Journal of Artificial Intelligence Research 75 (2022) 323-350 Submitted 03/2022; published 09/2022

Research Note

Multi-Agent Path Finding: A New Boolean Encoding

Roberto Asín Achá RASIN@UDEC.CL
Department of Computer Science,
Universidad de Concepción, Concepción, Chile

Rodrigo López RILOPEZ3@UC.CL
Department of Management Control and Information Systems,
School of Economics and Business, Universidad de Chile, Santiago, Chile, &
Department of Computer Science,
Pontificia Universidad Católica de Chile, Santiago, Chile

Sebastián Hagedorn SHAGEDORN@UC.CL
Department of Computer Science,
Pontificia Universidad Católica de Chile, Santiago, Chile

Jorge A. Baier JABAIER@ING.PUC.CL
Department of Computer Science,
Pontificia Universidad Católica de Chile, Santiago, Chile, &
Instituto Milenio Fundamentos de los Datos, Santiago, Chile

Abstract

Multi-agent pathfinding (MAPF) is an NP-hard problem. As such, dense maps may be very hard to solve optimally. In such scenarios, compilation-based approaches, via Boolean satisfiability (SAT) and answer set programming (ASP), have been shown to outperform heuristic-search-based approaches, such as conflict-based search (CBS). In this paper, we propose a new Boolean encoding for MAPF, and show how to implement it in ASP and MaxSAT. A feature that distinguishes our encoding from existing ones is that swap and follow conflicts are encoded using binary clauses, which can be exploited by current conflict-driven clause learning (CDCL) solvers. In addition, the number of clauses used to encode swap and follow conflicts do not depend on the number of agents, allowing us to scale better. For MaxSAT, we study different ways in which we may combine the MSU3 and LSU algorithms for maximum performance. In our experimental evaluation, we used square grids, ranging from 20×20 to 50×50 cells, and warehouse maps, with a varying number of agents and obstacles. We compared against representative solvers of the state-of-the-art, including the search-based algorithm CBS, the ASP-based solver ASP-MAPF, and the branch-and-cut-and-price hybrid solver, BCP. We observe that the ASP implementation of our encoding, ASP-MAPF2 outperforms other solvers in most of our experiments. The MaxSAT implementation of our encoding, MtMS shows best performance in relatively small warehouse maps when the number of agents is large, which are the instances with closer resemblance to hard puzzle-like problems.

1. Introduction

Given a graph G and K agents, each of which is associated with a start and a goal vertex of G, multi-agent pathfinding (MAPF) is the problem of finding K non-conflicting paths $\pi_1, \ldots, \pi_K$, such that π_i connects the start and goal vertex associated with agent i. MAPF has many applications. It is key for the implementation of automated warehouses, multi-agent videogames (Wang & Botea, 2008), and has become increasingly relevant in other important applications such as airport ground control (e.g., Li, Gong, Liang, Liu, Tong, Yi, Morris, Pasearanu, & Koenig, 2019b).

Optimal MAPF solving is NP-hard (Surynek, 2010; Yu & LaValle, 2013; Nebel, 2020). Not surprisingly, optimal solvers have scalability issues. Consequently, solving dense MAPF instances, that is, instances with a high proportion of space occupied by either agents or obstacles, can be challenging even when maps are relatively small.

Most MAPF solvers can be classified as either search-based, compilation-based or hybridized. While the former (e.g., Sharon, Stern, Felner, & Sturtevant, 2012; Felner, Li, Boyarski, Ma, Cohen, Kumar, & Koenig, 2018; Li, Felner, Boyarski, Ma, & Koenig, 2019a) can scale to large maps, they do not perform very well in relatively small, dense maps. Compilation-based techniques, instead, translate the MAPF instance to an instance of another problem, for example Boolean satisfiability (SAT) (e.g., Surynek, Felner, Stern, & Boyarski, 2016; Barták, Zhou, Stern, Boyarski, & Surynek, 2017; Barták & Svancara, 2019; Surynek, Stern, Boyarski, & Felner, 2022), answer set programming (ASP) (e.g., Erdem, Kisa, Öztok, & Schüller, 2013; Gebser, Obermeier, Otto, Schaub, Sabuncu, Nguyen, & Son, 2018; Nguyen, Obermeier, Son, Schaub, & Yeoh, 2017; Gómez, Hernández, & Baier, 2021), or Mixed-Integer Programming (MIP) (e.g., Barták et al., 2017). They do perform better than search-based solvers in dense, rather small maps, but do not scale to large maps. Hybridized methods (e.g., Lam, Bodic, Harabor, & Stuckey, 2019) incorporate elements of both.

The efficiency of compilation-based approaches depends on a number of factors, including the base SAT/ASP/MIP solver, but also on the encoding used. Recently, Gómez et al. (2021) showed that focusing on generating a smaller encoding, linear on the number of agents rather than quadratic, yielded significant benefits in practice.

In this paper, we propose a new Boolean encoding for sum-of-costs MAPF and show how it can be implemented in both MaxSAT and ASP. Our encoding shares common aspects with the encoding introduced by Gómez et al. (2021) for their ASP-MAPF solver. However, ours encodes decisions in a different way, allowing to represent swap and follow conflicts using binary clauses involving two decision variables. Specifically, our encoding improves upon the number and type of previous encodings in that we use binary or at-most-one (AMO) cardinality constraints when previous encodings did not use binary constraints. A summary of the relations is presented in Table 1.

Furthermore, this paper presents the first study and evaluation of MAPF encoding using MaxSAT technology. For this reason, even though the encoding can be represented in ASP, we study in detail how MaxSAT technology can be used for obtaining good results. Thus, we investigate the effectiveness of using different optimization algorithms (e.g., MSU3/LSU), and a number of encodings for cardinality constraints, and report the configurations that work best in practice. In addition, we investigate the benefits of including redundant clauses, which allow the MaxSAT solver to reduce the search space, resulting in better scaling.

In our experimental evaluation, we compare our MaxSAT and our ASP MAPF solvers with ASP-MAPF, MDD-SAT (Surynek et al., 2016), a SAT-based MAPF solver, and CBS-H2 (Li et al., 2019a), a state-of-the-art search-based solver. We use a number of grid and warehouse benchmarks. We observe that our ASP solver scales better when the number of

	MDD-SAT (Surynek et al., 2016)	**ASP-MAPF (Gómez et al., 2020)**	**Our encoding (this paper)**
vertex conflicts	$O(K^2 \cdot \lvert V\rvert \cdot T)$ binary	$O(K \cdot \lvert V\rvert \cdot T)$ non-binary	$O(K \cdot \lvert V\rvert \cdot T)$ AMO
follow conflicts	$O(K^2 \cdot \lvert E\rvert \cdot T)$ binary	N/A	$O(K \cdot \lvert E\rvert \cdot T)$ binary
swap conflicts	$O(K^2 \cdot \lvert E\rvert \cdot T)$ binary	$O(K \cdot \lvert E\rvert \cdot T)$ non-binary	$O(K \cdot \lvert E\rvert \cdot T)$ binary

Table 1: Number of and type of constraints needed to encode vertex, follow, and swap conflicts in MDD-SAT, ASP-MAPF, and our encoding.

agents is increased on grids with few obstacles, while our MaxSAT solver performs better in scenarios with more obstacles and fewer agents. Our results support the fact that our encoding is superior to previous ones, independently of the technology used.

This paper is an extension of a previous conference paper (Asín Achá, López, Hagedorn, & Baier, 2021). In addition to the material presented in the conference paper, this paper incorporates the following.

- A complete description of the ASP version of our encoding; previously only a few rules had been described.
- We extend the experimental evaluation significantly, by including larger maps and by analyzing the performance of the solvers used by the density of the maps.
- We include a complete proof that speaks to the correctness of the MaxSAT encoding we use.

The rest of the paper is organized as follows. First we present the background on MAPF, MaxSAT and ASP, and review relevant aspects of most relevant existing compilations of MAPF to SAT and ASP. We continue presenting the main variables of our encoding in both MaxSAT and ASP. Then we present our empirical evaluation and present a summary and conclusions.

2. Background

We start off with background on MAPF, following Stern, Sturtevant, Felner, Koenig, Ma, Walker, Li, Atzmon, Cohen, Kumar, Barták, and Boyarski (2019) rather closely. Then we present background on MaxSAT and MAPF compilations relevant for our work.

2.1 Multi-Agent Pathfinding

A MAPF instance is defined by a tuple $(G, K, \mathit{start}, \mathit{goal})$, where $G = (V, E)$ is a directed graph, K is the number of agents, and functions $\mathit{start} : \{1, \ldots, K\} \to V$ and $\mathit{goal} : \{1, \ldots, K\} \to V$ specify the start and goal vertex for each of the agents. In addition, E contains the edge (u, u), for every $u \in V$. This constraint is important, since it allows agents to 'wait' at their current location.

A *path over graph G* is a sequence of edges $\pi = (u_0, u_1)(u_1, u_2) \ldots (u_{n-1}, u_n)$, for every $i \in \{0, \ldots, n-1\}$. Given a path $\pi = (u_0, u_1) \ldots (u_{n-1}, u_n)$, we denote the vertex reached by traversing the first t edges of π by $\pi[t]$; formally $\pi[t] = u_t$, where $0 \leq t \leq |\pi|$. If $\pi[t] = v$ we say that the path *visits vertex v at time step t*. In this paper, like in most of the literature on MAPF (Stern et al., 2019), we consider that agents stay at their target after reaching it. Thus, for any $t > |\pi|$, we define $\pi[t] = \pi[|\pi|]$. A path π *connects* u *with* v iff $\pi[0] = u$ and $\pi[|\pi|] = v$. Given two nodes $u, v \in V$ we denote by $d(u, v)$ the length of the shortest path connecting u and v.

A *solution* to a MAPF instance $(G, K, \mathit{start}, \mathit{goal})$ is a tuple of K paths over G, $\Pi = (\pi_1, \pi_2, \ldots, \pi_K)$, where:

1. π_k connects $\mathit{start}(k)$ with $\mathit{goal}(k)$, for every $k \in \{1, \ldots, K\}$.
2. The last element of π_k is not of the form (u, u), for every $k \in \{1, \ldots, K\}$.
3. Π is *conflict-free*.

Intuitively, Π has a conflict if the paths of two or more agents interfere with each other. A relevant observation is that condition 2 is necessary for the definition of the cost of a solution, which we introduce below. A non-solution that satisfies conditions 1 and 3 but not 2 can always be transformed into a solution by removing the largest suffix of wait actions from every path.

Different types of conflicts have been considered in the MAPF literature. Those relevant to this paper follow.

