Solving Sum-of-Costs Multi-Agent Pathfinding with Answer-Set Programming

Rodrigo N. Gómez,[1] **Carlos Hernández,**[2] **Jorge A. Baier**[1,3]
[1]Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile, Chile
[2]Departamento de Ciencias de la Ingeniería, Universidad Andrés Bello, Chile
[3]Instituto Milenio Fundamentos de los Datos, Chile
rigomez@uc.cl, carlos.hernandez.u@unab.cl, jabaier@ing.puc.cl

Abstract

Solving a Multi-Agent Pathfinding (MAPF) problem involves finding non-conflicting paths that lead a number of agents to their goal location. In the sum-of-costs variant of MAPF, one is also required to minimize the total number of moves performed by agents before stopping at the goal. Not surprisingly, since MAPF is combinatorial, a number of compilations to Satisfiability solving (SAT) and Answer Set Programming (ASP) exist. In this paper, we propose the first family of compilations to ASP that solve sum-of-costs MAPF over 4-connected grids. Unlike existing compilations to ASP that we are aware of, our encoding is the first that, after grounding, produces a number of clauses that is *linear* on the number of agents. In addition, the representation of the optimization objective is also carefully written, 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- and SAT-based sum-of-costs MAPF solvers when grids are congested with agents.

Introduction

Given a graph G and k agents, each of which is associated with an initial and a goal vertex of G, *Multi-Agent Pathfinding* (MAPF) is the problem of finding k conflict-free paths connecting the initial vertex with the goal of each agent.

A number of applications of MAPF exist, ranging from industrial applications, in which the increase in automation may promote the need for dozens—perhaps hundreds—of robots navigating in indoor environments (e.g., warehouses), to aviation, underground mining, and multi-agent videogames (Wang and Botea 2008).

Solving MAPF optimally is NP-complete (Yu and LaValle 2013; Ma and Koenig 2017). 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. It is not surprising, then, that building algorithms that scale reasonably well with the number of agents has been challenging.

In its simplest version, that is, MAPF over 4-connected grids, a number of approaches have been proposed, but two classes of solvers are most relevant for the research we report here. First, *search-based solvers* (e.g., Standley 2010), which use heuristic search as the main component. A state-of-the-art search-based solver is Conflict-Based Search (CBS) (Sharon et al. 2012; Felner et al. 2018; Li et al. 2019), which uses A* at its core. A second class is *compilation-based solvers*; for example, compilers from MAPF to Satisfiability Testing (SAT) (e.g., Surynek et al. 2016; Barták et al. 2017; Barták and Svancara 2019), and Answer-Set Programming (ASP) (Erdem et al. 2013; Gebser et al. 2018).

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 paper, 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 (Surynek et al. 2016), SMT-CBS (Surynek 2019) two state-of-the-art compilation-based solvers, and iCBS-h (Felner et al. 2018), a representative of the state-of-the-art in search-based MAPF. We observe that our approach outperforms both MDD-SAT and iCBS-h when congestion is high. Specifically, our approach has a greater coverage as the number of agents increase.

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, most of the code needed to solve a MAPF instance is included in this paper.

Background

In this section we describe MAPF and ASP. Our definition of MAPF follows closely that of Stern et al. (2019).

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 $init : A \rightarrow V$ and $goal : A \rightarrow V$ 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, $v_1 v_2 \dots v_n$, 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 $i \in \{1, \dots, n-1\}$. Given a path π we denote by $\pi[i]$ the i-th element in π, where $i \in \{1, \dots, |\pi|\}$.

A solution to MAPF is a function $sol : \mathcal{A} \rightarrow V^*$, 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 $\{1, \dots, M\}$, where M is the size of any of the paths in $sols$. 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 iff $\pi[i] = \rho[i]$, 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 iff $(\pi[i], \rho[i]) = (\rho[i+1], \pi[i+1])$, for some $i \in \{1, \dots, M-1\}$.

