IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 30, 2020, accepted December 19, 2020, date of publication January 22, 2021, date of current version February 17, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3053547

A Compact Answer Set Programming Encoding of Multi-Agent Pathfinding

RODRIGO N. GÓMEZ[1], CARLOS HERNÁNDEZ[2], AND JORGE A. BAIER[1,3]
[1]Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile, Santiago 7820244, Chile
[2]Departamento de Ciencias de la Ingeniería, Universidad Andrés Bello, Santiago 7500000, Chile
[3]Instituto Milenio Fundamentos de los Datos, Santiago 7820244, Chile

Corresponding author: Rodrigo N. Gómez (rigomez@uc.cl)

This work was supported by the Fondo Nacional de Desarrollo Científico y Tecnológico (FONDECYT) under Grant 1150328 and Grant 1161526.

ABSTRACT Multi-agent pathfinding (MAPF) is the problem of finding k non-colliding paths connecting k given initial positions with k given goal positions on a given map. In its sum-of-costs variant, the total number of moves and wait actions performed by agents before they definitely reach the goal is minimized. Not surprisingly, since MAPF is combinatorial, a number of compilations to Boolean Satisfiability (SAT) and Answer Set Programming (ASP) exist. In this article, we describe in detail the first family of compilations to ASP that solve sum-of-costs MAPF over 4-connected grids. Compared to existing ASP compilations, a distinguishing feature of our compilation is that the number of total clauses (after grounding) grow linearly with the number of agents, while existing compilations grow quadratically. In addition, the optimization objective is such that its size after grounding does not depend on the size of the grid. In our experimental evaluation, we show that our approach outperforms search-based sum-of-costs MAPF solvers when grids are congested with agents. We also show that our approach is competitive with a SAT-based approach when follow conflicts are taken into account. We also explore the potential of our solver when finding makespan-optimal solutions, in which makespan is minimized first and then cost is minimized. Our results show that makespan-optimal solutions are slightly suboptimal in most benchmarks. Moreover, our MAPF solver, when run in that mode, is faster and scales better.

INDEX TERMS Answer set programming, multi-agent pathfinding.

I. INTRODUCTION

Multi-agent pathfinding (MAPF) is the problem of finding k non-conflicting paths connecting k given initial positions with k given goal positions on a given map. MAPF has many applications. It is key for the implementation of automated warehouses, in which crowds of robots usually share limited space, and multi-agent videogames, in which sometimes hundreds of agents need to move in crowded areas [1]. But it may also become increasingly relevant in other important applications such as underground mining and airport ground control [2].

MAPF is a hard computational problem. Unlike single-agent pathfinding, in which a path being sought should just avoid collisions with obstacles, in MAPF collisions between moving agents. Indeed, when the map is represented as a graph, finding a solution to MAPF that optimizes either total cost or makespan (i.e., last arrival time) are computationally difficult. Indeed, the decision problem associated to both optimizations are NP-complete [3], [4]. This should not be surprising. When viewed as a standard AI search problem, it is straightforward to notice that the branching factor of MAPF is exponential on the number of agents, since at each moment in time each agent can perform a number of actions, relatively fixed.

Several variants of MAPF have been considered in the literature. The simplest case, which is the focus of our paper, models the maps as 4-connected grid, in which some of the cells in the grid are marked as obstacles. Agents may move in parallel, in the sense that at each time instant we assume all agents can move from their current cell to any of the obstacle-free neighbor cells. A *conflict* is produced when two agents visit the same cell at the same time instant or when they traverse the same graph edge at the same time.

MAPF over 4-connected grids does not consider the kinematic constraints or physical dimensions of the agents

The associate editor coordinating the review of this manuscript and approving it for publication was Chao-Yang Chen.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

(e.g., [5]), and considers a rather limited range of motion for agents. This does not make this problem less relevant for the robotics community. Indeed, MAPF over grids can be used as the base solver for solutions to full-fledged multi-robot motion planning with dynamic constraints [6].

Even though a number of approaches to MAPF over grids have been proposed, two classes of solvers are most relevant for the research we report here. First, search-based solvers (e.g., [7]), which use heuristic search as the main component. A state-of-the-art search-based approach is Conflict-Based Search (CBS) [8]–[10], which uses A* at its core. A second class is compilation-based solvers; for example, compilers from MAPF to Satisfiability Testing (SAT) (e.g., [11]–[13]), and Answer-Set Programming (ASP) [14]–[16].

When seeking for an optimal solution for MAPF, different objective functions can be considered. Under sum-of-costs, the most popular variant of MAPF, the objective is to minimize the moves agents perform before stopping at the goal.

In this article, we continue to explore the potential of ASP solvers for MAPF, and propose the first compilation that solves MAPF optimally under the sum-of-costs assumption. A second contribution consists of proposing the first compilation from MAPF to ASP that grows linearly with the number of agents, unlike existing compilations to ASP that are quadratic on the number of agents. In addition, we propose an optimization which uses information drawn from a search algorithm that is run as a preprocessing step to make the encoding more compact.

We evaluate our approach on synthetic square grids and warehouse grids with an increasing number of agents. We compare against MDD-SAT [11], SMT-CBS [17] two state-of-the-art compilation-based solvers, and to ICBS-H [9], a representative of the state-of-the-art in search-based MAPF. We observe that our approach outperforms search-based solvers when agent congestion is high. Specifically, our approach has a greater coverage as the number of agents increase. When follow conflicts are considered, that is, when agents can only move to a previously unoccupied cell, our approach slightly outperforms MDD-SAT. In addition, we investigate the performance of our solver when finding so-called makespan-optimal solutions; that is solutions in which makespan is minimized first and then cost is minimized.

We conclude that ASP is a viable approach to solving MAPF problems. Another interesting conclusion is that ASP also provides a very compact and elegant representation of sum-of-costs MAPF. Indeed, all of the code needed to solve a MAPF instance is included in this article. We conclude in addition that finding makespan-optimal solutions allows ASP-based solvers to find solutions more quickly and to scale better, by sacrificing less than 1% average solution quality. This is important since it suggests that focusing on cost-optimal solutions instead of makespan-optimal solutions might not be a practical approach.

This article is a journal version of a previous AAAI-20 publication [18]. The following items describe material not included in the previous publication.

- The full details of the MAPF to ASP encoding, some of which were omitted from the conference publication.
- A proof of Theorem 1, which was not included in the conference publication.
- An extended experimental analysis, which allows us to draw additional conclusions. Specifically,
  - We analyze the performance of our solver over an extended set of grids with different sizes, and establish new relations between the relative performance of our solver and existing solvers.
  - We include an analysis of our solver configured to find makespan-optimal solutions. The conclusions we draw from these experiments are important: a solver that is makespan-optimal scales better and finds solutions that are only slightly suboptimal in terms of cost.
  - Finally, we include an analysis of grounding time versus execution time.

The rest of the paper is organized as follows. We start with a detailed description of MAPF and an introduction to ASP. Then we present a basic encoding to ASP, and we continue introducing the elements that allow the translation to be linear on the number of agents. We continue with our empirical analysis, and end with conclusions.

II. BACKGROUND

In this section we describe MAPF and ASP. Our definition of MAPF follows closely that of [19].

A. MULTI-AGENT PATHFINDING

A MAPF instance is defined by a tuple $(G, \mathcal{A}, init, goal)$, where $G = (V, E)$ is a graph, A is the set of agents, and $init : A \to V$ and $goal : A \to V$ are functions used to denote the initial and goal vertex for each of the agents.

At each time instant each agent at vertex v can either move to any of its successors in G or not move at all. When the graph G is a 4-connected grid, as we assume in the rest of the paper, at each time instant each agent can perform an action in the set $\{up, down, left, right, wait\}$. The *wait* action leaves the agent in the same position whereas the others move them in one of the four cardinal positions. A path over G is a sequence of vertices in V, $v_1 v_2 \ldots v_n$, where either $v_i = v_{i+1}$ (i.e., a *wait* action is performed) or $(v_i, v_{i+1}) \in E$ (i.e., a non wait action is performed) for every $i \in \{1, \ldots, n-1\}$. Given a path π we denote by $\pi[i]$ the i-th element in π, where $i \in \{1, \ldots, |\pi|\}$.

A solution to MAPF is a function $sol : \mathcal{A} \to V^*$, which associates a path to each of the agents, such that the first and last vertices of $sol(a)$ are, respectively, $init(a)$ and $goal(a)$. Without loss of generality, henceforth we assume that all paths in sol have the same size, since wait actions may be used at the end of any action sequence to remain on the same vertex. Below we denote by $\mathcal{T}$ the set $\{1, \ldots, M\}$, where M is the size of any of the paths in sol. We also refer to $M - 1$ as the makespan of the solution.

In addition, *sol* must be conflict-free, which means that if π and ρ are the paths in *sol* followed by two different agents, none of the following conflicts should arise.

- **Vertex Conflict**. Two agents cannot be at the same vertex at the same time instant. Formally, there is a vertex conflict if and only if $\pi[i] = \rho[i]$, for some $i \in \mathcal{T}$.
- **Swap Conflict**. Two agents cannot swap their positions. Intuitively, this conflict is justified by that fact that we assume that size of the agents prevent the connection between two vertices in opposite directions. Formally, there is a swap conflict if and only if $(\pi[i], \rho[i]) = (\rho[i+1], \pi[i+1])$, for some $i \in \{1, \ldots, M-1\}$.
- **Follow Conflict**. An agent cannot occupy the position of an agent that has just moved away from such a position; that is, it cannot follow another agent. Formally there is a follow conflict if and only if $\pi[i+1] = \rho[i]$.

Most of the literature in MAPF (e.g., [8]), considers vertex conflicts and swap conflicts. Fewer solvers consider follow conflicts. In our translation to ASP we show how to model all three types of conflicts but our experimental section focuses on the first two.

A standard assumption in MAPF is that all actions cost one unit except for waits performed at the goal when no other action is planned in the future. Note that this means *wait* actions do have a cost of 1 unless the agent performs such a wait at the goal, and does not move away from the goal in the future. Thus, the cost of path π for agent a is written as $|\rho|$, when ρ is the shortest sequence such that $\pi = \rho(goal(a))^k$, for some k. The cost of a solution *sol*, denoted by $c(sol)$, is defined as $\sum_{a \in A} c(sol(a))$.

