Reduction-based Solving of Multi-agent Pathfinding on Large Maps Using Graph Pruning

Matej Husár
Charles University
Prague, Czech Republic
husarmatej@gmail.com

Jiří Švancara
Charles University
Prague, Czech Republic
svancara@ktiml.mff.cuni.cz

Philipp Obermeier
Potassco Solutions, and
University of Potsdam
Potsdam, Germany
phil@cs.uni-potsdam.de

Roman Barták
Charles University
Prague, Czech Republic
bartak@ktiml.mff.cuni.cz

Torsten Schaub
Potassco Solutions, and
University of Potsdam
Potsdam, Germany
torsten@cs.uni-potsdam.de

ABSTRACT

Multi-agent pathfinding is the problem of finding collision-free paths for a set of agents. Solving this problem optimally is computationally hard, therefore many techniques based on reductions to other formalisms were developed. In comparison to search-based techniques, the reduction-based techniques fall behind on large maps even for a small number of agents. To combat this phenomenon, we propose several strategies for pruning vertices off large instances that will most likely not be used by agents. First, we introduce these strategies conceptually and prove which of them maintain completeness and optimality. Eventually, we conduct an exhaustive evaluation and show that graph pruning strategies make reduction-based solvers comparable to search-based techniques on large maps while maintaining their advantage on small dense maps.

KEYWORDS

Multi-agent pathfinding; Scalability; Answer set programming; Satisfiability; Subgraph; Graph pruning; Logic programming

ACM Reference Format:
Matej Husár, Jiří Švancara, Philipp Obermeier, Roman Barták, and Torsten Schaub. 2022. Reduction-based Solving of Multi-agent Pathfinding on Large Maps Using Graph Pruning. In *ACM Conference, Washington, DC, USA, July 2017*, IFAAMAS, 9 pages.

1 INTRODUCTION

Multi-agent pathfinding (MAPF) is the problem of navigating a fixed set of mobile agents in a shared environment (map) from their initial locations to target destinations without any collisions among the agents [24]. This problem has numerous practical applications in robotics, logistics, digital entertainment, automatic warehousing and more, and it has attracted significant research focus from various research communities in recent years [10, 14, 17, 18, 20, 27].

The optimal MAPF solvers can be in general split into two categories – search-based and reduction-based. The former algorithms search over possible locations or conflicts among the agents, the latter reduce the problem to some other well-defined formalism such as Answer Set Programming (ASP; [7, 19]) or Boolean Satisfiability (SAT; [4]). While it is not always the case, it is generally established that each of the approaches dominates on different types of instances [14, 28]. The search-based solvers are easily able to find solutions on large sparsely populated maps while having trouble dealing with small densely populated maps. On the other hand, the reduction-based solvers are able to deal with the small densely populated maps but are unable to find a solution for large maps even with a small number of agents.

Since the reduction-based solvers have trouble with solving instances on large maps, the main idea of the presented techniques is to prune the map of vertices that are most likely not needed to solve the instance while maintaining completeness and optimality.

1.1 Related Work

Our proposed methods are build on top of an existing ASP-based MAPF solver [9, 11] and SAT-based MAPF solver [2, 3]. We describe these two solvers in more detail in later sections.

A similar idea can be seen in Corridor-Map-Method (CMM) [12, 13] where an explicit corridor is defined for each agent and the agent is allowed to move only in that corridor, thus, decreasing the search space of the problem. As opposed to CMM, our approach allows the set of the pruned vertices to change over time to maintain optimality and completeness. In the work on CMM, the authors do not focus on optimality, since they use the technique for online planning with dynamic obstacles. Also, they are concerned with smooth trajectories rather than the shortest possible paths.

A different angle on helping the underlying solver by removing some unnecessary information is implemented by SMT-CBS [26]. This approach does not include all of the constraints on the agents' interactions. Only once it is found that a specific interaction needs to be forbidden, the solver is informed and the information is added. Similarly, in our techniques, we change the set of the excluded vertices based on the solutions provided by the underlying solver.

CBS is an optimal MAPF algorithm that is commonly studied and used in the literature and considered to be the state-of-the-art approach [5, 22]. We will also compare our results to CBS. It is a search-based algorithm that can be split into two levels. On the low level, a single-agent path for each agent is found (this can be done in polynomial time). The high level checks if there are any conflicts between the single-agent plans. If there are any, constraints that disallow this conflict are added and the low level repeatedly finds single-agent plans in accordance with the constraints. The high

level constraints tree is traversed in a best-first search so that the first found solution is guaranteed to be optimal.

If there are not many interactions between the agents, CBS is very successful even on large instances due to the polynomial low level. In a sense, the purpose of our proposed strategies is to help the underlying solver find easily some path for each agent (as the low level of CBS) and use the solver strength to deal with the conflicts among them (as the high level of CBS).

1.2 Contributions

The first contribution of this paper is identifying the types of instances that makespan optimal reduction-based MAPF solvers have trouble dealing with. We also reason why these instances are hard. The examples in this paper are always on a grid map, however, the idea and proposed techniques are valid for any type of graph.

The second contribution is describing three different strategies that solve the problematic instances. We also provide a theoretical analysis of the strategies on whether they maintain completeness and optimality.

The third contribution of this paper is an empirical evaluation of the proposed strategies along with different underlying reduction-based solvers. This also includes exploiting the preprocessing information commonly used by SAT-based solvers for ASP solving which is missing in the current encodings.

2 MAPF DEFINITIONS

MAPF instance $\mathcal{M}$ is a pair $\mathcal{M} = (G, A)$, where G is a graph $G = (V, E)$ and A is a set of agents. Each agent $a_i \in A$ is defined as a pair $a_i = (s_i, g_i)$, where $s_i \in V$ is a starting location of agent a_i and $g_i \in V$ is a goal location of agent a_i.

