for each instance. With this metric, Encoding 1 and Encoding 2 performed significantly better than Encoding 3. Likewise, Encoding 4 and Encoding 5 performed significantly better than Encoding 6. Overall Encoding 5 had the best performance. However, this does not mean that Encoding 5 had the best performance. These results can be attributed to the fact that Encoding 5 solved the least number of instances and this positively impacted the average solving time.

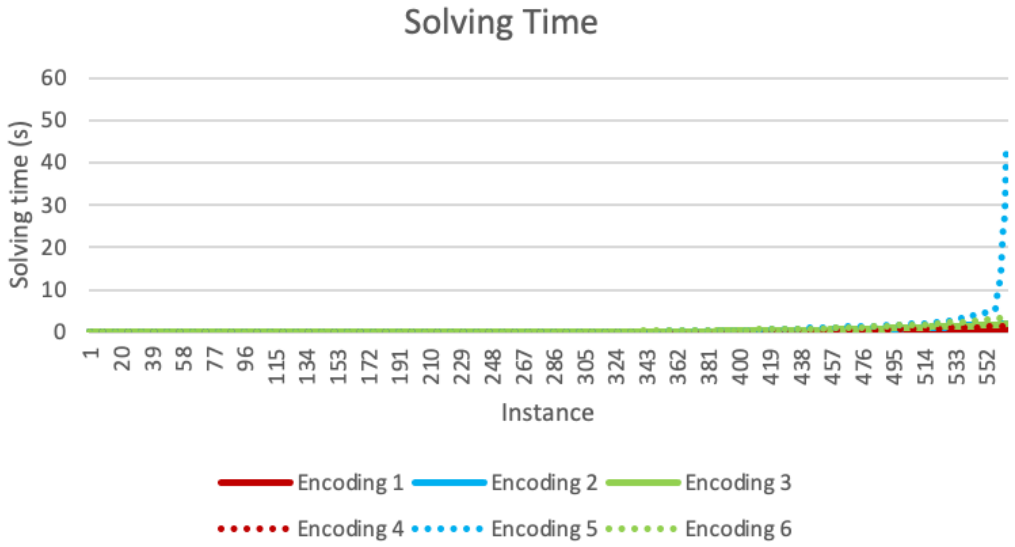

Figure 4.3: Average time for the SAT solver to find a satisfiable assignment.

From the graph 4.3, we can see that the dotted blue line (Encoding 5) had the highest line, and despite what was indicated by the table, actually had the worst performance. However, because the graph only plots the number of instances that were successfully solved by all encodings, it is not possible to tell which encoding had the best solving time.

That being said, the solving time for all encodings was significantly smaller compared to the clause-building time, and therefore determining the relative performance of the encodings in the context of average solving time is not very useful.

Average Total Runtime

	E1	E2	E3	E4	E5	E6
8x8	1.91	**1.71**	1.81	2.17	2.08	1.83
16x16	17.97	21.50	**17.47**	21.55	22.29	19.08
32x32	24.04	28.49	28.10	22.23	26.10	**21.79**
Empty	**12.86**	14.47	14.52	14.69	13.10	13.76
Random	13.08	13.03	13.50	14.32	12.32	**12.27**
Room	14.73	14.23	14.68	15.18	**13.93**	14.61
Average Runtime	13.52	13.90	14.22	14.71	**13.08**	13.55

Table 4.4: Average runtime (seconds) for each encoding on different sizes and types of maps.

Table 4.4 shows the average total runtime for each encoding across maps of different types and sizes. The average runtimes are similar for all encodings. The first three encodings had better performance for the smaller 8x8 maps. The encodings where the conflicts were added ad hoc (Encoding 4, Encoding 5, and Encoding 6) performed better than the rest for the bigger 32x32 maps.

