Which Objective Function is Solved Faster in Multi-Agent Pathfinding? It Depends

Jiří Švancara[1], Dor Atzmon[2], Klaus Strauch[3], Roland Kaminski[3] and Torsten Schaub[3]
[1]*Charles University, Czech Republic*
[2]*Bar-Ilan University, Israel*
[3]*University of Potsdam, Germany*

Keywords: Multi-Agent Pathfinding, Sum of Costs, Makespan, Search-Based Algorithm, Reduction-Based Algorithm.

Abstract: Multi-agent pathfinding (MAPF) is the problem of finding safe paths for multiple mobile agents within a shared environment. This problem finds practical applications in real-world scenarios like navigation, warehousing, video games, and autonomous intersections. Finding the optimal solution to MAPF is known to be computationally hard. In the literature, two commonly used cost functions are makespan and the sum of costs. To tackle this complex problem, various algorithms have been developed, falling into two main categories: search-based approaches (e.g., Conflict Based Search) and reduction-based approaches, including reduction to SAT or ASP. In this study, we empirically compare these two approaches in the context of both makespan and the sum of costs, aiming to identify situations where one cost function presents more challenges than the other. We compare our results with older studies and improve upon their findings. Despite these solving approaches initially being designed for different cost functions, we observe similarities in their behavior. Furthermore, we identify a tipping point related to the size of the environment. On smaller maps, the sum of costs is more challenging, while makespan poses greater difficulties on larger maps for both solving paradigms, defying intuitive expectations. Our study also offers insights into the reasons behind this behavior.

1 INTRODUCTION

Multi-agent pathfinding (MAPF) is the task of maneuvering a group of agents within a shared environment, ensuring they navigate without colliding. This problem has practical applications in various domains, including warehousing (Ma et al., 2017), robotics (Bennewitz et al., 2002), navigation (Dresner and Stone, 2008), and potentially the coordination of autonomous vehicles in the foreseeable future.

Since the agents collaborate, there are situations where an agent may prolong its path to facilitate another's passage, ultimately leading to a better global solution. To enable such cooperation, a centralized planner is typically employed, guaranteeing the optimality of the plan according to a desired cost function. Note that, decentralized planners exist but cannot ensure optimal solutions due to the problem's inherent combinatorial complexity (Alonso-Mora et al., 2010; Sartoretti et al., 2019).

In this paper, we explore the behavior and performance of two popular methods for achieving optimal MAPF solutions: search-based and reduction-based approaches, considering the two most commonly used cost functions, makespan and the sum of costs. While both approaches are capable of optimizing either cost function it is noteworthy that each of them was originally designed with a specific cost function in mind. Specifically, the search-based algorithm CBS was tailored for optimizing the sum of costs (Sharon et al., 2015), while the reduction-based solver was primarily geared towards optimizing makespan (Surynek, 2012). Intuition might suggest that each solver excels at optimizing the cost function it was initially designed for but we show that this is not always the case.

The contributions of this paper are threefold:

(1) We show that while both search-based and reduction-based solvers can find makespan and sum of costs optimal solutions, on smaller maps it is more challenging to find the sum of costs optimal solution, while on large maps it is more challenging to find the makespan optimal solution for both approaches. We show this empirically and provide insight into the workings of the algorithms that explain this behavior.

(2) We compare our results with a previous studies which concludes that makespan is always easier for

Švancara, J., Atzmon, D., Strauch, K., Kaminski, R. and Schaub, T.
Which Objective Function is Solved Faster in Multi-Agent Pathfinding? It Depends.
DOI: 10.5220/0012249400003636

In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024) - Volume 3*, pages 23-33
ISBN: 978-989-758-680-4; ISSN: 2184-433X

both approaches (Surynek et al., 2016b; Gómez et al., 2021). We arrive at different conclusions and explain why the results are different, mainly focusing on the size of the experimental setup.

(3) For the reduction-based solvers, we include two different approaches to find the sum of costs optimal solution based on previous work (Barták and Svancara, 2019). Again, we arrive at different conclusions, in terms of the performance of the proposed models, and explain why the results are different. The difference is again mainly due to the experiment size.

On the other hand, we do not deeply compare the performance of the solvers against each other, as our implementations are not state-of-the-art for all of the solvers, and as such, the comparison would be unfair.

2 DEFINITIONS

The *Multi-Agent Pathfinding* problem (MAPF) (Stern et al., 2019) is a pair (G,A), where G is an undirected graph $G=(V,E)$ and A is a list of n agents $A=(a_1,\ldots,a_n)$. Each agent $a_i \in A$ is associated with a start vertex $s_i \in V$ and a goal vertex $g_i \in V$. Time is considered discrete and between two consecutive timesteps, an agent can either move to an adjacent vertex (move action) or stay at its current vertex (wait action). The movement of an agent is captured by its path. A *path* π_i of agent a_i is a list of vertices that starts at s_i and ends at g_i. Let $\pi_i(t)$ be the vertex (i.e. location) of a_i at timestep t according to π_i. Therefore, $\pi_i(0)=s_i$, $\pi_i(|\pi_i|)=g_i$, and for all timesteps t, $(\pi_i(t),\pi_i(t+1)) \in E$ or $\pi_i(t)=\pi_i(t+1)$, i.e. at each timestep agent a_i either moves over an edge or waits in its current vertex, respectively.

As there are several agents, we are interested in the interaction of each pair of paths. A tuple $\langle a_i,a_j,x,t\rangle$ represents a *conflict* between paths π_i and π_j at timestep t if $\pi_i(t)=\pi_j(t)$ (*vertex conflict* at vertex $x=\pi_i(t)$) or $\pi_i(t)=\pi_j(t+1) \wedge \pi_j(t)=\pi_i(t+1)$ (*swapping conflict* over edge $x=(\pi_i(t),\pi_i(t+1))$). A *plan* Π is a list of n paths $\Pi=(\pi_1,\ldots,\pi_n)$, one for each agent. A *solution* is a conflict-free plan Π, where no two paths of distinct agents have any conflicts.

A solution is *optimal* if it has the lowest cost among all possible solutions. The cost $C(\pi_i)$ of path π_i equals the number of actions performed in π_i until the last arrival at g_i, not counting any subsequent wait actions. Formally, $C(\pi_i)=\max(\{0<t\leq|\pi_i| \mid \pi_i(t)=g_i,\pi_i(t-1)\neq g_i\}\cup\{0\})$. Note that waiting at the goal counts towards the cost if the agent leaves the goal at any time in the future.