Our task is to find a *valid plan* π_i for each agent $a_i \in A$ such that it is a valid path from s_i to g_i. We use $\pi_i(t) = v$ to denote that agent a_i is located in vertex v at timestep t. Time is discrete and at each timestep t, an agent can either wait in its current location or move to a neighboring location.

Furthermore, we require that each pair of plans π_i and π_j, $i \neq j$ is collision-free. Based on MAPF terminology [25], there are five types of collisions (see Figure 1). In this work, we forbid *edge*, *vertex*, and *swapping* conflict while allowing *following* and *cycle* conflicts since during the last two conflicts, the agents are not occupying the same physical location. We call this setting *parallel motion*, as opposed to *pebble motion* [16], where all of the conflicts are forbidden.

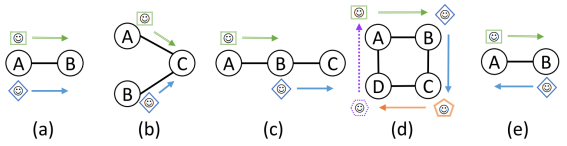


Figure 1: Conflicts between two or more agents. (a) *edge conflict*, (b) *vertex conflict*, (c) *following conflict*, (d) *cycle conflict*, (e) *swapping conflict*. Figure taken from [25].

In this paper, we are interested in finding a *makespan* optimal solution to MAPF problems. Makespan (or sometimes horizon) is the first timestep t at which all of the agents are located at their goal vertices. Once an agent arrives at its goal location it does not disappear. It may move out of the goal location again, however, the plan ends once all of the agents are at the goal location at the same time. This means that the length of the plan $|\pi_i|$ is the same for all of the agents. Another cost function often used in literature is *sum of costs* [23]. Note that finding an optimal solution for either of the cost functions is an NP-Hard problem [21, 29].

3 SOLVING MAPF VIA REDUCTION

3.1 SAT Encoding

Let's assume that we are looking for a solution to a MAPF problem with makespan (horizon) H using the parallel motion restriction on allowed conflicts. The SAT encoding in [2, 3] defines the following two sets of variables: $\forall v \in V, \forall a_i \in A, \forall t \in \{0, \ldots, H\} : At(v, i, t)$ meaning that agent a_i is at vertex v at timestep t; and $\forall (u, v) \in E, \forall a_i \in A, \forall t \in \{0, \ldots, H-1\} : Pass(u, v, i, t)$ 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 H. An auxiliary loop 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:

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

$$\forall a_i \in A : At(g_i, i, H) = 1 \tag{2}$$

$$\forall a_i \in A, \forall t \in \{0, \ldots, H\} : \sum_{v \in V} At(v, i, t) \leq 1 \tag{3}$$

$$\forall v \in V, \forall t \in \{0, \ldots, H\} : \sum_{a_i \in A} At(v, i, t) \leq 1 \tag{4}$$

$$\forall u \in V, \forall a_i \in A, \forall t \in \{0, \ldots, H-1\} : At(u, i, t) \implies \sum_{(u,v) \in E} Pass(u, v, i, t) = 1 \tag{5}$$

$$\forall (u, v) \in E, \forall a_i \in A, \forall t \in \{0, \ldots, H-1\} : Pass(u, v, i, t) \implies At(v, i, t+1) \tag{6}$$

$$\forall (u, v) \in E : u \neq v, \forall t \in \{0, \ldots, H-1\} : \sum_{a_i \in A} (Pass(u, v, i, t) + Pass(v, u, i, t)) \leq 1 \tag{7}$$

Constraints (1) and (2) ensure that the starting and goal positions are valid. Constraints (3) and (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 (5) – (7). In sequence, they ensure that if an agent is in a certain vertex, it needs to leave it by one of the outgoing edges (5). If an agent is using an edge, it needs to arrive at the corresponding vertex in the next timestep (6). Finally, (7) forbids two agents to traverse two opposite edges at the same time (forbidding swapping conflict). To find the optimal makespan, we iteratively increase H until a satisfiable formula is generated.

We provide the constraints as a set of inequalities rather than a CNF formula since it is more readable and there are tools that

automatically translate such inequalities into a CNF that is solvable by any SAT-solver.

3.2 ASP Encoding

To describe both movement actions and positional changes of agents, we use the ASP encoding[1] of an action theory for MAPF in Listing 1, introduced by [9, 11]. The encoding assumes that graph G is a grid and plans agents (here called *robots*) in parallel within a makespan while avoiding conflicts. Specifically, the plan's timesteps are bound by the `horizon` (or makespan) in Line 1. Line 3 gives the four cardinal directions, used in Line 4 to represent all transitions on the grid with its x,y-coordinates stated by predicate `position/1`. Possible movement actions, at most one per agent and timestep, are generated by Line 8. Related preconditions and positional changes are described in Lines 10-12: `position(R,C,T)` states that agent `R` is at x,y-coordinates `C` at time `T`. For an agent `R` sitting idle at time `T`, the frame axiom in Lines 14-15 propagates its unchanged position. Swapping conflicts are prevented by Lines 17-19, and both edge and vertex conflicts by Line 21.

```
1   time(1..horizon).

3   direction((X,Y)) :- X=-1..1, Y=-1..1, |X+Y|=1.
4   nextto((X,Y),(DX,DY),(X',Y')) :-
5       direction((DX,DY)), position((X,Y)), position((X',Y')),
6       (X,Y)=(X'-DX,Y'-DY), (X',Y')=(X+DX,Y+DY).

8   { move(R,D,T) : direction(D) } 1 :- isRobot(R), time(T).

10  position(R,C,T) :-
11      move(R,D,T), position(R,C',T-1),     nextto(C',D,C).
12   :- move(R,D,T), position(R,C ,T-1), not nextto(C ,D,_).

14  position(R,C,T) :-
15      position(R,C,T-1), not move(R,_,T), isRobot(R), time(T).

17  moveto(C',C,T) :-
18      nextto(C',D,C), position(R,C',T-1), move(R,D,T).
19   :- moveto(C',C,T), moveto(C,C',T), C < C'.

21   :- { position(R,C,T) : isRobot(R) }  > 1, position(C), time(T).
```

Listing 1: Action theory for agent movements.