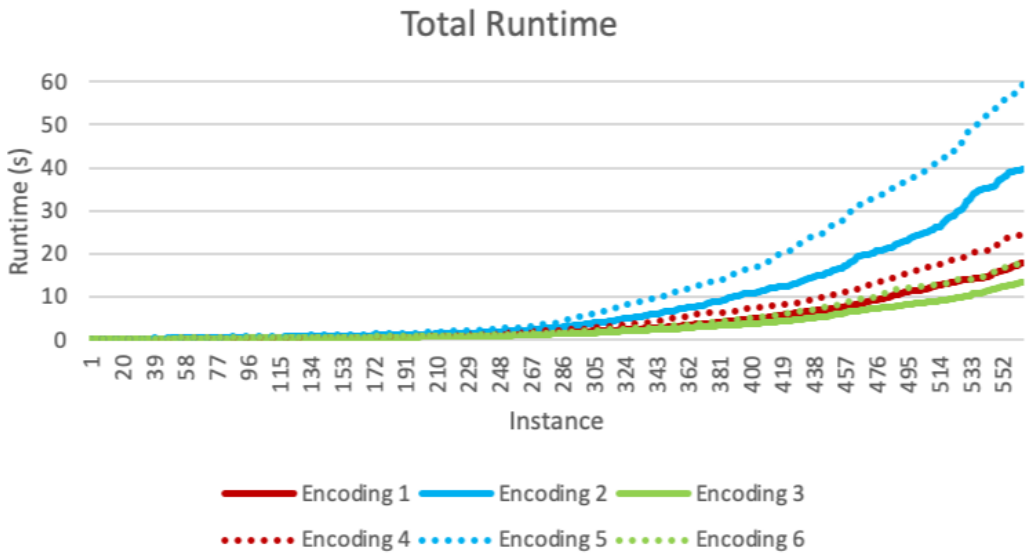

Figure 4.4: Average total runtime.

From the graph 4.4, we can deduce that Encoding 3 had the best performance and Encoding 5 had the worst performance. This is in accordance with the results of the average clause-building time. So, compared to the solving time, the time taken by each encoding to build its clauses had a greater contribution to its

overall performance.

Only 328 instances had a higher solving time compared to their clause-building time, with an equal split of around 100 instances between the empty, random and room maps. These instances were primarily on 16x16 maps (256 instances), with some 8x8 maps (72 instances). All these instances were densely populated maps. No instance with a 32x32 map had a higher solving time compared to the clause-building time. However, this might have been because the instances of 32x32 maps that were successfully solved within the given time limit were sparsely populated. The algorithms were unable to find solutions for densely populated 32x32 maps within the time limit of 60 seconds. From this, we can deduce that the clause-building time has a bigger impact on the runtime for sparsely populated maps, while the solving time has a bigger impact on the runtime for densely populated maps.

IPC

We also compute the IPC score for each instance. It gets its name from the International Planning Competition where it was introduced. The IPC score ranges from 0 to 1, where 0 indicates that an encoding did not finish within the given time. It is calculated as

$$IPC = \frac{\text{best score}}{\text{solver score}}$$

Therefore, the bigger the number, the better its performance. We sum up the IPC scores of each of the instances and present them in table 4.5.

	E1	E2	E3	E4	E5	E6
8x8	**123.70**	104.02	148.31	109.35	86.42	121.55
16x16	32.15	17.13	**34.77**	24.77	12.76	30.68
32x32	**15.79**	3.54	10.66	12.26	3.46	14.77
Empty	60.84	47.61	**72.67**	52.50	38.67	60.42
random	57.89	44.71	**68.82**	49.67	35.70	52.94
room	40.66	30.36	**44.60**	34.84	26.15	42.15
IPC Σ	157.71	121.38	**184.09**	135.57	99.48	153.96

Table 4.5: IPC Σ score for each encoding on different sizes and types of maps.

The table with the IPC sums shows the same results, with Encoding 3 having the highest score of 184.09, and Encoding 5 having the lowest score of 99.48.

4.3 Limitations

It is important to acknowledge the limitations of these results. Even though we chose maps of varying types and sizes to observe the changes in behaviour, the experiments were conducted on a limited set of instances, so they may not generalize to all MAPF scenarios.

The paper that Encodings 4-6 are based on (Surynek [2019]) claimed that these encodings had significantly better performance. However, our findings do not align with this. This might be because the largest maps we used were 32x32, while the experiments in the paper used much larger game maps.

In a paper by Barták and Svancara [2019], Encoding 2 (and by extension Encoding 5) perform well. This is in contradiction to the results obtained in this thesis. However, we conjecture that this is due to the implementation differences. In the paper, the authors use the Picat language to implement the encoding, while we used the Python language. Picat is a logic-based programming language that is similar to Prolog. Picat provides an easy representation of the constraints and then automatically translates these into a propositional formula. Since we found that the time taken to build clauses represents a big part of the runtime, using Picat as opposed to Python could have potentially led to better performance of the encoding.