A solution *sol* is optimal under sum-of-costs, or simply *cost-optimal*, if no other solution sol' exists such that $c(sol') < c(sol)$. A solution *sol* is *makespan-minimal* if no other solution exists whose makespan is smaller than the makespan of *sol*. A makespan-minimal solution is *makespan-optimal* if no other makespan-minimal solution sol' exists such that $c(sol') < c(sol)$. The class of cost-optimal solutions is not equal to the class of makespan-optimal solutions. This is illustrated in Figure 1.

In the rest of the paper, the notion of a solution that *relaxes all conflicts* will be relevant to obtain costs-optimal solutions. These solutions are such that swap and vertex conflicts are not considered and thus can be efficiently computed running any single-agent pathfinding algorithm. Below we use the notation c_a^* to refer to the number of steps required by agent a to reach the goal under the assumption that all (vertex and swap) conflicts are relaxed.

B. ANSWER-SET PROGRAMMING

ASP [20] is a logic-based framework for solving optimization problems. For space limitations, here we describe a subset of an ASP standard that is relevant to this article. An ASP basic program is a set of rules of the form:

$$p \leftarrow q_1, q_2, \ldots, q_n. \tag{1}$$

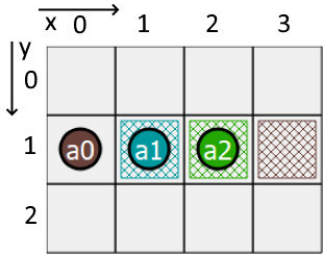

FIGURE 1. A MAPF instance in which an increase of makespan allows a solution of lower cost. The problem has 3 agents: a_0, who needs to go from (0, 1) to (3, 1). Agents a_1 and a_2, located at (1, 1) and (2, 1), respectively, are at their goal cell. The makespan-optimal solution has a makespan of 3 and a cost of 8, and it involves a_1 and a_2 moving away from their goal to let a_0 move straight towards its goal. In contrast, the cost-optimal solution has a cost of 5 and a makespan of 5, and only moves agent a_0.

where $n \geq 0$, and $p, q_1, \ldots, q_n$ are so-called atoms. The intuitive interpretation of this rule is as follows 'p is true/provable if so are $q_1, q_2, \ldots, q_n'$. When $n = 0$ rule (1) is considered to have an empty body. Such rules are called facts and are usually written as 'p' instead of '$p \leftarrow$'.

A model of an ASP basic program is a set of atoms M that intuitively contains all and only the atoms that are provable. Formally, M is a model of basic program Π if and only if M is the subset-minimal set such that for every rule $p \leftarrow q_1, q_2, \ldots, q_n \in \Pi$ such that $\{q_1, \ldots, q_n\} \subseteq M$, then $p \in M$.

An important syntactic element relevant to our paper is the so-called negation as failure. Rules containing such negated atoms look like:

$$p \leftarrow q_1, q_2, \ldots, q_n, \quad not\ r_1,\ not\ r_2, \ldots, not\ r_k. \tag{2}$$

Intuitively, rule (2) should be interpreted as 'p is provable if $q_1, \ldots q_n$ are provable a none of $r_1, \ldots, r_k$ are provable'. The semantics of programs that include negation as failure is simple but a little more involved, requiring the introduction of so-called stable models, whose formal definition we omit from this article. We direct the interested reader to [21].

Another relevant type of rule for our paper is:

$$|\{p_1, p_2, \ldots, p_n\}| = k \leftarrow q_1, q_2, \ldots, q_m,$$

where k is an integer. Intuitively here we say that if $q_1, q_2, \ldots, q_m$ are all provable, then k of the elements in $\{p_1, p_2, \ldots, p_n\}$ must appear in a model. This definition allows programs to have multiple models.[1] For example, the program $\{s, |p, q, r| = 1 \leftarrow s\}$ has three models: $\{p, s\}, \{q, s\}$, and $\{r, s\}$.

Another type of rule that is relevant to our translation is:

$$\leftarrow p_1, p_2, \ldots, p_n, \tag{3}$$

which is a constraint that prohibits the occurrence of $\{p_1, p_2, \ldots, p_n\}$ in the model. Technically these rules are a

[1]Technically, negation as failure alone also allows the user to create programs with multiple models, but in our translation we define multiple models exploiting this type of rule.

particular case of (1), but we treat them separately here to make the presentation simpler.

Finally, ASP programs may contain variables to represent rule schemas. As such, a rule like:

$$p(X) \leftarrow q(X), \tag{4}$$

where uppercase letters represent variables. Intuitively a variable occurring in a program Π can take any value among the set of terms of Π. As such, rule (4) represents that when c is a term and $q(c)$ is provable, so is $p(c)$. Intuitively a term represents an object that can be named in the program. The set of terms for a program is syntactically determined from the program using the constants mentioned in it. The set of terms has a theoretical counterpart, the so-called Herbrand base, whose definition we omit here, since it is not key for understanding the rest of the paper.

In the process of finding a model for a program, an initial step that is carried out is grounding. Grounding instantiates rules with variables, effectively removing all variables from the program. Since there are plenty of optimizations that solvers employ during grounding, it is not easy to describe the grounding process with complete precision here, and therefore we will just describe it intuitively. For example, the grounded version of program:

$$\{q(a), q(b), p(X) \leftarrow q(X)\} \tag{5}$$

may be

$$\{q(a), q(b), p(a) \leftarrow q(a), p(b) \leftarrow q(b)\}$$
$$\text{or } \{q(a), q(b), p(a), p(b)\},$$

depending on the optimizations applied at grounding time. What is however unavoidable is that grounding generates two instances for the rule $p(X) \leftarrow q(X)$ because the number of objects that satisfy predicate q is two. Thus, if we had declared n objects satisfying q we would expect the grounding process to generate n instances for $p(X) \leftarrow q(X)$. As we will see in the rest of the paper, the size of the grounded version of the program is key for performance.

III. A BASIC TRANSLATION OF MAPF TO ASP

We are now ready to describe our compilation of sum-of-costs MAPF to ASP. As we have mentioned above, this is the first compilation to ASP that handles sum-of-costs. Besides that aspect of novelty, the basic compilation that we present here is similar in many aspects to the compilation of [14] to ASP, and, in some aspects similar to the MAPF-to-SAT compilation of [22]. Below we are specific about these similarities.

As most compilations of planning problems into SAT/ASP, the makespan of the compilation is a parameter, which below we call $\mathtt{T}$.

A. ATOMS

We use the following atoms:

- $agent(a)$: to express that A is an agent,
- $goal(a, x, y)$: specifies that the goal cell for agent a is (x, y),
- $obstacle(x, y)$: specifies that cell (x, y) is an obstacle,
- $at(a, x, y, t)$: specifies that agent a is at (x, y) at time t,
- $exec(a, m, t)$: specifies that agent a executes action m at time t,
- $at\_goal(a, t)$: specifies that agent a is at the goal at time t,
- $time(t)$: t is a time instant,
- $action(m)$: m is an action,
- $cost(a, t, c)$: specifies that agent a executes an action of cost c at time t,
- $delta(m, x, y, x', y)$: specifies that the cell (x', y') is reached when performing action m at cell (x, y).

Finally, we use atoms $rangeX(x)$ and $rangeY(y)$ to specify that (X, Y) is within the limits of the grid.

B. INSTANCE SPECIFICATION

To specify a particular MAPF instance, we define facts for atoms of the form $agent(a)$, for each $a \in A$, $obstacle(x, y)$ for each (x, y) that is marked as an obstacle in the grid, $rangeX(x)$ for each $x \in \{1, \ldots, w\}$, where w is the width of the grid, and $rangeY(y)$ for each $y \in \{1, \ldots, h\}$, where h is the height of the grid. Additionally, we define the initial cells for each agent, adding one fact of the form $at(a, x_a, y_a, 0)$ for each agent $a \in \mathcal{A}$, where $(x_a, y_a) = init(a)$. Furthermore, we add an atom of the form $time(t)$ for every $t \in \{1, \ldots, \mathtt{T}\}$. The number of rules needed to encode a MAPF instance is therefore in $\Theta(|\mathcal{A}| + \mathtt{T} + |V|)$.

C. EFFECTS

To encode the effects of the five actions, we use a single rule written as follows:

$$\begin{aligned} at(A, X, Y, T) \leftarrow\ & exec(A, M, T-1), \\ & at(A, X', Y', T-1), \\ & delta(M, X', Y', X, Y). \end{aligned} \tag{6}$$

which specifies that if agent A is at position (X', Y') at time instant $T-1$, then it will be in position (X, Y) in time instant T if and only if (X, Y) and (X', Y') satisfy predicate $delta$. Auxiliary predicate $delta$ is used to establish a relation between (X, Y) and (X', Y') given a certain action M in the following way:

$$\begin{aligned} delta(right, X, Y, X+1, Y) &\leftarrow rangeX(X), \quad rangeY(Y). \\ delta(left, X, Y, X-1, Y) &\leftarrow rangeX(X), \quad rangeY(Y). \\ delta(up, X, Y, X, Y+1) &\leftarrow rangeX(X), \quad rangeY(Y). \\ delta(down, X, Y, X, Y-1) &\leftarrow rangeX(X), \quad rangeY(Y). \\ delta(wait, X, Y, X, Y) &\leftarrow rangeX(X), \quad rangeY(Y). \end{aligned} \tag{7}$$

A grounding time predicate $delta$ results in 5 rules per each position of the grid. This defines that the total number of grounded instances for rule (6) is proportional to the size of the grid, the number of agents and the number of time

instants. The total number of instances for rules of the form (6) and (7) is in $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$.

D. PARALLEL ACTION EXECUTION

We need to encode that each agent performs exactly one action at each time instant. To do this we write the following rule:

$$|\{exec(A, M, T-1) : action(M)\}| = 1 \leftarrow time(T),\ agent(A). \quad (8)$$

Upon grounding, the number of instances of this rule is in $\Theta(|\mathcal{A}| \cdot \mathtt{T})$.

E. LEGAL POSITIONS

We need to express that the agents move through the vertices in the graph; that is, they cannot exit the grid or visit an obstacle cell. We do so using the following three rules:

$$\begin{aligned} &\leftarrow at(A, X, Y, T), \quad not\ rangeX(X), \\ &\leftarrow at(A, X, Y, T), \quad not\ rangeY(Y), \\ &\leftarrow at(A, X, Y, T), \quad obstacle(X, Y). \end{aligned} \quad (9)$$

The total number of grounded rules for the rules of form (9) is in $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$, since it depends on the number of atoms of the form *at*, *obstacle*, *rangeX*, and *rangeY*.