Further, we augment the action theory encoding with the goal condition in Listing 2 to enforce that every agent `R` has reached its goal coordinates `C`, stated by `goal(R,C)`, at the time `horizon`.

```
1   :- not position(R,C,horizon), goal(R,C).
```

Listing 2: Goal condition for agents and assigned nodes.

Overall, our ASP encoding consists of the action theory (Listing 1) in conjunction with the goal condition (Listing 2) and expects an MAPF instance in form of the aforementioned ASP facts as input.

4 GRAPH PRUNING

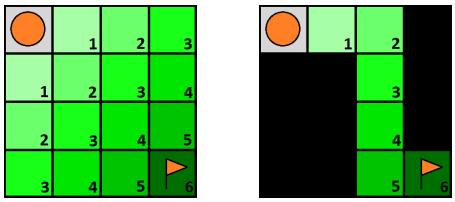
4.1 Motivation

There are two commonly used techniques to speed up computation, both applicable to the described reduction-based solvers. First, using a lower bound for the makespan instead of starting with $H = 1$. A simple lower bound is to compute for each agent a_i the shortest path from agent's start location s_i to agent's goal location g_i. The lower bound for H is then the longest of these shortest paths.

Another enhancement is to preprocess the variables representing the agent's location. 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 cannot be present at the specific time, because we know where the agent needs to be present at times 0 and H. Specifically, for agent a_i, if vertex v is distance d away from start location s_i, we know that the agent a_i cannot be present in vertex v at times $0, \ldots, (d-1)$ because it cannot travel the distance in time. Similarly, if vertex v is distance d away from goal location g_i, agent a_i cannot be present in vertex v at times $H-d+1, \ldots, H$. For SAT encoding, we set the corresponding variables $At(v, i, t)$ to *False*. For ASP, we add the integrity constraint in Listing 3 to ensure that agent `R` occupies an eligible position `C` at time `T`, expressed by a fact `poss_loc(R,C,T)`.

```
1   :- not poss_loc(R,C,T), position(R,C,T), isRobot(R),
2          poss_loc(_,_,_).
```

Listing 3: Eligible agent locations from pre-processing.



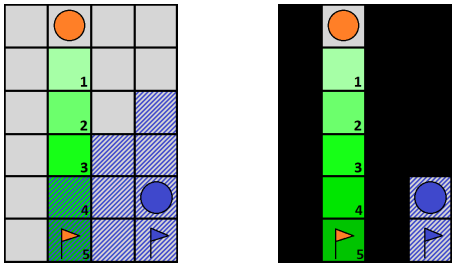
(a) **(b)**

Figure 2: An agent moving on a grid map from a corner to the opposite one. The numbers represent at what timesteps the agent can reach the given vertex.

Both of these techniques maintain completeness and optimality. However, there are situations, where too many possibilities for the agent's location remain, which may overwhelm the underlying solver. As a motivation example, see Figure 2a. The agent is placed on a 4-connected grid map going from one corner to the diagonally opposite corner. With just one agent and no obstacles, there are $\binom{2(N-1)}{N-1}$ possible shortest paths if the size of the grid is $N \times N$. As seen in the figure, the preprocessing correctly finds at what timesteps the agent can be located at which vertices, noted by the number in the corner of each vertex. However, the number of choices for the solver is still too large. We propose to pick just one of the shortest paths and treat the other vertices as an impassable obstacle. Hence, for these vertices, there are no variables entering the solver. Such pruning of the graph can be seen in Figure 2b.

Another example where this approach is helpful can be seen in Figure 3a. The two agents have different lengths of shortest paths. For the orange agent with the longer path, preprocessing correctly finds the only shortest path. However, the blue agent with a much shorter path may move anywhere in the shaded area since it has enough time. Recall that we are computing makespan optimal solutions, so we are interested in the timestep when all agents are at their goal locations. This time is prolonged by the orange agent, therefore the blue agent has many more choices.

[1] https://github.com/potassco/asprilo-encodings/blob/master/m/action-M.lp



(a) (b)

Figure 3: An instance with two agents, one with longer path allowing the other to move freely.

If we use our pruning technique the number of choices reduces dramatically. The blue agent can still choose when to move to the goal location, or even move back and forth a few times, however, it may use significantly fewer vertices.

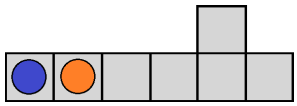


Figure 4: An instance with two agents that want to swap their positions.

Of course, this pruning does not maintain completeness in general. A simple counterexample can be seen in Figure 4. The two agents want to swap their location (ie. their goal location is identical with the starting location of the other agent). To do this, the only solution is for both of them to travel to the right and use the top vertex to switch their position. If we were to use our pruning technique, this would not be possible, making the example unsolvable. To mitigate these instances, we propose several strategies how to change which vertices are pruned.

4.2 Solving Strategies

First, we establish some notation. Let SP_i be the vertices on a chosen shortest path for agent $a_i \in A$ (ie. a single shortest path from s_i to g_i). The length of the path is $|SP_i|$. The union of vertices on the shortest paths of all agents is $SP_A = \bigcup_{a_i \in A} SP_i$. Note that for each agent we consider just one shortest path. If multiple shortest paths exist for an agent, one is chosen at random. Given this notation the lower bound on makespan of an instance $\mathcal{M} = (G, A)$ can be written as $LB_{mks}(G, A) = \max_{a_i \in A} |SP_i|$. For short, we refer to such lower bound just by LB.

A *k-restricted graph* $Gres_k^{SP_A}$ is a subgraph of G containing only vertices in SP_A and vertices that are at most distance k away from some vertex in SP_A, ie. $Gres_k^{SP_A} = \{v \in V \mid \exists u \in SP_A,\ dist(u, v) \leq k\}$. Since we always fix SP_A, we write for simplicity only $Gres_k$. Note that $Gres_k \subseteq Gres_{k'}$ for $k \leq k'$. An example of such k-restricted graph can be seen in Figure 5. For a 0-restricted graph, only the shortest path is part of the graph. A 3-restricted graph is the whole initial graph in case of the example in Figure 5.