These limitations provide the opportunity to improve the experiments and can potentially yield more accurate, or even different, results.

5. Conclusion

In this thesis, we compared different ways of reducing a multi-agent pathfinding (MAPF) problem to a SAT problem. We first introduced the MAPF problem, including relevant definitions and a brief background on related topics. We also explained the different possible behaviours of agents, the different conflicts, and the different metrics to measure the optimality of an MAPF solution. Then, we defined the specific settings that were used throughout this thesis. We forbade vertex conflicts and swapping conflicts. Edge conflicts were implicitly forbidden. The optimality of plans was measured using the makespan objective function.

An overview of the current research into MAPF was provided, with a more detailed emphasis on approaches that involve a reduction to a SAT problem. Both makespan optimal models and sum of costs optimal models were discussed, with more detail in the former. Six different encodings from the existing literature were introduced and implemented using the Python programming language. The first three encodings, Encoding 1, Encoding 2, and Encoding 3 used different combinations of variables to reduce the MAPF instance to a SAT problem. On the other hand, Encoding 4, Encoding 5, and Encoding 6 were built upon the previous three encodings but differed in that they added clauses to forbid conflicts when a conflict was found rather than all at once in the beginning.

We ran experiments on maps of different sizes and types to see if there was any performance difference between the encodings. The results we looked at were the number of solved instances, and the average clause-building time, solving time, and runtime. The results were presented in two ways: in a tabular format and a graphical format. Additionally, the IPC scores for each instance were calculated and their sums were presented in a table.

Our analysis showed that the clause-building time had a significant impact on the overall performance of an encoding while the solving time had a minimal impact for sparsely populated instances, and vice versa for densely populated instances. We found that Encoding 3 had the best performance overall. The first three encodings had better performance than the last three encodings in general, except for the larger 32x32 maps (that tended to be more sparse), where adding conflict clauses ad hoc proved to be beneficial in reducing the total runtime. This trend was confirmed both by the graph and by the IPC score sums.

5.1 Future Work

In this thesis, we explored six different encodings to solve MAPF problems. This was the combination of the variables At, $Move$, and $Shift$ along with the option of either starting by preventing all conflicts or by preventing conflicts once they are found. There is still scope for further work in the same direction.

Currently, if the formula is not satisfiable in the given T, we increase the

value of T by 1 until it is satisfiable. We can implement variations of encodings by changing the way we increment T. One such encoding can use binary search to find the optimal T while another encoding could estimate the T value from the agent's position.

Bibliography

Roberto Javier Asín Achá, Rodrigo López, Sebastián Hagedorn, and Jorge A. Baier. A new boolean encoding for MAPF and its performance with ASP and maxsat solvers. In Hang Ma and Ivan Serina, editors, *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, pages 11–19. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/SOCS/article/view/18546.

Roman Barták and Jirí Svancara. On sat-based approaches for multi-agent path finding with the sum-of-costs objective. 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 10–17. AAAI Press, 2019. URL https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18323.

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. IEEE Computer Society, 2017. doi: 10.1109/ICTAI.2017.00147. URL https://doi.org/10.1109/ICTAI.2017.00147.

Adi Botea and Pavel Surynek. Multi-agent path finding on strongly biconnected digraphs. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 2024–2030. AAAI Press, 2015. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9653.

Eli Boyarski, Ariel Felner, Guni Sharon, and Roni Stern. Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In Ronen I. Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015*, pages 47–51. AAAI Press, 2015a. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/view/10616.

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 Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 740–746. AAAI Press, 2015b. URL http://ijcai.org/Abstract/15/110.

Liron Cohen, Tansel Uras, and Sven Koenig. Feasibility study: Using highways for bounded-suboptimal multi-agent path finding. In Levi Lelis and Roni Stern, editors, *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel*, pages 2–8. AAAI Press, 2015. URL http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/11301.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. Push and rotate: a complete multi-agent pathfinding algorithm. *J. Artif. Intell. Res.*, 51:443–492, 2014. doi: 10.1613/jair.4447. URL https://doi.org/10.1613/jair.4447.