There are two commonly used cost functions to evaluate a plan's Π quality: (1) *sum of costs* (SOC),

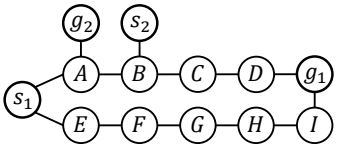

Figure 1: Example of SOC and MKS optimal solutions.

which is the sum of costs of all paths ($C_{SOC}(\Pi)=\sum_i C(\pi_i)$); (2) *makespan* (MKS), which is the maximum cost among all paths ($C_{MKS}(\Pi)=\max_i C(\pi_i)$).

Solving MAPF optimally for either sum of costs or makespan is known to be NP-hard (Yu and LaValle, 2013; Surynek, 2010). Figure 1Example of SOC and MKS optimal solutions. presents a MAPF problem instance with two agents a_1 and a_2. Here, the optimal solution for sum of costs is $\pi_1=(s_1,E,F,G,H,I,g_1)$ and $\pi_2=(s_2,B,A,g_2)$, which yields $C_{SOC}(\Pi)=9$ and $C_{MKS}(\Pi)=6$. The optimal solution for makespan is $\pi_1=(s_1,A,B,C,D,g_1)$ and $\pi_2=(s_2,s_2,s_2,B,A,g_2)$, which yields $C_{SOC}(\Pi)=10$ and $C_{MKS}(\Pi)=5$. This illustrates that optimizing one cost function may increase the other.

3 SEARCH-BASED SOLVER

Optimally solving MAPF can be naively performed using a simple heuristic search algorithm, such as A* (Hart et al., 1968), executing on a shared search space containing all agents. In such a search space, each node contains the locations of all agents at a specific timestep. The start node contains all start vertices, the goal node contains all goal vertices, and a transition between two nodes represents any possible combination of movement of all agents. As the number of agents increases, the number of such combinations increases exponentially. Therefore, problems with more than a few agents become unsolvable in practice when using such a coupled approach.

Conflict-Based Search (CBS) (Sharon et al., 2015) is a prominent decoupled search-based algorithm for optimally solving MAPF that overcomes the exponential increase described above by finding paths for the agents separately and resolving conflicts iteratively by imposing constraints on conflicting agents until a conflict-free plan is found.

CBS builds a *Constraint Tree* (CT), where each CT node N contains a set of constraints $constraints(N)$, a plan $\Pi(N)$, and a cost $cost(N)$ (i.e. $C(\Pi(N))$). A constraint $\langle a_i,x,t\rangle$ prohibits agent a_i from occupying vertex x at timestep t or traversing edge x between timesteps t and $t+1$. The search of a path for an agent under a set of constraints is called a *low-level* search and can be performed by searching

Algorithm 1: High level of CBS.

```
1  CBS (MAPF problem instance)
2      Init OPEN
3      Init Root with an initial plan and no
         constraints
4      Insert Root into OPEN
5      while OPEN is not empty do
6          N ← Pop the node with the lowest cost in
             OPEN // according to SOC or MKS
7          if Π(N) is conflict-free then
8              return Π(N)
9          ⟨a_i, a_j, x, t⟩ ← get-conflict(N)
10         N_i ← GenerateChild(N, ⟨a_i, x, t⟩)
11         N_j ← GenerateChild(N, ⟨a_j, x, t⟩)
12         Insert N_i and N_j into OPEN
13     return No Solution
14 GenerateChild (Node N, Constraint ⟨a', x, t⟩)
15     constraints(N') ← constraints(N) ∪ {⟨a', x, t⟩}
16     Π(N') ← Π(N)
17     Update Π(N') to satisfy constraints(N')
18     cost(N') ← C(Π(N'))
19     return N'
```

in a space-time configuration, e.g., using Space-Time A* (Silver, 2005). Space-Time A* is similar to the classical A*, but it also maintains time and can return a path that satisfies a given set of constraints.

The search performed on the CT is called a *high-level* search. The pseudo-code of the high-level search of CBS is presented in Algorithm 1. The search begins by initializing OPEN and a root node *Root* with no constraints (lines 2-3). The plan $\Pi(Root)$ is planned by calling a chosen low-level solver for each agent. The leaves of the CT are maintained in the priority queue OPEN, ordered by their costs (according to sum of costs or makespan), and OPEN is initialized with *Root* (line 4). Then, the high-level search performs the following expansion cycle (lines 5-14). CBS extracts the node N with the lowest cost from OPEN (line 6) and performs a solution check. If $\Pi(N)$ is conflict-free, N is a solution, and its plan is returned (lines 7-8). If $\Pi(N)$ is not conflict-free, a conflict $\langle a_i, a_j, x, t \rangle$ in $\Pi(N)$ is chosen to be resolved (line 9). As both agents a_i and a_j cannot occupy x together at timestep t, CBS generates two new nodes N_i and N_j (lines 10-11). When the new node N' is generated, it sets $constraints(N') = constraints(N) \cup \{\langle a', x, t \rangle\}$ (line 15). Then, it calls the low-level solver to replan for the constrained agent a', and recomputes the updated plan cost (lines 16-18). N_i and N_j are, therefore, generated with the additional constraints $\langle a_i, x, t \rangle$ and $\langle a_j, x, t \rangle$, respectively. If OPEN is empty, *No Solution* is returned (line 13). Note that CBS can optimally solve MAPF for either sum of costs or makespan and the only modification for this purpose is ordering OPEN according to the selected cost function.

In recent years, different improvements were developed for CBS, including prioritizing conflicts (Boyarski et al., 2015), heuristics for CBS (Felner et al., 2018), and symmetry-breaking (Li et al., 2019). We do not discuss these here as they all aim to improve CBS only to minimize the sum of costs. Including these improvements would most likely affect the results in favour of sum of costs, however, we aim to explore the behaviour of the base algorithm.