Following most of the literature in MAPF (e.g., Sharon et al. 2012), we consider only these two types of conflicts, and no other conflicts. For example, we do not consider so-called *following conflicts* (Stern et al. 2019), which do not allow an agent to occupy at time $t+1$ the position of another agent had at time t.

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 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-optimal if no other solution exists whose makespan is smaller than the makespan of sol. A makespan-optimal solution is not necessarily a cost-optimal solution. This is illustrated in Figure 1.

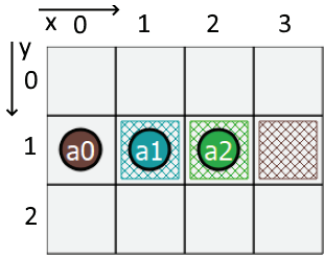

Figure 1: Problem instance where the increase of makespan yields better *sum-of-costs* solutions. The problem has 3 agents: a_0, who needs to go from $(0,1)$ to $(3,1)$. And agents a_1 and a_2 who are already at the goal at the positions $(1,1)$ and $(2,1)$ respectively. The cost-optimal solution with makespan 3 is 8, since it involves a_1 and a_2 moving away from their goal to make space for a_0. In contrast the optimal cost solution with makespan 5 is 5 as it only needs to move a_0.

Answer-Set Programming

ASP (Lifschitz 2008) 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 paper. An ASP *basic program* is a set of rules of the form:

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

where $n \geq 0$, and p, $q_1, \dots, q_n$ are so-called *atoms*. The intuitive interpretation of this rule is as follows 'p is true/provable if so are $q_1, q_2, \dots, q_n$'. 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 iff M is the subset-minimal set such that for every rule $p \leftarrow q_1, q_2, \dots, q_n \in \Pi$ such that $\{q_1, \dots, q_n\} \subseteq M$, 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:

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

Intuitively, rule (2) should be interpreted as 'p is provable if $q_1, \dots q_n$ are provable a none of $r_1, \dots, r_k$ 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 paper. We direct the interested reader to the paper by Ferraris and Lifschitz (2005).

Another relevant type of rule for our paper is:

$$|\{p_1, p_2, \dots, p_n\}| = k \leftarrow q_1, q_2, \dots, q_n.$$

Intuitively here we say that if $q_1, q_2, \dots, q_n$ are all provable, then k of the elements in $\{p_1, p_2, \dots, p_n\}$ must appear in

the 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 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.

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 Erdem et al.'s compilation (2013) to ASP, and, in some aspects similar to the MAPF-to-SAT compilation of Surynek (2014). 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 $\mathtt{T}$.

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.

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

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 $\Theta(|\mathcal{A}| + \mathtt{T} + |V|)$.

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 iff (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:

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

A grounding time predicate $delta$ results in 5 rules per each position of the grid. This defines that the total number of

[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.

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 $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$.

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:

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

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

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:

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

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 $rangeY$.

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 by Erdem et al.; Gebser et al. (2013; 2018) and Surynek et al. (2016):

$$\leftarrow at(A, X, Y, T), at(A', X, Y, T) \tag{10}$$

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.

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

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

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

Goal Achievement We specify via a constraint that no agent is away from its goal at time $\mathtt{T}$:

$$\begin{aligned} &at\_goal(A, T) \leftarrow at(A, X, Y, T), goal(A, X, Y), \\ &\leftarrow agent(A), not\ at\_goal(A, \mathtt{T}). \end{aligned} \tag{12}$$

The number of instances for rules (12) is $\Theta(\mathcal{A} \cdot |V| \cdot \mathtt{T})$.

Size of 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 $\mathtt{T}$.

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 $\mathtt{T}$ 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.