- **Vertex Conflicts**. Π has a *vertex conflict* if two paths π_i, π_j in Π, where $i \neq j$, are such that $\pi_i[t] = \pi_j[t]$, for some $t \geq 0$. Intuitively, a vertex conflict occurs when two agents visit the same vertex at the same time.
- **Swap Conflicts**. Π has an *swap conflict* if two paths π_i, π_j in Π, where $i \neq j$, are such that $(\pi_i[t], \pi_i[t+1]) = (\pi_j[t+1], \pi_j[t])$, for some $t \geq 0$. Intuitively, a vertex conflict occurs when two agents traverse the connection between two nodes in different directions at the same time.
- **Follow Conflicts**. Π has a *follow conflict* if two paths π_i, π_j in Π, where $i \neq j$, are such that $\pi_i[t] = \pi_j[t+1]$, for some $t \geq 0$. Intuitively, a follow conflict occurs when at time $t+1$ an agent occupies the position another agent had at time step t. When a solution is free of follow conflicts, it is also free of swap conflicts.

Most literature on MAPF (e.g., Sharon et al., 2012) has focused on solvers for solutions free of vertex and swap conflicts. The most relevant compilation-based solvers that we consider in this paper (Surynek et al., 2016; Surynek, 2019b), compute solutions free of vertex, swap, and follow conflicts.

The *sum-of-costs* (SOC) of a solution $\Pi = (\pi_1, \pi_2, \ldots, \pi_K)$ is $c(\Pi) = \sum_{i=1}^{K} |\pi_i|$, whereas its *makespan* is $\mathit{makespan}(\Pi) = \max_{i \in \{1,\ldots,K\}} |\pi_i|$. A solution Π to an instance of MAPF is SOC-optimal iff the SOC of every other solution is equal to or greater than $c(\Pi)$. A solution Π to an instance of MAPF is makespan-optimal iff the makespan of every other solution is equal to or greater than $\mathit{makespan}(\Pi)$.

2.2 Answer-Set Programming

ASP (Lifschitz, 2008) is a logic-based framework for solving optimization problems. In this section we cover the aspects of ASP that are most relevant to our paper. A *basic program* over a set of atoms A is a set of rules of the form:

$$Head \leftarrow Body, \tag{1}$$

where *Head* and *Body* are sets of atoms. To simplify notation rule $\{p\} \leftarrow \{q_0, q_2, \ldots, q_n\}$ is written as $p \leftarrow q_0, q_2, \ldots, q_n$.

In our encoding we only consider the case in which $|Head| \leq 1$. If $\mathcal{P}$ is a basic program then its model is a subset-minimal set of atoms M such that for each $Head \leftarrow Body \in \Pi$ such that $Body \subseteq M$, then $Head \cap M \neq \emptyset$. From the definition it follows that if a program contains a rule with an empty body such as:

$$p \leftarrow,$$

then every model of the program must contain p. Moreover, if a program contains a rule of the form $q \leftarrow q_1, \ldots, q_n$, then if $q_1 \ldots, q_n$ are in the model M it follows that $q \in M$. Finally, a rule of an empty head such as:

$$\leftarrow p_1, p_2, \ldots, p_n,$$

is in Π then no model of Π can simultaneously contain the atoms $p_1, \ldots, p_n$.

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

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

Intuitively, rule (2) should be interpreted as 'p appears in the model if $q_1, \ldots q_n$ appear too but none of $r_1, \ldots, r_k$ do'. 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 (Ferraris & Lifschitz, 2005).

Another syntactic element relevant to our work is:

$$|\{p_1, p_2, \ldots, p_n\}| = k$$

Intuitively, this says that 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, since if $n > k$ the solver may consider all subsets of k elements to appear in the model. For example, the program containing just $|\{p, q, r\}| = 1$ has three models: $\{p\}$, $\{q\}$, and $\{r\}$.

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

$$p(X) \leftarrow q(X), \tag{3}$$

where uppercase letters represent *variables*. Intuitively a variable occurring in a program $\mathcal{P}$ can take any value among the set of *terms* of $\mathcal{P}$. As such, rule (3) represents that when c is a term and $q(c)$ appears in a model M, then $p(c)$ should also. 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, removing all variables from the program. Since there are plenty of optimizations that solvers employ during grounding, it is not simple to describe it here, and therefore we will only describe it intuitively. For example, the grounded version of program:

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

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 $p(X) \leftarrow q(X)$.

2.3 Maximum Boolean Satisfiability

Given a set X of *variables*, ℓ is a *literal* over X iff $\ell = x$ or $\ell = \neg x$, for some $x \in X$. A *clause* over X is a set of literals over X. Given a literal ℓ over X, its *complement* $\overline{\ell}$ is defined as x if $\ell = \neg x$, and as $\neg x$ if $\ell = x$, for some variable x. A *boolean assignment* for X is a function $\sigma : X \to \{\mathit{true}, \mathit{false}\}$. An assignment *satisfies* Y iff $\sigma \models Y$, where the binary relation $\models$ is such that $\sigma \models x$ iff $\sigma(x) = \mathit{true}$, and $\sigma \models \neg x$ iff $\sigma(x) \not\models x$, for every $x \in X$. If C is a clause $\sigma \models C$ iff $\sigma \models \ell$, for some $\ell \in C$. If S is a set of clauses $\sigma \models S$ iff $\sigma \models C$, for every $C \in S$.

Given a set of clauses S over variables X, the *boolean satisfiability* problem (SAT) consists of computing an assignment σ over X such that $\sigma \models S$. Given a pair of two sets of clauses (H, S), where H corresponds to a set of *hard* clauses, and S is a set of *soft* clauses, the *(partial) maximum boolean satisfiability problem* (MaxSAT), consists of finding an assignment σ that satisfies H and maximizes the number of clauses of S which it satisfies.

Given a set of literals $L = \{\ell_1, \ldots, \ell_n\}$, $\bigwedge L$ stands for $\ell_1 \wedge \ldots \wedge \ell_n$, $\bigvee L$ stands for $\ell_1 \vee \ldots \vee \ell_n$, and $\overline{C}$ stands for $\{\overline{\ell_1}, \ldots, \overline{\ell_n}\}$. A clause C logically corresponds to the disjunction $\bigvee C$, which in turn is logically equivalent to the implication $\bigwedge \overline{B} \to \bigvee T$, where B and T are disjoint and such that $C = B \cup T$.

For our experimental evaluation, it is relevant to review the state-of-the-art approaches to MaxSAT. The following are approaches that have been used by solvers participating in recent MaxSAT competitions (e.g., Bacchus, Berg, Järvisalo, & Martins, 2020).

SAT-based These algorithms iteratively call an underlying SAT solver where each iteration imposes a stronger bound for the number of unsatisfied clauses in S. Between consecutive calls, the bound is incremented by 1. The the first call that returns UNSAT yields a MaxSAT assignment. The most used SAT-based algorithm is known as LSU (Biere, Heule, & van Maaren, 2009). Although not usually competitive in the MaxSAT competitions, a strength of this approach is its anytime property which allows it to output a sequence of assignments of increasing quality.

UNSAT-based Also an iterative approach. In the first iteration all clauses are treated as hard. In subsequent iterations, depending on the reasoning of the algorithm, some or all original soft clauses are relaxed by adding cardinality constraints imposing that at most a certain number of soft clauses may be unsatisfied. Bounds imposed on the relaxation are sound, which guarantees that the first SAT call returns an optimal assignment. MSU3 (Morgado, Liffiton, & Marques-Silva, 2012), RC2 (Ignatiev, Morgado, & Marques-Silva, 2019) and EvalMaxSAT (Avellaneda, 2020) are examples of UNSAT-based solvers.

SAT-UNSAT-based Algorithms belonging to this category mix the two strategies from above. Pacose (Paxian, Reimer, & Becker, 2019), QMaxSAT (Koshimura, Zhang, Fujita, & Hasegawa, 2012), UWrMaxSAT (Piotrów, 2019), are examples of solvers based on this paradigm.

Hitting-set-based The solvers belonging to this category separate the problem in two: core extraction, performed by a SAT solver and optimisation, performed by an Integer Linear Programming (ILP) solver. By doing so, solvers of this type avoid increasing the complexity of the SAT problem, but instead they use the ILP optimizer for doing the numerical reasoning. The main representative of this paradigm is MaxHS (Berg, Bacchus, & Poole, 2020).

2.4 Existing Compilation-Based Approaches

We briefly review relevant aspects of the two most related compilation-based approaches: MDD-SAT (Surynek et al., 2016, 2022) and ASP-MAPF (Gómez et al., 2021). Both approaches encode the problem using propositional variables relative to a certain time instant.

2.4.1 MDD-SAT

MDD-SAT is a SAT-based solver which encodes MAPF solutions forbidding follow conflicts. Below we assume the set of vertices is of the form $\{1, \ldots, m\}$. The main variables used by the encoding are:

- $\mathcal{X}_j^t(a_i)$: expresses that agent i is at vertex j at time step t.
- $\mathcal{E}_{j,k}^t(a_i)$: expresses that edge (j, k) is traversed by agent i at time step t.

MDD-SAT encodes the absence of follow conflicts simultaneously, with the constraint:[1]

$$\mathcal{E}_{j,k} \rightarrow \bigwedge_{a_\ell \in A, a_\ell \neq a_i, u_k^{t+1} \in V_\ell} \neg \mathcal{X}_k^{t+1}(a_\ell), \tag{5}$$

which expands to a number of binary clauses that is quadratic on the number of agents, and linear on the number of edges of the graph; that is $O(K^2 \cdot |V| \cdot T)$, where T is the makespan of the encoding. Since the absence of follow conflicts also implies the absence of swap conflicts, no additional constraints is needed for swap conflicts.

Absence of vertex conflicts is encoded via the following constraint:

$$\bigwedge_{u_j^t \in V_i \cap V_l} \neg \mathcal{X}_j^t(a_i) \vee \neg \mathcal{X}_j^t(a_l), \tag{6}$$

which expands into $O(K^2 \cdot |V| \cdot T)$ binary clauses.

1. To simplify our presentation, we omit a formal definition of V_i here; we note however that its size is $O(|V|)$.

2.4.2 ASP-MAPF

The main variables used by ASP-MAPF are as follows.

- $at(r, x, y, t)$: expresses that agent r is at cell (x, y) at time step t.
- $exec(r, a, t)$: expresses that agent r performs action a at time t, where a is an action in $\{up, down, left, right, wait\}$.

In the encoding of ASP-MAPF it is assumed the that graph is a grid, which is consistent with the fact that MAPF's benchmarks are gridworlds.

To define swap, follow, and vertex conflicts, ASP-MAPF uses five predicates: rt, lt, ut, dt (encoding, respectively, right-, left-, upwards-, downwards-traversals) and st (stay) predicate. More specifically,

- $rt(x, y, t)$: expresses that the undirected edge connecting (x, y) and $(x + 1, y)$ was traversed at time t from left to right. Variable $lt(x, y, t)$ is used to express that the same undirected edge is traversed from right to left.
- $ut(x, y, t)$: expresses that the edge connecting (x, y) and $(x, y + 1)$ was traversed upwards at time t. Variable $dt(x, y, t)$ expresses the very same edge was traversed downwards.
- $st(x, y, t)$: expresses that an agent at (x, y) at time t executed a $wait$.