4 REDUCTION-BASED SOLVERS

Reduction-based approaches translate the input problems into another formalism, such as Boolean satisfiability (SAT) (Surynek, 2017), answer set programming (ASP) (Nguyen et al., 2017), or integer linear programming (ILP) (Yu and LaValle, 2012). We describe reductions to SAT and ASP. However, all of the reduction procedures are based on the idea of creating variables representing agents' positions at a time.

Reduction to SAT. To model the positions and transitions of the agents, variables $At(t, a_i, v)$ and $Pass(t, a_i, (u, v))$ are created, representing that at time t, agent a_i is located at vertex v, and that at time t, agent a_i is moving along edge (u, v), respectively. Note that loop edges (v, v) for all $v \in V$ are added to E to model wait actions. Given some bound on the number of timesteps (i.e., the makespan) T, the following constraints are created. Note that several different encodings exist in the literature (Zhou and Barták, 2017; Surynek, 2017; As'in Ach'a et al., 2021). For this paper, we follow the one in (Barták and Svancara, 2019).

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

$$\forall a_i \in A : At(T, a_i, g_i) \tag{2}$$

$$\forall 0 \leq t \leq T, a_i \in A, u, v \in V, u \neq v : \neg At(t, a_i, u) \vee \neg At(t, a_i, v) \tag{3}$$

$$\forall 0 \leq t < T, u \in V, a_i \in A : At(t, a_i, u) \implies \bigvee\nolimits_{(u,v) \in E} Pass(t, a_i, (u, v)) \tag{4}$$

$$\forall 0 \leq t < T, (u, v) \in E, a_i \in A : Pass(t, a_i, (u, v)) \implies At(t+1, a_i, v) \tag{5}$$

$$\forall 0 \leq t \leq T, v \in V, a_i, a_j \in A, a_i \neq a_j : \neg At(t, a_i, v) \vee \neg At(t, a_j, v) \tag{6}$$

$$\forall 0 \leq t < T, (u, v) \in E, a_i, a_j \in A, a_i \neq a_j : \neg Pass(t, a_i, (u, v)) \vee \neg Pass(t, a_j, (v, u)) \tag{7}$$

Constraints (1) – (7) ensure that the movement of each agent is valid and that there are no conflicts

among the agents. Specifically, (1) and (2) ensure that each agent starts and ends at s_i and g_i respectively. (3) forbids an agent to be located at two vertices at the same time. (4) states that if an agent is present at a vertex, it leaves through one of the outgoing edges. (5) ensures that if an agent is moving along an edge, it arrives at the correct vertex at the next timestep. Together, constraints (1) – (5) ensure that each agent is moving along a *path*. To make sure there are no conflicts among the paths, (6) forbids vertex conflicts, while (7) forbids swapping conflicts.

As the plan length is unknown in advance, T is iteratively increased until a solvable formula is produced. This produces a makespan optimal solution.

Two important enhancements are used. Let $dist(u,v)$ denote the shortest distance between vertices u and v and let $D_i = dist(s_i, g_i)$ be the shortest distance from the start to the goal of agent a_i. The initial T is then set as $max_{a_i \in A} D_i$ as it is clear that at least this number of timesteps is needed to find a solution. Secondly, only variables representing reachable positions are created. A variable $At(t, a_i, v)$ is created only if $dist(s_i, v) \leq t$ and $dist(v, g_i) < T - t$, meaning that there is enough time for a_i to move from start s_i to vertex v and enough remaining time to move from v to goal vertex g_i. A variable $Pass(t, a_i, (u,v))$ is created only if both $At(t, a_i, u)$ and $At(t+1, a_i, v)$ exist.

Changing the objective function is not as straightforward in the reduction-based approach as compared to CBS. To model sum of costs, the same $At(t, a_i, v)$ and $Pass(t, a_i, (u,v))$ variables are created. As opposed to the makespan optimal version, we do not create a global bound on timesteps T, but rather each agent has their limit T_i. The same constraints (1) – (7) are used with a slight modification to (2) as follows:

$$\forall a_i \in A : At(T_i, a_i, g_i) \tag{8}$$

To limit the sum of costs, a numerical constraint is introduced stating that at most k extra actions may be used (Surynek et al., 2016a). An extra action refers to an action that is performed after the timestep D_i by agent a_i. We iteratively increase k and each T_i by one until a solvable formula is created. Specifically, after δ iterations, the plan is allowed to contain δ extra actions, however, since we do not know which agent uses these actions, we increase all T_is so that any agent can use them. It can be shown that this approach yields a sum of costs optimal solution (Surynek et al., 2016a), where the final cost is $C_{SOC}(\Pi) = k + \sum D_i$. We refer to this model as *iterative*.

An alternative approach to finding the appropriate k has been proposed (Barták and Svancara, 2019). First, a makespan optimal solution Π_{MKS} is found. This solution has some (most likely suboptimal) sum of costs value $C_{SOC}(\Pi_{MKS})$. It can be shown that setting $k = C_{SOC}(\Pi_{MKS}) - \sum D_i$ is sufficient to find the sum of costs optimal solution. The optimal solution is found by a branch-and-bound search on sum of costs values (i.e. the number of extra actions) in the interval $[0, C_{SOC}(\Pi) - \sum D_i]$. We refer to this model as *jump*.

Similar enhancements as the ones used for makespan optimization are used for both *iterative* and *jump* models. The initial T_i is set as D_i and only variables representing a reachable position are created. Furthermore, variables $At(t, a_i, g_j)$ are not created for $a_i \in A, t \geq D_j + k$ to prevent any agent entering a goal g_j of agent a_j after it finishes their plan.

4.1 Reduction to ASP

For the reduction to ASP, we use the encoding in Listing 1 to model bounded MAPF problems. In the following, we only give a high level description of the rules used in this encoding; we refer the interested reader to (Gebser et al., 2015) for a precise ASP semantics.

```
1  #program sum_of_costs.
2  horizon(A,H+D) :- dist(A,H), delta(D).
3  #program makespan.
4  horizon(A,H) :- agent(A), makespan(H).
5  #program mapf.
6  time(A,1..T) :- horizon(A,T).

8  {move(A,U,V,T): edge(U,V), reach(A,V,T)} 1
9   :- reach(A,U,T-1).
10 at(A,V,0) :- start(A,V), agent(A).
11 at(A,V,T) :- move(A,_,V,T).
12 at(A,V,T) :- at(A,V,T-1), not move(A,V,_,T),
13              time(A,T).
14 :- move(A,U,_,T), not at(A,U,T-1).
15 :- at(A,V,T), not reach(A,V,T).
16 :- {at(A,V,T)} != 1, time(A,T).

18 :- {at(A,V,T)} > 1, vertex(V), time(_,T).
19 :- move(_,U,V,T), move(_,V,U,T), U<V.
20 :- goal(A,V), not at(A,V,H), horizon(A,H).
```

Listing 1: ASP encoding for bounded MAPF.