F. VERTEX CONFLICTS

To express that no agents can be at the same vertex we use the following constraint, which is similar to those used in the encodings to ASP of [14], [15] and [11]:

$$\leftarrow at(A, X, Y, T), \quad at(A', X, Y, T),\ A \neq A'. \quad (10)$$

The number of instances for rule (10) after grounding is $\Theta(|\mathcal{A}|^2 \cdot |V| \cdot \mathtt{T})$. Note that this is the first rule so far whose instantiation is quadratic on the number of agents. This motivates the improvement we present in the following section.

G. SWAP CONFLICTS

No pair of agents can swap their positions. We express this avoiding horizontal and vertical swaps using, respectively, the following constraints.

$$\begin{aligned} &\leftarrow at(A, X+1, Y, T-1), \quad at(A', X, Y, T-1), \\ &\quad at(A, X, Y, T), \quad at(A', X+1, Y, T). \end{aligned} \quad (11)$$

$$\begin{aligned} &\leftarrow at(A, X, Y+1, T-1), \quad at(A', X, Y, T-1), \\ &\quad at(A, X, Y, T), \quad at(A', X, Y+1, T). \end{aligned} \quad (12)$$

The number of ground rules for (11) is in $\Theta(|\mathcal{A}|^2 \cdot |V| \cdot \mathtt{T})$.

H. FOLLOW CONFLICT

An agent cannot occupy in $t+1$ the position an agent had at time t.

$$\leftarrow at(A, X, Y, T), \quad at(A', X, Y, T+1),\ A \neq A'. \quad (13)$$

The number of instances for rule (13) after grounding is $\Theta(|\mathcal{A}|^2 \cdot |V| \cdot \mathtt{T})$, hence quadratic on the number of agents.

I. GOAL ACHIEVEMENT

We specify using a constraint expressing that no agent is away from its goal at time T, using the following two rules:

$$at\_goal(A, T) \leftarrow at(A, X, Y, T), \quad goal(A, X, Y). \quad (14)$$

$$\leftarrow agent(A), \quad not\ at\_goal(A, \mathtt{T}). \quad (15)$$

The number of instances for rules (14) is $\Theta(\mathcal{A} \cdot |V| \cdot \mathtt{T})$.

J. SIZE OF THE BASIC ENCODING

After grounding, it follows that the size of the total encoding is in $\Theta(|\mathcal{A}|^2 \cdot |V| \cdot \mathtt{T})$. That is, it is quadratic in the number of agents, linear in the size of the grid, and linear in the makespan parameter T.

IV. SUM-OF-COSTS IN ASP

The encoding we presented in the previous section still does not produce cost-optimal solutions. Indeed, once fed into an ASP solver, it will return a model only if a solution with makespan T exists. In this section we present how we can obtain solutions for sum-of-costs MAPF.

There is a natural way to encode sum-of-costs minimization: to minimize the number of actions performed by each agent before stopping at the goal. We noticed, however, that this yields an encoding whose size grows linearly with the size of the grid, $|V|$. This motivated us to look for a more compact encoding which would not depend on $|V|$. Even though, as we see in our empirical evaluation below, the grid-independent encoding performs better in practice, we describe both approaches here since the grid-dependent encoding is more natural and is a contribution on its own since sum-of-costs had not been encoded in ASP before.

A. GRID-DEPENDENT ENCODING

This encoding is similar to the approach used in MDD-SAT [11]: the idea to minimize the actions performed by the agent at each cell before stopping at the goal. At a first glance one might think that we just need to count every action performed away from the goal and minimize this number. This approach, however does not work because a *wait* at the goal at time t should be counted if the agent will move away from the goal at some instant t' greater than t.

To identify time instants at which we know the agent will not move away from the goal, we introduce the predicate $at\_goal\_back(a, t)$, which specifies that agent a has reached the goal at time t and will not move away in the future:

$$\begin{aligned} at\_goal\_back(A, \mathtt{T}) &\leftarrow agent(A). \\ at\_goal\_back(A, T-1) &\leftarrow at\_goal\_back(A, T), \\ &\quad exec(A, wait, T-1). \end{aligned}$$

Now we define predicate *cost*, such that there is an atom of the form $cost(a, t, 1)$ in the model whenever agent a performs an action at time t before stopping at the goal. First we express that moving an agent from a cell that is not the goal is penalized by one unit:

$$cost(A, T, 1) \leftarrow at(A, X, Y, T), \quad not\ goal(A, X, Y).$$

Second, moving an agent away from the goal is also penalized by one:

$$cost\,(A, T, 1) \leftarrow at\,(A, X, Y, t), \quad goal\,(A, X, Y), \\ exec(A, M, t), \quad M \neq wait.$$

Third, if an agent performs a *wait* at the goal, but moves at a later time instant, then this is also penalized:

$$cost(A, T, 1) \leftarrow at\,(A, X, Y, t), \quad goal\,(A, X, Y), \\ exec(A, wait, T), \\ not\ at\_goal\_back\,(A, T).$$

Finally, via an optimization statement, we minimize the number of atoms of the form $cost(A, T, 1)$ in the model:

$$\#minimize\,\{C, T, A : cost(A, T, C)\}.$$

After grounding, the number of rules is in $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$.

B. GRID-INDEPENDENT ENCODING

For this encoding, we define the atom $optimal(a, c_a)$, for each agent $a \in \mathcal{A}$, where c_a corresponds to the cost of the optimal path from $init(a)$ to $goal(a)$ ignoring both vertex and swap conflicts. In other words, c_a is the result of solving a relaxation of the problem that ignores other agents. We compute such a value using Dijkstra's algorithm, before generating the encoding.

In contrast to the first encoding, we maximize the slack between the makespan T and the time instant at which an agent has stopped at the goal, by simply adding:

$$penalty(A, T, 1) \leftarrow optimal(A, C), \quad T > C, \\ at\_goal\_back(A, T - 1).$$

Note that since no reference to the grid cells is made, the grounding generates a number of rules in $\Theta(|\mathcal{A}| \cdot \mathtt{T})$. Finally, we use the following maximization statement.

$$\#maximize\,\{P, T, A : penalty(A, T, P)\}.$$

C. FINDING COST-OPTIMAL SOLUTIONS

The encoding proposed so far can find the minimum sum-of-cost solution for a given makespan. We still need to define how to find a true cost-optimal solution.

Following the approach used for SAT encodings for planning [23], in our approach we attempt to solve instances for increasing makespan $\mathtt{T}$, until a solution, say sol_{min}, is found. Two observations with this process are important. First, we do not need to start increasing $\mathtt{T}$ from 1. As mentioned above, at preprocessing time, for each agent we compute cost the cost c_a^* which ignores other agents. The makespan of any solution must be at least $\max_{a\in\mathcal{A}} c_a^*$ so this can be the inferior limit of our iteration.

Second, let sol_{min} be the solution that is found first. Unfortunately, sol_{min} is a makespan-optimal solution but not necessarily a cost-optimal solution. Now, we can compute a bound for the largest makespan $\mathtt{T}_{max}$ at which the cost-optimal solution is found, using the following theoretical result first proposed by [11]. Below we state the theorem and provide a different proof.

Theorem 1: Let sol_{min} be the makespan-optimal solution for MAPF problem P, let sol^- denote a cost-optimal solution to P that ignores all conflicts, and let $\mathtt{T}^-$ denote its makespan. Then the makespan of the cost-optimal solution is at most at $\mathtt{T}_{max} = \mathtt{T}^- + c(sol_{min}) - c(sol^-) - 1$.

Proof: Let sol_{min} be a makespan-optimal solution for P such that there is no other solution with the same makespan and lower cost. Let sol^- be a solution for P that ignores all agent conflicts. For this solution, $c(sol^-) = \sum_{a\in A} c_a^*$. In addition, let $\mathtt{T}^- = \max_{a\in\mathcal{A}} c_a^*$ be the makespan of sol^-, and let sol_{opt} be the sum-of-costs optimal solution for P, where $\mathtt{T}_{opt}$ is its makespan.

Because makespan is always an integer, we prove that $\mathtt{T}_{opt} < \mathtt{T}^- + c(sol_{min}) - c(sol^-)$, which is equivalent to the statement of the theorem. We do so by contradiction, assuming that:

$$\mathtt{T}_{opt} \geq \mathtt{T}^- + c(sol_{min}) - c(sol^-) \tag{16}$$

Because $\mathtt{T}_{opt}$ is the makespan of sol_{opt} there exists one agent, say a_i, that is such that $c(sol_{opt}(a_i)) = \mathtt{T}_{opt}$, therefore:

$$c(sol_{opt}) = \mathtt{T}_{opt} + \sum_{a\in A\setminus a_i} c(sol_{opt}(a)) \tag{17}$$

Because $c_a^* \leq c(sol_{opt}(a))$, for every $a \in A$:

$$c(sol_{opt}) \geq \mathtt{T}_{opt} + \sum_{a\in A\setminus a_i} c_a^* \tag{18}$$

Now we use (16) to obtain:

$$c(sol_{opt}) \geq \mathtt{T}^- + c(sol_{min}) - c(sol^-) + \sum_{a\in A\setminus a_i} c_a^* \tag{19}$$

Since $c(sol_{min}) > c(sol_{opt})$, we substitute with (19) to write:

$$c(sol_{min}) \geq \mathtt{T}^- + c(sol_{min}) - c(sol^-) + \sum_{a\in A\setminus a_i} c_a^*. \tag{20}$$

Canceling $c(sol_{min})$ in (20), we obtain:

$$c(sol^-) > \mathtt{T}^- + \sum_{a\in A\setminus a_i} c_a^* = c(sol^-),$$

which is a contradiction.

Thus, after we find the first solution sol_{min}, we run the solver again for makespan $\mathtt{T}_{max}$ given by Theorem 1, as specified by Algorithm 1. We assume that function Solve invokes an ASP solver for the given ASP program, and returns a model, if one exists, and `unsatisfiable` otherwise. The approach implemented by Algorithm 1 was recently evaluated by [13] for their Picat-based MAPF solver.