These five predicates are defined in terms of at and $exec$, using non-binary rules. For example, the definition for rt is as follows.

$$rt(X, Y, T) \leftarrow exec(R, right, T), at(R, X, Y, T). \tag{7}$$

Swap conflicts are encoded using the following binary constraints:

$$\leftarrow lt(X, Y, T), rt(X - 1, Y, T). \tag{8}$$

$$\leftarrow ut(X, Y - 1, T), dt(X, Y, T). \tag{9}$$

Grounding generates $O(K \cdot |V| \cdot T)$ non-binary rules from (7), and $O(|V| \cdot T)$. In total, swap conflicts are encoded with $O(K \cdot |V| \cdot T)$ rules, which include non-binary rules.

Follow and vertex conflicts are also encoded using the 5 traversal predicates, and require a total of $O(K \cdot |V| \cdot T)$ non-binary clauses. The full encoding of ASP-MAPF is shown in Figure 1.

3. Our Encoding

In this section we present our new encoding for MAPF, and how it can be implemented in MaxSAT and ASP. We follow the same principle of previous compilation-based approaches to MAPF, encoding the position of the agents using propositional variables, for different time steps in $\{0, \ldots, T\}$. As previous approaches (e.g., Surynek et al., 2016; Gómez et al., 2021), we optimize the number of variables by, in practice, removing those variables that would encode an agent a being at a vertex v that is not reachable by a given the constraints

```
% robot(r):             express that r is an agent in R,
% rangeX(x):            expresses that x is within the legal range for the X axis
% rangeY(y):            expresses that y is within the legal range for the Y axis
% time(t):              t is a time instant,
% action(a):            a is an action,
% obstacle(x,y):        specifies that cell (x,y) is an obstacle,
% goal(r,x,y):          specifies that the goal cell for agent r is (x,y).
% cost_to_go(r,x,y,c): specifies that the shortest path between (x,y) and r’s goal is c
% delta(a,x,y,x’,y’):  specifies that the cell (x’,y’) is reached when performing action a at cell (x,y).
% at_goal(r,t):         specifies that agent r is at the goal at time t.
% penalty(r,t,1):       specifies the slack between the makespan T and the time instant
                        at which an agent has stopped at the goal.

% dynamics of the at predicate
at(R,X,Y,T) :- exec(R,A,T-1), at(R,Xp,Yp,T-1), delta(A,Xp,Yp,X,Y), cost_to_go(R,X,Y,C), T + C <= bound.

% the 5 possible actions
action(up; down; left; right; wait).

% a specification of the moves
free(X,Y) :- rangeX(X), rangeY(Y), not obstacle(X,Y).
delta(right,X,Y,X+1,Y) :- rangeX(X), rangeX(X+1), rangeY(Y), free(X,Y).
delta(left,X,Y,X-1,Y)  :- rangeX(X), rangeX(X-1), rangeY(Y), free(X,Y).
delta(up,X,Y,X,Y+1)    :- rangeX(X), rangeY(Y),   rangeY(Y+1), free(X,Y).
delta(down,X,Y,X,Y-1)  :- rangeX(X), rangeY(Y),   rangeY(Y-1), free(X,Y).
delta(wait,X,Y,X,Y)    :- rangeX(X), rangeY(Y),   free(X,Y).

% definition of the 5 traversal predicates
rt(X,Y,T)   :- exec(R,right,T), at(R,X,Y,T).
lt(X-1,Y,T) :- exec(R,left,T),  at(R,X,Y,T).
ut(X,Y,T)   :- exec(R,up,T),    at(R,X,Y,T).
dt(X,Y-1,T) :- exec(R,down,T),  at(R,X,Y,T).
st(X,Y,T)   :- exec(R,wait,T),  at(R,X,Y,T).

% A single action per agent can be performed at each time instant.
1 {exec(R,A,T-1) : action(A)} 1 :- robot(R), time(T).

:- at(R,X,Y,T),obstacle(X,Y).

% Vertex conflicts, part 1; agent stays at (x,y), while another enters
:- st(X,Y,T), lt(X,Y,T).
:- st(X,Y,T), rt(X-1,Y,T).
:- st(X,Y,T), ut(X,Y-1,T).
:- st(X,Y,T), dt(X,Y,T).

% Vertex conflicts, part 2: two agents enter the same cell at the same time
:- rt(X,Y,T), lt(X,Y,T).
:- ut(X,Y,T), dt(X,Y,T).
:- lt(X,Y,T), dt(X,Y,T).
:- lt(X,Y,T), ut(X,Y-1,T).
:- rt(X-1,Y,T), ut(X,Y-1,T).
:- rt(X-1,Y,T), dt(X,Y,T).

% Swap conflicts
:- lt(X,Y,T), rt(X-1,Y,T).
:- ut(X,Y-1,T), dt(X,Y,T).

% Goal achievement
at_goal(R,T) :- at(R,X,Y,T), goal(R,X,Y).
:- robot(R), not at_goal(R,bound).
at_goal_back(R,bound) :- robot(R).
at_goal_back(R,T-1) :- at_goal_back(R,T),exec(R,wait,T-1).

% Cost of the optimal path.
penalty(R,T,1) :- time(T), dijkstra(R,Tp), T>Tp, at_goal_back(R,T-1).

Optimization statement.
#maximize {P,R,T : penalty(R,T,P)}.
```

Figure 1: ASP-MAPF’s encoding (Gómez et al., 2021).

imposed by the bound T. To that end, we define $\mathit{Feasible}(a,t)$ as the set of vertices that can be *feasibly reached* by a. An agent can feasibly reach a vertex v at time t if it can reach v in t steps or less and the goal is reachable from v by time step T. Formally,

$$\mathit{Feasible}(a,t) = \{u \in V : d(\mathit{start}(a), u) \leq t, d(u, \mathit{goal}(a)) \leq T - t\}.$$

To compute set $\mathit{Feasible}(a,t)$ efficiently, before encoding our problem we run Dijkstra's algorithm to compute the d function, for each start and goal state as a source.

Below we use $\mathcal{A}$ to abbreviate $\{1,\ldots,K\}$. The variables we use in our encoding are as follows.

3.1 Variables

The following variables are used in both of our encodings.

1. $\mathit{at}_{a,u,t}$: agent a is at vertex u at time step t, where $a \in \mathcal{A}$, $u \in \mathit{Feasible}(a,t)$, and $t \in \{0,\ldots,T\}$.

2. $\mathit{shift}_{u,v,t}$: vertex u *shifts towards* v at time step t, where $(u,v) \in E$, and $t \in \{0,\ldots,T-1\}$. Specifically, this means that if an agent is at u, then it should move towards v. Vertices shift regardless of whether or not there is an agent at them.

3. $\mathit{finalState}_{a,t}$: agent a is at its goal vertex, $\mathit{goal}(a)$, at time t, and will not move in the future of t to another vertex. This variable is defined for every $a \in \mathcal{A}$, and every $t \in \{0,\ldots,T\}$. These variables allow us to define our optimization function. They are inspired by ASP-MAPF.

It is important to note here the use of $\mathit{shift}_{u,v,t}$, which is a variable that does not refer to the agent. This is the main difference of our encoding with the encodings of MDD-SAT and of ASP-MAPF; and, as we see later, this allows us to write binary constraints for certain conflicts. Indeed, both of these approaches use variables that involve the agents for action decisions. As mentioned above, MDD-SAT uses a variable $\mathcal{E}^t_{j,k}(a_i)$, to express that agent a_i moves from vertex j to vertex k at time t, while ASP-MAPF uses a variable $\mathit{exec}(A,M,T)$ to express that agent A performs move M at time T. As we see later in our experimental section, this rather simple change yields important speedups experimentally.

3.2 MaxSAT Encoding

Now we present a MaxSAT encoding which uses the variables described above.

A constraint on *shift* variables We start off by expressing that every vertex u shifts to exactly one of its neighbors at every time interval, by including the cardinality constraint:

$$\sum_{v:(u,v)\in E} \mathit{shift}_{u,v,t} = 1, \tag{C1}$$

for every $u \in V$, and every $t \in \{0,\ldots,T-1\}$.

Encoding the Agents' locations Now we define the relationship between the *at* variables and the *shift* variables. If one understands the *at* variable actually as a *fluent*, as one may do when understanding our model as a theory of action (e.g., Reiter, 2001), we would generate a so-called *successor state axiom*, which, in the context of (C1) is as follows:

$$at_{a,v,t+1} \leftrightarrow \bigvee_{u:(u,v)\in E} at_{a,u,t} \wedge \mathit{shift}_{u,v,t}, \tag{SSA}$$

for every $v \in V$. It is easy to see that axiom (SSA) expresses that agent a is at vertex v at $t+1$ if and only if it was at some vertex u at time t and vertex u shifted towards v at time t.

Unfortunately, converting (SSA) to clauses using standard logical manipulation results in many clauses; indeed, it results in $\Theta(K \cdot |V| \cdot 2^n)$ clauses, where n is the average number of successors for a node in the graph. Most of such clauses, however, are redundant with each other. Moreover (SSA) does not take into account that $at_{a,u,t}$ is only defined when $u \in \mathit{Feasible}(a,t)$. Instead of (SSA), we use the clauses corresponding to the following four families of formulas.

$$at_{a,u,t} \wedge \mathit{shift}_{u,v,t} \rightarrow at_{a,v,t+1}, \tag{H1}$$

$$at_{a,u,t} \wedge at_{a,v,t+1} \rightarrow \mathit{shift}_{u,v,t}, \tag{H2}$$

for every $a \in \mathcal{A}$, every $u \in \mathit{Feasible}(a,t)$ every v such that $(u,v) \in E$ which is such that $v \in \mathit{Feasible}(a,t+1)$, and every $t \in \{0,\ldots,T-1\}$. In addition,

$$at_{a,v,t+1} \rightarrow \bigvee_{u:(u,v)\in E, u\in \mathit{Feasible}(a,t)} at_{a,u,t}, \tag{H3}$$

for every $a \in \mathcal{A}$ and every $v \in \mathit{Feasible}(a,t+1)$, and every $t \in \{0,\ldots,T\}$, and:

$$at_{a,u,t} \rightarrow \overline{\mathit{shift}_{u,v,t}}, \tag{H4}$$

for every $u \in \mathit{Feasible}(a,t)$, every $t \in \{0,\ldots,T\}$, and every v such that $(u,v) \in E$ and such that $v \in V \setminus \mathit{Feasible}(a,t+1)$.

Formula (H1) can be regarded as a *positive effect axiom* (Reiter, 1991, 2001), which establishes that when agent a is at vertex u and a shift from u to v occurs at time t, then a is at v at time $t+1$. Formula (H2) could be viewed as an explanation axiom: if an agent has moved from u to v between time t and time $t+1$, then such a move is explained by a shift from u to v at time t. Formula (H3) establishes that if an agent is at a certain vertex u at time $t+1$, then at the previous time instant, it must have been at one of the predecessors of u. Finally, formula (H4) could be viewed as a special case of (H1) had (H1) been defined for any $(u,v) \in E$. In such a case, we would have needed to account for the fact that $at_{a,v,t+1}$ is not defined—and thus should be considered equivalent to *false*—when $v \notin \mathit{Feasible}(a,t+1)$. In words, (H4) says it is not possible to shift an agent to an unfeasible vertex.

The following result provides a formal account of the correctness of our approach to encoding the agents' locations, by identifying the relationship between formulas (H1)–(H4) and the successor state axiom of (SSA).

Proposition 1. *The conjunction of:*

F1. any formula logically equivalent to (C1), *F2. a formula specifying that every agent is in exactly one location at $t = 0$, and*

F3. clauses of the form $\overline{at_{a,u,t}}$, for every $a \in \mathcal{A}$, $u \in V \setminus Feasible(a,t)$, and $t \in \{0,\ldots,T\}$,

implies that (SSA) *is equivalent to the conjunction of* (H1)–(H4).

Proof. As a first step of our proof, we will prove a lemma (henceforth referred to as LEM) that establishes that F1, F2 and (SSA) imply that every agent is at a single location, for every $t \in \{0,...,T\}$. The proof is by induction on t. For the base case, the result comes directly from assumption F2 in a time $t = 0$. Now if we assume that the lemma holds for some $t \in \{1,...,T\}$, we need to prove that it holds for $t + 1$. Notice that (C1) establishes that every vertex u shifts to exactly one of its neighbors at every time interval. Also, from the assumption for a time t, every agent must be in a vertex at a time t. Then, by using the $\Leftarrow$ of (SSA) this proves that LEM holds for every $t \in \{1,...,T\}$.

Now we start the proof for Proposition 1. We have two cases:

1. Case 1: (SSA) implies (H1)–(H4).

   Obtaining (H1) and (H3) is straightforward by writing (SSA) as a double implication. can use the double implication law on (SSA) to get (H1) and (H3). (H1) is obtained by separating the following formula into multiple implications using the fact that $(p \vee q) \to r$ is equivalent to $(p \to r) \wedge (q \to r)$. To obtain (H3), we write the $\to$ part of (SSA) as a disjunction:

$$\neg at_{a,v,t+1} \vee \bigvee_{u:(u,v)\in E} at_{a,u,t} \wedge shift_{u,v,t}, \tag{10}$$

   which is equivalent to (H3).

   We now transform (10) into conjunctive normal form. This generates a number of clauses, including this one:

$$\neg at_{a,v,t+1} \vee \bigvee_{u':(u',v)\in E, u\neq u'} at_{a,u,t} \vee shift_{u,v,t}, \tag{11}$$

   Now, to obtain (H2), we use the fact that LEM allows us to simplify formulas of the form $\bigvee_{j:(j,v)\in E; j\neq u} at_{a,j,t}$ into $\neg at_{a,u,t}$.

   We now use LEM, which implies that the clause $\neg at_{a,u',t} \vee \neg at_{a,u,t}$ holds for every pair $u, u' \in V$. In particular, this implies that (11) entails:

$$\neg at_{a,v,t+1} \vee shift_{u,v,t} \vee \neg at_{a,u,t},$$

   which is equivalent to (H2).

   By conjoining F3 with (H2) we conclude that for the clauses that a vertex v is not Feasible by an agent a in a vertex u, we obtain (H4).

2. Case 2: (H1)–(H4) imply (SSA).

   We observe that the right-to-left implication of (SSA) comes directly from (H1). For the left-to-right implication of (SSA) we use resolution between clauses in (H2), (H4) and (H3) by using syntactic manipulation.

□

Encoding *finalState* variables We continue defining the dynamics of the *finalState* variables. For every $a \in \mathcal{A}$ and every $t \in \{d(start(a), goal(a)), \ldots, T-1\}$ we define:

$$\mathit{finalState}_{a,T}, \tag{H5}$$

$$at_{a,goal(a),t} \wedge \mathit{finalState}_{a,t+1} \leftrightarrow \mathit{finalState}_{a,t}. \tag{H6}$$

The first formula, (H5), expresses that, at time T, agent a is at its final state. Indeed, as required below by (H8), each agent should reach its goal at time T. Formula (H6) expresses that in t, a is at its final state if and only if it was already at its final state at $t+1$ and a is still at its goal location at time instant t.

Initial and Goal Conditions To define the start location of the agents, we add, for every $a \in \mathcal{A}$:

$$at_{a,start(a),0}. \tag{H7}$$

Note that it is not necessary to explicitly specify that the agent is *not* at other locations different from $start(a)$ at $t = 0$, since $start(a)$ is the only location that is in $\mathit{Feasible}(a, 0)$.

For the goal condition, we add, for every $a \in \mathcal{A}$:

$$at_{a,goal(a),T}. \tag{H8}$$

Vertex Conflicts To make solutions comply with vertex conflicts, we prohibit two or more agents occupying the same vertex at a particular time, using the following AMO cardinality constraint:

$$\sum_{a:u \in Feasible(a,t)} at_{a,u,t} \leq 1, \tag{C2}$$

for every $t \in \{0, \ldots, T\}$.

Swap Conflicts To enforce solutions respect swap conflicts, we forbid one edge being used in two different directions at the same time step, using:

$$\mathit{shift}_{u,v,t} \rightarrow \overline{\mathit{shift}_{v,u,t}}, \tag{H9}$$

for every $(u, v) \in E$, and every $t \in \{0, \ldots, T-1\}$.

Follow Conflicts To get solutions to respect follow conflicts, if there is a shift towards a specific vertex v, we force that such a vertex to perform a shift towards itself, using the following formula:

$$\mathit{shift}_{u,v,t} \rightarrow \mathit{shift}_{v,v,t}. \tag{H10}$$

Cost Minimization To minimize the sum-of-costs, like Gómez et al. (2021), we maximize the number of time intervals for which each agent has been at its goal state. To that end, we add the following *soft* clauses:

$$\mathit{finalState}(a, t), \tag{S1}$$

for each $a \in \mathcal{A}$, and each $t \in \{d(start(a), goal(a)), \dots, T\}$. We associate the same violation cost, equal to 1, with each clause.

Additional Constraints Up to this point our encoding does not include redundant clauses. However, as we see later in our experimental section, the MaxSAT solver benefits from the inclusion of redundant clauses. This is because these may trigger propagations that are helpful for the solver. Below, we consider two redundant constraints, the first of which is given by the following formula:

$$at_{a,u,t} \rightarrow \bigvee_{v:(u,v)\in E, v\in Feasible(a,t)} at_{a,v,t+1}, \tag{H11}$$

for every $a \in \mathcal{A}$, every $u \in Feasible(a,t)$ and every $t \in \{0, \dots, T-1\}$, establishes that if an agent is at certain vertex u at time t, then at time $t+1$ it will be at a neighbor v of u. The second constraint establishes that each agent is at some vertex of the graph at any given time. For every $a \in \mathcal{A}$ and $t \in \{0, \dots, T\}$:

$$\sum_{u\in Feasible(a,t)} at_{a,u,t} = 1 \tag{C3}$$

The following results show in what sense these constraints are redundant with the rest of the encoding.

Proposition 2. *Formula* (H11) *is implied by the conjunction of* (H1) *and any formula logically equivalent to* (C1)*.*

Proof. (C1) implies $\bigvee_{v:(u,v)\in E} shift_{u,v,t}$, for every $u \in V$, which together with (H1) implies (H11). □

Proposition 3. *The conjunction of* (H1)–(H4)*, and* (H7) *and any formula logically equivalent to* (C1) *entails any formula logically equivalent to* (C3)*.*

Proof. By induction on the total number of time steps of the encoding. In the base case ($t = 0$) (C3) holds by definition. For the induction, if the property holds at t, using (SSA) and (C1) we prove that, in $t+1$, (C3) holds. □

Another observation that is important to make at this point is that the inclusion of (C3) in the encoding allows us to remove (H3), resulting in a more *compact* encoding. Indeed, this is because the number of clauses produced by (H3) is $O(K \cdot |E| \cdot T)$, whereas the number of clauses produced by (C3) is $O(S \cdot |V| \cdot T)$, where S is a function that depends on the way we encode cardinality constraints. Thus, substituting H3 may actually generate a more compact encoding. Such a substitution is justified by the following result.

Proposition 4. *The conjunction of* (H1)*,* (H2)*,* (H4)*,* (H7)*, and* (C1) *implies* (H3)*.*

Proof. We prove that the conjunction of (H1), (H2), (H4), (H7), and the negation of (H3) implies the negation of (C1), since it require agents to be at certain location in $t+1$ which is not a neighbor of a location in t, effectively 'duplicating' agents on the graph. □

3.3 Size of the Encoding

Any encoding constructed using the formulas above is at least linear in the size of the graph, number of agents, and time bound T, but the final size of the encoding depends on the particular way we encode cardinality constraints:

Proposition 5. *Let (ALL) denote the set that results from taking the union of the clauses generated by* (H1)–(H11) *and* (C1)–(C3)*. The size of (ALL) is* $O(K \cdot |E| \cdot T) + O(S \cdot (|V| + |E|) \cdot T)$*, where* S *is the size of the encoding for the cardinality constraints.*

Proof. To prove the result, we observe that the number of clauses for (H1)-(H11) is bounded from above by the number of edges of the graph, $|E|$, similarly, the number of constraints of the form (C1). The number of constraints of the form (C2) and (C3), however, are bounded from above by $|V|$. □

4. SOC-optimal MAPF via MaxSAT and ASP

With the encoding in hand, now we describe how we find SOC-optimal solutions to MAPF. A standard approach to find a plan in SAT-based planning (e.g., Kautz & Selman, 1992) is to iteratively increase the time bound of the encoding until a solution is found. If one aims at cost optimality, this approach does not necessarily guarantees finding a SOC-optimal solution since SOC-optimal solutions are not necessarily makespan-minimal.

We follow a two-phase approach similar to the one described by Barták and Svancara (2019) and Gómez et al. (2021). Phase 1 finds a Makespan-minimal with minimum cost. Let such a solution be Π_{min}, and its makespan be T_{min}. To compute it, we iteratively increment the bound of the encoding until a solution is found. The lower bound of the iteration is set to the makespan of a *relaxed* solution Π_{rel}, in which no conflicts are considered, and which is efficiently computed using Dijkstra's algorithm. In phase 2, we call the MaxSAT solver with an encoding whose time bound is set to $T_{min} + c(\Pi_{min}) - c(\Pi_{rel}) - 1$, which provably (Surynek et al., 2016; Gómez et al., 2021) allows finding a SOC-optimal solution.

In our experimental evaluation we exploit the following two observations, which despite their simplicity have not been exploited by previous compilation-based approaches to MAPF. First, there is no need to run the same solver configuration on both phases. Second, the solution found on the first phase is a feasible (possibly non-optimal) solution for the second phase call. This allows us to pass such solution together with the input encoding to the MaxSAT call of the second phase. As a result, the solver may perform better, especially when using LSU as the optimization algorithm.

5. ASP Version of our Encoding

As we have explained above, the main difference between our encoding and ASP-MAPF's is our decision variable $\mathit{shift}_{v,u,t}$ which does not refer to the agent. We build the ASP version of our encoding on top of ASP-MAPF's and represent $\mathit{shift}_{v,u,t}$ as `shift(X,Y,A,T)` to express the fact that cell `(X,Y)` shifts the agent in the direction of action `A` at time `T`. In practice this means that we change the following ASP-MAPF rule for encoding the dynamics:

$$\begin{aligned} at(R,X) \leftarrow & exec(R,A,T-1), at(R,Xp,Yp,T-1), \\ & delta(A,Xp,Yp,X,Y), cost\_to\_go(R,X,Y,C), T+C \leq bound. \end{aligned}$$

by:

$$\begin{aligned} at(R,X,Y,T) \leftarrow & shift(Xp,Yp,A,T-1), at(R,Xp,Yp,T-1), \\ & delta(A,Xp,Yp,X,Y), cost\_to\_go(R,X,Y,C), T+C \leq bound. \end{aligned}$$

Swap conflicts are encoded with the two constraints, analogous to (H9):

$$\leftarrow shift(X,Y,right,T), shift(X+1,Y,left,T). \tag{12}$$
$$\leftarrow shift(X,Y,up,T), shift(X,Y+1,down,T). \tag{13}$$

Follow conflicts are expressed by translating (H10):

$$shift(X+1,Y,wait,T) \leftarrow shift(X,Y,right,T). \tag{14}$$
$$shift(X-1,Y,wait,T) \leftarrow shift(X,Y,left,T). \tag{15}$$
$$shift(X,Y+1,wait,T) \leftarrow shift(X,Y,up,T). \tag{16}$$
$$shift(X,Y-1,wait,T) \leftarrow shift(X,Y,down,T). \tag{17}$$

Finally, vertex conflicts are prevented via the following constraint:

$$\leftarrow \#count\{R : at(R,X,Y,T)\} > 1, free(X,Y), time(T).$$

The complete encoding is shown in Figure 2. We call the resulting solver ASP-MAPF2.

Proposition 6. *The number of clauses generated after grounding for the encoding of ASP-MAPF2 is $O(K \cdot |V| \cdot T)$. This holds when follow conflicts are supported (that is, by including* (14)–(17)*) and when they are not (that is, when* (14)–(17) *are not included).*

Proof. Straightforward from the structure of the program. □

6. Experimental Evaluation

In this section we present our experimental evaluation, whose objective was to evaluate the performance of the solvers based on our new encoding, compared to other state-of-the-art solvers.

6.1 Results on Previously Proposed Benchmarks

First we present our results on benchmark instances proposed by Gómez et al. (2021).

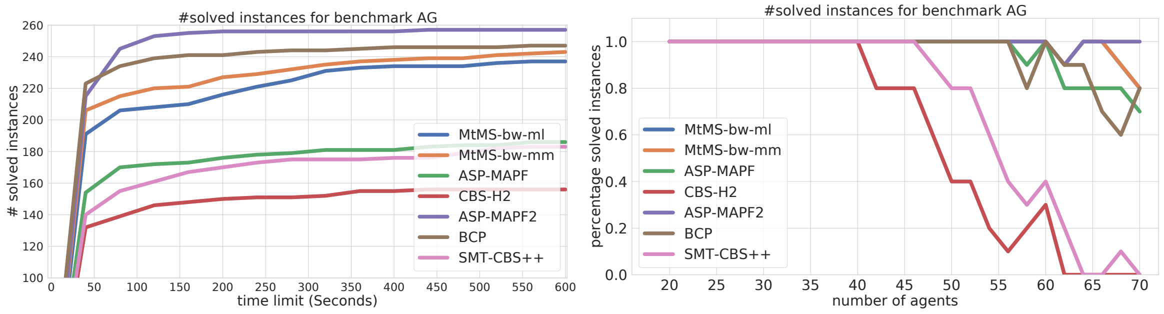
Benchmarks The benchmarks proposed by Gómez et al. (2021) are medium-scale maps, which are challenging for the both types of MAPF solvers we used: compilation-based and search-based. The characteristics of such benchmarks are:

AG: Instances in this set are 20 × 20 grids with 40 (10%) randomly placed obstacles with a number of agents that varies between 20 and 70, randomly distributed. Densest instances have around 27.5% of their cells occupied. This benchmark has 260 instances.

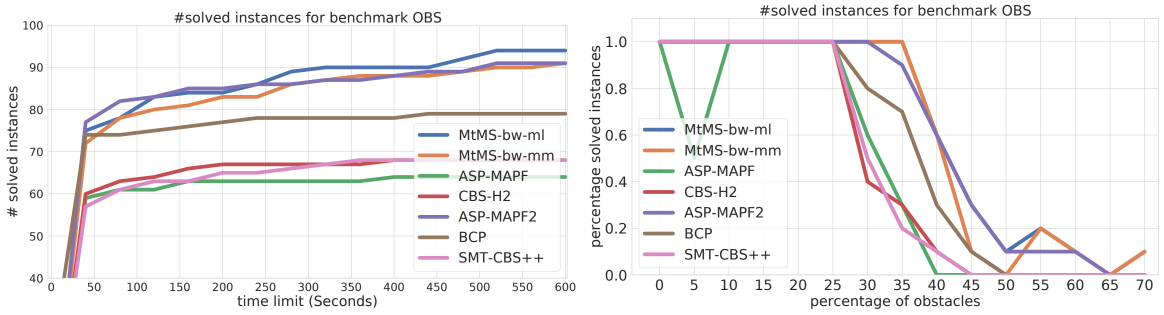
OBS: Instances in this benchmark are 20 × 20 with 20 randomly placed agents. These instances have between 0 and 70% of obstacles, randomly placed on the grid. The densest instances have around 75% of cells occupied. This benchmark has 150 instances.

```
% robot(r):             express that r is an agent in R,
% rangeX(x):            expresses that x is within the legal range for the X axis
% rangeY(y):            expresses that y is within the legal range for the Y axis
% time(t):              t is a time instant,
% action(a):            a is an action,
% obstacle(x,y):        specifies that cell (x,y) is an obstacle,
% goal(r,x,y):          specifies that the goal cell for agent r is (x,y).
% cost_to_go(r,x,y,c): specifies that the shortest path between (x,y) and r's goal is c
% delta(a,x,y,x',y'):  specifies that the cell (x',y') is reached when performing action a at cell (x,y).
% shift(x,y,a,t):      specifies that the cell (x,y) produce the action a to any agent on it at time t.
% at_goal(r,t):        specifies that agent r is at the goal at time t.
% penalty(r,t,1):      specifies the slack between the makespan T and the time instant
                       at which an agent has stopped at the goal.


% dynamics of the at predicate
at(R,X,Y,T) :- shift(Xp,Yp,A,T-1),at(R,Xp,Yp,T-1),delta(A,Xp,Yp,X,Y), cost_to_go(R,X,Y,C), T + C <= bound.
% the 5 possible actions
action(up; down; left; right; wait).

% a specification of the moves
free(X,Y)               :- rangeX(X), rangeY(Y), not obstacle(X,Y).
delta(right,X,Y,X+1,Y) :- rangeX(X), rangeX(X+1), rangeY(Y), free(X,Y).
delta(left,X,Y,X-1,Y)  :- rangeX(X), rangeX(X-1), rangeY(Y), free(X,Y).
delta(up,X,Y,X,Y+1)    :- rangeX(X), rangeY(Y), rangeY(Y+1), free(X,Y).
delta(down,X,Y,X,Y-1)  :- rangeX(X), rangeY(Y), rangeY(Y-1), free(X,Y).
delta(wait,X,Y,X,Y)    :- rangeX(X), rangeY(Y), free(X,Y).

% A single action per agent can be performed at each time instant.
1 {shift(X,Y,A,T-1) : action(A)} 1 :- free(X,Y), time(T).

:- at(R,X,Y,T),obstacle(X,Y).

% Vertex conflicts
:- #count{R : at(R,X,Y,T)} > 1, free(X,Y), time(T).

% Swap conflicts
:- shift(X,Y,right,T),shift(X+1,Y,left,T).
:- shift(X,Y,up,T),shift(X,Y+1,down,T).

% Goal achievement
at_goal(R,T)          :- at(R,X,Y,T), goal(R,X,Y).
:- robot(R), not at_goal(R,bound).
at_goal_back(R,bound) :- robot(R).
at_goal_back(R,T-1)   :- at_goal_back(R,T),at_goal(R,T-1).

% Cost of the optimal path.
penalty(R,T,1) :- time(T), dijkstra(R,Tp), T>Tp, at_goal_back(R,T-1).

Optimization statement.
#maximize {P,R,T : penalty(R,T,P)}.
```

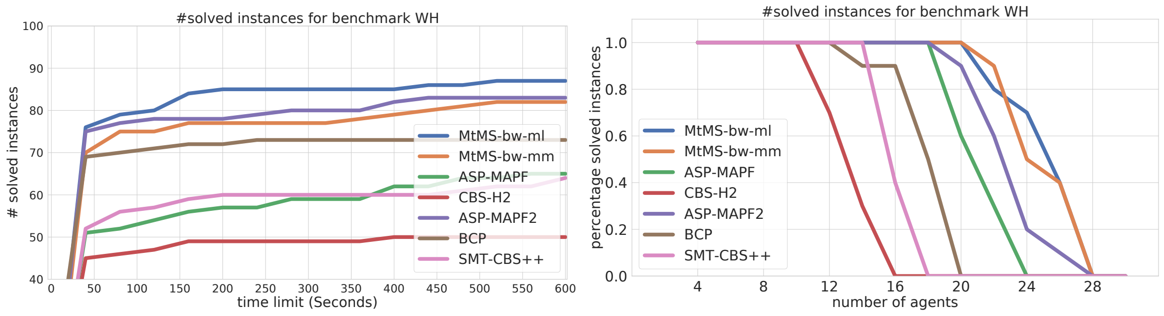
Figure 2: ASP-MAPF2's encoding



(a) AG: 20 × 20, increasing agents



(b) OBS: 20 × 20, increasing % of obstacles



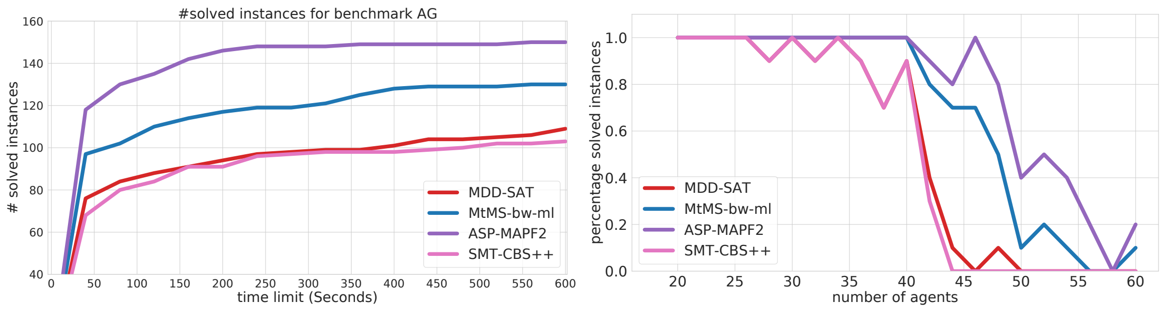
(c) WH: Warehouse, increasing agents

Figure 3: Comparison with BCP, ASP-MAPF and CBS-H2 on previously proposed random and warehouse maps, without follow conflicts.

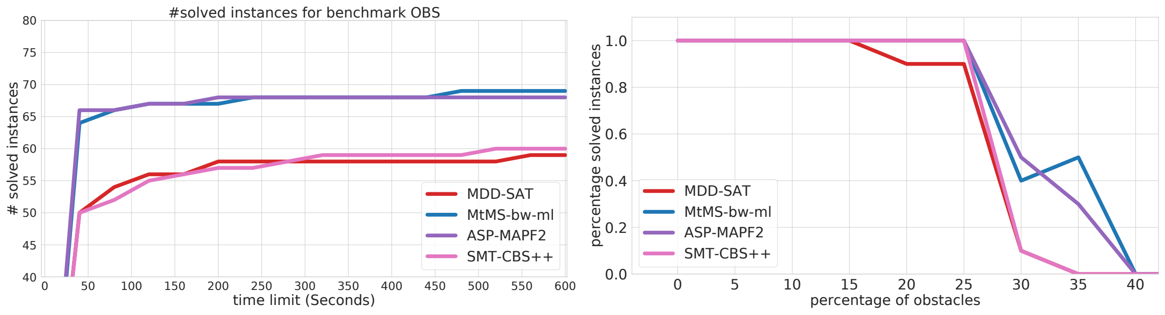
WH: All the instances in this set correspond to warehouses with a layout of size 9×21 with aisles of width 1. These instances vary in the number of agents, randomly distributed on the grid, which can be between 4 and 30. The densest instances on the set have around 63.5% of cells occupied. This benchmark has 140 instances.

The experiments reported in this subsection were done on a computer running Linux Mint 19.3 with an Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz and 16 Gbytes of RAM memory.

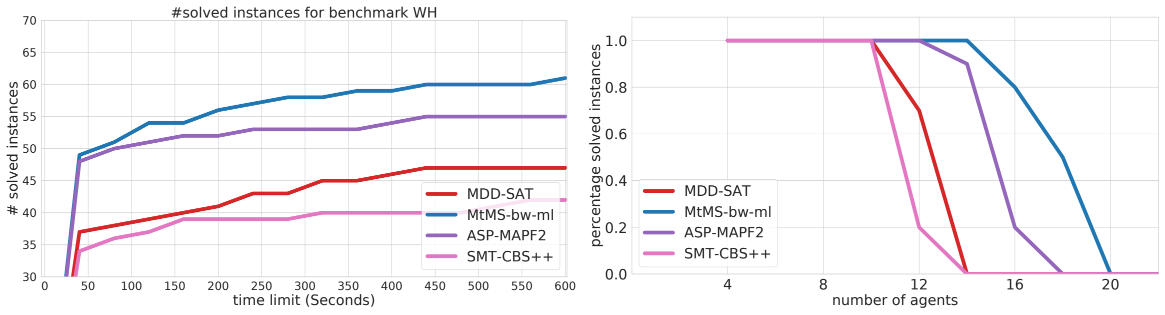
For these experiments, we tested several state-of-the-art solvers. Besides our new solvers based on MaxSAT (MtMS) and ASP (ASP-MAPF2), we chose to test ASP-MAPF, MDD-SAT, and SMT-CBS++ (Surynek, 2019a), since they are three good representatives of



(a) AG: 20 × 20, increasing agents



(b) OBS: 20 × 20, increasing % of obstacles



(c) WH: Warehouse, increasing agents

Figure 4: Comparison with MDD-SAT on previously proposed random and warehouse maps, using follow conflicts.

compilation-based SOC-optimal solvers. Moreover, we chose CBS-H2 (Li et al., 2019a) as a reference for search-based approaches and we also included BCP (Lam et al., 2019), an optimal MAPF solver which is a hybrid solver integrating search-based MAPF techniques with and integer programming compilation. For all solvers, we use versions provided by their authors. ASP-MAPF2 uses the same codebase as ASP-MAPF.

For MtMS[2] here we report results of the two configurations that yielded best results on a fix-all-but-one dimension tuning, as explained by Asín Achá et al. (2021). MtMS-bw-ml, refers to the MaxSAT encoding=0, using the bitwise encoding for at-most-one constraints and running MSU3 algorithm for Phase1 and LSU algorithm for Phase2. MtMS-bw-mm

2. Source code available at https://github.com/robertoasin/MtMF

	MtMS -bw-ml	MtMS -bw-mm	ASP -MAPF	CBS -H2	SMT -CBS++	ASP -MAPF2
AG	237	243	186	156	185	**257**
OBS	**94**	91	64	68	68	91
WH	**87**	82	65	50	64	83
Total	418	416	315	274	317	**431**

Table 2: # of solved instances at time limit (600 seconds).

only differs in the sence that MSU3 is also used for Phase2. An important characteristic of the chosen encoding is that it includes constraints H3, even though H3 is entailed by H1,H2,H4, H7 and C1. This redundant constraint seems to enhance the behavior of the Unit Propagation procedure of CDCL solvers, which, in turn, speeds up search. This is clear, at least in contrast with the minimal encoding composed by all constraints but H3 and H11.

Since MDD-SAT produces solutions strictly forbidding follow conflicts, but BCP, ASP-MAPF and CBS-H2 do not, we separate our experimental comparison in two parts. In the first part we compare versions of our solver that allow follow conflicts with BCP, ASP-MAPF and CBS-H2 (Figure 3). In the second part, we compare versions of our solvers that forbid follow conflicts with MDD-SAT. Likewise, SMT-CBS can be configured to allow or forbid follow conflicts, and we run it using the corresponding configuration. We compare on all three benchmarks, with the time limit set to 600 seconds per instance. We can make the observations that follow.

Allowing Follow Conflicts. Both variants of our MaxSAT approach: MtMS-bw-ml and MtMS-bw-mm, our ASP-MAPF2 solver and BCP clearly outperform ASP-MAPF and CBS-H2 in all benchmark sets by a substantial margin (cf. Table 2). Regarding the comparison between the three best solvers, MtMS-bw-ml seems to perform better on domains with more obstacles, while ASP-MAPF2 shows the best performance on instances with an increasing number of agents. BCP outperforms MtMS on instances with many agents and ASP-MAPF2 outperforms BCP on instances with increasing number of obstacles.

Forbidding Follow Conflicts. For these experiments, MtMS and ASP-MAPF2 are configured to return solutions free of follow conflicts. Specifically, in ASP-MAPF2 we replace the constraints for swap conflicts (12)–(13) with those for follow conflicts[3] (14)–(17). Likewise for MtMS we replace the constraints of the form (H9) with those of the form (H10).

For this comparison, we only use MtMS-bw-ml, since this solver seemed more robust in the previous setting. Our solvers outperform MDD-SAT by a substantial margin (cf. Table 3). Like in the previous benchmark, ASP-MAPF2 scales better in domains with fewer obstacles and an increasing number of agents, while MtMS scales better in domains with more obstacles.

3. Recall that a solution that is free of follow conflicts is also free of swap conflicts.

	MDD-SAT	MtMS-bw-ml	SMT-CBS++ (follow)	ASP-MAPF2 (follow)
AG	109	130	106	**150**
OBS	59	**69**	61	68
WH	47	**61**	42	55
Total	215	260	209	**273**

Table 3: # of solved instances using follow conflicts, at time limit (600 seconds).

6.2 Scalability and New Instances

Here we present our results on new benchmarks we generated in order to test the scalability of the new methods proposed in this paper. For this, we only considered the problem allowing follow conflicts, as more solvers can be compared. We did not include SMT-CBS++ since it was not clearly superior to CBS-H2 or ASP-MAPF in our previous suit of experiments. Here, MtMS refers to the MaxSAT solver, using bitwise encoding for at-most-one constraints, and the MSU3-LSU combination for the different phases of the solving process. Since this dataset is much bigger than the previous one, we used a Cluster composed of machines with the following characteristics: CPU: Intel(R) Xeon(R) CPU E3-1270 v6 @ 3.80GHz, RAM: 64 GBytes, OS: Ubuntu 18.04.5 LTS.

The new benchmark sets were generated as follows:

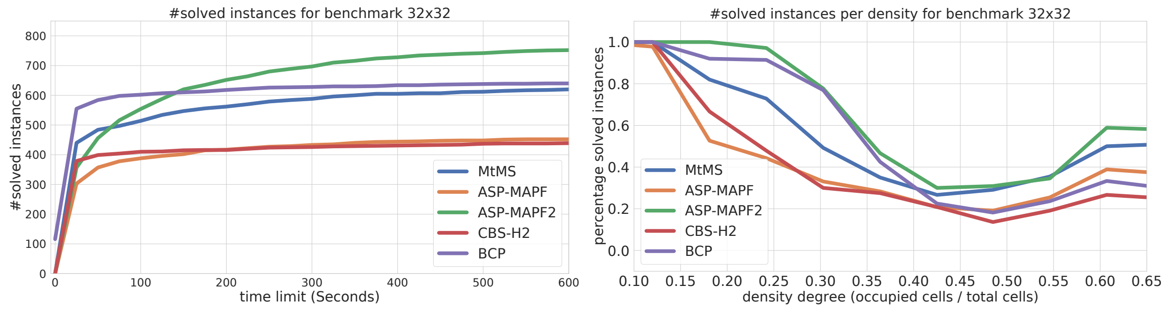
32×32 Instances in this set are 32×32 grids with a varying number of agents and a varying percentage of obstacles. The number of agents varies from 10 up to 120 and the percentage of obstacles in the grid varies from 5% to 60%. Densest instances have around 68% of their cells occupied. This benchmark has 1160 instances.

40×40 Instances in this set are 40×40 grids with a varying number of agents and a varying percentage of obstacles. The number of agents varies from 10 up to 120 and the percentage of obstacles in the grid varies from 5% to 60%. Densest instances have around 66% of their cells occupied. This benchmark has 1280 instances.

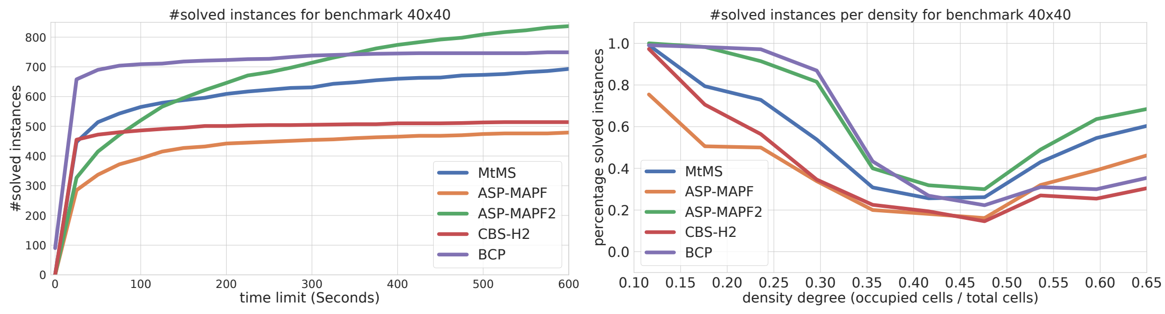
50×50 Instances in this set are 50×50 grids with a varying number of agents and a varying percentage of obstacles. The number of agents varies from 10 up to 120 and the percentage of obstacles in the grid varies from 5% to 64%. Densest instances have around 66% of their cells occupied. This benchmark has 1280 instances.

18×33 All the instances in this set correspond to warehouse layouts of size 18×35 with aisles of width 1. These instances vary in the number of agents, randomly distributed on the grid, which can be between 10 and 60. The densest instances on the set have around 57% of cells occupied. This benchmark has 510 instances.

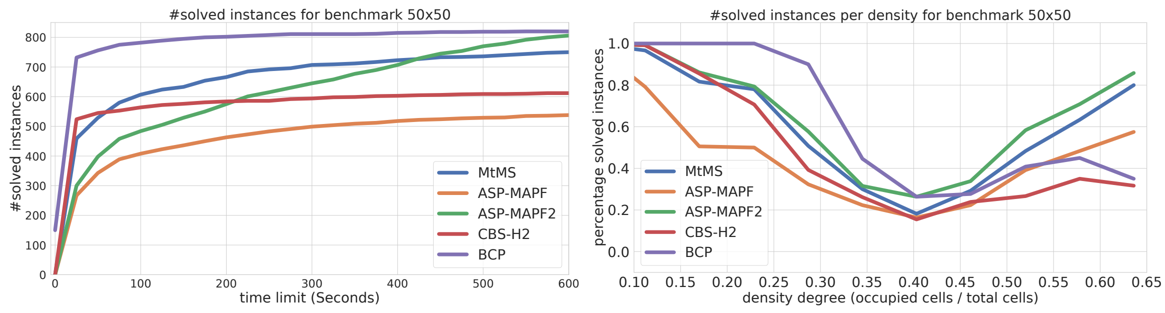
30×57 All the instances in this set correspond to warehouse layouts of size 30×57 with aisles of width 1. These instances vary in the number of agents, randomly distributed on the grid, which can be between 10 and 60. The densest instances on the set have around 56% of cells occupied. This benchmark has 510 instances.



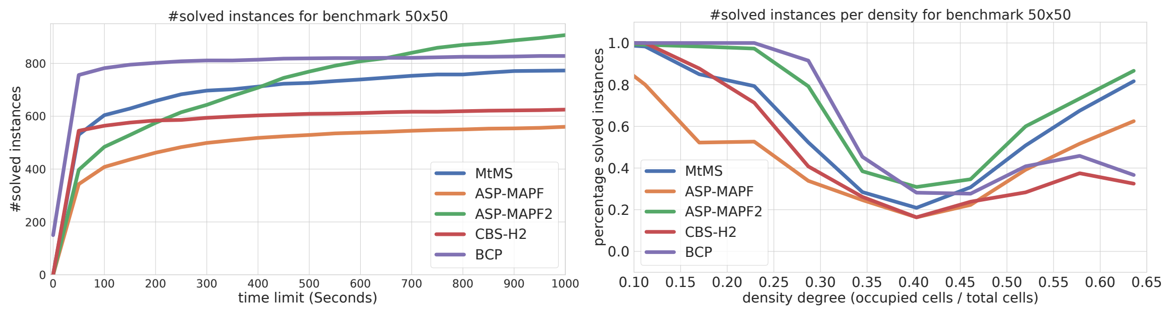
(a) 32×32 maps, 600 seconds time limit



(b) 40×40 maps, 600 seconds time limit

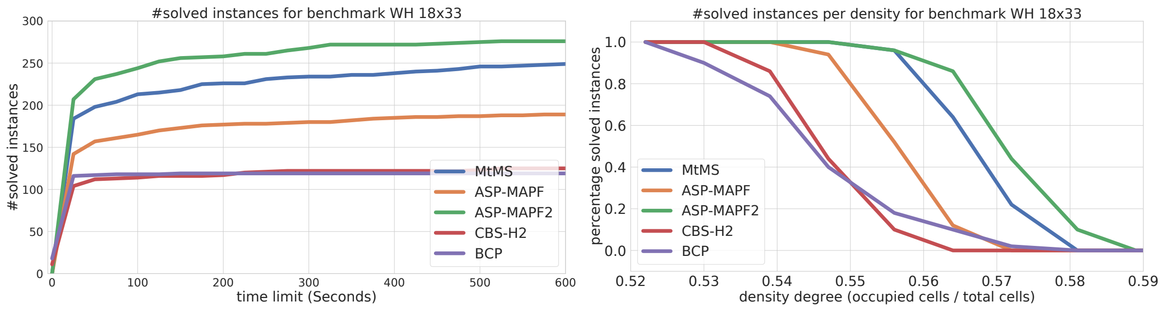


(c) 50×50 maps, 600 seconds time limit

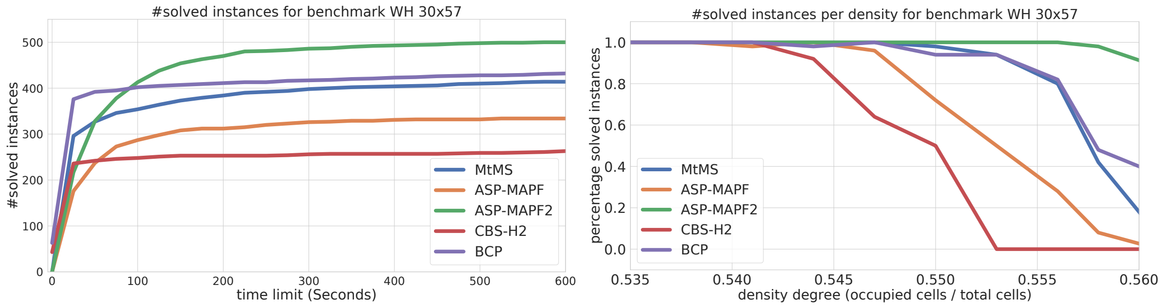


(d) 50×50 maps, 1000 seconds time limit

Figure 5: Comparison with BCP, ASP-MAPF and CBS-H2 on random maps with varying sizes, number of obstacles and number of agents, without follow conflicts.



(a) 18×33 warehouse maps, 600 seconds time limit



(b) 30×57 warehouse maps, 600 seconds time limit

Figure 6: Comparison with BCP, ASP-MAPF and CBS-H2 on random maps with varying sizes, number of obstacles and number of agents, without follow conflicts on Warehouses instances

Figure 5 shows the results the different solvers obtained on 32×32, 40×40 and 50×50 random maps with varying number of agents and obstacles. In the same way, Figure 6 shows the results obtained on Warehouses of size 18×33 and 30×57.

Regarding the total number of instances solved within the 600 seconds time limit we imposed, our new ASP-MAPF2 solver outperforms the other solvers on all scenarios, with exception of the 50×50 maps, where BCP prevails. BCP is second on most cases and the MaxSAT-based MtMS solver occupies the third position in this comparison. The three of them outperform CBS-H2 and the previous ASP-based solver by a big margin, with the exception of the 18×33 warehouses, where ASP-MAPF outperforms BCP, which behaves as CBS-H2[4].

In all cases, BCP shows a clear advantage in the time it needs to solve the instances which have an occupation density low-to-medium, but it is unable to solve many more of the higher-density instances. As can be seen in the plots, the peak of solved instances for BCP is achieved very early on the 600 second time horizon. On the opposite extreme is our new ASP-MAPF2 solver, which can be considered as the second slowest solver for handling the easier instances, but which is able to solve more instances than the rest of the competitors in the portfolio. Given this observation and the trend on the curves of the 50×50 maps, we decided to rerun the experiments increasing the time limit for this benchmark to 1000

4. BCP behavior in this dataset is unusual, since "easy" instances make the solver abort with an `overflow_error` error

seconds. The results for this can be seen on Figure 5d. There, we can observe that, given more time, ASP-MAPF2 continues solving more instances, while the rest of the solvers do not, showing it the solver that scales best.

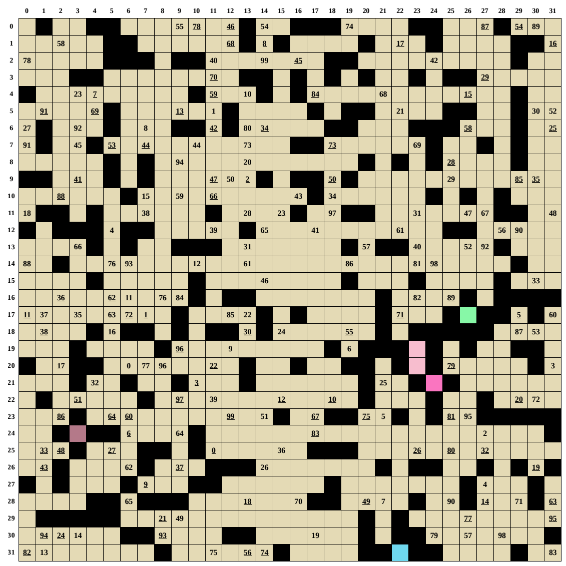
Besides analyzing the total number of solved instances, we study which instances are more challenging to solve. To this end, we analyzed the percentage of instances solved according to the instance density. The density of an instance is defined as the total number of cells occupied (either by an agent or by an obstacle) divided by the total number of cells in the map. Figures 5 and 6 also show the percentage of instances solved, by each solver, per density degree. The interpretation of the plot for the Warehouses maps, on Figure 6, is straightforward, as the percentage of solved instances keep decreasing for all solvers as density increases. But this does not happen for the random map instances. Figure 5 shows that the harder instances have densities around $40-50\%$ and, above that value, the random instances seem to be *easier* to solve. Although counter-intuitive, this can be explained on the way the random maps were generated. First, we populated the maps with a varying number of obstacles, for, later, defining connected starting and ending points for a varying number of agents. The first step can split the map in many connected components in which the starting and ending positions of the agents are disposed. What affects the solvers' behavior is how dense each connected component is, since each component can be seen as an independent MAPF problem. When the overall density of the maps is around $40-50\%$, we induce to the densest connected components. To illustrate this, Figure 7 shows 32×32 example maps with 30%, 40%, 50% and 60% density degree.

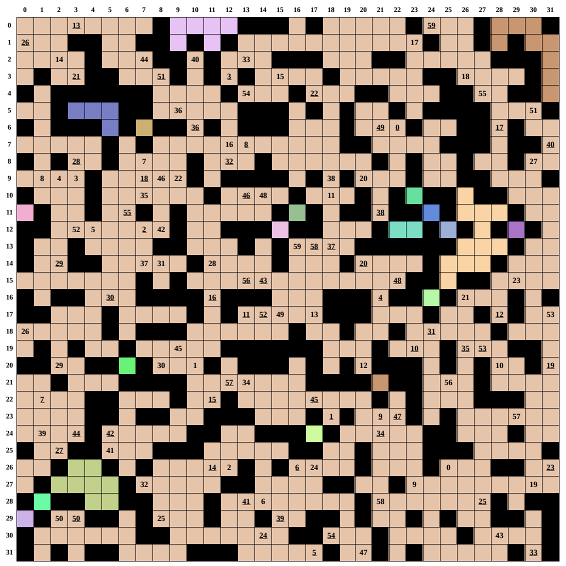
Overall, we make the following observations on the results the different solvers obtain in these larger maps:

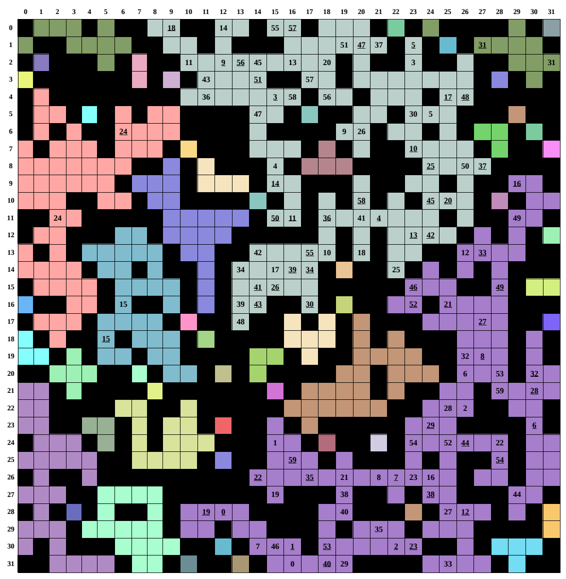
- Contrary with what happens on the scenarios of Section 6.1, our scalability tests shows that our new ASP-MAPF2 solver behaves better than the MaxSAT solver.
- In many cases, the grounding time for the ASP encoding, and the encoding time for the MaxSAT one, take more time than the search procedure.
- Our new encoding, either implemented through MaxSAT or through ASP, scales better and allows us to solve medium-sized problems to optimality.
- Particularly, the use of at-most-one constraints in the encoding allows ASP-MAPF2 scale better than MaxSAT, since ASP solvers are able to handle this type of constraints natively (Bomanson et al., 2020).
- BCP scales better than our encodings in the size of the map, but does not do so while increasing the number of agents and, in general, increasing the overall density of the scenarios.

7. Summary and Future Work

We presented a new Boolean encoding for SOC-optimal MAPF. We studied extensively how this encoding can be exploited within MaxSAT and ASP solvers. The encoding is compact. When implemented in ASP, after grounding, it produces a number of clauses that is linear on the number of agents, size of the map, and makespan. When implemented in MaxSAT,


(a) 32 × 32 maps with 30% occupation


(b) 32 × 32 maps with 40% occupation


(c) 32 × 32 maps with 50% occupation

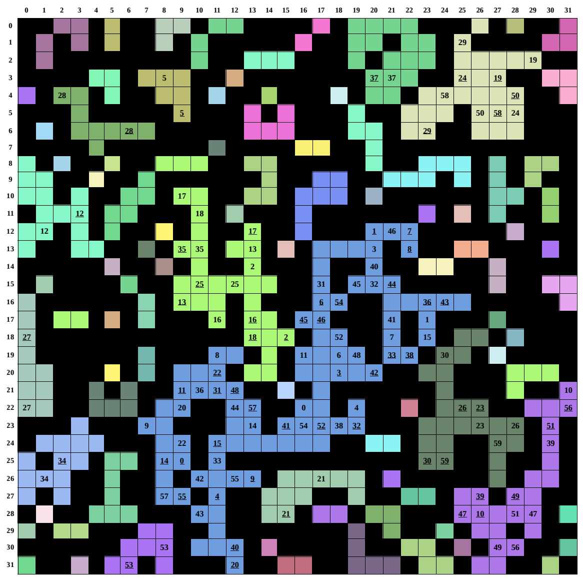

(d) 32 × 32 maps with 60% occupation

Figure 7: Example of 32 × 32 instances with 40, 50 and 60% of occupied cells. Each color represents a connected component of the graph.

the number of resulting clauses depends on the encoding used for cardinality constraints. Specifically, our encoding for vertex conflicts, done via a cardinality constraint, may result in a number of clauses that may be linear, quadratic or superlinear on the number of agents, depending on the algorithm used to encode at-most-one (AMO) constraints. Our experimental evaluation showed that among a number of possible logically equivalent formulas for MaxSAT, the one that yields best result is a redundant encoding whose performance is explained by the fact that it strengthens unit propagation.

An important advantage of our encoding is that it can be exploited with the latest state-of-the-art ASP and MaxSAT technology, which has been designed for optimization. In our experimental evaluation, we evaluate over a number of square grids and warehouse settings. We show that ASP-MAPF2, the solver that implements our encoding, outperforms other algorithms in dense maps. The ASP encoding seems superior to the MaxSAT encoding in relatively large maps—larger than 20×20—but the MaxSAT solver seems superior in relatively small maps in which there is limited space for agents to move; specifically in smaller warehouses with a large number of agents. We observe that one of the reasons for the superior performance of ASP solvers over MaxSAT solvers is that ASP solvers handle cardinality constraints natively.

References

Asín Achá, R., López, R., Hagedorn, S., & Baier, J. A. (2021). A new boolean encoding for MAPF and its performance with ASP and MaxSAT solvers. In *Proceedings of the 14th Symposium on Combinatorial Search (SoCS)*, pp. 11–19.

Avellaneda, F. (2020). A short description of the solver EvalMaxSAT. MaxSAT Evaluation 2020.

Bacchus, F., Berg, J., Järvisalo, M., & Martins, R. (2020). *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*. University of Helsinki, Department of Computer Science.

Barták, R., & Svancara, J. (2019). On SAT-based approaches for multi-agent path finding with the sum-of-costs objective. In Surynek, P., & Yeoh, W. (Eds.), *Proceedings of the 12th Symposium on Combinatorial Search (SoCS)*, pp. 10–17. AAAI Press.

Barták, R., Zhou, N., Stern, R., Boyarski, E., & Surynek, P. (2017). Modeling and solving the multi-agent pathfinding problem in picat. In *29th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 959–966, Boston, MA, USA. IEEE Computer Society.

Berg, J., Bacchus, F., & Poole, A. (2020). Abstract cores in implicit hitting set maxsat solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pp. 277–294. Springer.

Biere, A., Heule, M., & van Maaren, H. (2009). *Handbook of satisfiability*, Vol. 185. IOS press.

Bomanson, J., et al. (2020). *Normalization and Rewriting for Answer Set Programming and Optimization*. Ph.D. thesis, Aalto University.

Erdem, E., Kisa, D. G., Öztok, U., & Schüller, P. (2013). A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press.

Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, T. K. S., & Koenig, S. (2018). Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 83–87, Delft, The Netherlands.

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

Gebser, M., Obermeier, P., Otto, T., Schaub, T., Sabuncu, O., Nguyen, V., & Son, T. C. (2018). Experimenting with robotic intra-logistics domains. *TPLP*, *18*(3-4), 502–519.

Gómez, R. N., Hernández, C., & Baier, J. A. (2020). Solving sum-of-costs multi-agent pathfinding with answer-set programming. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*, pp. 9867–9874, New York, NY, USA. AAAI Press.

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

Ignatiev, A., Morgado, A., & Marques-Silva, J. (2019). Rc2: An efficient maxsat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *11*(1), 53–64.

Kautz, H. A., & Selman, B. (1992). Planning as satisfiability. In Neumann, B. (Ed.), *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, pp. 359–363.

Koshimura, M., Zhang, T., Fujita, H., & Hasegawa, R. (2012). Qmaxsat: A partial max-sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *8*(1-2), 95–100.

Lam, E., Bodic, P. L., Harabor, D. D., & Stuckey, P. J. (2019). Branch-and-cut-and-price for multi-agent pathfinding. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1289–1296.

Li, J., Felner, A., Boyarski, E., Ma, H., & Koenig, S. (2019a). Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 442–449.

Li, J., Gong, M., Liang, Z., Liu, W., Tong, Z., Yi, L., Morris, R., Pasearanu, C., & Koenig, S. (2019b). Departure scheduling and taxiway path planning under uncertainty. In *AIAA Aviation 2019 Forum (AIAA)*.

Lifschitz, V. (2008). What is answer set programming?. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1594–1597.

Morgado, A., Liffiton, M., & Marques-Silva, J. (2012). Maxsat-based mcs enumeration. In *Haifa Verification Conference*, pp. 86–101. Springer.

Nebel, B. (2020). On the computational complexity of multi-agent pathfinding on directed graphs. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 212–216.

Nguyen, V., Obermeier, P., Son, T. C., Schaub, T., & Yeoh, W. (2017). Generalized target assignment and path finding using answer set programming. In Sierra, C. (Ed.), *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1216–1223, Melbourne, Australia. ijcai.org.

Paxian, T., Reimer, S., & Becker, B. (2019). Pacose: An Iterative SAT-based MaxSAT Solver. MaxSAT Evaluation 2019.

Piotrów, M. (2019). Uwrmaxsat-a new minisat+-based solver in maxsat evaluation 2019. MaxSAT Evaluation 2019.

Reiter, R. (1991). *The Frame Problem in the Situation Calculus: A Simple Solution (sometimes) and a completeness result for goal regression*, pp. 359–380. Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. Academic Press, San Diego, CA.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, Cambridge, MA.

Sharon, G., Stern, R., Felner, A., & Sturtevant, N. R. (2012). Conflict-based search for optimal multi-agent path finding. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI)*, Toronto, Canada.

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., & Boyarski, E. (2019). Multi-agent pathfinding: Definitions, variants, and benchmarks. In Surynek, P., & Yeoh, W. (Eds.), *Proceedings of the 12th Symposium on Combinatorial Search (SoCS)*, pp. 151–159. AAAI Press.

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In Fox, M., & Poole, D. (Eds.), *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*, p. 1261–1263. AAAI Press.

Surynek, P. (2019a). Multi-agent path finding with continuous time and geometric agents viewed through satisfiability modulo theories (SMT). In Surynek, P., & Yeoh, W. (Eds.), *Proceedings of the 12th Symposium on Combinatorial Search (SoCS)*, pp. 200–201. AAAI Press.

Surynek, P. (2019b). Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In Kraus, S. (Ed.), *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1177–1183.

Surynek, P., Felner, A., Stern, R., & Boyarski, E. (2016). Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI)*, pp. 810–818. IOS Press.

Surynek, P., Stern, R., Boyarski, E., & Felner, A. (2022). Migrating techniques from search-based multi-agent path finding solvers to sat-based approach. *JAIR*, *73*, 553–618.

Wang, K. C., & Botea, A. (2008). Fast and memory-efficient multi-agent pathfinding. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 380–387.

Yu, J., & LaValle, S. M. (2013). Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*, Bellevue, Washington.