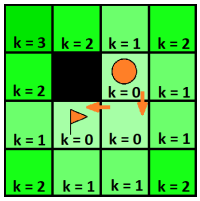


Figure 5: An instance with a single agent. Each vertex is labeled into which k-restricted graph it belongs.

Since finding the makespan optimal solution is done by iteratively increasing the makespan, we define a *makespan-restricted* MAPF instance $\mathcal{M} = (G, A, H)$. This is the same problem as finding the solution for $\mathcal{M} = (G, A)$ in makespan H.

The *(k,m)-relaxation* of $\mathcal{M}$ is the makespan-restricted MAPF instance

$$\mathcal{M}_{k,m} = (Gres_k, A, LB + m)$$

This relaxation means that instead of the whole graph G we consider only $Gres_k$ and we are finding a solution with extra makespan m – extra over the lower bound on makespan. Also note that $Gres_k$ is constructed such that $LB_{mks}(G, A) = LB_{mks}(Gres_k, A)$ for any k, therefore, we do not need to change the notation of LB.

We can build a partial order $\prec_{relax}$ over the (k,m)-relaxations $\mathcal{M}_{k,m}$ such that

$$\mathcal{M}_{k,m} \prec_{relax} \mathcal{M}_{k',m'}$$

if $k \leq k'$, $m \leq m'$ and $k + m < k' + m'$

There is an upper bound on k such that for some k_{max} we have $Gres_{k_{max}} = G$. There is also a theoretical upper bound on makespan for a given MAPF instance of $O(V^3)$ [16], however, in this paper we work only with solvable instances (this can be checked by polynomial-time algorithm) and we do not need to know the exact upper bound on makespan. Just for the next example assume that $k_{max} = 3$ and $m_{max} = 2$. Then, Figure 6 depicts the space of possible relaxations induced by $\prec_{relax}$. Note that the partial ordering forms a lattice.

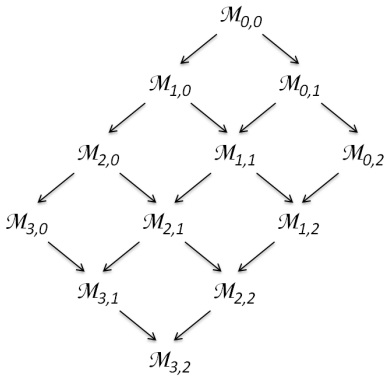


Figure 6: MAPF instance relaxations for $k_{max} = 3, m_{max} = 2$.

The generic algorithm to solve MAPF using the relaxed instances can be seen in Algorithm 1. First, we build an initial (k,m)-relaxation and we iteratively change k and m until the instance is solvable.

This corresponds to a traversal of the lattice formed by the partial ordering $\prec_{relax}$. Note that the shortest path for each agent is fixed for all of the iterations. Next we identify four reasonable traversals.

Algorithm 1 Generic algorithm solving MAPF using relaxation.

```
function GENERIC MAPF RELAXATION(M = (G, A))
    LB = max_{a_i ∈ A} |SP_i|
    (k, m) ← Initial_Candidate()
    while not solve_MAPF(M_{k,m}) do
        (k, m) ← Relax()
    end while
    return LB + m
end function
```

4.2.1 Baseline Strategy. The classical approach to solving MAPF makespan optimally can be expressed in the relaxed instances as follows. We start with an initial candidate of k_{max} (ie. the whole graph G) and $m = 0$. If the relaxed instance is unsolvable, only the additional makespan is increased as $m = m + 1$. In terms of the Figure 6, the first solved relaxation is $\mathcal{M}_{3,0}$ and then we are moving only to the right-hand side. We shall refer to this strategy as *baseline* or **B** for short.

Proposition 1. *If a MAPF instance $\mathcal{M}$ has a solution,* baseline *strategy finds an optimal solution.*

Proof. Since $\mathcal{M}$ has a solution, there needs to be an optimal solution with some makespan H such that $LB \leq H$. The *baseline* strategy will try all of the possible makespans $LB, \ldots, H$, with H being the first solvable. □

4.2.2 Makespan-add Strategy. The first smarter solution is to keep only the vertices on the shortest paths and the immediately adjacent ones. The initial candidate is $k = 1$ and $m = 0$. Otherwise, the strategy is the same as the baseline strategy – if the relaxed instance is unsolvable, we increase $m = m + 1$ while the k is never changed. We refer to this strategy as *makespan-add* or **M** for short.

Proposition 2. Makespan-add *strategy is both suboptimal and incomplete.*

Proof. For a simple example where *makespan-add* cannot find a solution recall Figure 4. No matter how the initial constant of k is set, we can create a graph where the extra vertex needed for the two agents to swap is not part of $Gres_k$.

For an example where *makespan-add* finds a suboptimal solution see figure 8 with blue agent choosing the blue path. In this case *makespan-add* needs to increase m two times to find a solution, while it would be possible to find a solution in LB steps if the vertices of the black path were included. □

On the other hand, in most cases, this simple strategy can find a solution, and due to the great reduction of vertices of the graph, the solution may be found quickly. We choose to start with $k = 1$ rather than $k = 0$ to increase the probability for a solution to exist while keeping the number of vertices to a minimum.

In terms of Figure 6, the strategy first moves to the left once and then only to the right.

4.2.3 Prune-and-cut Strategy. The previous strategies either use unnecessary large restricted graph or do not guarantee to find a solution. Strategy *prune-and-cut* (**P** for short) guarantees both completeness and optimality. We start with initial candidate $k = 0$ and $m = 0$. In case the relaxed instance is unsolvable, we cannot be sure if the reason is the restriction on k or on m. However, since we do not want to overestimate m, we first need to increase k potentially up to k_{max}. Once a restricted instance $\mathcal{M}_{k_{max},m}$ is unsolvable, we are sure that m needs to be increased. Since we proved that we require at least $m + 1$ extra makespan, we can optimistically assume that the whole $Gres_{k_{max}}$ is not needed and we restrict the graph back to $k = 0$ producing $\mathcal{M}_{0,m+1}$.