V. A LINEAR ENCODING

The encoding we have proposed is quadratic in the number of agents. In this section we show how to make it linear by introducing new atoms to the encoding. Specifically, we introduce the following atoms:

Algorithm 1 Sum-of-Costs MAPF via ASP

Input: A MAPF instance $I = (G, \mathcal{A}, init, goal)$

```
T⁻ ← max_{a∈A} c*(a)
T ← T⁻
C⁻ ← Σ_{a∈A} c*(a)
while true do
    P ← Encode I with makespan T
    M ← Solve P
    if M ≠ unsatisfiable then
        C ← cost of the solution given by M
        Δ ← C − C⁻ − 1 // bound of Thm. 1
        if T⁻ + Δ ≤ T then
            return solution given by M
        else
            P ← Encode I with makespan T⁻ + Δ
            M* ← Solve P
            return solution given by M*
    T ← T + 1
```

- $rt(x, y, t)$ (resp. $lt(x, y, t)$) which specifies that the edge between (x, y) and $(x+1, y)$ was traversed by some agent at time t from left to right (resp. from right to left).
- $ut(x, y, t)$ (resp. $dt(x, y, t)$) which specifies that the edge between (x, y) and $(x, y+1)$ was traversed upwards (resp. downwards) by some agent at time t.
- $st(x, y, t)$, indicates that some agent stayed at (x, y), that is, it performed a wait action at time t.

The dynamics of these atoms are defined using one rule with variables for each one of the possible move directions:

$$\begin{aligned} rt(X, Y, T) &\leftarrow exec(A, right, T), \quad at(A, X, Y, T), \\ lt(X-1, Y, T) &\leftarrow exec(A, left, T), \quad at(A, X, Y, T), \\ ut(X, Y, T) &\leftarrow exec(A, up, T), \quad at(A, X, Y, T), \\ dt(X, Y-1, T) &\leftarrow exec(A, down, T), \quad at(A, X, Y, T). \end{aligned} \tag{21}$$

while the rule for st is:

$$st(X, Y, T) \leftarrow at(A, X, Y, T), \quad exec(A, wait, T). \tag{22}$$

It is easy to verify that these rules do not mention pairs of different agents, unlike (10) and (11), and as such after grounding we end with $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$ rules, and therefore the resulting encoding is linear in $|\mathcal{A}|$.

A. VERTEX CONFLICTS

Using these predicates we now express the fact that a cell cannot be entered at the same time instant by two different agents. This requires six rules each of which corresponds to a pair of actions in $\{right, left, up, down\}$. For example, the following rule expresses that (X, Y) cannot be entered by an agent performing a *down* action at the same time that is entered by another agent performing *up* :

$$\leftarrow lt(X, Y, T), \quad dt(X, Y, T). \tag{23}$$

Finally, we express that if an agent enters a cell, then another agent could have not stayed at that cell. We do this with the following rules:

$$\leftarrow st(X, Y, T), \quad lt(X, Y, T). \tag{24}$$
$$\leftarrow st(X, Y, T), \quad rt(X-1, Y, T). \tag{25}$$
$$\leftarrow st(X, Y, T), \quad ut(X, Y-1, T). \tag{26}$$
$$\leftarrow st(X, Y, T), \quad dt(X, Y, T). \tag{27}$$

B. SWAP CONFLICTS

For swap conflicts we simply express that a single edge cannot be traversed in two different directions. We do this using the following rules:

$$\leftarrow rt(X, Y, T), \quad lt(X, Y, T).$$
$$\leftarrow ut(X, Y, T), \quad dt(X, Y, T).$$

After grounding the encoding of vertex and swap conflicts is $\Theta(|V| \cdot \mathtt{T})$.

C. FOLLOW CONFLICTS

To encode follow conflicts we prevent an agent entering a cell at time t which was occupied by another agent, also at time t, using the following rules:

$$\leftarrow at(A, X, Y, T), \quad rt(X-1, Y, T). \tag{28}$$
$$\leftarrow at(A, X, Y, T), \quad lt(X, Y, T). \tag{29}$$
$$\leftarrow at(A, X, Y, T), \quad ut(X, Y-1, T). \tag{30}$$
$$\leftarrow at(A, X, Y, T), \quad dt(X, Y, T). \tag{31}$$

Observe that, under follow conflicts, some vertex conflicts are implicit, and it is not necessary to include rules (24)-(27), nor (22). After grounding, follow conflicts expand into $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$ rules.

VI. USING SEARCH TO REDUCE THE ATOMS

We can exploit our run of Dijkstra's algorithm during preprocessing time to generate an even smaller encoding by replacing rule (6) by the following rule.

$$\begin{aligned} at(A, X, Y, T) \leftarrow\ & at\,(A, X', Y', T-1), \\ & exec\,(A, M, T), \\ & delta\,(M, X', Y', X, Y), \\ & cost\_to\_go(A, X, Y, C), \\ & T + C <= \mathtt{T}. \end{aligned} \tag{32}$$

where $cost\_to\_go(A, X, Y, C)$ specifies that C is the minimum number of actions needed to go from (X, Y) to the goal of agent A. This way we can ignore the generation of rules to positions that will not reach the goal, generating a much more compact encoding. This idea is related to the use of MDDs graphs in MDD-SAT [11], but does not require the generation of the MDD, so it is conceptually simpler.

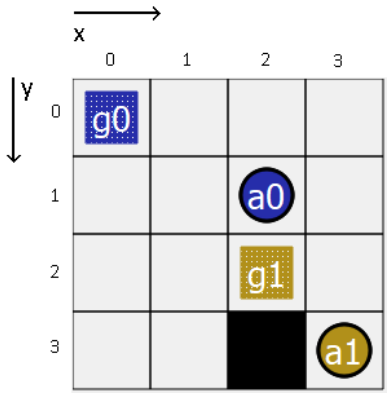

FIGURE 2. An example MAPF instance.

A. REPRESENTING AN INSTANCE

To represent a MAPF instance, we define predicates *at* for time instant 0, and predicates *goal* and *obstacle*. Figure 2 shows an example of an instance which is represented using the following rules.

$$at(a_0, 2, 1, 0),$$
$$at(a_1, 3, 3, 0),$$
$$goal(a_0, 0, 0),$$
$$goal(a_1, 2, 2),$$
$$obstacle(2, 3).$$

A model returned by the solver contains *exec* predicates, from where the solution can be extracted. In our example, a model of the program may contain:

$$exec(a_0, left, 0),$$
$$exec(a_0, left, 1),$$
$$exec(a_0, down, 2),$$
$$exec(a_1, right, 0),$$
$$exec(a_1, up, 1).$$

VII. EMPIRICAL EVALUATION

The objective of our empirical evaluation was to compare the performance of the different variants of our translation against representatives of search-based and SAT-based solvers. We compared to the publicly available SAT-based solver MDD-SAT [22] (enc = mdd), the SMT compilation-based solver SMT-CBS [17], and the search-based solver ICBS-h [9]. Our evaluation is focused on relatively small grids since grounding time grows too much for larger grids (e.g., 512 × 512), making the approach impractical.

It is important to note that both MDD-SAT and SMT-CBS are solvers that do respect follow conflicts, while the search-based solver ICBS-h does not. Since most part of the solvers proposed in the MAPF literature do not take into account follow conflicts, in the first part of our analysis (Subsections VII-A–VII-D) we evaluate our solver considering vertex and swap conflicts only. Thus the performance of MDD-SAT and SMT-SAT should be considered as a reference; a version of these solvers without follow conflicts is not available. In Subsection VII-E, we compare our solver with follow conflicts activated, to MDD-SAT.

We compared different encodings based on each of our improvements: ASP-basic is the basic encoding that uses quadratic conflict resolution and grid-dependent penalties. To name the non-basic variants of our solver, we use ASP followed by some of the identifiers GI, LC, and CG. GI refers to the use of grid-independent penalties. LC refers to the use of our linear conflict encoding. Finally, CG refers to the use of Dijkstra's algorithm as seen on rule (14).

The code used for our implementation was written in Python 3.7 using Clingo 5.3 [24] for the ASP solver. Clingo was run with 4 threads in parallel mode, and using USC as the optimization strategy, unless otherwise stated. All algorithms we compared with were obtained from their authors.

All experiments were run on a 3.40GHz Intel Core i5-3570K with 8GB of memory running Linux. We set a runtime limit of 5 minutes for all problems.

A. N × N GRIDS WITH RANDOM OBSTACLES

First, we experimented on 8 × 8 and 20 × 20 randomly generated problems with 10% obstacles. For 8 × 8 (resp. 20 × 20) we generate 150 (resp. 260) problems with the number of agents in {4, . . . , 18} (resp. {20, 22, . . . , 70}), where for each number of agents we generate 10 instances. Success rates—defined as the proportion of instances that can be solved within the runtime limit—and number of problems solved versus time for the 20 × 20 are shown in Figure 3. We omit the results for 8 × 8 since they look very similar to those for 20 × 20.

In the 8 × 8 grids, as the number of agents increase, all our encodings outperform the other algorithms in terms of success rate. We also observe that our modifications to the basic encoding pay off substantially. We observe a substantial difference between our grid-dependent encoding and our grid-independent encoding.

For 20 × 20 grids, we observe that our linear encoding solves almost all problems and substantially outperforms our quadratic (basic) encoding. We do not observe in this case an important impact of the grid-dependent encoding over grid-independent encoding. In contrast, it is interesting to see the benefits of using the CG mode as a way to generate a smaller encoding, as shown in Figure 3 CG greatly improves the solving time. In fact we found that the average solving time is a factor of 1.74 smaller using ASP-GI-LC-CG in comparison to ASP-GI-LC. Also we found a decrease by a factor of 1.98 on the average runtime of the grounding process when using CG.