The encoding assumes as input a MAPF problem given by predicates `vertex`, `edge`, `agent`, `start`, and `goal`. The objective function is given by predicate `makespan` for the makespan objective, and predicates `dist` and `delta` for the sum of costs objective. Finally, predicate `reach` gives reachable vertices at time points computed as described in the previous section. The rules in lines 1–6 setup the required timesteps and horizons for each agent as determined by the objective function. In lines 8–9, we generate a set of move candidates only considering reachable vertices; there can be up to one move per agent. Based on this predicate, the agent locations are inferred by the following

three rules. The first establishes the initial positions of agents, the next specifies the effects of moves, and the last encodes inertia in case an agent is not moved. The following three integrity constraints in lines 14–16 prune invalid solution candidates. The first discards candidates having moves without an agent at its source, the second discards candidates with agents at unreachable positions, and the last ensures that an agent has exactly one position at all time points. In the last block of rules in lines 18–20, we ensure that a solution corresponds to a plan. The first rule ensures that the solution has no vertex conflict, the second that there is no swapping conflict, and the last one that each agent reaches its goal vertex.

We implement the same approaches to compute makespan and sum of costs optimal solutions (including the *iterative* and *jump* variants) as described above. To compute makespan optimal solutions, the encoding can be used as is. However, for sum of costs, additional rules are needed. The following three rules are used to accumulate a penalty for each agent not located at its goal vertex at a timestep:

```
penalty(A,N) :- dist(A,N+1), N>=0.
penalty(A,T) :- dist(A,N), at(A,U,T),
                not goal(A,U), T>=N.
penalty(A,T) :- penalty(A,T+1), T>=0.
```

For the *iterative* approach, we then add the following integrity constraints to ensure that a solution is indeed sum of costs optimal:

```
bound(H+D) :- H=#sum{T,A: dist(A,T)}, delta(D).
:- #sum{1,A,T: penalty(A,T)} > B, bound(B).
```

Finally, for the *jump* approach, we simply use ASP's inbuilt optimization facilities to minimize the accumulated penalties:

```
#minimize{1,A,T : penalty(A,T)}.
```

5 EXPERIMENTS

Instance Setup. To test and evaluate the behavior of the search-based algorithm CBS and the two reduction-based algorithms w.r.t. both described cost functions, we created a set of experiments inspired by the commonly used benchmark set (Stern et al., 2019). We created a variety of 4-connected grid maps with different obstacle structures and with increasing size. There are two types of maps with regards to the obstacle placement – *empty*, i.e. there are no obstacles placed, and *random*, i.e., 20% of randomly selected vertices are marked as impassable obstacles. The size of the maps starts with grids of size 8 by 8 and increases by 8 (i.e., 8 by 8, 16 by 16, 24 by 24, etc.) until 64 by 64. As we show below, the increase in size has a high impact on the performance of the algorithms, for this reason, we created our own maps with a finer increment as opposed to using already existing instances from the benchmark set, which does not contain such a set of maps.

Table 1: The number of solved instances. The results are split based on the used cost function and based on the size of the instance grid graph. For each solver and map size, the most number of solved instances is highlighted.

	CBS		SAT			ASP		
	mks	soc	mks	soc iter	soc jump	mks	soc iter	soc jump
8	**196**	171	**461**	218	194	**457**	243	241
16	**362**	270	**1526**	445	230	**1660**	461	359
24	**373**	313	**1433**	505	197	**2088**	572	415
32	323	**375**	**873**	564	114	**1292**	645	448
40	310	**441**	470	**677**	80	666	**721**	411
48	263	**473**	321	**543**	77	398	**671**	292
56	299	**524**	241	**559**	41	297	**670**	235
64	293	**510**	149	**580**	33	134	**645**	122
total	2419	3077	5474	4091	966	6992	4628	2523

For each map, we created 5 scenario files, resulting in 80 scenarios in total. Each scenario file contains different start and goal locations of agents. The intended use is to create an instance with one agent from the scenario, and if solvable in the given time limit, add further agents creating new instances. This process is repeated until the solver is unable to solve an instance with the given number of agents in the given time limit. The time limit is set to 60 seconds per instance. The number of agents in each scenario file was chosen so that no solver was able to solve all of the agents, with the exception of the 8 by 8 maps which include only 50 agents. We mention this below when relevant while describing the results.

The tests were performed on a desktop computer with Intel® Core™ i5-6600 CPU @ 3.30GHz × 4 and 16GB of RAM. We used an implementation of CBS from (Boyarski et al., 2015), for the "at most K" constraint we used PBLib (Philipp and Steinke, 2015), the underlying SAT solver used is Kissat (Biere et al., 2020), the underlying ASP solver used is Clingo (Kaminski et al., 2020). All source codes and results are available online – https://github.com/svancaj/mks_vs_soc.

Results. Table 1The number of solved instances. The results are split based on the used cost function and based on the size of the instance grid graph. For each solver and map size, the most number of solved instances is highlighted. shows the number of solved instances each solver was able to solve within the time limit. The results are split based on the solver used, the optimized cost function (in case of reduction to

SAT and ASP, we present both of the described approaches for solving the sum of costs optimization), and the size of the input instance map. We do not split the results based on the obstacle structure as this does not provide any significant insights. We observed that the results for *empty* and *random* maps are similar.

Based on the presented numbers, we see that for the smaller maps (8 by 8 through 24 by 24), both of the approaches were more successful using the makespan optimization, while for the larger maps (40 by 40 and larger) the advantage heavily shifts to optimizing the sum of costs. The turning points are the maps of size 32 by 32 for CBS and 40 by 40 for the reduction-based solvers.

The *jumping* model was always outperformed by the *iterative* model for both reductions, ASP and SAT. The results were close only for maps of size 8 by 8.