Proposition 3. *If a MAPF instance $\mathcal{M}$ has a solution,* prune-and-cut *strategy finds an optimal solution.*

Proof. Before m is increased, we always check if there is a solution using the original G. The rest of the proof is the same as for Proposition 1. □

During our initial experiments, it turned out that the whole $Gres_{k_{max}}$ is usually not necessary. Therefore, increasing k by 1 each time may prove inefficient, since most of the calls are unsolvable and we just need to prove that we can increase m. For this reason we increase k by powers of 2 (ie. $k = k + 1, k = k + 2, k = k + 4, \ldots$).

Another implementation improvement is to not increase up to k_{max} but rather to some $k \leq k_{max}$ that produces $Gres_k$ that includes all of the vertices reachable in given $LB + m$ by some agent. This information can be obtained by the preprocessing.

The visualization of solver calls of the *prune-and-cut* strategy over the lattice can be seen in Figure 7.

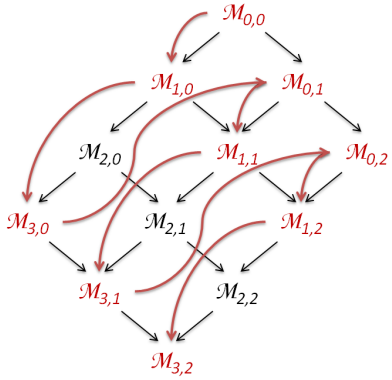


Figure 7: The traversal of the lattice by strategy *prune-and-cut*. The highlighted relaxed instances are being solved.

4.2.4 Combined Strategy. The drawback of the *prune-and-cut* strategy is that in the case the makespan needs to be increased, we first increase k up to k_{max} before increasing m. To mitigate this problem, we present the *combined* strategy (**C** for short). The initial candidate is again $k = 0$ and $m = 0$. If the relaxed instance is unsolvable, we increase both $k = k + 1$ and $m = m + 1$ at the same time. This way, we save the number of calls to the solver because we do not need to explore all of the possible reductions in the k direction. On the other hand, this strategy is no longer optimal.

Proposition 4. *If a MAPF instance $\mathcal{M}$ has a solution,* combined *strategy is guaranteed to find a solution (completeness) but not necessarily an optimal one.*

Proof. If it is necessary to use all of the vertices in the graph G to find a solution, *combined* strategy will eventually increase k up to k_{max} since k_{max} is a finite number. However, in doing so, it may overestimate the m needed. Figure 8 with blue agent choosing the blue path is again such an example. □

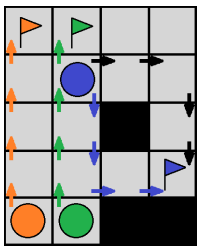


Figure 8: An example instance where the blue agent has two choices of the shortest path. If the blue path is chosen, the proposed strategies perform worse.

The described strategies (with the exception of *baseline*) may suffer from poor choices of the initial shortest paths for each agent. See example in Figure 8. The blue agent has two possible shortest paths. If the algorithm by random chooses the blue path, none of the sophisticated strategies is able to solve the relaxed instance in the first solver call. *Makespan-add* would find a suboptimal solution with makespan $LB + 2$, *prune-and-cut* would require to increase k two times to be able to use the black path, and *combined* strategy would also find a suboptimal solution with makespan $LB + 2$.

This issue can be mitigated by including all of the vertices on all of the possible shortest paths into the $Gres_k$, however, this goes against the logic of the motivational example in Figure 2, therefore we still include only one of the shortest paths for each agent.

5 EXPERIMENTAL EVALUATION

To test and compare the proposed strategies and the underlying solvers, we set up empirical experiments[2].

The SAT-based solver is implemented in Picat programming language [30] and is run on Picat version 3.1. For ASP, we used the grounding-and-solving system *clingo* [6, 15] version 5.5.1.

Furthermore, we used an implementation of a makespan optimal CBS algorithm [1] to compare to a state-of-the-art search-based approach. We are using CBS as a black-box without influencing which shortest paths the algorithm chooses. As shown in previous examples, using poorly chosen paths may lead to worse performance in our strategies. On the other hand, one of the first steps of CBS is to choose different shortest paths if the initial ones are chosen poorly. Therefore, we argue that this comparison is fair even if the initial paths are chosen differently by our strategies and the CBS algorithm.

[2] https://github.com/potassco/mapf-subgraph-system

We ran the experiments on an Intel Xeon E5-2650v4 under Debian GNU/Linux 9, with each instance limited to 300s processing time and 28 GB of memory.

5.1 Instances

The instances used in our experiments are inspired by commonly used benchmark instances available online [25]. To see the effect of the increase in the size of the map, we chose maps such that they fall into one of three size categories – *small* (32 by 32), *medium* (64 by 64), and *large* (128 by 128). Furthermore, the structure of the impassable obstacles in the map may affect the paths of the agents and thus the solver performance. We picked the following types – *empty*, *maze*, *random*, and *room* (see Figure 9 for reference). Unfortunately, some of the combinations of size and type were not available in the benchmark set, therefore, we had to create our own following the structure of the existing maps.

Figure 9: Types of maps used in the experimental evaluation. From left to right: *maze, random,* and *room.*

For the placement of the agents (called *scenarios*), we used the available scenarios from the benchmark sets or, if not present, created our own. For each map, we used 5 different scenarios. Furthermore, we created new scenarios for each map such that the distance from start to goal of each agent is similar and the paths of the agents need to cross more often. We did this because the makespan optimal solution for the random scenarios rarely differ from the lower bound. This is caused by one of the agents having a much longer path than the others leaving them with enough time to solve any conflicts. The behavior of our strategies may be gravely affected by many conflicts and the need to increase the makespan. We are also using 5 different scenarios with this setting.

The intended way to use the benchmark set is to create an instance of MAPF from a map and a number of agents from a scenario. If the instance is solved in the given time-limit, additional agents from the same scenario are added and thus a new MAPF instance is produced. Once the instance cannot be solved in the time-limit, it is reasoned that increasing further the number of agents cannot make the instance solvable. We are aware that using reduction-based solvers, this may not always hold. Also, some of our strategies may benefit from additional agents which change the restricted graph. However, these cases are extremely rare and therefore, we decided to use the benchmark as intended starting with 5 agents, adding 5 more each time the instance is solvable in given time-limit.

5.2 Results

Table 1 shows the measured results for all of the possible combinations of strategies and underlying solvers we used. For the ASP-based solver, we differentiate between using and not using preprocessing, since the preprocessing is a new addition to the existing encoding. For the SAT-based solver, we did not include the setting

		ASP				ASP with preprocessing				SAT with preprocessing				CBS
		Optimal		Sub-optimal		Optimal		Sub-optimal		Optimal		Sub-optimal		
	size	**B**	**P**	**M**	**C**	**B**	**P**	**M**	**C**	**B**	**P**	**M**	**C**	
Solved instances	32	816 (0.91)	806 (0.90)	801 (0.89)	832 (0.93)	**851 (0.95)**	836 (0.93)	845 (0.94)	**860 (0.96)**	472 (0.53)	475 (0.53)	486 (0.54)	526 (0.59)	199 (0.22)
	64	459 (0.68)	567 (0.84)	522 (0.77)	610 (0.91)	516 (0.77)	**612 (0.91)**	573 (0.85)	**670 (0.99)**	141 (0.21)	325 (0.48)	324 (0.48)	369 (0.55)	172 (0.26)
	128	162 (0.42)	287 (0.74)	280 (0.72)	335 (0.86)	194 (0.50)	**315 (0.81)**	332 (0.86)	**382 (0.98)**	15 (0.04)	178 (0.46)	173 (0.45)	216 (0.56)	119 (0.31)
∑ IPC	32	422.4	520.2	506.4	640.7	546.8	**582.5**	647.6	**773.3**	46.9	84.7	108.6	132.3	87.1
	64	129.3	347.1	265.9	452.7	210.6	**417.0**	387.0	**611.1**	9.4	55.2	71.1	95.5	82.1
	128	24.8	147.9	116.4	198.3	42.6	**203.1**	213.6	**314.7**	0.1	38.3	38.6	58.5	72.0
	total	576.5	1015.2	888.8	1291.7	799.9	**1202.6**	1248.2	**1699.1**	56.3	178.1	218.3	286.3	241.3
Max. agents	32	102	101	100	104	**107**	105	106	**108**	59	59	61	66	25
	64	57	71	65	76	65	**77**	72	**84**	18	41	41	46	22
	128	20	36	35	42	24	**39**	42	**48**	2	22	22	27	15
Conflicts	32	396	380	385	408	**269**	334	**310**	317	-	-	-	-	-
	64	242	110	204	187	172	**91**	169	**134**	-	-	-	-	-
	128	272	**68**	119	**86**	176	72	151	89	-	-	-	-	-
Constraints [millions]	32	22,7	17,6	21,4	17,6	8,1	**6,6**	7,8	**6,6**	-	-	-	-	-
	64	37,7	11,0	20,0	11,0	10,3	**3,1**	6,0	**3,1**	-	-	-	-	-
	128	43,4	7,9	11,3	7,4	2,6	**0,3**	0,5	**0,3**	-	-	-	-	-

Table 1: Number (ratio) of solved instances, IPC score, average maximal number of agents in a solved scenario, average number of conflicts, and average number of constraints. The results are split by the map size. Strategies are *baseline* (B), *prune-and-cut* (P), *makespan-add* (M), and *combined* (C).

without preprocessing, since these results were not competitive at all. For all of the underlying reduction-based solvers, we used all of the four proposed strategies. Lastly, we also include results obtained by CBS. The results are split by the size of the maps.

The number of solved instances (absolute value and ratio to the total number of instances) indicates that the most successful combination is ASP with preprocessing with **C**. This combination was the most successful on all sizes and in fact on all of the types of instances. The strategies are clearly ordered with **C** being the most successful, closely followed by **M** and **P** with **B** being the worst. This ordering is maintained for all of the underlying solvers. Also using the preprocessing maintains this ordering.

While **B** is similarly successful on the smallest instances, With the increase of the map size, the success ratio of **B** falls drastically, while for all of the remaining proposed strategies the ratio decreases only slightly. This is an indication that the motivation of the strategies – to make reduction-based approaches more competitive on large maps – is indeed fulfilled. Also note, that for the CBS the ratio increases with the increase of the map size. This also indicates that the intuition that CBS is more successful on sparse maps is valid.

The IPC score (introduced at International Planing Competition, hence the name) is computed as 0 if the solver did not finish in time, otherwise 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 are summed in Table 1. The IPC score tells a similar story as the solved instances. The ordering of the strategies is maintained and the preprocessing also helps. On the other hand, we can see that **B** lacks behind even on the smallest instances. This means that while it managed to solve many of the smallest instances, it did not solve them fast. This trend can also be seen in Figure 10 which shows the instances ordered by their runtime. For better readability, we omitted the results of SAT-based solver, since they are not competitive.

Among all algorithms and map sizes, the maximum number of successfully planned agents was 225, solved for some small maps by ASP with **P** and **C**. The average number of maximum agents

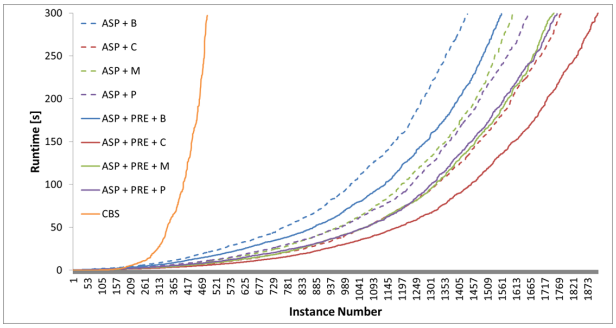


Figure 10: Number of instances (x-axis) solved in a given time-limit (y-axis).

per scenario solved on different map sizes is shown in Table 1. The number falls with the increase in map size as excepted. Again, we observe that the decrease of the maximal number of agents solved is higher for strategy **B** than the other strategies, even though **B** solves the most agents on the smallest maps. On larger maps, strategy **P** with ASP with preprocessing is the most successful optimal approach, while strategy **C** with ASP with preprocessing is the most successful overall. The least successful approach on large maps is SAT with **B** followed by CBS. If we inspect the results of individual instances (not included in the table due to readability and space limitations, but available in our repository[2]), we can see that the easiest map types (ie. most solved agents) are empty and random, while for maps maze and room the algorithms were able to solve only instances with fewer agents.

The SAT-based solver lacks behind ASP-based solver significantly, with CBS reaching similar results as the SAT-based solver. We conjecture that the reason for the poor performance of CBS is threefold. (1) The map types maze and room are much more challenging for CBS because of the extra congestion [14, 28]. The success ratio and IPC score for empty and random are much higher for CBS than for the other types of maps. (2) The scenario setting

where agents' paths cross each other is more challenging for the same reason. Again, we can see this in both the IPC score and success ratio. We do not include map types and agent start/goal locations in Table 1 since, for the other solvers, the difference is less significant. (3) The two previous reasons are based on our observation of the results, however, even on the favorable setting, the CBS lacks behind the ASP-based solver. We conjecture that the makespan objective is harder for CBS than the sum of cost objective which is the objective used in most CBS publications. There are significantly more solutions with the same makespan than with the same sum of costs making the best-first search of CBS less prominent.

The ordering of the performance of the strategies can be explained by looking at the average number of vertices and number of calls to the underlying solver. Strategies **C**, **M**, and **P** used on average 22%, 24%, and 20% of the vertices used by **B**. For **M** it was unnecessary in some cases to start with $k = 1$. On the other hand in the cases where m needed to be increased, **P** needed to solve greater number of relaxed instances. **C** takes the best of both, which explains its prominent results.

As for optimality, both **B** and **P** are optimal. **C** found an optimal solution in 85% of cases, while **M** found an optimal solution in 76% of cases. When the found solution was not optimal, it was on average only 4% and 6.4% over the optimum for **C** and **M** respectively.

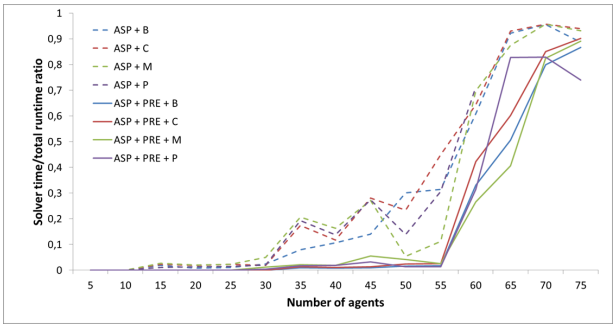


Figure 11: Solver time to total runtime ratio per number of agents for specific scenario *maze* 32 by 32.

Since our strategies performed best in conjunction with ASP-based solving, we focus on an in-depth analysis of the related benchmark runs, as summarized in the first two main columns of Table 1. The last two main rows state, per *clingo* solve call, the average number of search conflicts and the size of the internal problem representation in terms of number of constraints. At first, let us consider the strategies without preprocessing, detailed in the left-most main column of Table 1. In comparison to our baseline **B**, the strategies using a graph pruning (viz. **P**, **M** and **C**) show a significant gain in solved instances and IPC score, for medium and large instances. E.g. for large instances (size 128), **B** solves 162 instances and has IPC score of 24.8 whereas the graph pruning, optimal strategy **P** solves 287 instances and has an IPC score of 147.9. Similarly, the graph pruning strategies reduce the number of constraints by one order of magnitude, for large instances, e.g. **B** has 44.2 million constraints whereas **P** has only 7.8 million. Secondly, augmenting the strategies with preprocessing, detailed in the mid column of Table 1, leads to another substantial increase of solved instances and IPC scores. Analogously, preprocessing reduces the number of constraints by an additional order of magnitude. E.g. for large instances, **P** improves from 286 to 315 solved instances, from 147.9 to 203.1 IPC score and from 7.8 down to 0.3 million constraints. Independent of the employed strategy or preprocessing, the number of encountered conflicts, an indicator for the complexity of *clingo*'s search for a solution, stays very low, ranging between 67 and 760 conflicts. Based on these observations, we conjecture that — on their own and in combination — the graph pruning strategies and the addition of preprocessing information lead to a significant simplification of *clingo*'s grounding of the ASP input program which is, eventually, reflected by the substantial decrease of the internal problem size and overall runtime improvement. Hence, in general, the ASP computation seems to be dominated by grounding whereas the search difficulty is trivial. However, this trend does not apply to every single problem instance: a closer look into the consecutive test runs on single scenarios, such as the specific maze 32 by 32 scenario in Figure 11, shows that the ratio between solve time and overall runtime will gradually increase, over the course of adding more and more agents. That is, as witnessed in [14], a higher congestion of agents will generally lead to a higher number of potential agent collisions and, by that, to more conflicts during search. Considering that ASP typically outperforms search-based MAPF solvers for highly congested maps, we selected the top 20 instances with the most search conflicts still solvable before timeout by ASP with our best-performing, optimal strategy **P** and preprocessing. Out of those instances, 6 could be also solved by ASP with **P** but without preprocessing, 8 by SAT with preprocessing and **P**, and 2 by CBS.

6 DISCUSSION

We proposed several graph pruning strategies to mitigate the lack of scalability of reduction-based MAPF solvers on large maps. We did this by removing some of the vertices from the map thus lowering the number of variables entering the underlying solvers. We showed that one of the strategies maintains completeness and optimality. In our empirical evaluation, the strategies were able to solve more instances than both the baseline approach and the CBS algorithm. The best performance was achieved by the suboptimal *combined* strategy that while suboptimal was able to find an optimal solution in most cases and was on average only 4% away from optimum.

In future work, we aim to modify the strategies to be able to find the sum of costs optimal solutions rather than makespan optimal to provide a better comparison to search-based algorithms.

ASP is the most effective approach in conjunction with our strategies. However, in most cases the computation is dominated by (re)-grounding of the problem. Fortunately, *clingo* offers an advanced methodology called *multi-shot solving* [8] that supports the operative processing of multiple consecutive problems. Instead of restarting *clingo* whenever we relax the instance, we may reuse the process by directly updating its internal knowledge base.

ACKNOWLEDGMENTS

Research is supported by project P103-19-02183S of the Czech Science Foundation, the Czech-USA Cooperative Scientific Research Project LTAUSA19072, and DFG grant SCHA 550/15, Germany.

REFERENCES

[1] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. 2018. Robust Multi-Agent Path Finding. In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, Vadim Bulitko and Sabine Storandt (Eds.). AAAI Press, 2–9. https://aaai.org/ocs/index.php/SOCS/SOCS18/paper/view/17954

[2] Roman Barták and Jirí Svancara. 2019. 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*, Pavel Surynek and William Yeoh (Eds.). AAAI Press, 10–17. https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18323

[3] Roman Barták, Neng-Fa Zhou, Roni Stern, Eli Boyarski, and Pavel Surynek. 2017. 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*. IEEE Computer Society, 959–966. https://doi.org/10.1109/ICTAI.2017.00147

[4] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.

[5] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. 2015. 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*. 740–746. http://ijcai.org/Abstract/15/110

[6] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele. 2019. *Potassco User Guide* (version 2.2.0 ed.). http://potassco.org

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. 2012. *Answer Set Solving in Practice*. Morgan and Claypool Publishers.

[8] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1 (2019), 27–82. https://doi.org/10.1017/S1471068418000054

[9] M. Gebser, P. Obermeier, T. Otto, T. Schaub, O. Sabuncu, V. Nguyen, and T. Son. 2018. Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 502–519. https://doi.org/10.1017/S1471068418000200

[10] M. Gebser, P. Obermeier, T. Schaub, M. Ratsch-Heitmann, and M. Runge. 2018. Routing Driverless Transport Vehicles in Car Assembly with Answer Set Programming. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 520–534.

[11] M. Gebser, P.Obermeier, T. Schaub, and P. Wanko. 2020. Collection of ASP encodings for asprilo. https://github.com/potassco/asprilo-encodings

[12] R. Geraerts. 2010. Planning short paths with clearance using explicit corridors. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'10)*. IEEE, 1997–2004. https://doi.org/10.1109/ROBOT.2010.5509263

[13] R. Geraerts and M. Overmars. 2007. The Corridor Map Method: Real-Time High-Quality Path Planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'07)*. IEEE, 1023–1028. https://doi.org/10.1109/ROBOT.2007.363119

[14] Rodrigo N. Gómez, Carlos Hernández, and Jorge A. Baier. 2021. A Compact Answer Set Programming Encoding of Multi-Agent Pathfinding. *IEEE Access* 9 (2021), 26886–26901. https://doi.org/10.1109/ACCESS.2021.3053547

[15] R. Kaminski, J. Romero, T. Schaub, and P. Wanko. 2020. How to build your own ASP-based system?! *CoRR* abs/2008.06692 (2020). https://arxiv.org/abs/2008.06692

[16] Daniel Kornhauser, Gary L. Miller, and Paul G. Spirakis. 1984. 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*. 241–250. https://doi.org/10.1109/SFCS.1984.715921

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

[18] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. 2020. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, Amal El Fallah Seghrouchni, Gita Sukthankar, Bo An, and Neil Yorke-Smith (Eds.). International Foundation for Autonomous Agents and Multiagent Systems, 1898–1900. https://dl.acm.org/doi/abs/10.5555/3398761.3399020

[19] Vladimir Lifschitz. 2019. *Answer Set Programming*. Springer. https://doi.org/10.1007/978-3-030-24658-7

[20] Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. 2017. Generalized Target Assignment and Path Finding Using Answer Set Programming. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 1216–1223. https://doi.org/10.24963/ijcai.2017/169

[21] Daniel Ratner and Manfred K. Warmuth. 1990. NxN Puzzle and Related Relocation Problem. *J. Symb. Comput.* 10, 2 (1990), 111–138. https://doi.org/10.1016/S0747-7171(08)80001-6

[22] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. 2012. 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*. http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5062

[23] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. 2011. 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*. 662–667. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-117

[24] David Silver. 2005. Cooperative Pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. 117–122.

[25] 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. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, Pavel Surynek and William Yeoh (Eds.). AAAI Press, 151–159. https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18341

[26] Pavel Surynek. 2019. Lazy Compilation of Variants of Multi-robot Path Planning with Satisfiability Modulo Theory (SMT) Approach. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China, November 3-8, 2019*. IEEE, 3282–3287. https://doi.org/10.1109/IROS40897.2019.8967962

[27] Pavel Surynek. 2019. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 1177–1183. https://doi.org/10.24963/ijcai.2019/164

[28] Jirí Svancara and Roman Barták. 2019. 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*, Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik (Eds.). SciTePress, 226–231. https://doi.org/10.5220/0007470002260231

[29] Jingjin Yu and Steven M. LaValle. 2013. 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*. http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6111

[30] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. 2015. *Constraint Solving and Planning with Picat*. Springer. https://doi.org/10.1007/978-3-319-25883-6