Grid-Dependent Encoding

This encoding is similar to the approach used in MDD-SAT (Surynek et al. 2016): 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 performes 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:

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

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

$$\begin{aligned} cost(A, T, 1) \leftarrow &at(A, X, Y, t), goal(A, X, Y), \\ &exec(A, M, t), M \neq wait. \end{aligned}$$

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

$$\begin{aligned} cost(A, T, 1) \leftarrow &at(A, X, Y, t), goal(A, X, Y), \\ &exec(A, wait, T), not\ at\_goal\_back(A, T). \end{aligned}$$

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 $\Theta(|\mathcal{A}| \cdot |V| \cdot \mathtt{T})$.

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:

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

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.

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

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 (Kautz and Selman 1992), in our approach we attempt to solve instances for increasing makespan T, until a solution, say sol_{min}, is found. Two observations with this process are important. First, we do not need to start increasing 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 Surynek et al. (2016):

Theorem 1 (Surynek et al. 2016) *Let sol_{min} be the makespan-optimal solution for MAPF problem P, let sol^- denote a 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 $\mathtt{T}_{max} = \mathtt{T}^- + c(sol_{min}) - c(sol^-) - 1$.*

Thus, after we find the first solution sol_{min}, we run the solver again for makespan $\mathtt{T}_{max}$ given by Theorem 1. The approach described in this section was recently evaluated by Barták and Svancara (2019) for their Picat-based MAPF solver.

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:

- $rt(x, y, t)$ (resp. $lt(x, y, t)$) which specifies that the edge between (x, y) and $(x, y+1)$ 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. The rule for rt is:

$$rt(X, Y, T) \leftarrow exec(A, right, T), at(A, X, Y, T), \quad (13)$$

while the rule for st is:

$$st(X, Y, T) \leftarrow at(A, X, Y, T), exec(A, wait, T).$$

We omit the rules for dt, lt, and ut since they are analogous to (13).

Using these predicates we now express the fact that a single cell cannot be entered at the same time instant by two different agents. This requires six rules each of which corresponds to a pairs 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:

$$\leftarrow lt(X, Y, T), dt(X, Y, T)$$

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 $|\mathcal{A}|$.

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.

$$at(A, X, Y, T) \leftarrow at(A, X', Y', T-1), exec(A, M, T), \\ delta(M, X', Y', X, Y), \\ cost\_to\_go(A, X, Y, C), T + C <= \mathtt{T}, \quad (14)$$

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 (Surynek et al. 2016), but does not require the generation of the MDD, so it is conceptually simpler.

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 (Surynek 2014) (enc=mdd), the SMT compilation-based solver SMT-CBS (Surynek 2019), and the search-based solvers EPEA* (Goldenberg et al. 2014), and ICBS-h (Felner et al. 2018). Our evaluation is focused

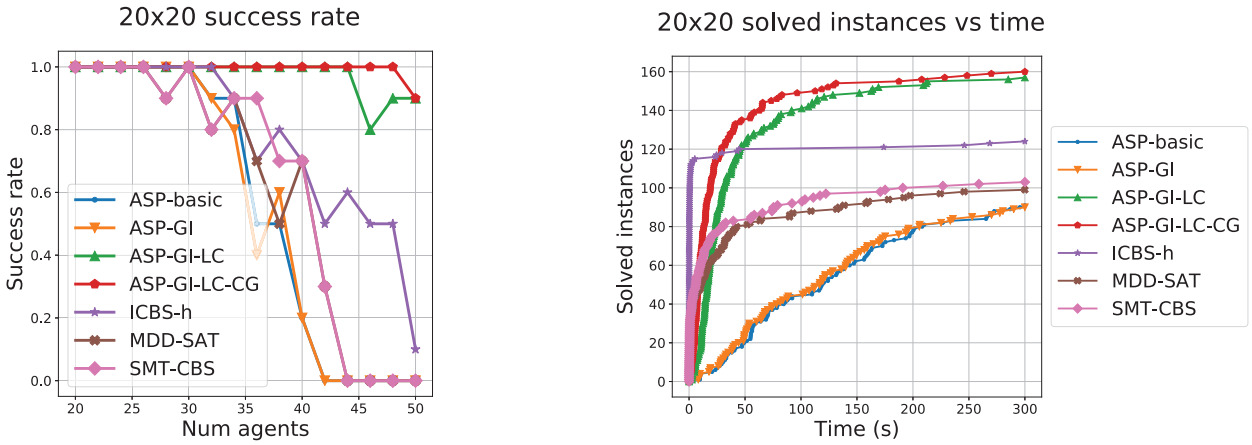

Figure 2: Success rate and number of instances solved versus time on 20×20 grids.