As we already stated, the aim of this paper is not to compare the algorithms against each other as the performance may be affected by our implementation. Nevertheless, we can see that for the smaller maps, the reduction-based approaches solve significantly more instances compared to CBS, especially under the makespan objective. On the other hand, as the size of the maps increases (from 40 by 40 onward) the number of instances solved by the reduction-based approaches decreases, while the number of instances solved by CBS tends to increase under the sum of costs optimization. The phenomenon that reduction-based solvers struggle with large instances is explored by other studies (Husár et al., 2022), even though they explore only the makespan optimization. The tendency of CBS is intuitive: as there are fewer chances for collision on large maps, fewer conflicts need to be resolved and, thus, a smaller CT is created. But, again, this is explored only for the sum of costs optimization in the literature (Sharon et al., 2015).

We compare our results to the results presented in a previous works focusing on empirical evaluation of the sum of costs and makespan optimal solvers (Surynek et al., 2016b). In the paper, one of the main conclusions is that "results show that the makespan optimal variant tends to be easier (except for the EPEA* solver), all the other solvers are faster in their makespan optimal configuration." The experimental evaluation used grid maps of sizes 16 by 16 and a smaller set of experiments on maps of size 6 by 6 and a 4-dimensional hypercube (such a hypercube has 16 vertices). We see that on these smaller maps, our results are comparable. However, as we show above, we can extend these results and observe a switch in performance as the maps' size increases. Similarly, a work focusing on ASP encodings for MAPF (Gómez et al., 2021) concludes that "ASP-makespan scales substantially better than ASP-cost". Again, the experiments are performed only on small grid maps of size 20 by 20.

6 DISCUSSION

Search-Based CBS. As mentioned above, CBS mainly performed better for minimizing sum of costs than for minimizing makespan, except for tiny maps. Here, we try to explain this phenomenon.

Let Π_{SOC} and Π_{MSK} denote the optimal solutions for minimizing the sum of costs and makespan, respectively, and $C_{SOC}(\Pi)$ and $C_{MSK}(\Pi)$ denote the sum of costs and makespan of a given plan Π, respectively. Clearly, $C_{SOC}(\Pi_{SOC}) \leq C_{SOC}(\Pi_{MSK})$. According to (Boyarski et al., 2021), when a conflict is resolved, the sum of costs of the child node may increase by one relative to the sum of costs of its parent node. Conflicts that increase the sum of costs in both child nodes are known as *cardinal* conflicts, conflicts that increase the sum of costs in only one child node are known as *semi-cardinal* conflicts, and other conflicts are called *non-cardinal* conflicts (Boyarski et al., 2015).[1] Therefore, any plan Π' can only be found in the CT at a depth *greater than or equal to* $\Delta_{\Pi'} = C_{SOC}(\Pi') - C_{SOC}(\Pi(Root))$. $\Delta_{\Pi'}$ can be seen as a lower bound on the depth of the closest CT node in the CT that contains plan Π'. As mentioned above, $C_{SOC}(\Pi_{SOC}) \leq C_{SOC}(\Pi_{MSK})$. By subtracting $C_{SOC}(\Pi(Root))$ from each side of the inequality, we get $\Delta_{\Pi_{SOC}} \leq \Delta_{\Pi_{MSK}}$, i.e, the lower bound on the depth of the optimal SOC solution is lower than or equal to the one of the optimal makespan solution. Thus, it may be found faster with fewer node expansions.

Figure 2CBS's CTs for SOC (left) and MKS (right). presents partial CTs for minimizing sum of costs (left) and makespan (right) created when CBS is executed on the problem instance shown in Figure 1Example of SOC and MKS optimal solutions.. The figure only presents the shortest branches of the CTs leading to optimal solutions. Here, $C_{SOC}(Root) = 8$, $C_{SOC}(\Pi_{SOC}) = 9$, and $C_{SOC}(\Pi_{MSK}) = 10$. Thus, $\Delta_{\Pi_{SOC}} = 1$ and $\Delta_{\Pi_{MSK}} = 2$, corresponding to the depths of the optimal solutions.

While the lower bound on the depth of the optimal sum of costs is lower than that of the optimal makespan, it may not always be the case that the optimal makespan solution is deeper in the CT than the optimal sum of costs solution; it depends on the paths the low-level solver returns, e.g., the low-level solver

[1] It is shown (Boyarski et al., 2015) that resolving conflicts in this order (cardinal, semi-cardinal, and non-cardinal) often results in fewer CT node expansions.

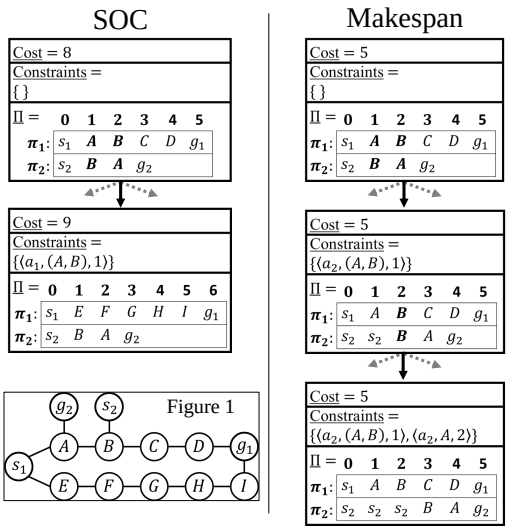

Figure 2: CBS's CTs for SOC (left) and MKS (right).

may return conflicting paths with non-cardinal conflicts. Then the depth of the optimal solution may be higher. Moreover, for CBS, some problem instances are easier for sum of costs and some for makespan.

Figure 3Example of problem instances where CBS performs better for SOC (left) and MKS (right). presents two problem instances and their corresponding CTs, where each CT node shows the sum of costs and makespan of its plan. On the left instance, after the conflict of the root is resolved, the sum of costs and makespan increase in the left child CT node, and only the sum of costs increases in the right one. As the left node contains a conflict-free plan and the sum of costs is equal for both nodes, an optimal sum of costs solution is found, and the search can stop. However, to find the optimal makespan solution, the right node must be explored, as the makespan of that node is lower. On the right instance, after the conflict of the root is resolved, only the sum of costs in the left CT node increases. Here, the optimal makespan solution is found in the left node, as both nodes have the same makespan. However, to find the optimal SOC solution, the right node must be explored. Therefore, the left instance is easier for finding the optimal sum of costs solution and the right one is easier for finding the optimal makespan solution.