To understand the influence of the number of obstacles on the grid, we experimented on 20 × 20 randomly generated problems with 20 agents. We evaluated the best-performing configuration of the previous experiments using 1 and 4 threads. We generate 100 problems by varying the percentages of obstacles in {0%, 5%, 10%, . . . , 50%} (for each

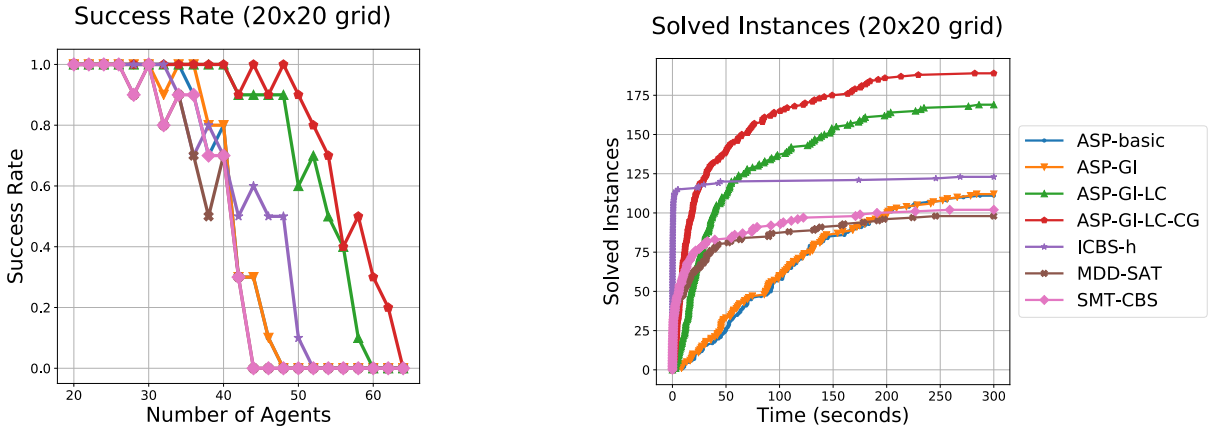

FIGURE 3. Success rate and number of instances solved versus time on 20 × 20 grids.

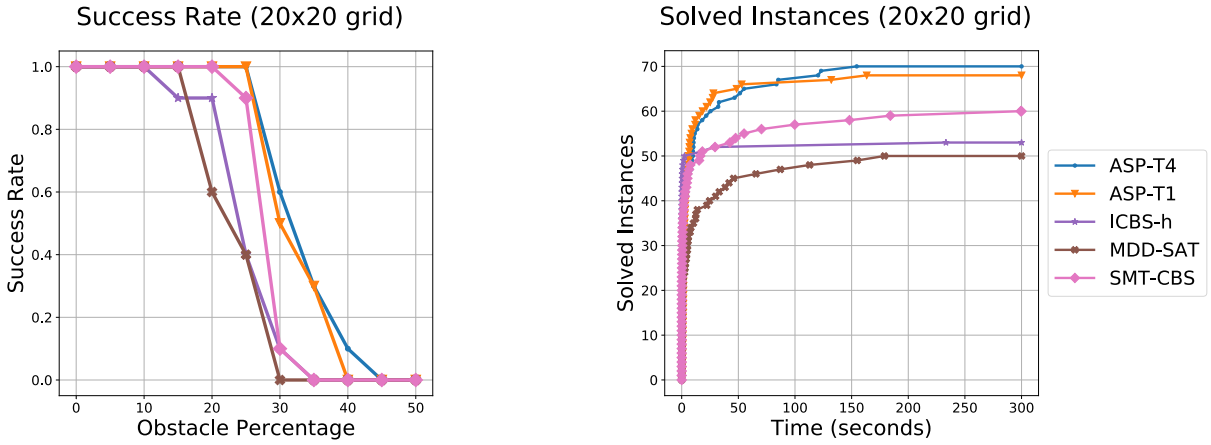

FIGURE 4. Success rate and number of instances solved versus time on 20 × 20 grids. ASP-T1 is the best configuration (ASP-GI-LC-CG) ran with 1 thread, and ASP-T4 is ASP-GI-LC-CG ran with 4 threads.

percentage we generate 10 random instances). Figure 4 shows the results. Again our ASP formulations outperform other algorithms as the number of agents increases. In addition, no significant differences are observed between the 1- and 4-thread variants.

B. WAREHOUSE EXPERIMENTS

We experimented with a 9 × 21 warehouse grid used in the MAPF literature (e.g., [9]), shown in Figure 6. We selected random initial and goal locations over the left and right borders of the grid. We generated 10 instances for each number of agents in {4, . . . , 20}. In this experiment we only used ASP-GI-LC-CG, the encoding with best results on the grid experiments. Also, because clingo offers two optimization strategies: Branch-and-bound (BB) [25] based optimization and unsatisfiable-core (USC) based optimization [26], we wanted to test our encoding on both of them to analyze the effects. Success rates, and number of problems solved versus time are shown in Figure 5.

Results show the benefits of our approach on this type of grids, where we outperform substantially the planners we compare with. Given the limited amount of free space on these grids, $|V|$ is smaller and thus our encoding is rather compact. These results also illustrate that our approach scales better than other approaches when the number of potential conflicts grow.

Regarding USC versus BB. USC starts with a minimum cost solution that does not necessarily satisfy the constraints given in the ASP program. If the program has no model, USC attempts to find a higher cost solutions incrementally, until one is found. BB, on the other hand, finds a solution and uses the cost of this solution to prune the search. We observe that USC is the best approach for cost-optimal MAPF.

We also experimented with warehouses with sizes 18×21, 27×21, 36×21, 9×39, 9×57, and 36×57. In the largest warehouse (36×57) our approach is slightly outperformed by ICBS-h, whereas on the rest of the warehouses, our approach exhibits a tendency similar to that of Figure 5.

C. MAKESPAN-OPTIMAL SOLUTIONS

Most of the MAPF literature has focused on finding cost-optimal solutions, and thus the main contribution of this

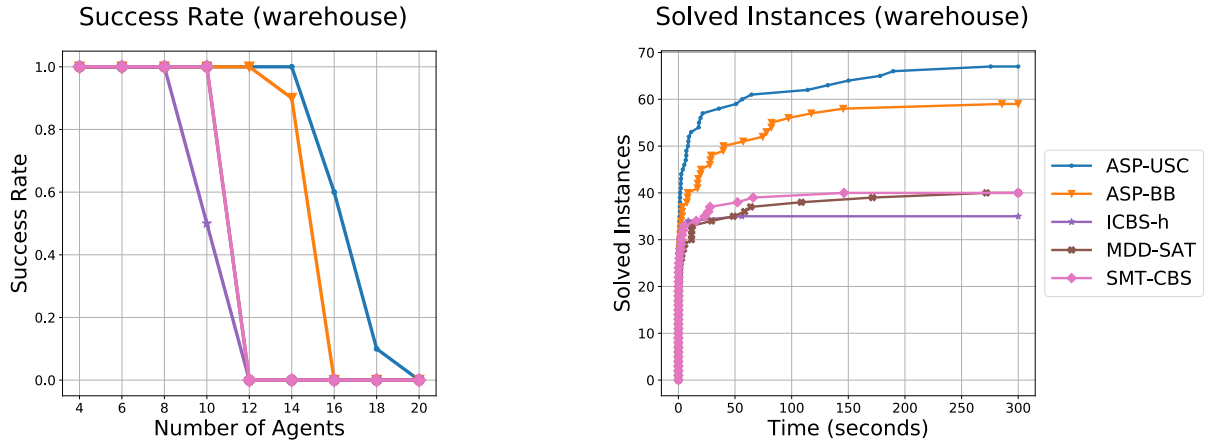

FIGURE 5. Success rate and number of instances solved versus time on the Warehouse problem.

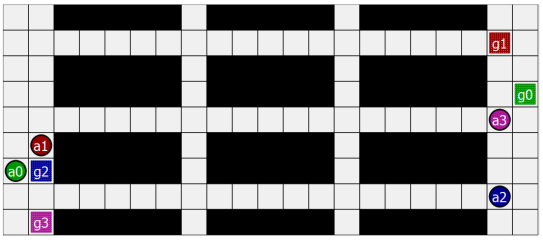

FIGURE 6. Warehouse problem example.

article is to propose a cost-optimal approach to MAPF using ASP. Notwithstanding, finding makespan-optimal solutions (which, as seen above, are not cost-optimal) requires less time. Indeed, Algorithm 1 incrementally runs the ASP solver for increasing makespan values until one first solution is found. After that, it uses the bound of Theorem 1 to find another solution, with larger makespan, which is guaranteed to be cost-optimal. Because this last run requires uses a larger makespan, the encoding is also larger, which has an impact over runtime. This raises a number of questions regarding the difference between finding makespan-optimal and cost-optimal solutions. How much solving time would we save if we simply computed makespan-optimal rather than cost-optimal solutions? What is the suboptimality of makespan-optimal solutions compared to cost-optimal solutions? We attempt to answer these questions here.

For the first experiment, we use the same set of 20×20 grids of the previous section, but since we noted the makespan-optimal solver scales better, we extended the percentage of obstacles to include values in $\{50\%, 55\%, \ldots, 70\%\}$. We record the time taken to find the makespan-optimal solution, and its makespan.

For the analysis of the results, first we focus on the suboptimality of makespan-optimal solutions. We define percentage suboptimality as the percentage of the difference between the cost of both solutions with respect to the cost of the cost-optimal solution. Tables 1 and 2 show the average percentage suboptimality, respectively, when the number of agents is

TABLE 1. Average percentage suboptimality of the makespan-optimal solution and percentage of instances in which makespan-optimal and cost-optimal solutions have the same cost, for obstacle-free 20 × 20 with 10%-obstacle grids in which we vary the number of agents.

# agents	Avg. Suboptimality	Same cost %
20	0.000%	100.0%
22	0.496%	80.0%
24	0.062%	90.0%
26	0.000%	100.0%
28	0.000%	100.0%
30	0.000%	100.0%
32	0.485%	90.0%
34	0.000%	100.0%
36	0.000%	100.0%
38	0.000%	100.0%
40	0.323%	80.0%
42	0.000%	100.0%
44	0.000%	100.0%
46	0.000%	100.0%
48	0.340%	80.0%
50	0.000%	100.0%
52	0.000%	100.0%
54	0.019%	85.7%
56	0.000%	100.0%
58	0.000%	100.0%
60	0.000%	100.0%
62	0.000%	100.0%
64	–	–
66	0.000%	100.0%
68	0.000%	100.0%
70	–	–

increased and when the percentage of obstacles is increased. Finally, Table 3 shows the average percentage suboptimality of the warehouse map of Figure 6 when the number agents is increased.

We observe that average percentage suboptimality very low in general: for most configurations is less than 0.5%, and its maximum value is only 1.668% on a 20×20 with a relatively high number of obstacles. This means that makespan-optimal and cost-optimal solutions are very similar in general, and almost always equivalent.