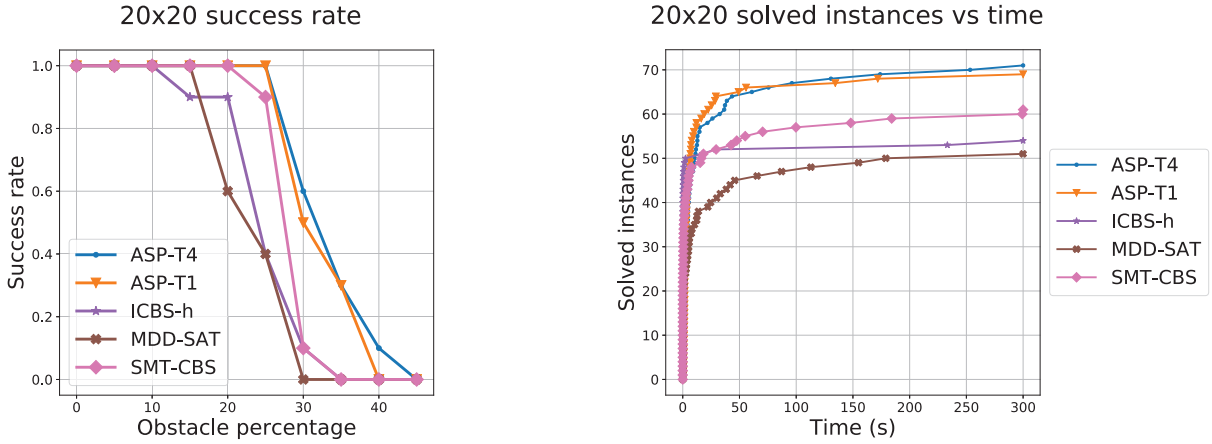

Figure 3: Success rate and number of instances solved versus time on 20×20 grids. ASP-T1 is the best configuration (ASP-GI-LC-CG) ran with 1 thread, and ASP-T4 is ASP-GI-LC-CG ran with 4 threads.

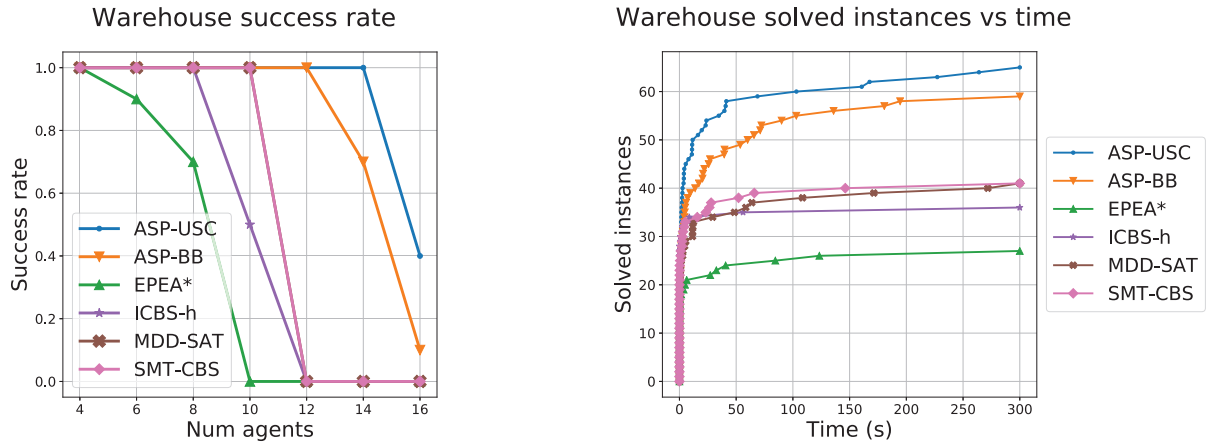

Figure 4: Success rate and number of instances solved versus time on the Warehouse problem.

on relatively small grids since grounding time grows too much for larger grids (e.g., 512×512), making the approach impractical.

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. GI refers to the use of grid-independent penalties. LC refers to the use of 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 (Gebser et al. 2014) 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.

$N \times N$ 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. 160) problems with the number of agents in $\{4, \ldots, 18\}$ (resp. $\{20, 22, \ldots, 50\}$). Success rates, and number of problems solved versus time for the 20×20 are shown in Figure 2. The results for 8×8 are omitted since they look very similar to 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 rule as a way to generate a smaller encoding, as shown in Figure 2 CG greatly improves the solving time. In fact we found that the average solving time is 1.74 smaller using ASP-GI-LC-CG in comparison to ASP-GI-LC. Also we found a decrease of factor 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 $\{0, 5\%, 10\%, \ldots, 45\%\}$. Figure 3 shows the results. Again our ASP formulations outperform other algorithms. In addition, no significant differences are observed between the 1- and 4-thread variants.

Obstacle-Free $N \times N$ Grids

Next, we experimented on $N \times N$ obstacle-free grids with $N \in \{8, 16, 32\}$. Here we wanted to understand how success rate is affected by the number of agents. For each grid we generated 10 random instances for each number of agents in $\{2, 4, \ldots N^2 - 2\}$.

We define the *breaking point* of an algorithm, as the number of agents at which the success rate in the success rate plot drops below 0.5, never to go back up again. The breaking point for each algorithms and grid size is shown at Table 1.

	Breaking-point by grid size		
Algorithm	8×8	16×16	32×32
ASP-basic	26	48	20
ASP-GI-LC-CG	32	70	78
ICBS-h	16	52	108

Table 1: Breaking point on empty $N \times N$ grids

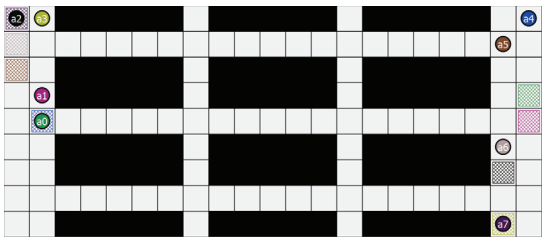

Figure 5: Warehouse problem example.

In the results we observe how critical is grid size to the performance of our algorithm. Because our encoding is so heavily dependent on $|V|$ and `T`, an instance that will seem easier in terms of possible conflicts—such as a (32×32) grid with 70 agents—will be just as hard for our algorithm to solve as a (16×16) grid with roughly the same number of agents. In comparison, ICBS-h does not get affected as much with the increase of the grid size.

Warehouse Experiments

We experimented with a warehouse grid used in the MAPF literature (e.g., Felner et al. 2018), shown in Figure 5. 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 $\{4, \ldots, 16\}$. In this evaluation we only used the encoding with the best results shown on the $N \times N$ grids experiments (ASP-GI-LC-CG). Also, because clingo offers two optimization strategies: Branch-and-bound (BB) (Gebser et al. 2011) based optimization and unsatisfiable-core (USC) based optimization (Andres et al. 2012), 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 4.

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 can cope nicely and more effectively 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 is unsolvable,

USC attempts to find a higher cost solutions incrementally, until one is found. BB, on the other hand, find some 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 4.

Conclusions and Future Work

In this paper we proposed the first compilation of MAPF under the sum-of-costs assumption to ASP. We proposed a number of variants of the approach. In its first, basic form, the encoding is quadratic on the number of agents, just like existing approaches to ASP (e.g. Erdem et al. 2013; Gebser et al. 2014), and like the encoding of MDD-SAT (Surynek et al. 2016), a state of the art SAT-based solver. We also propose an encoding that is linear on 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 on square grids, we observed that the linear encoding substantially outperforms the quadratic encoding. In general our approach outperforms the search-based and SAT-based state-of-the-art solvers we compared with as the number of agents increases on small grids. This suggests that ASP is a competitive approach for solving highly congested MAPF instances.

An important drawback of our approach, which only becomes apparent as larger grids are used, is grounding time. A potential (yet not elegant) line of improvement is to perform grounding manually.

An important observation is that ASP being competitive or superior to SAT for MAPF offers a clearly more elegant alternative than SAT. Indeed, the complete formulation of MAPF has been almost completely included in this paper and does not require clause-generation algorithm like SAT approaches.

References

Andres, B.; Kaufmann, B.; Matheis, O.; and Schaub, T. 2012. Unsatisfiability-based optimization in clasp. In Dovier, A., and Costa, V. S., eds., *Technical Communications of the 28th International Conference on Logic Programming (ICLP)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 211–221. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Barták, R., and Svancara, J. 2019. On SAT-based approaches for multi-agent path finding with the sum-of-costs objective. In Surynek, P., and Yeoh, W., eds., *SoCS*, 10–17. AAAI Press.

Barták, R.; Zhou, N.; Stern, R.; Boyarski, E.; and Surynek, P. 2017. Modeling and solving the multi-agent pathfinding problem in picat. In *ICTAI*, 959–966. Boston, MA, USA: IEEE Computer Society.

Erdem, E.; Kisa, D. G.; Öztok, U.; and Schüller, P. 2013. A general formal framework for pathfinding problems with multiple agents. In *AAAI*. AAAI Press.

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*, 83–87.

Ferraris, P., and Lifschitz, V. 2005. Mathematical foundations of answer set programming. In Artëmov, S. N.; Barringer, H.; d'Avila Garcez, A. S.; Lamb, L. C.; and Woods, J., eds., *We Will Show Them! Essays in Honour of Dov Gabbay, Volume One*, 615–664. College Publications.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2011. Multi-Criteria Optimization in Answer Set Programming. In Gallagher, J. P., and Gelfond, M., eds., *Technical Communications of the 27th International Conference on Logic Programming (ICLP)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 1–10. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + control: Preliminary report. *CoRR* abs/1405.3694.

Gebser, M.; Obermeier, P.; Otto, T.; Schaub, T.; Sabuncu, O.; Nguyen, V.; and Son, T. C. 2018. Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming* 18(3-4):502–519.

Goldenberg, M.; Felner, A.; Stern, R.; Sharon, G.; Sturtevant, N. R.; Holte, R. C.; and Schaeffer, J. 2014. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research* 50:141–187.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In Neumann, B., ed., *ECAI*, 359–363. John Wiley and Sons.

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

Lifschitz, V. 2008. What is answer set programming? In *AAAI*, 1594–1597.

Ma, H., and Koenig, S. 2017. AI buzzwords explained: multi-agent path finding (MAPF). *AI Matters* 3(3):15–19.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Conflict-based search for optimal multi-agent path finding. In *AAAI*.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, 173–178.

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 Surynek, P., and Yeoh, W., eds., *SoCS*, 151–159. AAAI Press.

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

Surynek, P. 2014. Compact representations of cooperative pathfinding as SAT based on matchings in bipartite graphs. In *ICTAI*, 875–882. IEEE Computer Society.

Surynek, P. 2019. Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In Kraus, S., ed., *IJCAI*, 1177–1183. ijcai.org.

Wang, K. C., and Botea, A. 2008. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, 380–387.

Yu, J., and LaValle, S. M. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*.