Martin Gebser, Philipp Obermeier, Torsten Schaub, Michel Ratsch-Heitmann, and Mario Runge. Routing driverless transport vehicles in car assembly with answer set programming. *Theory Pract. Log. Program.*, 18(3-4):520–534, 2018. doi: 10.1017/S1471068418000182. URL https://doi.org/10.1017/S1471068418000182.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136. URL https://doi.org/10.1109/TSSC.1968.300136.

Florence Ho, Rúben Geraldes, Artur Goncalves, Bastien Rigault, Benjamin Sportich, Daisuke Kubo, Marc Cavazza, and Helmut Prendinger. Decentralized multi-agent path finding for UAV traffic management. *IEEE Trans. Intell. Transp. Syst.*, 23(2):997–1008, 2022. doi: 10.1109/TITS.2020.3019397. URL https://doi.org/10.1109/TITS.2020.3019397.

Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018. doi: 10.1007/978-3-319-94144-8_26. URL https://doi.org/10.1007/978-3-319-94144-8_26.

Ilya Ivanashev, Anton Andreychuk, and Konstantin S. Yakovlev. Analysis of the anytime MAPF solvers based on the combination of conflict-based search (CBS) and focal search (FS). In Obdulia Pichardo-Lagunas, Juan Martínez-Miranda, and Bella Martínez-Seis, editors, *Advances in Computational Intelligence - 21st Mexican International Conference on Artificial Intelligence, MICAI 2022, Monterrey, Mexico, October 24-29, 2022, Proceedings, Part I*, volume 13612 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2022. doi: 10.1007/978-3-031-19493-1_29. URL https://doi.org/10.1007/978-3-031-19493-1_29.

Henry A. Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pages 359–363. John Wiley and Sons, 1992.

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. IEEE Computer Society, 1984. doi: 10.1109/SFCS.1984.715921. URL https://doi.org/10.1109/SFCS.1984.715921.

Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Scalable rail planning and replanning: Winning the 2020 flatland challenge. In Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankz Hankui Zhuo, editors, *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, pages 477–485. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/ICAPS/article/view/15994.

Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 294–300. IJCAI/AAAI, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-059. URL https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-059.

Raymond Reiter. On knowledge-based programming with sensing in the situation calculus. *ACM Trans. Comput. Log.*, 2(4):433–457, 2001. doi: 10.1145/383779.383780. URL https://doi.org/10.1145/383779.383780.

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. IEEE, 2010. doi: 10.1109/ROBOT.2010.5509582. URL https://doi.org/10.1109/ROBOT.2010.5509582.

Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors, *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press, 2012. URL http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5385.

Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent path finding. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5062.

Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artif. Intell.*, 195:470–495, 2013. doi: 10.1016/j.artint.2012.11.006. URL https://doi.org/10.1016/j.artint.2012.11.006.

David Silver. Cooperative pathfinding. In R. Michael Young and John E. Laird, editors, *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*, pages 117–122. AAAI Press, 2005.

Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1926.

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. URL https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18341.

Pavel Surynek. An optimization variant of multi-robot path planning is intractable. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1768.

Pavel Surynek. Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1177–1183. ijcai.org, 2019. doi: 10.24963/ijcai.2019/164. URL https://doi.org/10.24963/ijcai.2019/164.

Pavel Surynek. A sat-based approach to cooperative path-finding using all-different constraints. *Proceedings of the International Symposium on Combinatorial Search*, 3(1):191–192, 2021. doi: 10.1609/socs.v3i1.18220.

Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In Gal A. Kaminka, Maria Fox, Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum, and Frank van Harmelen, editors, *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)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 810–818. IOS Press, 2016. doi: 10.3233/978-1-61499-672-9-810. URL https://doi.org/10.3233/978-1-61499-672-9-810.

Jirí Svancara, 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. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33017732. URL https://doi.org/10.1609/aaai.v33i01.33017732.

Jirí Svancara, Philipp Obermeier, Matej Husár, Roman Barták, and Torsten Schaub. Multi-agent pathfinding on large maps using graph pruning: This way or that way? (extended abstract). In Lukás Chrpa and Alessandro Saetti, editors, *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, pages 320–322. AAAI Press, 2022. URL https://ojs.aaai.org/index.php/SOCS/article/view/21799.

Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. AAAI Press, 2013. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6111.