The next question we address is how is performance affected when we configure the solver to find makespan-optimal solutions instead of cost-optimal solutions. Figures 7,

TABLE 2. Average percentage suboptimality of the makespan-optimal solution and percentage of instances in which makespan-optimal and cost-optimal solutions have the same cost, for 20-agent 20 × 20 grids in which we vary the percentage of obstacles.

Obstacle %	Avg. Suboptimality	Same cost %
0	0.000%	100.0%
5	0.112%	90.0%
10	0.000%	100.0%
15	0.359%	80.0%
20	0.575%	80.0%
25	0.238%	80.0%
30	0.801%	50.0%
35	1.668%	33.3%
40	0.909%	0.0%

TABLE 3. Average percentage suboptimality of the makespan-optimal solution and percentage of instances in which makespan-optimal and cost-optimal solutions have the same cost, on the warehouse map of Figure 6.

# agents	Avg. Suboptimality	Same cost %
4	0.000%	100.0%
6	0.073%	90.0%
8	0.054%	90.0%
10	0.258%	80.0%
12	0.036%	90.0%
14	0.120%	80.0%
16	0.570%	66.7%
18	0.000%	100.0%

8, and 9 show the performance of ASP-makespan, our ASP encoding configured to return makespan-optimal solutions, versus ASP-cost, our cost-optimal solver. We compare the performance of both solvers over the same scenarios in which we compared above (warehouses with varying number of agents and 20 × 20 grids with varying number of agents and obstacle rates).

We observe that ASP-makespan scales substantially better than ASP-cost when the number of agents or the obstacle rate is increased. Indeed, in the 20 × 20 grid experiment when we vary the number of agents, ASP-makespan solves 46 more instances than ASP-cost (a 24% increase), in the 20 × 20 grid experiment where we vary the obstacle rate, ASP-makespan solves 31 more instances than ASP-optimal (a 44% increase), and, finally, in the warehouse experiment, where we vary the number of agents, ASP-makespan solves 19 more instances than ASP-cost (a 28% increase).

Both ASP-cost and ASP-makespan invoke the ASP solver iteratively increasing the makespan until one solution is found. While ASP-makespan returns the first solution found, ASP-cost needs to run the solver one more time for an encoding with makespan $\mathtt{T}^- + \Delta$ (Line 14; Algorithm 1). The value Δ is given by the theoretical bound of Theorem 1.

Given that ASP-cost is outperformed quite significantly by ASP-makespan, a natural question that arises is *how tight is such a bound*. To answer this question, let us define δ as $\mathtt{T}^- + \Delta - \mathtt{T}$, that is, the difference between the makespan of the last and the makespan used in the second-last call to the ASP solver. Hence, intuitively, δ is a proxy to the "extra effort" that the ASP-solver needs to make in order to find the last solution (recall that the encoding is linear on the makespan). Furthermore, let us define δ_{oracle} as the difference between the makespan of the cost-optimal solution and the makespan used in the second-last run of ASP-cost. That is, intuitively, δ_{oracle} is a the minimum possible extra effort that ASP-cost would need to make to find the optimal solution if we had an oracle for the makespan of the cost-optimal solution.

TABLE 4. Average δ and average δ_{oracle} which are, respectively, proxies to the actual and ideal additional effort needed to find the cost-optimal solution. The number of solved instances is also shown for reference. We experiment over a 20 × 20 grid with a 10% obstacle rate and an increasing number of agents.

# agents	solved	Avg. δ_{oracle}	Avg. δ
20	10	0.0	0.0
22	10	0.3	2.0
24	10	0.2	2.0
26	10	0.0	1.5
28	10	0.0	3.5
30	10	0.0	4.0
32	10	0.2	4.0
34	10	0.0	6.0
36	10	0.0	9.0
38	10	0.0	4.0
40	10	0.4	7.5
42	9	0.1	9.0
44	10	0.1	7.5
46	9	0.0	11.0
48	10	0.4	13.0
50	9	0.0	17.0
52	8	0.1	12.5
54	7	0.1	16.0
56	4	0.5	12.0
58	5	0.0	17.0
60	3	0.0	16.0
62	2	0.0	23.0
64	0	–	–
66	1	0.0	26.0
68	2	0.0	24.0
70	0	–	–

Tables 4, 5, and 6 show the average δ and δ_{oracle} for the grid and warehouse benchmarks. We observe that δ tends to be higher than δ_{oracle}. Moreover, the growth rate of δ is higher than that of δ_{oracle}. This suggests that there is significant potential for the improvement of ASP-cost. A key improvement factor would come from finding a bound better than that given by Theorem 1.

D. OBSTACLE-FREE N × N GRIDS

In the experiments presented above on a 20 × 20 grid, we observed that success rate drops as the number of agents increases. Indeed, in Figure 7 it can be observed that when we solve instances with 58 agents we obtain a success rate of 50%, but when we attempt to solve larger problems, we obtain lower success rates. Therefore one could argue that for 20 × 20, 58 agents is the point at which problems start becoming hard. In other words, given that the area of the 20 × 20 grid is 400, we say that problems become harder when the occupation rate is around 58/400 or 14.5%.

We wanted to understand for which occupation rate we observe that success rate drops below 50% as a function of grid size. To do this we designed a new set of experiments in which we use obstacle-free grids of size $N \times N$ with $N \in \{8, 16, 32\}$. For grids of size 8×8 we set the number of agents to a number in $\{2, 4, \ldots, N^2 - 2\}$, and generated 10 random

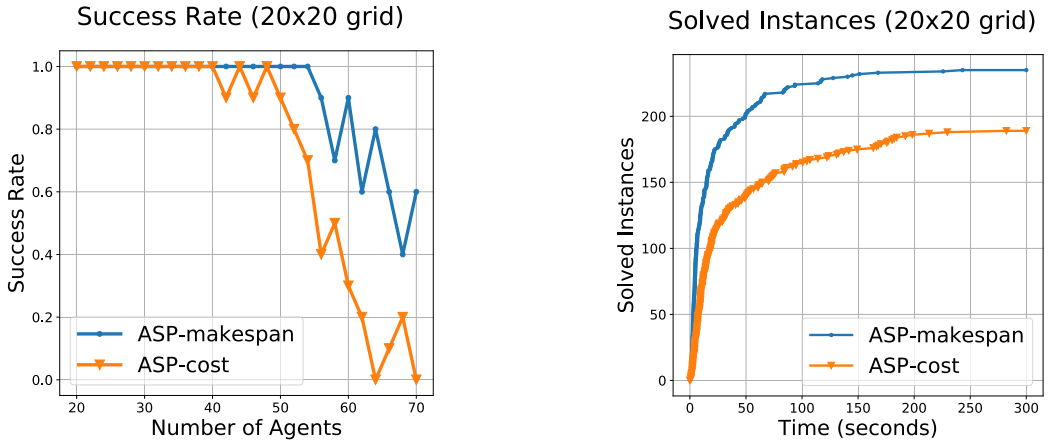

FIGURE 7. Success rate and number of solved instances as the number of agents is varied on a 20 × 20 grid with 10% obstacle rate. ASP-makespan finds a makespan-optimal solution whereas ASP-cost finds a cost-optimal solution.

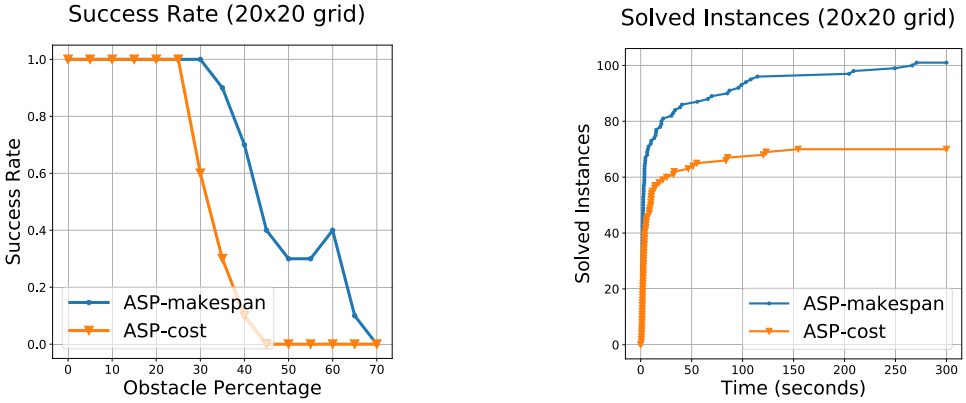

FIGURE 8. Success rate and number of solved instances as the obstacle rate is varied on a 20 × 20 grid with 20 agents. ASP-makespan finds a makespan-optimal solution whereas ASP-cost finds a cost-optimal solution.

TABLE 5. On a 20 × 20 grid with 20 agents and an increasing obstacle percentage, the average δ and average δ_{oracle} are shown, which are, respectively, proxies to the actual and ideal additional effort needed to find the cost-optimal solution. The number of solved instances is also shown for reference.

Obstacle %	solved	Avg. δ_{oracle}	Avg. δ
0	10	0.0	0.0
5	10	0.2	1.0
10	10	0.0	0.5
15	10	0.4	2.5
20	10	0.4	8.5
25	10	0.6	12.5
30	6	2.0	26.0
35	3	2.3	52.0
40	1	5.0	37.0

TABLE 6. On a warehouse map, with an increasing number of agents, the average δ and average δ_{oracle} are shown, which are, respectively, proxies to the actual and ideal additional effort needed to find the cost-optimal solution. The number of solved instances is also shown for reference.

# agents	solved	Avg. δ_{oracle}	Avg. δ
4	10	0.1	0.0
6	10	0.2	1.0
8	10	0.3	3.0
10	10	0.7	8.0
12	10	0.8	13.0
14	10	0.4	17.5
16	6	1.8	21.0
18	1	1.0	23.0

problems for each number of agents. For the 16 × 16 and 32 × 32 grids we set the number of agents to a value in {5, 10, . . . , 150} and also generate 10 random problems.

Figure 10 allows us to make two observations.

1) The point at which the success rate crosses 50% is approximately 31 agents (48% occupation) for 8 × 8, 66 agents (25% occupation) for 16 × 16, and 70 agents (1.7 % occupation) for 32 × 32. Thus the occupation rate at which problems become harder drops quickly as the size of the grids increase.
2) On 8 × 8 and 16 × 16 grids our solver outperforms ICBS-h, but it does not do so on 32 × 32 grids.