While there are instances easier for either sum of costs or for makespan, CBS still often performs better for sum of costs. When resolving a conflict in CBS, the sum of costs and makespan in the two child CT nodes may either remain the same as their parent or increase. Increasing the cost in the child nodes usually reduces the size of the CT, as one of these branches may not be explored, e.g., if the cost exceeds the cost of the optimal solution. When the makespan increases, the sum of costs always also increases, e.g., the left node of the left instance in Figure 3Example of problem instances where CBS performs better for SOC (left) and MKS (right).. However, when the sum of costs increases, the makespan may not always increase as well, e.g., the left node of the right instance. Therefore, we conjecture that this also gives an advantage to sum of costs in CBS.

In the tiny maps (form size 8 by 8 to 24 by 24), CBS performed slightly better for makespan than for sum of costs. We conjecture that the reason is that such maps are more crowded, the paths of the agents are relatively short, and there is more than a single agent influencing the makespan, i.e., multiple agents have the longest path. Therefore, the makespan increases more often in these maps when CBS resolves a conflict between two agents.

Reduction to SAT. Both makespan and the sum of costs optimizations pose specific challenges, which we explain using the average number of variables in instances solved by both the makespan optimal model and *iterative* sum of costs optimal model shown in Table 2Average number of variables (in thousands) entering the SAT solver using the makespan optimal model and the *iterative* sum of costs optimal model. From left to right, the number of variables of the last call, the number of times the SAT solver was invoked, and the cumulative number of variables across all solver calls are listed.. Of course, other factors that are hard to measure may play a role in performance in the case of reduction-based solving, however, as we show, the number of variables may provide an explanation.

Table 2: Average number of variables (in thousands) entering the SAT solver using the makespan optimal model and the *iterative* sum of costs optimal model. From left to right, the number of variables of the last call, the number of times the SAT solver was invoked, and the cumulative number of variables across all solver calls are listed.

	vars last call		solver calls		vars cumulative	
	mks	soc	mks	soc	mks	soc
8	6	**12**	1,0	5,4	6	**158**
16	**140**	76	1,0	5,8	140	**748**
24	**454**	127	1,0	5,0	454	**1269**
32	**1439**	144	1,0	4,2	**1439**	1143
40	**1685**	130	1,0	4,5	**1685**	1183
48	**1498**	67	1,0	2,7	**1498**	451
56	**1475**	60	1,0	2,7	**1475**	382
64	**1239**	18	1,0	1,8	**1239**	53

The number of variables modeling the movement

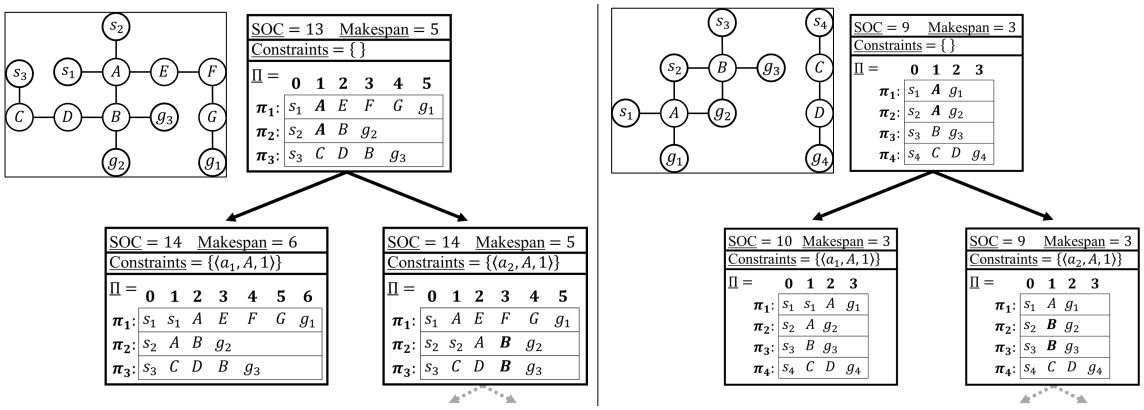

Figure 3: Example of problem instances where CBS performs better for SOC (left) and MKS (right).

of the agents depends on the number of agents $|A|$, number of vertices $|V|$, and number of timesteps T. Increasing $|V|$ (i.e., increasing map size) increases also T as the start and goal placement is random, it is more likely that the start-goal distance is greater. In the makespan optimization, there is a global time limit T for all agents. This means that if the start-goal distance of agent a_i is much lower than T, a_i has enough time and freedom to move around the map. For this movement, variables have to be created and the underlying solver has to assign them in compliance with the constraints. These variables may be unnecessary for finding the solution, however, to ensure that the found solution is optimal, they have to be included. The increase in the number of variables with the size of the map can be seen in Table 2Average number of variables (in thousands) entering the SAT solver using the makespan optimal model and the *iterative* sum of costs optimal model. From left to right, the number of variables of the last call, the number of times the SAT solver was invoked, and the cumulative number of variables across all solver calls are listed.. The fact that the number of variables decreases for map sizes 40 by 40 and onward is caused by both solvers not being able to solve instances with a high number of agents. On the other hand, as T is usually dictated by a single agent with a large start-goal distance, T is rarely increased as all conflicts are solved without disturbing the critical agent. Again, this can be seen in Table 2Average number of variables (in thousands) entering the SAT solver using the makespan optimal model and the *iterative* sum of costs optimal model. From left to right, the number of variables of the last call, the number of times the SAT solver was invoked, and the cumulative number of variables across all solver calls are listed. in the number of solver calls, which indicates how many times the bound on the cost was increased.