List of Figures

1.1 Example of an MAPF map where each cell is a vertex 5
1.2 Types of MAPF Conflicts (a) *edge conflict* (b) *vertex conflict* (c) *following conflict* (d) *cycle conflict* (e) *swapping conflict*. Figure taken from Stern et al. [2019] 6
1.3 Sum of Costs vs Makespan. Figure taken from Barták and Svancara [2019] 9

2.1 An example of a sub-optimal vs optimal MAPF solution (Ivanashev et al. [2022]) 10
2.2 Example of a 3-vertex graph being transformed to a T-layered time-expanded graph (Barták and Svancara [2019]) 13

3.1 Types of maps available at Moving AI Lab 21
3.2 Example of an 8x8 map file 22

4.1 (a) 8x8 random map (b) 8x8 room map 23
4.2 Average time to build clauses. 25
4.3 Average time for the SAT solver to find a satisfiable assignment. . 26
4.4 Average total runtime. 27

List of Tables

1.1 Example solution to the MAPF instance in Figure 1.1. The numbers correspond to the vertex each agent occupies at that timestep. 5

4.1 Number of solved instances by each encoding on different sizes and types of maps. 24
4.2 Average time (seconds) to build clauses for each encoding on different sizes and types of maps. 25
4.3 Average time (seconds) for the SAT solver to try to find a satisfying assignment for each encoding on different sizes and types of maps. 26
4.4 Average runtime (seconds) for each encoding on different sizes and types of maps. 27
4.5 IPC Σ score for each encoding on different sizes and types of maps. 28

A. User Documentation

A.1 Introduction

This program reduces multi-agent pathfinding problems into SAT problems, using a SAT solver to find valid solutions. There are six different encodings available to reduce an MAPF problem to a SAT problem.

A.2 System Requirements

To use this program, you will need a computer with Python3 installed.

A.3 Getting Started

To use this program, follow these steps:

- Download the source code and instances (maps and scenarios) from the GitLab repository `https://gitlab.mff.cuni.cz/teaching/nprg045/svancara/samyuktha-ramesh-isp`.
- Install the necessary dependencies using pip.
- Open a terminal and navigate to the directory containing the code and instances.
- Run the program by executing the "Program.py" script, passing in any desired arguments.

A.4 Arguments

This program accepts arguments while running the program. A list of possible arguments is given below. Note that all arguments have a default value so the program can be run without any arguments.

- **--scen**: Specifies the path to the file containing the MAPF scenario to be solved. The path to the corresponding map file is present in the scenario file so an additional argument is not necessary. The default file path is "scenarios/empty-8-8-random-1.scen".
- **--number_of_agents**: Specifies the number of agents to include in the MAPF scenario. The program will select the first number_of_agents agents in the scenario file. The default value is 1.
- **--encoding**: Specifies the name of the encoding function used to reduce the MAPF problem to a SAT problem. The available encoding options are "encoding_1", "encoding_2", "encoding_3", "encoding_4", "encoding_5", and "encoding_6". The default encoding is "encoding_1".

- **--timeout**: Specifies the maximum number of seconds the solver will run for before timing out. The default is 60 seconds.
- **--increment**: This specifies the number of agents to add at each iteration, given that the previous iteration was solved in the given time limit. Note that the increment value must be a positive integer greater than 0.

A.5 Using the Program

To solve an MAPF problem using this program, run the program specifying the required arguments.

Here is an example command to run the program with a timeout of 200 seconds, using encoding_2, and with 3 agents:

```
python src/Program.py --timeout 200 --number_of_agents 3 --encoding encoding_2
```

A.6 Instances

Compatible maps and scenarios that this program can try to solve can be found at `https://movingai.com/benchmarks/mapf/index.html`.

B. Attachments

This thesis includes an attachment containing the source code for the program, as well as the instances that the experiments were run on. The attachment can be found on the Student Information System (SIS). A brief overview of the folders is provided here:

- **src** - Source code containing the main file *Program.py*, a map loader *MapLoader.py*, and a scenario loader *ScenLoader.py*. It also contains six different encodings, as well as a file with code common to all encodings *Encodings_common.py*, and a conflict checker *Conflict_check.py*.
- **maps** - Maps used in the experiments. Types: empty, random, room. Sizes: 8x8, 16x16, 32x32.
- **scenarios** - Scenarios used in the experiments. There are three scenarios for every map.
- **results** - CSV files with the results of the experiments.