Together these two observations may seem to indicate that our solver does not scale well for larger grids. We argue that this is not necessarily the case. Indeed, what happens is that problems are becoming much harder when the grid size is

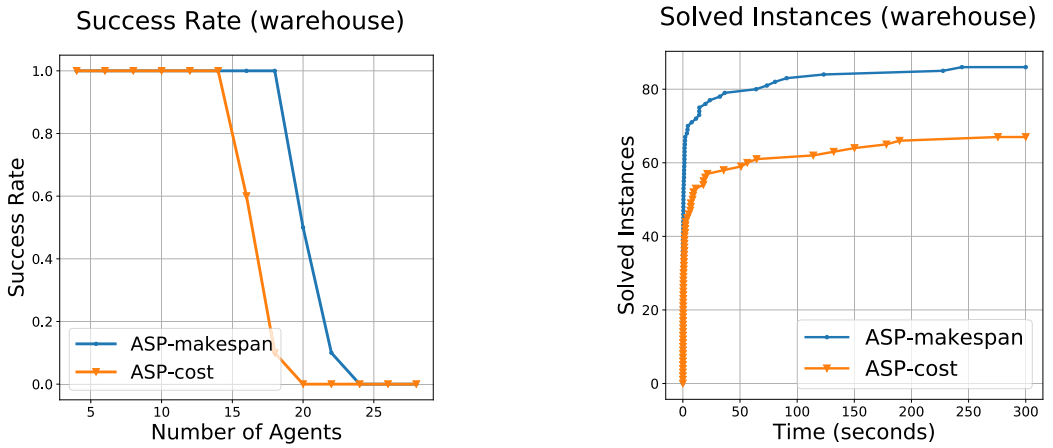

FIGURE 9. Success rate and number of solved instances as the number of agents is varied on the warehouse map of Figure 6. ASP-makespan finds a makespan-optimal solution whereas ASP-cost finds a cost-optimal solution.

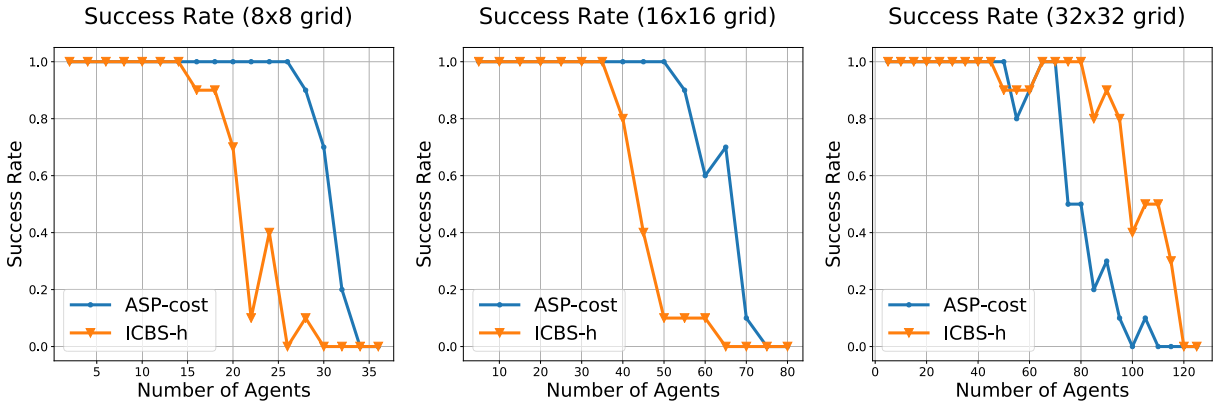

FIGURE 10. Success rate of our solver versus ICBS-h on obstacle-free grids of different sizes.

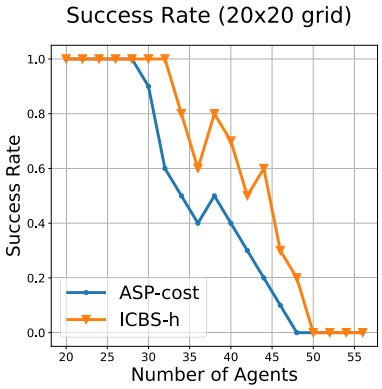

FIGURE 11. Success rate on a 20 × 20 grid with 10% obstacle rate, where timeout per instance is set to 15 seconds, and the number of agents varies.

increased and thus they require more time. The apparent drop in performance is therefore due to the fact that we did not increase the runtime limit (set for 300 seconds for every run).

To test this hypothesis, we use our 20 × 20 data and show how would success rate would look like if the runtime limit had been set to 15 seconds (Figure 11). We observe that given just 15 seconds per instance, ICBS-h outperforms our solver.

Another piece of evidence suggesting that 32 × 32 grids require more execution time per instance comes from the form of the average runtime curves. Figure 12 shows the total runtime and grounding time (i.e., the time spent on the grounding phase) on a logarithmic scale. In both the 8 × 8 and the 16 × 16 grids, using a logarithmic scale, we observe that the runtime curve is essentially a straight line, showing that average runtime grows exponentially with the number of agents. For the 32 × 32 grids, curves resemble a logarithm.

Interestingly, when we limit the runtime to 15 seconds on our 20 × 20 grid we also obtain a logarithmic-like curve (Figure 13). Our ASP encodings grow linearly with the number of agents and the fact that runtime grows exponentially is consistent with that fact. But since runtime on our 32 × 32 grid experiments does not exhibit this tendency, we conclude that more timeout is needed per instance. More research is necessary to establish whether or not in any setup it is the case that a linear rather than an exponential increase in runtime is due to a time limit set too low.

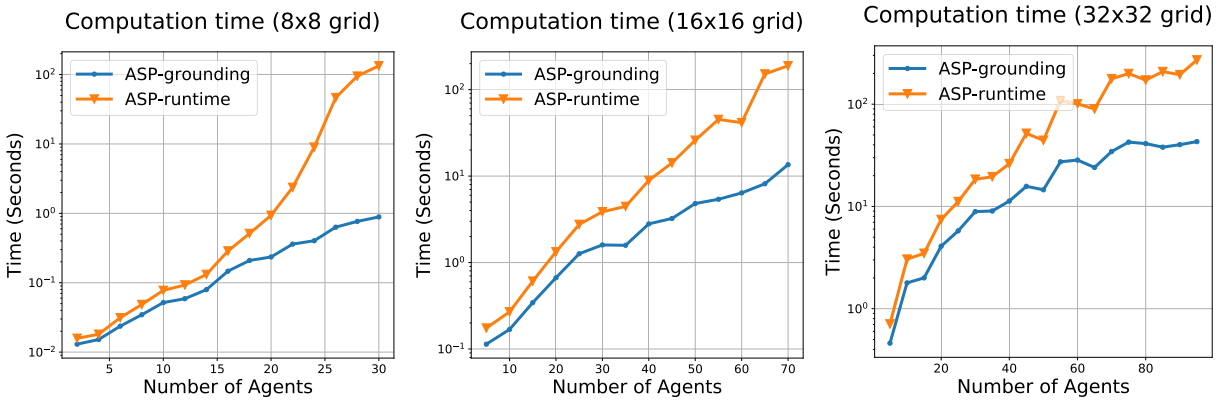

FIGURE 12. Execution time and grounding time required for the instances on obstacle-free grids of different sizes.

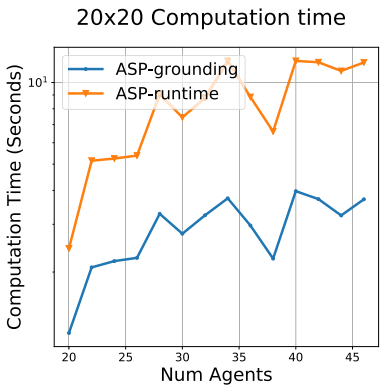

FIGURE 13. Total solving and grounding time for the 20 × 20 grid with 10% obstacle rate, where timeout per instance is set to 15 seconds, and the number of agents varies.

E. INCORPORATING FOLLOW CONFLICTS

As we mentioned above, follow conflicts are not taken into account by many of the solvers in the literature. In this Section, however, we compare our approach, with follow conflicts turned on, to MDD-SAT, a solver that natively supports these conflicts.

Figure 14 shows results obtained for the 8×8 and 20×20 grids described in Section VII-A. We observe that in 8×8 grids the performance of both solvers is very similar, while in 20×20 grids our approach seems to scale slightly better than MDD-SAT.

VIII. SUMMARY AND PERSPECTIVES

In this article we proposed the first compilation of MAPF to ASP for sum-of-costs optimization. We started off by proposing a basic encoding that is quadratic on the number of agents, just like existing approaches to MAPF via ASP (e.g. [14], [16], [24], [27]), and like the encoding of MAPF to SAT of MDD-SAT [11], a state-of-the-art SAT-based sum-of-costs MAPF solver. Second, we proposed an encoding that grows linearly with the number of agents, and show how we can benefit by running Dijkstra's algorithm during preprocessing time to generate a more compact encoding.

In our empirical evaluation we used square grids with increasing number of agents/obstacles, and a warehouse map. We observed that our linear encoding outperforms our quadratic encoding, solving substantially more instances given the same amount of time, leading to a higher success rate. Moreover, our approach outperforms the search-based solver ICBS-h, and is competitive with the SAT-based solver MDD-SAT when follow conflicts are considered. We conclude that the compilation of MAPF to ASP is a competitive approach for solving highly congested MAPF instances.

In the second part of our empirical evaluation, we studied the performance of our approach for finding makespan-optimal solutions, that is, solutions in which, lexicographically, makespan is minimized first and cost is minimized second. We observe that this approach outperforms the original approach substantially in terms of solving time and success rate, while only sacrificing optimality slightly; indeed, makespan-optimal solutions are often less than 1% suboptimal. This suggests that from a practical perspective, in applications in which there is high congestion, makespan-optimal solutions may be preferable to cost-optimal solutions.

Our approach, like other compilation approaches, does not scale to large grids. Indeed, in such grids grounding time becomes exceedingly large, making our approach impractical. An open line of research is to incorporate our ideas to distributed, non-optimal compilation-based approaches, that scale to much larger grids. An example of such an approach is [27], which can scale to grids of size 100×100 with more than 1,000 agents. In its core, [27] uses a quadratic compilation of MAPF to ASP, which could be made linear—and hence likely faster—using our ideas.