In the sum of costs optimization, each agent has a separate time limit T_i. This means that the movement of the agents is much more restricted and the number of variables is lowered. On the other hand, a numeric constraint “at most K” is introduced which causes an overhead in terms of the number of variables. Table 2Average number of variables (in thousands) entering the SAT solver using the makespan optimal model and the *iterative* sum of costs optimal model. From left to right, the number of variables of the last call, the number of times the SAT solver was invoked, and the cumulative number of variables across all solver calls are listed. shows that the overhead over the makespan model is most prominent in the smallest maps. As the size of the map increases, the makespan model overtakes the sum of costs model due to the behavior explained above. Since the sum of costs model poses a larger restriction on the movement, the collisions among agents must be often resolved by increasing the cost function, meaning a new call to the underlying solver (shown in Table 2Average number of variables (in thousands) entering the SAT solver using the makespan optimal model and the *iterative* sum of costs optimal model. From left to right, the number of variables of the last call, the number of times the SAT solver was invoked, and the cumulative number of variables across all solver calls are listed.). Since there are multiple solver calls, we also present a cumulative number of variables across all of the calls. These numbers more closely correspond to the results in Table 1The number of solved instances. The results are split based on the used cost function and based on the size of the instance grid graph. For each solver and map size, the most number of solved instances is highlighted.. Again, the fact that the number of variables decreases for map sizes 40 by 40 and onward is caused by both solvers not being able to solve instances with many agents.

Our results in Table 1The number of solved instances. The results are split based on the used cost

function and based on the size of the instance grid graph. For each solver and map size, the most number of solved instances is highlighted. also provide an insight into the two different sum of costs optimal models – *iterative* and *jump*. In (Barták and Svancara, 2019), the latter showed to outperform the former. In our experiments, the *iterative* model always performed better. The experiments in the original publication of the *jump* model were performed only on small instances of sizes 8 by 8 up to 16 by 16, where the model also performed the best in our experiments. On these maps, the model can indeed outperform the *iterative* model with improved implementation. As stated, our implementation consists of hand-crafted translation to SAT, while in the original paper, Picat language is used (Zhou et al., 2015) to automatically translate constraints to SAT and to minimize the sum of costs. We suspect that the built-in libraries for minimization perform better than rebuilding the formula from scratch with a different bound on the sum of costs.

The *jump* model behaves differently as compared to the *iterative* one. The number of solved instances tend to decrease with the increase in map size for the *jump* model, while the *iterative* model has a peak at size 40 by 40. This is to be expected since the first step of the *jump* model is to find the makespan optimal solution and we showed that for the larger maps, it is harder to find the makespan optimal solution than the sum of costs optimal solution. On the other hand, the numerical constraints are still present and, as shown, introduce a significant overhead for the smaller maps. In a sense, the *jump* model combines the hardest parts of both optimizations.

On the other hand, the *jump* model performs fewer solver calls to find the appropriate k. The model may be improved by finding a suboptimal solution first (instead of a makespan optimal one) to set the k.

Reduction to ASP. The results for the reduction to ASP are similar to the ones for SAT. In Table 3Average number of variables, solve calls, and reachable positions per instance for each cost function. Numbers for variables and positions are in thousands and the number of variables is accumulated over all solve calls., we see that the cumulative number of Boolean variables follows a similar trend to that of SAT. However, the numbers are not directly comparable because they depend on the number of solved instances and the ASP-based reduction solves slightly more instances. The table also reports the number of reachable positions. We observe that this number is proportional to the number of variables, which is to be expected because they directly influence the size of the ground instances (the internal representation of the encoding, closely related to a SAT formula, the solver is working on). We observe that for the makespan objective, reachable positions increase for larger maps but there is a drop for the largest maps. This can be explained by the smaller number of instances solved for larger maps; we can only solve for a low number of agents. This behavior is even more pronounced for the sum of costs objective. Furthermore, we observe that there is a much higher number of solve calls for smaller instances in this setting. This shows that we can solve instances with more agents here. For larger maps, the number of solve calls drops due to timeouts coupled with a decrease in reachable positions. This shows that the reachability optimization is especially effective for the sum of costs objective where the shortest distances between start and goal vertices are taken into account for each agent as opposed to the makespan objective where all agents move within the same fixed horizon. Hence, we observe a much smaller number of reachable positions and a larger number of solved instances for the sum of costs objective as compared to the makespan objective.

Table 1The number of solved instances. The results are split based on the used cost function and based on the size of the instance grid graph. For each solver and map size, the most number of solved instances is highlighted. shows that the *jump* model for ASP performs worse than the *iterative* model. As discussed in the previous section, this can be improved by changing the way the initial plan is computed. Furthermore, we used the default configuration of the ASP solver in the benchmarks. We suspect that the *jump* model would be faster than the *iterative* model (at least for the smaller instances) if an alternative algorithm for optimization is chosen, such as optimization based on unsatisfiable cores (Andres et al., 2012).

Finally, we comment on the difference in the number of solved instances between the SAT- and ASP-based reductions. The instantiation of rules in the ASP encoding is linear in the number of agents, while for the reduction to SAT, the number of clauses for conflicts is quadratic in the number of agents. Reducing the number of rule instances/clauses often has a big impact on solving. We note that further size reductions for the SAT model are possible as well.

7 CONCLUSION

We compared the behavior of the search-based algorithm CBS and the reduction-based approaches while optimizing MAPF under two cost functions – makespan and sum of costs. We empirically showed

Table 3: Average number of variables, solve calls, and reachable positions per instance for each cost function. Numbers for variables and positions are in thousands and the number of variables is accumulated over all solve calls.

	vars cumulative		solver calls		reach pos	
	mks	soc	mks	soc	mks	soc
8	12	**264**	1,0	7,0	2	**67**
16	189	**436**	1,0	5,3	28	**119**
24	692	**699**	1,0	5,1	102	**196**
32	**2336**	447	1,0	3,4	**342**	111
40	**2900**	273	1,0	2,7	**426**	60
48	**2815**	91	1,0	1,8	**424**	14
56	**2646**	180	1,0	2,3	**396**	37
64	**1488**	12	1,0	1,1	**216**	0,5

that on small-size maps, it is easier for both approaches to solve the makespan optimization, while on larger maps, it is easier to solve the sum of costs optimization. This is counter-intuitive since both of the solving approaches were first conceived for different cost functions – search-based for the sum of costs and reduction-based for makespan.

We provided insights into the phenomenon. For CBS, the lower depth of the optimal makespan solution is larger than or equal to that of the optimal sum of costs solution, which may require more node expansions when minimizing makespan. Moreover, when the makespan increases at a child node due to conflict resolution, the sum of costs also increases, but not vice versa. Increasing that cost often reduces the size of the CBS tree. Therefore, this also gives an advantage to the sum of costs. For the reduction-based approaches, solving for sum of costs introduces overhead for the numerical constraints. This overhead is outweighed on larger maps by the freedom of movement of the agents with less restricted paths. This freedom is modeled by a large number of variables that overwhelm the underlying solver.