Another interesting line of research is to construct linear MAPF-to-SAT encodings, were, based on the improvements seen with ASP, we would expect significant improvements in performance. An important observation, however, is that ASP compilations are more elegant than SAT compilations since grounding in ASP is carried out starting from a lifted representation.

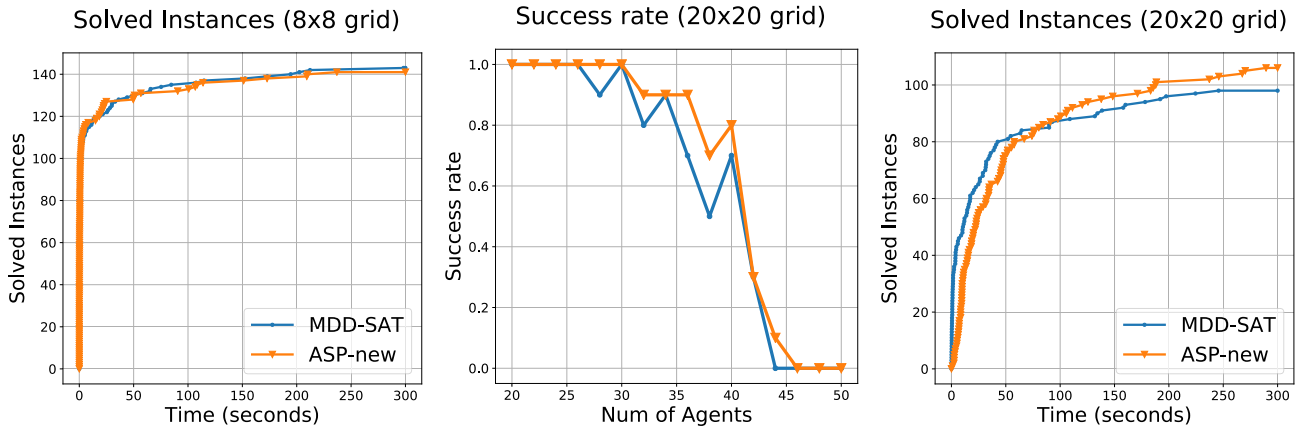

FIGURE 14. A comparison of our approach with follow conflicts and MDD-SAT on 8 × 8 and 20 × 20 grids, with 10% obstacles and an increasing number of agents.

REFERENCES

[1] K. C. Wang and A. Botea, "Fast and memory-efficient multi-agent pathfinding," in *Proc. 18th Int. Conf. Automated Planning Scheduling ICAPS*, Sep. 2008, pp. 380–387. [Online]. Available: http://www.aaai.org/Library/ICAPS/2008/icaps08-047.php

[2] J. Li, M. Gong, Z. Liang, W. Liu, Z. Tong, L. Yi, R. Morris, C. Pasearanu, and S. Koenig, "Departure scheduling and taxiway path planning under uncertainty," in *Proc. AIAA Aviation Forum*, Jun. 2019, p. 2930.

[3] P. Surynek, "An optimization variant of multi-robot path planning is intractable," in *Proc. 24th AAAI Conf. AI AAAI*, M. Fox and D. Poole, Eds. Palo Alto, CA, USA: AAAI Press, 2010, pp. 1261–1263. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1768

[4] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Proc. 27th AAAI Conf. AAAI*, Bellevue, WA, USA, 2013, pp. 1443–1449.

[5] W. Honig, J. A. Preiss, T. K. S. Kumar, G. S. Sukhatme, and N. Ayanian, "Trajectory planning for quadrotor swarms," *IEEE Trans. Robot.*, vol. 34, no. 4, pp. 856–869, Aug. 2018.

[6] D. Le and E. Plaku, "Multi-robot motion planning with dynamics via coordinated sampling-based expansion guided by multi-agent search," *IEEE Robot. Autom. Lett.*, vol. 4, no. 2, pp. 1868–1875, Apr. 2019, doi: 10.1109/LRA.2019.2898087.

[7] T. S. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Proc. 24th AAAI Conf. AAAI*, Atlanta, Georgia, 2010, pp. 173–178. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1926

[8] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent path finding," in *Proc. 26th AAAI Conf. AAAI*, Toronto, ON, Canada, 2012, pp. 563–569.

[9] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T. K. S. Kumar, and S. Koenig, "Adding heuristics to conflict-based search for multi-agent path finding," in *Proc. 18th Int. Conf. Automated Planning Scheduling ICAPS*, Delft, The Netherlands, Jun. 2018, pp. 83–87. [Online]. Available: https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17735

[10] J. Li, A. Felner, E. Boyarski, H. Ma, and S. Koenig, "Improved heuristics for multi-agent path finding with conflict-based search," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 442–449, doi: 10.24963/ijcai.2019/63.

[11] P. Surynek, A. Felner, R. Stern, and E. Boyarski, "Efficient SAT approach to multi-agent path finding under the sum of costs objective," in *Proc. 22nd Eur. Conf. ECAI*, 2016, pp. 810–818.

[12] R. Bartak, N.-F. Zhou, R. Stern, E. Boyarski, and P. Surynek, "Modeling and solving the multi-agent pathfinding problem in picat," in *Proc. IEEE 29th Int. Conf. Tools Artif. Intell. (ICTAI)*, Nov. 2017, pp. 959–966, doi: 10.1109/ICTAI.2017.00147.

[13] R. Barták and J. Svancara, "On SAT-based approaches for multi-agent path finding with the sum-of-costs objective," in *Proc. 12th Symp. Combinat. Search SoCS*, P. Surynek and W. Yeoh, Eds. Palo Alto, CA, USA: AAAI Press, 2019, pp. 10–17. [Online]. Available: https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18323

[14] E. Erdem, D. G. Kisa, U. Öztok, and P. Schüller, "A general formal framework for pathfinding problems with multiple agents," in *Proc. 27th AAAI Conf. AI AAAI*, Palo Alto, CA, USA: AAAI Press, 2013, pp. 290–296.

[15] M. Gebser, P. Obermeier, T. Otto, T. Schaub, O. Sabuncu, V. Nguyen, and T. C. Son, "Experimenting with robotic intra-logistics domains," *Theory Pract. Log. Program.*, vol. 18, nos. 3–4, pp. 502–519, Jul. 2018.

[16] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, and W. Yeoh, "Generalized target assignment and path finding using answer set programming," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, Aug. 2017, pp. 1216–1223, doi: 10.24963/ijcai.2017/169.

[17] P. Surynek, "Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 1177–1183, doi: 10.24963/ijcai.2019/164.

[18] R. N. Gómez, C. Hernández, and J. A. Baier, "Solving sum-of-costs multi-agent pathfinding with answer-set programming," in *Proc. 34th AAAI Conf. AAAI*, New York, NY, USA: AAAI Press, 2020, pp. 9867–9874.

[19] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, R. Barták, and E. Boyarski, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *Proc. 12th Symp. Combinat. Search SoCS*, P. Surynek and W. Yeoh, Eds. New York, NY, USA: AAAI Press, 2019, pp. 151–159. [Online]. Available: https://aaai.org/ocs/index.php/SOCS/SOCS19/paper/view/18341

[20] V. Lifschitz, "What is answer set programming," in *Proc. 23rd AAAI Conf. AAAI*, 2008, pp. 1594–1597. [Online]. Available: http://www.aaai.org/Library/AAAI/2008/aaai08-270.php

[21] P. Ferraris and V. Lifschitz, "Mathematical foundations of answer set programming," in *We Will Show Them! Essays in Honour of Dov Gabbay*. U.K.: King's College Publications, 2005, pp. 615–664.

[22] P. Surynek, "Compact representations of cooperative path-finding as SAT based on matchings in bipartite graphs," in *Proc. IEEE 26th Int. Conf. Tools Artif. Intell.*, Nov. 2014, pp. 875–882.

[23] H. A. Kautz and B. Selman, "Planning as satisfiability," in *Proc. 10th Eur. Conf. ECAI*, B. Neumann, Ed. Hoboken, NJ, USA: Wiley, 1992, pp. 359–363.

[24] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Clingo = ASP + control: Preliminary report," 2014, *arXiv:1405.3694*. [Online]. Available: https://arxiv.org/abs/1405.3694

[25] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-criteria optimization in answer set programming," in *Proc. Tech. Commun. 27th Int. Conf. Log. Program. (ICLP)* in Leibniz International Proceedings in Informatics (LIPIcs), vol. 11, J. P. Gallagher and M. Gelfond, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 1–10. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2011/3161

[26] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub, "Unsatisfiability-based optimization in clasp," in *Proc. Tech. Commun. 28th Int. Conf. Log. Program. (ICLP)* in Leibniz International Proceedings in Informatics (LIPIcs), vol. 17, A. Dovier and V. S. Costa, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012, pp. 211–221. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2012/3623

[27] P. Pianpak, T. C. Son, Z. O. Toups, and W. Yeoh, "A distributed solver for multi-agent path finding problems," in *Proc. 1st Int. Conf. Distrib. Artif. Intell.*, Oct. 2019, pp. 2:1–2:7, doi: 10.1145/3356464.3357702.

RODRIGO N. GÓMEZ received the bachelor's degree from the Pontificia Universidad Católica de Chile, Chile, where he is currently pursuing the master's degree in computer science. His research interests include automated planning, heuristic search, and the application of logic programming for search problems.

CARLOS HERNÁNDEZ received the bachelor's degree from the Universidad de Concepción, Chile, and the Ph.D. degree in computer science from the Universidad Autónoma de Barcelona, Spain. He is currently a Full Professor with the Department of Engineering Sciences, Universidad Andrés Bello de Chile (UNAB). His research interests include heuristic search, automated planning, and knowledge representation, with a focus on real-time, on-line, and multi-objective problems. He was the President of the Chilean Association of Computer Science (SCCC) from 2019 to 2020.

JORGE A. BAIER received the bachelor's and master's degrees from the Pontificia Universidad Católica de Chile, Chile, and the Ph.D. degree in computer science from the University of Toronto, Canada. He is currently an Associate Professor with the Department of Computer Science, Pontificia Universidad Católica de Chile (PUC), and the Associate Dean of engineering education with the School of Engineering, PUC. His research interests include automated planning, heuristic search, and knowledge representation, with a focus on the use of learning techniques in these areas. He is a member of the Association for the Advancement of Artificial Intelligence (AAAI).