We also compared our results with three previous studies (Surynek et al., 2016b; Barták and Svancara, 2019; Gómez et al., 2021) and showed that our results are different from theirs due to the small-scale experiments they used. On smaller maps, we observed similar behavior as was reported in the studies, however, on larger maps, the behavior diverged.

Based on our work, we propose open questions for future work. The CBS algorithm may be improved for makespan. When finding a new single-agent path, this path does not have to be the shortest possible, if the makespan is dictated by a path of a different agent. This may reduce the number of future conflicts.

Furthermore, the *jumping* model may be improved by changing the approach to finding the initial solution, as finding a makespan optimal solution first and then creating the numerical constraints combines the hardest parts of both cost functions.

ACKNOWLEDGEMENTS

This work was funded by DFG grant SCHA 550/15, by project 23-05104S of the Czech Science Foundation, and by Charles University project 24/SCI/008.

REFERENCES

Alonso-Mora, J., Breitenmoser, A., Rufli, M., Beardsley, P. A., and Siegwart, R. (2010). Optimal reciprocal collision avoidance for multiple non-holonomic robots. In *DARS*, pages 203–216.

Andres, B., Kaufmann, B., Matheis, O., and Schaub, T. (2012). Unsatisfiability-based optimization in clasp. In *ICLP*, pages 211–221.

As'in Ach'a, R. J., López, R., Hagedorn, S., and Baier, J. A. (2021). A new boolean encoding for MAPF and its performance with ASP and maxsat solvers. In *SoCS*, pages 11–19.

Barták, R. and Svancara, J. (2019). On sat-based approaches for multi-agent path finding with the sum-of-costs objective. In *SoCS*, pages 10–17.

Bennewitz, M., Burgard, W., and Thrun, S. (2002). Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics Auton. Syst.*, 41(2-3):89–99.

Biere, A., Fazekas, K., Fleury, M., and Heisinger, M. (2020). CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proceedings of the SAT Competition 2020 – Solver and Benchmark Descriptions*, pages 51–53.

Boyarski, E., Felner, A., Bodic, P. L., Harabor, D. D., Stuckey, P. J., and Koenig, S. (2021). f-aware conflict prioritization & improved heuristics for conflict-based search. In *AAAI*, pages 12241–12248.

Boyarski, E., Felner, A., Stern, R., Sharon, G., Tolpin, D., Betzalel, O., and Shimony, S. E. (2015). ICBS: improved conflict-based search algorithm for multi-agent pathfinding. In *IJCAI*, pages 740–746.

Dresner, K. M. and Stone, P. (2008). A multiagent approach to autonomous intersection management. *JAIR*, 31:591–656.

Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, T. K. S., and Koenig, S. (2018). Adding heuristics to conflict-based search for multi-agent path finding. In *ICAPS*, pages 83–87.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., and Schaub, T. (2015). Abstract Gringo. *Theory and Practice of Logic Programming*, 15(4-5):449–463.

Gómez, R. N., Hernández, C., and Baier, J. A. (2021). A compact answer set programming encoding of multi-agent pathfinding. *IEEE Access*, 9:26886–26901.

Hart, P., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107.

Husár, M., Svancara, J., Obermeier, P., Barták, R., and Schaub, T. (2022). Reduction-based solving of multi-agent pathfinding on large maps using graph pruning. In *AAMAS*, pages 624–632.

Kaminski, R., Romero, J., Schaub, T., and Wanko, P. (2020). How to build your own asp-based system?! *CoRR*, abs/2008.06692.

Li, J., Felner, A., Boyarski, E., Ma, H., and Koenig, S. (2019). Improved heuristics for multi-agent path finding with conflict-based search. In *IJCAI*, pages 442–449.

Ma, H., Li, J., Kumar, T., and Koenig, S. (2017). Lifelong multi-agent path finding for online pickup and delivery tasks. In *AAMAS*, pages 837–845.

Nguyen, V., Obermeier, P., Son, T. C., Schaub, T., and Yeoh, W. (2017). Generalized target assignment and path finding using answer set programming. In *IJCAI*, pages 1216–1223.

Philipp, T. and Steinke, P. (2015). Pblib – a library for encoding pseudo-boolean constraints into cnf. In *Theory and Applications of Satisfiability Testing – SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer International Publishing.

Sartoretti, G., Kerr, J., Shi, Y., Wagner, G., Kumar, T. K. S., Koenig, S., and Choset, H. (2019). PRIMAL: pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics Autom. Lett.*, 4(3):2378–2385.

Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66.

Silver, D. (2005). Cooperative pathfinding. In *AIIDE*, pages 117–122.

Stern, R., Sturtevant, N. R., Felner, A., Koenig, S., Ma, H., Walker, T. T., Li, J., Atzmon, D., Cohen, L., Kumar, T. K. S., Barták, R., and Boyarski, E. (2019). Multi-agent pathfinding: Definitions, variants, and benchmarks. In *SoCS*, pages 151–159.

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In *AAAI*, pages 1261–1263.

Surynek, P. (2012). A sat-based approach to cooperative path-finding using all-different constraints. In *SoCS*.

Surynek, P. (2017). Time-expanded graph-based propositional encodings for makespan-optimal solving of cooperative path finding problems. *Ann. Math. Artif. Intell.*, 81(3-4):329–375.

Surynek, P., Felner, A., Stern, R., and Boyarski, E. (2016a). Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*, pages 810–818.

Surynek, P., Felner, A., Stern, R., and Boyarski, E. (2016b). An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. In *SoCS*, pages 145–147.

Yu, J. and LaValle, S. M. (2012). Multi-agent path planning and network flow. In *WAFR*, pages 157–173.

Yu, J. and LaValle, S. M. (2013). Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*, pages 1444–1449.

Zhou, N. and Barták, R. (2017). Efficient declarative solutions in picat for optimal multi-agent pathfinding. In *ICLP*, pages 11:1–11:2.

Zhou, N., Kjellerstrand, H., and Fruhman, J. (2015). *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer.