A Summary of Adaptation of Techniques from Search-based Optimal Multi-Agent Path Finding Solvers to Compilation-based Approach

Pavel Surynek
Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00 Praha 6, Czech Republic

Abstract

In the multi-agent path finding problem (MAPF) we are given a set of agents each with respective start and goal positions. The task is to find paths for all agents while avoiding collisions aiming to minimize an objective function. Two such common objective functions is the sum-of-costs and the makespan. Many optimal solvers were introduced in the past decade - two prominent categories of solvers can be distinguished: search-based solvers and compilation-based solvers.

Search-based solvers were developed and tested for the sum-of-costs objective while the most prominent compilation-based solvers that are built around Boolean satisfiability (SAT) were designed for the makespan objective. Very little was known on the performance and relevance of the compilation-based approach on the sum-of-costs objective.

In this paper we show how to close the gap between these cost functions in the compilation-based approach. Moreover we study applicability of various techniques developed for search-based solvers in the compilation-based approach.

A part of this paper introduces a SAT-solver that is directly aimed to solve the sum-of-costs objective function. Using both a lower bound on the sum-of-costs and an upper bound on the makespan, we are able to have a reasonable number of variables in our SAT encoding. We then further improve the encoding by borrowing ideas from ICTS, a search-based solver.

Experimental evaluation on several domains show that there are many scenarios where our new SAT-based methods outperform the best variants of previous sum-of-costs search solvers - the ICTS, CBS algorithms, and ICBS algorithms.

Keywords: Multi-agent path finding (MAPF), sum-of-costs, makespan, Boolean satisfiability (SAT), optimality, suboptimality, propositional encoding, cardinality constraint

1. Introduction and Background

The multi-agent path finding (MAPF) problem consists a graph, \( G = (V,E) \) and a set \( A = \{a_1, a_2, \ldots, a_k\} \) of \( k \) agents. Time is discretized into time steps. The arrangement of agents at
time-step $t$ is denoted as $\alpha_t$. Each agent $a_i$ has a start position $\alpha_0(a_i) \in V$ and a goal position $\alpha_+(a_i) \in V$. At each time step an agent can either move to an adjacent empty location or wait in its current location. The task is to find a sequence of move/wait actions for each agent $a_i$, moving it from $\alpha_0(a_i)$ to $\alpha_+(a_i)$ such that agents do not conflict, i.e., do not occupy the same location at the same time. Formally, an MAPF instance is a tuple $\Sigma = (G = (V, E), A, \alpha_0, \alpha_+)$. A solution for $\Sigma$ is a sequence of arrangements $S(\Sigma) = [\alpha_0, \alpha_1, ..., \alpha_\mu]$ such that $\alpha_{\mu+1}$ results from valid movements from $\alpha_t$ for $t = 1, 2, ..., \mu - 1$. An example of MAPF and its solution are shown in Figure 1.

MAPF has practical applications in video games, traffic control, robotics etc. (see [31] for a survey). The scope of this paper is limited to the setting of fully cooperative agents that are centrally controlled. MAPF is usually solved aiming to minimize one of the two commonly-used global cumulative cost functions:

1. **sum-of-costs** (denoted $\xi$) is the summation, over all agents, of the number of time steps required to reach the goal location [11] [39] [32] [31]. Formally, $\xi = \sum_{i=1}^{\mu} \xi(a_i)$, where $\xi(a_i)$ is an individual path cost of agent $a_i$.

2. **makespan** (denoted $\mu$) is the total time until the last agent reaches its destination (i.e., the maximum of the individual costs) [41] [51] [44].

In the individual path cost, each action of an agent (move action or wait action) is assumed to have a unit cost. It is important to note that in any solution $S(\Sigma)$ it holds that $\mu \leq \xi \leq m \cdot \mu$. Thus the optimal makespan is usually smaller than the optimal sum-of-costs.

Intuitively, sum-of-costs can be regarded as total energy consumption of all agents such that at each time step spent before reaching the goal the agent consumes one unit of energy. In this respect, it is not surprising that optimization of one of these two objectives goes against the other - total time can be saved at the cost of increased energy consumption and vice versa. An example of MAPF instance where any makespan optimal solution has sum-of-costs that is greater than the optimum and any sum-of-costs optimal solution has makespan that is greater than the optimum makespan is shown in Figure 2.

Finding optimal solutions for both variants with any standard style of agents’ movement is

---

1Some variants of MAPF relax the empty location requirement by allowing a chain of neighboring agents to move, given that the head of the chain enters an empty locations. Most MAPF algorithms are robust (or at least easily modified) across these variants.
Figure 2: An instance of MAPF where makespan and sum-of-costs optimal solutions differ - that is, any makespan optimal solution is strictly sum-of-costs suboptimal and any sum-of-costs optimal solution is strictly makespan suboptimal.

NP-hard even on planar graphs [23, 24, 41, 54, 62, 60]. Therefore, many suboptimal solvers were developed and are usually used when \( m \) is large or when the graph is large [8, 18, 26, 28, 36, 58]. In contrast to difficulty of finding optimal solutions, finding any feasible solution or detecting unsolvability of a given instance can be done polynomial time [9, 19, 20, 45, 53].

1.1. Optimal MAPF Solvers

Many optimal solvers were introduced in the past decade but they all focus on one of these cost functions:

- **(1) optimal sum-of-costs solvers.** Most of them are based on search. Some of these search-based solvers are variants of the A* algorithm on a global search space in which all different ways to place \( m \) agents into \( V \) vertices, one agent per vertex, are considered [39, 56]. Other employ novel search trees [7, 31, 32]. Search-based solvers feature various search space compilation techniques like independence detection (ID) [39] or multi-value decision diagrams (MDDs) [32].

- **(2) optimal makespan solvers.** Many optimal solvers were developed for the makespan variant. Most of them are compilation-based solvers which reduce MAPF to known problems such as Constraint Satisfaction (CSP) [28], Boolean Satisfiability (SAT) [42], Inductive Logic Programming (ILP) [59] and Answer Set Programming (ASP) [12]. These works mostly prove a polynomial-time reduction from MAPF to these problems. Existing reductions are usually designed for the makespan variant of MAPF; they are not applicable for the sum-of-costs variant.
1.2. Current Shortcomings and Contribution

A major weakness across all these works is that each of these algorithms was introduced and applied for one of these objective functions only. Furthermore, the connection/comparison between different algorithms was usually done only within a given class of algorithms and cost objective but not across these two classes. Finally, experiments were always performed on one objective-function and very little is known on the performance and relevance of any given algorithm (developed for one cost function) on the other objective function.

This paper aims to close the gap. First, we discuss how to migrate algorithms across the different objective functions. Most of the search-based algorithms developed for the sum-of-costs objective function can be modified to the makespan variant with some technical adaptations such as modifying the cost function and the way the state-space is represented. Some initial directions are given by [31] and we give a complete picture here.

By contrast, the compilation-based algorithms that were developed for the makespan objective function are not trivially modified to the sum-of-costs variant and sometimes a completely new encoding is needed.

A major algorithmic contribution of this paper is that we develop the compilation-based solver for the sum-of-costs variant to SAT. Our SAT-based solver is based on establishing relations between the maximum makespan under the given sum-of-costs which enables to build SAT encodings that represent all feasible solutions for the given sum-of-costs. Bounds on the sum-of-costs in the SAT encoding are established by cardinality constraints [3, 35]. We show how to use known lower bounds on the sum-of-costs to reduce the number of variables that encode these cardinality constraints so as to be practical for current SAT solvers.

We then present how to migrate various techniques used in search-based approach to our new SAT-based solver. First, we adapt ideas from the ICTS algorithm [32] that uses multi-value decision diagrams (MDDs) [38] to further reduce the size of SAT encodings. Next, we show how to integrate a modification of independence detection [39] technique into the SAT-based solver. Finally, we demonstrate flexibility of our SAT-based solver by modifying it into a bounded sum-of-costs suboptimal solver - a modification applicable in search-based approach to trade-off quality of solutions and runtime [4].

Successful migration of techniques demonstrates the potential of combining ideas from both classes of approaches - search-based and compilation-based. Experimental results show that our SAT solver with various enhancements outperforms the best existing search-based solvers for the sum-of-costs variant on a number scenarios.

Hence as a results of our unification provided in the beginning of this paper we have an arsenal of algorithms which can be applied for both objective functions. We conclude this paper by providing experimental results comparing the hardness of solving MAPF with SAT-based and search-based solvers under the makespan and the sum-of-costs objectives in a number of domains.

2. Related Work

We summarize existing algorithmic approaches to MAPF in this section. We categorize algorithms into two streams according to the objective function they use. For optimization of sum-of-costs great variety of algorithms has been proposed. On the other hand, previous makespan optimal algorithms are limited to compilation-based approach where the target formalism is represented by Boolean satisfiability. Many sum-of-costs optimal algorithms can be directly modi-
fied for the makespan variant. The opposite migration from the makespan optimal case to sum-of-costs optimality in compilation-based algorithms is however not straightforward.

2.1. Previous Sum-of-Costs Optimal Algorithms and Techniques

A*-based Algorithms. A* is a general-purpose algorithm that is well suited to solve MAPF. A common straightforward state-space where the states are the different ways to place $k$ agents into $|V|$ vertices, one agent per vertex is used. In the start and goal states agent $a_i$ is located at vertices $s_i$ and $g_i$, respectively. Operators between states are all non-conflicting combinations of actions (including wait) that can be taken by the agents.

Branching factor in A*-based algorithms is an important measure. Denote $b(a_i)$ the branching factor of single agent $a_i$. Then the effective branching factor for $k$ agents, denoted by $b = \prod_{i=1}^{k} b(a_i)$. For example, in a 4-connected grid $b(a_i) = 5$ for most of agents; an agent can either move in four cardinal directions or wait at its current location. Then $b$, is roughly $5^k$; though usually a bit smaller because many possible combinations of moves result in immediate conflicts, especially when the environment is dense with agents.

A simple admissible heuristic that is used within A* for MAPF is to sum the individual heuristics of the single agents such as Manhattan distance for 4-connected grids or Euclidean distance for Euclidean graphs [27]. A more-informed heuristic is called the sum of individual costs heuristic. For each agent $a_i$ we calculate its optimal path cost from its current state (position) $\alpha(a_i)$ to $\alpha(a_i)$ assuming that other agents do not exist. Then, we sum these costs over all agents. More-informed heuristics uses forms of pattern-databases [16, 15].

The most important drawback of A*-based algorithms is they need to tackle with is the branching factor $b$ of a given state may be exponential in $k$. We briefly summarize attempts to overcome the high branching factor.

Operator Decomposition (OD). Instead of moving all the agents to their next positions at once, agents advance to the next position one by one in a fixed order within the OD concept. The original operator for obtaining the next state is thus decomposed into a sequence of operators for individual agents each of branching factor $b(a_i)$. OD together with a reservation table enabled computations of next states where agents do not collide with each other in CA*, HCA*, and W_HCA* [36].

Pruning of states by OD with respect to a given admissible heuristic was suggested by Stanley in [39]. Two conceptually different states are distinguished - standard and intermediate. Intermediate state correspond to the situation when not all the agents finished their move while standard states correspond to states in the original representation with no OD. The major strength of OD lies in the fact that top-level A* algorithm does not need to distinguish between standard and intermediate states. The next node for expansion is selected among both standard and intermediate states while the cost function applies to both types of states. It may thus happen that a certain intermediate state is not expanded towards a standard state because other states turned out to be better according to the cost function. Such a kind of search space pruning cannot be done without operator decomposition as there would be standard states only.

Independence Detection (ID). Closely related to OD also introduced in [39] is a concept of independence detection that can also regarded as a branching factor reduction technique. The main idea behind this technique is that difficulty of MAPF solving optimally grows exponentially with the number of agents. It would be ideal, if we could divide the problem into a series of smaller sub problems, solve them independently, and then combine them.

The simple approach, called simple independence detection (SID), assigns each agent to a group so that every group consists of exactly one agent. Then, for each of these groups, an
optimal solution is found independently. Every pair of these solutions is evaluated and if the two groups solutions are in conflict (that is, when a collision of agents belonging to different group occurs), the groups are merged together and a new optimal solution is found for the group (now considering composite search space obtained as a Cartesian product of search spaces of individual groups). If there are no conflicting solutions, the solutions can be merged to a single solution of the original problem. This approach can be further improved by deliberate avoiding of groups merging.

Generally, each agent has more than one possible optimal path and also group of agents has more that one optimal solution. However, SID considers only one of these optimal paths/solutions. The improvement of SID known as independence detection (ID) is as follows. Lets have two conflicting groups $G_1$ and $G_2$. First, we try to replan $G_1$ so that the new solution has the same cost but actions that are in conflict with $G_2$ are forbidden. If no such solution is possible, try to similarly replan $G_2$. If this is also not possible, then merge $G_1$ and $G_2$ into a new group. In case either of the replanning was successful, that group needs to be evaluated with every other group again. This can lead to infinite cycle. Therefore, if two groups were already in conflict before, merge them without trying to re-plan. See Figure 3 for illustration.

Standley uses ID in combination with the A* algorithm. In case A* has a choice between several nodes with the same minimal cost, the one with least amount of conflicts is expanded first. This technique yields an optimal solution that has a minimal number of conflicts with other groups. This property is useful when replanning of a groups solution is needed. Both SID and ID do not solve MAPF on their own, they only divide the problem into smaller sub-problems that are solved by any possible MAPF algorithms. Thus, ID and SID are general frameworks which can be executed on top of any MAPF solver.

**More A*-based Algorithms.** Enhanced Partial Expansion (EPEA*) [15] avoids the generation of surplus nodes (i.e., nodes $n$ with $f(n) > C^*$ where $C^*$ is the optimal cost; we assume standard A* notation with $f(n) = g(n) + h(n)$) by using a priori domain knowledge. When expanding a node $n$ EPEA* generates only the children $n_c$ with $f(n_c) = f(n)$ and the smallest $f$-value among those children with $f(n_c) > f(n)$ ($\xi$ stands for $f$ in the context of MAPF). The other children of $n$ are discarded. This is done with the help of a domain-dependent operator selection function (OSF). The OSF returns the exact list of operators which will generate nodes $n$ with the desired $f(n)$. Node $n$ is then re-inserted into OPEN setting $f(n)$ to the $f$-value of the next best child of $n$. In this way, EPEA* avoids the generation of surplus nodes and dramatically reduces the number of generated nodes. An OSF for MAPF can be efficiently built as the effect on the $f$-value of moving a single agent in a given direction can be easily computed. For more details see [15].

M* [57, 56] and its enhanced recursive variant (RM*) are important A*-based algorithms related to ID. M* dynamically changes the dimensionality and branching factor based on con-
flicts. The dimensionality is the number of agents that are not allowed to conflict. When a node is expanded, M* initially generates only one child in which each agent takes (one of) its individual optimal move towards the goal (dimensionality 1). This continues until a conflict occurs between \( q \geq 2 \) agents at node \( n \). At this point, the dimensionality of all the nodes on the branch leading from the root to \( n \) is increased to \( q \) and all these nodes are placed back in OPEN list. When one of these nodes is re-expanded, it generates \( b^q \) children where the \( q \) conflicting agents make all possible moves and the \( k - q \) non-conflicting agents make their individual optimal move. An enhanced variant of M* called ODRM* \[13\] builds RM* on top of Standley’s OD rather than plain A*.

**Increasing Cost Tree Search.** The *Increasing Cost Tree Search* algorithm (ICTS) \[34, 32\] is a two-level MAPF solver which is conceptually different from A*. This algorithm is particularly important as its concepts will be be migrated into the SAT framework we are about to introduce. ICTS works as follows.

At its high level, ICTS searches the increasing cost tree (ICT). Every node in the ICT consists of a \( k \)-ary vector \([\xi(a_1), \xi(a_2), \ldots, \xi(a_k)]\) which represents all possible solutions in which the individual path cost of agent \( a_i \) is exactly \( \xi(a_i) \). The root of the ICT is \([\xi^*(a_1), \xi^*(a_2), \ldots, \xi^*(a_k)]\), where \( \xi^*(a_i) \) is the optimal individual path cost for agent \( a_i \) ignoring other agents, i.e., it is the length of the shortest path from \( a_0(a_i) \) to \( a_t(a_i) \) in \( G \). A child in the ICT is generated by increasing the cost for one of the agents by 1. An ICT node \([\xi(a_1), \xi(a_2), \ldots, \xi(a_k)]\) is a goal if there is a complete non-conflicting solution such that the cost of the individual path for any agent \( a_i \) is exactly \( \xi(a_i) \). Figure 4 illustrates an ICT with 3 agents, all with optimal individual path costs of 10. Dashed lines mark duplicate children which can be pruned. The total cost of a node is \( \sum_{i=0}^{k} \xi(a_i) \). For the root this is exactly \( h_{ICT}(a_0) = \sum_{i=0}^{k} \xi^*(a_i) \). Since all nodes at the same height have the same total cost, a breadth-first search of the ICT will find the optimal solution.

The low level acts as a goal test for the high level. For each ICT node \([\xi(a_1), \xi(a_2), \ldots, \xi(a_k)]\) visited by the high level, the low level is invoked. Its task is to find a non-conflicting complete solution such that the cost of the individual path of agent \( a_i \) is exactly \( \xi(a_i) \). For each agent \( a_i \), ICTS stores all single-agent paths of cost \( \xi(a_i) \) in a special compact data-structure called a multi-value decision diagram (MDD) \[38\] - MDD will be defined precisely later.

The low level searches the cross product of the MDDs in order to find a set of \( k \) non-conflicting paths for the different agents. If such a non-conflicting set of paths exists, the low level returns true and the search halts. Otherwise, false is returned and the high level continues to the next high-level node (of a different cost combination).

ICTS also implements various pruning rules to enhance the search. A full study of these pruning rules and their connection to CSP is provided in \[32\].

**Conflict-based Search (CBS).** Another optimal MAPF solver not based on A* is Conflict-Based Search (CBS) \[39, 31\]. In CBS, agents are associated with constraints. A constraint for agent \( a_i \) is a tuple \( (a_i, v, t) \) where agent \( a_i \) is prohibited from occupying vertex \( v \) at time step \( t \). A consistent path for agent \( a_i \) is a path that satisfies all of \( a_i \)’s constraints, and a consistent solution
conflict is resolved by adding constraints. If a conflict occurs during the validation, a conflict is found for two (or more) agents, the validation halts and the agents’ plans are modified to avoid the conflict by replacing part of their plans with optimal ones that avoid the conflict. A sub-optimal solution is available at any stage of the algorithm. The length of replaced sub-sequences is increased in each iteration of the algorithm until it eventually covers the entire makespan. This ensures that given enough time the algorithm returns a makespan optimal solution. A sub-optimal solution is available at any stage of the algorithm.

2.2. Previous Makespan Optimal Algorithms

Major development in the makespan optimal MAPF solving has been done over the Boolean satisfiability (SAT) paradigm. Early works that compile MAPF to SAT focused on solution improvements in terms of shortening the makespan towards the optimum in anytime manner. Historically the first encoding of MAPF to SAT INVERSE relies on log-space encoded variables that represent what agent is located in vertex at each time step that is, the inverse assignment over bit vectors is constructed on top of variables such that it is satisfiable if and only if a solution to the input encoding is that various efficient encodings of the A-encoding are used. The next development has been done in SAT encoding called INVERSE SAT encoding. The INVERSE encoding employs log-space encoded variables but position of agent at time step that is, is represented instead of representing vertex occupancy - that is, variables are represented using log-space encoding. To ensure that conflicts among agents in vertices do not occur, the ALL-DIFFERENT constraint is introduced for variables over all agents for each timestep. The advantage of the ALL-DIFFERENT encoding is that various efficient encodings of the ALL-DIFFERENT constraint over bit vectors can be integrated.

MATCHING SAT encoding. The next development has been done in SAT encoding called MATCHING that separates conflict rules in MAPF and agents transitions between time steps.
Conflict rules are expressed over anonymized agents that are encoded by direct variables $M_i \in \{True, False\}$.

The presence of some agent in vertex $v$ at timestep $t$ is indicated by a single propositional variable ($M_i^v = True$) if and only if $\exists a_i \in A$ such that $\sigma(a_i) = v$. Using anonymized agents is however not enough as agents may end up in other agent’s goal - see Figure 5. For transitions where individual agents need to be distinguished, log-space encoded variables $\mathcal{A}_i^t \in \{0, 1, 2, ..., k\}$ represent what agent occupies a given vertex ($\mathcal{A}_i^t$ if and only if $\sigma(a_i) = v$). The advantage of MATCHING over previous encodings INVERSE and ALL-DIFFERENT is that movement conflict rules can expressed in a simpler way over direct variables $M_i^v$ for anonymized agents. Compared to doing so over log-space encoded variables $\mathcal{A}_i^t$ or $\mathcal{D}_i^t$ that distinguish individual agents, smaller formula can be obtained with conflict reasoning over $M_i^v$.

**DIRECT SAT encoding.** Lessons taken from the previous development was that introduction of directly encoded variables leads to significant performance improvements although encoding set of states by direct variable is not as space efficient as the in the log-space case. The next encoding purely based on direct variables - called DIRECT MAPF encoding [52] - introduces a single propositional variable for every triple of agent, vertex, and time step; formally there was a propositional variable $X(a_i)^t_v$ such that it is $True$ if and only if agent $a_i$ occurs in $v$ at timestep $t$ (some triples may be forbidden as unreachable). In this work we are partly inspired by the DIRECT encoding as for the of direct variables.

**ASP, CSP, and ILP approach.** Although lot of work in makespan optimal solving has been done for SAT other compilation-based approaches to MAPF like ASP-based [12] and CSP-based [28] exist. Both ASP and CSP offer rich formalism to express various objective functions in MAPF. The ASP-based approach adopts a more specific definition of MAPF where bounds on

---

Algorithm 1: Framework of makespan optimal SAT-based MAPF solving

```plaintext
1 Solve-MAPF-SAT\_MAKESPAN(G = (V, E, A, α₀, α₊))
2 paths ← \{shortest path from α₀(a_i) to α₊(a_i) | i = 1, 2, ..., k\}
3 \(μ \leftarrow \max_i^{k}_{a_i} (\text{length(path}\_\text{a}(a_i)))\)
4 while True do
5 \(F(μ) \leftarrow \text{encode}(G, A, α₀, α₊)\)
6 assignment ← consult-SAT-Solver(\(F(μ)\))
7 if assignment ≠ UNSAT then
8    paths ← extract-Solution(assignment)
9    return paths
10 \(μ \leftarrow μ + 1\)
```

Figure 5: Searching of non-conflicting paths over anonymized agents - conflicts are reflected but an agent may end up in the wrong goal (lower right part).
lengths of paths for individual agents are specified as a part of the input. Except the bound on sum-of-costs the ASP formulation works with other constraints such as no-cycle (if the agent shall not visit the same part of the environment multiple times), no-intersection (if only one agent visits each part of the environment), or no-waiting (when minimization of idle time is desirable). The ASP program for a given variant of MAPF consisting of a combination of various constraints is solver by the CLASP ASP solver [14].

Ryan in the CSP-based approach focuses on the structure of the underlying graph $G$. The graph is partitioned into halls (singly-linked chain of vertices with any number of entrances and exits) and cliques (represents large open spaces with many entrances and exists) commonly referred to as sub-graphs. The plan is searched using CSP techniques over an abstract graph whose nodes are represented by sub-graphs. Specific properties of different sub-graphs are reflected in constraints - for example, agents in a clique sub-graph never exceed the capacity and the agents preserve their ordering in a hall sub-graphs. The resulting CSP is eventually solved using the GECODE solver [55].

The similarity of MAPF and multi-commodity flows is studied in [62] where each agent is regarded as a different commodity. Depths of the multi-commodity flow are associated with individual time steps of MAPF solution. Finding optimal solutions of MAPF with respect to various objective functions can be then modeled as finding optimal solution of Integer Linear Programming (ILP) problem [61].

3. The New SAT-based Solvers

SAT solvers [5] encompass Boolean variables and answer binary questions. The challenge is to apply SAT for MAPF where there is a cumulative cost function. This challenge is stronger for the sum-of-costs variant where each agent has its own cost. We first recall main ideas of SAT encodings for makespan. Then, we present our SAT encoding for sum-of-costs.

3.1. SAT Encoding for Optimal Makespan

A time expansion graph (denoted TEG) is a basic concept used in SAT solvers for makespan optimal MAPF solving [51]. We use it too in the sum-of-costs variant below. A TEG is a directed acyclic graph (DAG). First, the set of vertices of the underlying graph $G$ is duplicated for all time-steps from 0 up to the given makespan bound $\mu$. Then, possible actions (move along edges or wait) are represented as directed edges between successive time steps. Figure 6 shows a graph and its TEG for time steps 0, 1 and 2 (vertical layouts).

It is important to note that in this example (1) horizontal edges in TEG correspond to wait actions. (2) diagonal moves in TEG correspond to real moves. Formally a TEG is defined as follows:

**Definition 1.** Time expansion graph (TEG) of depth $\mu$ is a digraph $(V', E')$ derived from $G=(V,E)$ where $V' = \{u'_t \mid t = 0, 1, ..., \mu \land u_j \in V\}$ and $E' = \{(u'_t, u'_t) \mid t = 0, 1, ..., \mu - 1 \land (u_j, u_k) \in E \lor j = k\}$.

The encoding for MAPF introduces TEGs for individual agents. That is, we have $TEG_i = (V'_i, E'_i)$ for each agent $i \in \{1, 2, ..., k\}$. Directed non-conflicting paths in TEGs correspond to valid non-conflicting movements of agents in the underlying graph $G$. The existence of non-conflicting paths in TEGs will be encoded as satisfiability of a Boolean formula. We will describe in more details encoding style used in the DIRECT encoding.
Boolean variables and constraints (clauses) for a single time-step \( t \in \{0, 1, \ldots, \mu\} \) in \( T E G_i \) in order to represent any possible location of agent \( a_i \) at time \( t \); that is, we have \( X(a_i)^t \). Boolean variables for all TEGs together represent all possible arrangements of agents from timestep 0 up to timestep \( \mu \). It is ensured by constraints that arrangements of agents in consecutive time steps of TEGs correspond to valid actions: agent \( a_i \) can appear in vertex \( u^t_j \) if it can move there from the previous time step in \( T E G_i \) along a directed edge, that is, if \( a_i \) is in some \( u^{t-1}_k \) such that \((u^{t-1}_k, u^t_j) \in E_i \). We also have inter-TEG constrains ensuring that agents do not collide with each other (detailed list of constraints will be introduced for the sum-of-costs variant).

Given a desired makespan \( \mu \), formula \( F_\mu \) represents the question of whether there is a collection of non-conflicting directed paths in \( T E G_1, \ldots, T E G_k \) of depth \( \mu \) such that the first arrangement equals to \( \alpha_0 \) and the last one equals \( \alpha_\mu \). The search for optimal makespan is done by iteratively incrementing \( \mu = 0, 1, 2, \ldots \) until a satisfiable formula \( F_\mu \) is obtained as shown in Algorithm 1.

This process ensures finding makespan optimal solution in case of a solvable input MAPF instance since satisfiability of \( F_\mu \) is a non-decreasing function of \( \mu \). It is important to note that solvability of a given MAPF can be checked in advance by a fast polynomial algorithm like PUSH-AND-ROTATE [10]. More information on SAT encoding for the makespan variant can be found, e.g. in [51, 43, 52]. The detailed transformation of a question of whether there are non-conflicting paths in TEGs will shown in following sections.

4. Basic-SAT for Optimal Sum-of-costs

The general scheme described above for finding optimal makespan is to convert the optimization problem (finding minimal makespan) to a sequence of decision problems (is there a solution of a given makespan \( \mu \)). The decision problem was: is there a solution of makespan \( \mu \), and the sequence of decision problems was to increment \( \mu \) until the minimal makespan is found (this works due to monotonicity of existence of solution w.r.t. increasing makespan; wait action can prolong a solution arbitrarily). The questions are which decision problem to encode, how to encode it, and how to devise an appropriate sequence of these decision problems that will guarantee a solution to the optimization problem at hand.

In the makespan MAPF variant, the numeric objective function to minimize, i.e., the makespan \( \mu \) corresponds directly to the number of time expansions of the underlaying graph \( G \) in TEGs. Thus, the decision problem was: is there a solution in a TEGs of depth \( \mu \). This decision problem can be regarded as a question: are there non-conflicting directed paths in TEGs that interconnects agents’ starting positions and goals. The existence of such paths is then encoded into a Boolean formula.

We apply the same scheme for finding optimal sum-of-costs, converting it to a sequence of decision problems – is there a solution of a given sum-of-costs \( \xi \) where the decision problem...
is whether there is a solution of sum-of-costs $\xi$, and the sequence of decision problems is to increment $\xi$ until finding the minimal sum-of-costs is found. Again the solvability of a MAPF instance is monotonic w.r.t. increasing sum-of-costs hence the above incremental strategy works.

It is important to note that incremental strategy to obtain the optimal value of the objective function is suitable only when the cost of query is exponential in $\mu$ or $\xi$ (in case of uniform query cost different strategies like binary search would be more suitable). This roughly holds in the MAPF as increasing $\mu$ corresponds to adding a fresh time step in TEGs which is reflected in the encoded Boolean formula by adding a number of variables and constraints proportional to the size of $G$. Since the runtime of a SAT solver is exponential in the size of the input formula in the worst case we have that runtime for answering $F_\mu$ is exponential in $\mu$ in the worst case. In such a setup with incremental strategy, the cost/runtime of the last query is roughly the same as the total cost/runtime of previous queries. As we will see later, the same applies also for the sum-of-costs $\xi$.

However, encoding the decision problem for the sum-of-costs is more challenging than the makespan case, because one needs to both bound the sum-of-costs, but also to predict how many time expansions are needed. We address this challenge by using two key novel techniques described next: (1) Cardinality constraint for bounding $\xi$ and (2) Bounding the Makespan.

- **Cardinality constraints.** This is a technique from the SAT literature that enables counting and bounding a numeric cost in a Boolean formula \[3, 35, 37\]. This enables encoding a constraint that bounds the sum of cost inside Boolean formulae. Typically the encoding of cardinality constraints is based on simulation of arithmetic circuits for calculating summations. While most of logic circuits assume binary encoding of input values where weights of individual bits/propositional variables is determined by their position, here we work with unary encoding of inputs where each single propositional variable contributes by 1 to the overall sum (see Section 4.4 for details).

- **Upper bound on the required time expansions.** We show below how to compute for a given sum of cost value $\mu$ such that all possible solutions with sum-of-costs $\xi$ must be possible for a makespan of at most $\mu$ (details in Section 4.2). This enables encoding the decision problem of whether there is a solution of sum-of-costs $\xi$ by using a SAT encoding similar to the makespan encoding with $\mu$ time expansions. In other words, it will be sufficient to use TEGs of depth $\mu$ in order to represent all solutions that fits under the given sum-of-costs $\xi$.

Next, we explain each of these techniques in detail, along with theoretical analysis and additional implementation details.

### 4.1. Cardinality Constraint for Bounding $\xi$

The SAT literature offers a technique for encoding a cardinality constraint \[3, 35, 37\], which allows calculating and bounding a numeric cost within the Boolean formula. Formally, for a bound $\lambda \in \mathbb{N}$ and a set of Boolean variables $X = \{x_1, x_2, \ldots, x_k\}$ the cardinality constraint

\[ \leq \lambda \{x_1, x_2, \ldots, x_k\}\]

is satisfied iff the number of variables from the set $X$ that are set to TRUE is $\leq \lambda$. There are various ways how to encode cardinality constraints in Boolean formulae. The standard approach is to simulate arithmetic circuits \[3\] inside the formula. Arithmetic circuits for cardinality constraints usually assume unary encoding of inputs where each propositional variable (bit) from $X$ contributes by 1 to the sum.
In our SAT encoding, we bound the sum-of-costs by mapping every agent’s action to a Boolean variable, and then encoding a cardinality constraint on these variables. Thus, one can use the general structure of the makespan SAT encoding (which iterates over possible makespans), and add such a cardinality constraint on top. Next we address the challenge of how to connect these two factors together.

4.2. Bounding the Makespan for the Sum of Costs

Next, we compute how many time expansions \( \mu \) are needed to guarantee that if a solution with sum-of-costs \( \xi \) exists then it will be found within at most \( \mu \) time expansions. In other words, in our encoding, the values we give to \( \xi \) and \( \mu \) must fulfill the following requirement:

\[ \text{R1: All possible solutions with sum-of-costs } \xi \text{ must be possible for a makespan of at most } \mu. \]

To find a \( \mu \) value that meets R1 for given \( \xi \), we require the following definitions. Let \( \xi_0(a_i) \) be the cost of the shortest individual path for agent \( a_i \) \( (\xi_0(a_i) = \text{length(path}(a_i))) \), and let \( \xi_0 = \sum_{a_i \in A} \xi_0(a_i). \xi_0 \) was called the sum of individual costs (SIC)\(^{[32]} \). \( \xi_0 \) is an admissible heuristic for optimal sum-of-costs search algorithms, since \( \xi_0 \) is a lower bound on the minimal sum-of-costs. \( \xi_0 \) is calculated by relaxing the problem by omitting the other agents (collisions with them). Similarly, we define \( \mu_0 = \max_{a_i \in A} \xi_0(a_i). \) \( \mu_0 \) is length of the longest of the shortest individual paths and is thus a lower bound on the minimal makespan. Finally, let \( \Delta \) be the extra cost over SIC (as done in \([32]\)). That is, let \( \Delta = \xi - \xi_0. \)

**Proposition 1.** For makespan \( \mu \) of any solution with sum-of-costs \( \xi \), R1 holds for \( \mu \leq \mu_0 + \Delta. \)

**Proof:** The worst-case scenario, in terms of makespan, is that all the \( \Delta \) extra moves belong to a single agent. Given this scenario, in the worst case, \( \Delta \) is assigned to the agent with the largest shortest-path. Thus, the resulting path of that agent would be \( \mu_0 + \Delta \), as required. \( \square \)

Using Proposition 1, we can safely encode the decision problem of whether there is a solution with sum-of-costs \( \xi \) and makespan \( \mu \leq \mu_0 + \Delta \). In every iteration, \( \mu \) is set to \( \mu_0 + \Delta \) (Line 4) and the relevant TEGs of depth \( \mu \) (described below) for the various agents are built. Using TEGs of individual agents a formula \( F(\mu, \Delta) \) is constructed that encodes a decision problem whether there is a solution with sum-of-costs \( \xi \) and makespan \( \mu \). Afterwards the formula is queried to the SAT solver (Line 8). The first iteration starts with \( \Delta = 0. \) If such a solution exists, it is returned. Otherwise \( \xi \) is incremented by one, \( \Delta \) and consequently \( \mu \) are modified accordingly and another iteration of SAT consulting is activated.

**Proposition 2.** The algorithm SAT consult is sound and complete.

**Proof:** This algorithm clearly terminates; for unsolvable instances after the initial solvability test; for solvable MAPF instances as we start seeking a solution of \( \xi = \xi_0 (\Delta = 0) \) and increment \( \Delta \) (which increments \( \xi \) and \( \mu \) as well) to all possible values. Hence we eventually encounter \( \Delta (\xi \text{ and } \mu) \) for which \( \Sigma \) is solvable and valid solution is calculated and returned. \( \square \)

The initial unsolvability check of an MAPF instance can be done by any polynomial-time complete sub-optimal algorithm such as \textsc{Push-And-Rotate} \([9]\).
Algorithm 2: Basic SAT-based sum-of-costs optimal MAPF solving.

Solve-MAPF-SAT$_{SUM-OF-COSTS}$(MAPF $\Sigma = (G, A, \alpha_0, \alpha_+$))

1. if $\Sigma$ is unsolvable then
   return (Solution not found)
2. else
   $\mu_0 = \max_{a \in A} \xi_0(a_i)$
   $\Delta \leftarrow 0$
   while True do
     $\mu \leftarrow \mu_0 + \Delta$
     for each agent $a_i$ do
       $\text{T EG}_i(\mu) \leftarrow \text{Construct-TEG}(\mu, \xi_0(a_i), \alpha_0(a_i), \alpha_+(a_i), G)$
       $F(\mu, \Delta) \leftarrow \text{encode}(\mu, \Delta, \alpha_0, \alpha_+, \text{T EG}_1(\mu), \ldots, \text{T EG}_k(\mu))$
       assignment $\leftarrow \text{Consult-SAT-Solver}(F(\mu, \Delta))$
       if assignment $\neq UNSAT$ then
         paths $\leftarrow \text{extract-Solution}(assignment)$
         return paths
       else
         $\Delta \leftarrow \Delta + 1$
     end
   end

4.3. Efficient Use of the Cardinality Constraint

The complexity of encoding a cardinality constraint depends linearly in the number of con-
strained variables [35, 37]. Since each agent $a_i$ must move at least $\xi_0(a_i)$, we can reduce the
number of variables counted by the cardinality constraint by only counting the variables corre-
sponding to extra movements over the first $\xi_0(a_i)$ movement $a_i$ makes. We implement this by
introducing a TEG for a given agent $a_i$ (labeled $\text{T EG}_i$).

$\text{T EG}_i$ differs from TEG (Definition 1) in that it distinguishes between two types of edges:
$E_i$ and $F_i$. $E_i$ are (directed) edges whose destination is at time step $\leq \xi_0(a_i)$. These are called
standard edges. $F_i$ denoted as extra edges are directed edges whose destination is at time step
$> \xi_0(a_i)$. A formal construction of the time expansion graph is shown using pseudo-code as
Algorithm 3.

Figure 7 shows an underlying graph for agent $a_i$ (left) and the corresponding $\text{T EG}_i$. Note
that the optimal solution of cost 2 is denoted by the diagonal path of the TEG. Edges that belong
to $F_i$ are those that their destination is time step 3 (dotted lines), only these edges can contribute...
to the sum-of-costs above $\xi_0(a_i) = 2$. That is, we will only bound the number of extra edges (they sum up to $\Delta$) making the encoding of the cardinality constraint more efficient.

4.4. Detailed Description of the SAT Encoding

Agent $a_i$ must go from its initial position to its goal within $TEG_i$. This simulates its location in time in the underlying graph $G$. That is, the task is to find a path from $a_i^{0}(a_i)$ to $a_i^{\mu}(a_i)$ in $TEG_i$. The search for such a path will be encoded within the Boolean formula. Additional constraints will be added to capture all movement constraints such as collision avoidance etc. And, of course, we will encode the cardinality constraint that the number of extra edges must be exactly $\Delta$.

We want to ask whether a sum-of-costs solution of $\xi$ exist. For this we build $TEG_i$ for each agent $a_i \in A$ of depth $\mu_0 + \Delta$. We use $V_t$ to denote the set of vertices in $TEG_i$ that agent $a_i$ might occupy during the time steps. Next we introduce the basic Boolean encoding (denoted $BASIC-SAT$) which has the following Boolean variables.

1): $X_t(a_i)$ for every $t \in \{0, 1, ..., \mu\}$ and $u_t^i \in V_t$ — Boolean variable of whether agent $a_i$ is in vertex $u_t^i$ at time step $t$.

2): $E_t(a_i)$ for every $t \in \{0, 1, ..., \mu - 1\}$ and $(u_t^j, u_{t+1}^j) \in (E_t \cup F_t)$ — Boolean variables that model transition of agent $a_i$ from vertex $u_t^j$ to vertex $u_{t+1}^j$ through any edge (standard or extra) between time steps $t$ and $t+1$ respectively.

3): $C_t(a_i)$ for every $t \in \{0, 1, ..., \mu - 1\}$ such that there exist $u_t^j \in V_t$ and $u_{t+1}^j \in V_t$ with $(u_t^j, u_{t+1}^j) \in F_t$ — Boolean variables that model cost of movements along extra edges (from $F_t$) between time steps $t$ and $t+1$.

We now introduce constraints on these variables to restrict illegal values as defined by our variant of MAPF. Other variants may use a slightly different encoding but the principle is the
same. Let $T_\mu = \{0, 1, \ldots, \mu - 1\}$. Several groups of constraints are introduced for each agent $a_i \in A$ as follows:

**C1:** If an agent appears in a vertex at a given time step, then it must follow through exactly one adjacent edge into the next time step. This is encoded by the following pseudo-Boolean constraint \[5\], which is posted for every $t \in T_\mu$ and $u_j^t \in V_i$:

$$X_j^t(a_i) \Rightarrow \sum_{(u_j^t, u_j^{t+1}) \in E_i \cup F_i} E_j^{t+1}(a_i) = 1 \quad (1)$$

The above pseudo-Boolean can be translated to clauses in multiple ways. One simple and efficient way is to rewrite the constraint as follows:

$$X_j^t(a_i) \Rightarrow \bigvee_{(u_j^t, u_j^{t+1}) \in E_i \cup F_i} E_j^{t+1}(a_i), \quad (2)$$

$$\bigwedge_{(u_j^t, u_j^{t+1}) \in E_i \cup F_i \land l > h} \neg E_j^t(a_i) \vee \neg E_j^{t+1}(a_i) \quad (3)$$

**C2:** Whenever an agent occupies an edge it must also enter it before and leave it at the next time-step. This is ensured by the following constraint introduced for every $t \in T_\mu$ and $(u_j^t, u_j^{t+1}) \in E_i \cup F_i$:

$$E_j^t(a_i) \Rightarrow X_j^t(a_i) \land X_j^{t+1}(a_i) \quad (4)$$

**C3:** The target vertex of any movement except wait action must be empty. This is ensured by the following constraint introduced for every $t \in T_\mu$ and $(u_j^t, u_j^{t+1}) \in E_i \cup F_i$ such that $j \neq l$.

$$E_j^t(a_i) \Rightarrow \bigwedge_{a_h \in A \land a_h \neq a_i \land u_j^{t+1} \in V_h} \neg X_j^t(a_h) \quad (5)$$

**C4:** No two agents can appear in the same vertex at the same time step. Again this can be expressed by the following pseudo-Boolean constraint for every vertex $u_j \in V$ and $t \in T_\mu$:

$$\sum_{a_i \in A \land u_j \in V_i} X_j^t(a_i) \leq 1 \quad (6)$$

Equivalently this can be expressed by following binary clauses for every pair of of agents $a_i, a_l \in A$ such that $i \neq l$:

$$\bigwedge_{u_j \in V \cap V_h} \neg X_j^t(a_i) \lor \neg X_j^t(a_h) \quad (7)$$

**C5:** Whenever an extra edge is traversed the cost needs to be accumulated. In fact, this is the only cost that we accumulate as discussed above. This is done by the following constraint for every $t \in T_\mu$ and extra edge $(u_j^t, u_j^{t+1}) \in F_i$:

$$E_j^t(a_i) \Rightarrow c_j^t(a_i) \quad (8)$$
C6: The cost of wait action followed by a non-wait action need to be accumulated. This is ensured by the following constraint for every $t \in T_\mu$:

$$C'(a_i) \Rightarrow \bigwedge_{\sigma \in [0,1, \ldots, t-1]} \bigwedge_{((u_r^\sigma, u_r^{\sigma+1}) \in F_i) \neq \emptyset} C''(a_i)$$  \hspace{1cm} (9)

C7: Cardinality constraint. Finally, the bound on the total cost needs to be introduced. Reaching the sum-of-costs of $\xi$ corresponds to traversing exactly $\Delta$ extra edges from $F_i$. The following cardinality constraint ensures this:

$$\leq \Delta \left\{ C'(a_i) \mid i = 1, 2, \ldots, n \land t \in T_\mu \land \{(u_r^\mu, u_r^{\mu+1}) \in F_i\} \neq \emptyset \right\}$$  \hspace{1cm} (10)

Final formula. The resulting Boolean formula that is a conjunction of C1 \ldots C7 will be denoted as $F_{\text{BASIC}}(\mu, \Delta)$ and is the one that is consulted by Algorithm 2 (lines 11-12).

The following proposition summarizes the correctness of our encoding.

Proposition 3. MAPF $\Sigma = (G = (V,E), A, a_0, a_\ast)$ has a sum-of-costs solution of $\xi$ if and only if $F_{\text{BASIC}}(\mu, \Delta)$ is satisfiable. Moreover, a solution of MAPF $\Sigma$ with the sum-of-costs of $\xi$ can be extracted from the satisfying valuation of $F_{\text{BASIC}}(\mu, \Delta)$ by reading its $X'(a_i)$ variables.

Proof: The direct consequence of the above definitions is that a valid solution of a given MAPF $\Sigma$ corresponds to non-conflicting paths in the TEGs of the individual agents. These non-conflicting paths further correspond to the satisfying variable assignment of $F_{\text{BASIC}}(\mu, \Delta)$, i.e., that there are $\Delta$ extra edges in TEGs of depth $\mu = \mu_0 + \Delta$. □

Proposition 4. Let $D$ be the maximal degree of any vertex in $G = (V,E)$ and let $k$ be the number of agents. If $|E| \geq \mu$ and $k \geq D$ then the number of clauses in $F_{\text{BASIC}}(\mu, \Delta)$ is $O(k^2 \cdot |\cdot |E|)$, and the number of variables is $O(k \cdot |\cdot |E|)$.

Proof: The components of $F_{\text{BASIC}}(\mu, \Delta)$ are described in equations 2-10. Equations 2 and 3 introduce at most $O(k \cdot |\cdot |V| \cdot D^2)$ clauses. Equation 4 introduces at most $O(k \cdot |\cdot |E|)$ clauses. Equation 5 introduces at most $O(k^2 \cdot |\cdot |E|)$ clauses. Equation 7 introduces at most $O(k^2 \cdot |V|)$ clauses. Equation 8 introduces at most $O(k \cdot |\cdot |E|)$ clauses. Equation 9 introduces at most $O(k \cdot \mu^2)$ clauses. And finally equation 10 introduces at most $O(k \cdot \mu \cdot \Delta)$ clauses, since a cardinality constraint checking that $n$ variables has a cardinality constraint of $k$ requires $O(n \cdot m)$ clauses [27].

Summing all the above results in a total of $O(k \cdot |\cdot |(|V| \cdot D^2 + k \cdot |E| + |\cdot | + \Delta))$. If we assume that $k \geq D, |E| \geq \mu$, and that $k \cdot |E| \geq \mu$ (by definition $\mu \geq \Delta$) then the number of clauses is $O(k^2 \cdot |\cdot |E|)$. The number of variables is easily computed in a similar way. □

Various optimizations of the encoding could be done at the level of using alternative encodings of the cardinality constraints and/or by eliminating edge variables $E'_{ij}(a_i)$ via equivalence $E'_{ij}(a_i) \Leftrightarrow X'(a_i) \land X''(a_i)$. We observed that these optimizations represent minor changes in the overall size and efficiency of the encoding.
Figure 8: Results on $8 \times 8$ grid (left). Number of solved instances in the given runtime on $16 \times 16$ and $32 \times 32$ grids. (right)

5. Experimental Evaluation

We experimented on 4-connected grids with randomly placed obstacles [36, 39] and on Dragon Age maps [31, 40]. Both settings are a standard MAPF benchmarks. The initial position of the agents was randomly selected. To ensure solvability the goal positions were selected by performing a long random walk from the initial arrangement.

We compared our SAT solvers to several state-of-the-art search-based algorithms: the increasing cost tree search - ICTS [32], Enhanced Partial Expansion A* - EPEA* [15] and improved conflict-based search - ICBS [7]. For all the search algorithms we used the best known setup of their parameters and enhancements suitable for solving the given instances over 4-connected grids.

The SAT approaches were implemented in C++. The implementation consists of a top level algorithm for finding the optimal sum-of-costs $\xi$ and CNF formula generator [5] that prepares input formula for a SAT solver into a file. The SAT solver is an external module our this architecture. We used Glucose 3.0 [2, 1] which is a top performing SAT solver in the SAT Competition [17, 51].

The cardinality constraint was encoded using a simple standard circuit based encoding called sequential counter [37]. In our initial testing we considered various encodings of the cardinality constrain such as those discussed in [3, 35]. However, it turned out that changing the encoding
ICTS and ICBS were implemented in C#, based on their original implementation (here we used a slight modification in which the target vertex of a move must be empty). All experiments were performed on a Xeon 2Ghz, and on Phenom II 3.6Ghz, both with 12 Gb of memory.

5.1. Square Grid Experiments

We first experimented on $8 \times 8$, $16 \times 16$, and $32 \times 32$ grids with $10\%$ obstacles while varying the number of agents from 1 up to the number where at least one solver was able to solve an instance (in case of the $8 \times 8$ grid this is 20 agents; and 32 and 58 in case of $16 \times 16$ and $32 \times 32$ grids respectively). For each number of agents 10 random instances were generated.

Figure 8 presents results where each algorithm was given a time limit of 300 seconds (as was done by [32, 7, 30]). The leftmost plot (Plot (a)) shows the success rate (=percentage out of given 10 random instances solved within the time limit) as a function of the number of agents for the $8 \times 8$ grid (higher curves are better). The next plot (Plot (b)) reports the average runtime for instances that were solved by all algorithms (lower curves are better). Here, we required 100% success rate for all the tested algorithms to be able to calculate average runtime; this is also the reason why the number of agents is smaller. The two right plots visualize the results on $16 \times 16$ grid (Plot (c)) and $32 \times 32$ grid (Plot (d)) but in a different way. Here, we present the number of instances (out of all instances for all number of agents) that each method solved (=y-axis) as a function of the elapsed time (=x-axis). Thus, for example Plot (c) says that MDD-SAT was able to solve 145 instances in time less than 10 seconds (higher curves are better).

The first clear trend is that MDD-SAT significantly outperforms BASIC-SAT in all aspects. This shows the importance of developing enhanced SAT encodings for the MAPF problem. The performance of the BASIC-SAT encoding compared to the search-based algorithm degrades as the size of the grids grow larger: in the $8 \times 8$ grids it is second only to MDD-SAT, in the $16 \times 16$ grid it is comparable to most search-based algorithms, and in the $32 \times 32$ grid it is even substantially worse. For the rest of the experiments we did not activate BASIC-SAT.

In addition, a prominent trend observed in all the plots is that MDD-SAT has higher success rate and solves more instances than all other algorithms. In particular, in highly constrained instances (containing many agents) the MDD-SAT solver is the best option.

However, on the $32 \times 32$ grid (rightmost figure) for easy instances when the available runtime was less than 10 seconds, MDD-SAT was weaker than the search-based algorithms. This is mostly due to the architecture of the MDD-SAT solver which has an overhead of running the external SAT solver and passing input in the textual form to it. This effect is also seen in the $8 \times 8$ plot (Plot (b)) as these were rather easy instances (solved by all algorithms) and the extra overhead of activating the external SAT solver did not pay off.

Next, we varied the number of obstacles for the $8 \times 8$ grid with 10 agents to see the impact of shrinking free space and increasing the frequency of interactions among agents. Results are shown in Figure 9. Again, MDD-SAT clearly solves more instances over all settings. MDD-SAT was always faster except for some easy instances (that needed up to 1 second) where ICBS was slightly faster which is again due to the overhead in setup of the SAT solving by an external solver. Interestingly, increasing the number of obstacles reduces the number of open cells. This is an advantage for the SAT formula generator in MDD-SAT as the formula has less variables.

\footnote{Due to the knowledge of lower bounds on the sum-of-costs, the number of variables involved in the cardinality constraint is relatively small and hence the different encoding style has not enough room to show its benefit.}
Figure 9: Success rate and runtime on the 8 × 8 grid with increasing number of obstacles (out of 64 cells).

Figure 10: Three structurally diverse Dragon-Age maps used in the experimental evaluation. This selection includes: narrow corridors in brc202d, large open space in den520d, and open space with almost isolated rooms in ost003d.

and constraints. By contrast, the combinatorial difficulty of the instances increases with adding obstacles for all the solvers as it means that the graphs get denser and harder to solve.

5.2. Results on the Dragon Age Maps

Next, we experimented on three structurally different Dragon-Age maps - ost003d, den520d, and brc202d, that are commonly used as testbeds [32, 15, 7] - see Figure 10. On these maps we only evaluated the most efficient algorithms, namely, MDD-SAT, ICTS, and ICBS. Generally, in these maps there is a large number of open cells but the graph is sparse with agents but there are topological differences. brc202d has many narrow corridors. ost003d consists of few open areas interconnected by narrow doors. Finally, den520d has wider open areas.

To obtain instances of various difficulties we varied the distance between start and goal locations. Ten random instances were generated for each distance in the range: {8, 16, 24, . . . , 320} in order to have instances of different difficulties (total of 400 instances). With larger distances, the problems are more difficult as the probability for interactions (avoidance) among agents increases as they need to travel through a larger part of the graph.

The results for the three Dragon-Age maps are shown in Figure 11 (brc202d), Figure 12 (den520d), and Figure 13 (ost003d). Two setups were used for each map - one with 16 agents, the other with 32 agents. The left plot of each figure shows the number of solved instances (y-axis) as a function of the elapsed time (x-axis). Again, higher curves correspond to better
Figure 11: Results for dragon age map brc202d with 16 and 32 agents. The left part shows the number of instances (y-axis) a solver manages to solve in the given timeout (x-axis). The right part shows all the runtimes for a given solver sorted in the ascending order.

Performance). The right plot is interpreted as follows. For each solver the 400 instances are ordered in increasing order of their solution time (this has strong correlation with the distance between the start and goal configurations). Thus, the numbers in the x-axis give the relative location (out of the 400) in this sorted order. The y-axis gives the actual running time for each instance. Here, lower curves correspond to better performance.

All these figures show a similar clear trend with the exception of ost003d with 32 agents (discussed below). On the easy instances where little time is required (left of the figures), MDD-SAT is not the best. But, for the harder instances that need more time (right of the figures), MDD-SAT clearly outperform all the other solvers.

Intuitively, one might think that the search-based solvers will have an advantage in these domains since they contain many open spaces (low combinatorial difficulty) while the MDD-SAT approach will suffer here as it will need to generate a large number of formulae (as the domains are large). This might be true for the easy instances. Nevertheless, the effectiveness of MDD-SAT was clearly seen on the harder instances where generating the formulae and the external time to activate the architecture of the SAT solver seemed to pay off. This trend was also seen in in the case of small densely occupied grids discussed above.

The ost003d map with 32 agents is the only case where MDD-SAT was outperformed by ICTS. This is probably due to the specific structure of ost003d which has a number of isolated
Figure 12: Results for dragon age map den520d with 16 and 32 agents. MDD-SAT is the best option on hard instances with more agents.

open spaces. This gives an advantage to ICTS with relatively many agents (32) as conflicts mostly occur at the exits/doors of the open areas. ICTS handles this on a per-agent cost basis while the other solvers are less effective here.

The entire set of experiments show a clear trend. For the easy instances when a small amount of time is given the search-based algorithm may be faster. But, given enough time MDD-SAT is the correct choice, even in the large maps where it has an initial disadvantage. One of the reasons for this is modern SAT solvers have the ability to learn and improve their speed during the process of answering a SAT question. But, this learning needs sufficient time and large search trees to be effective. By contrast, search algorithms do not have this advantage.

5.3. Size of the Formulae

Concrete runtimes for 10 instances of ost003d are given in Table 1. MDD-SAT solves the hardest instance (#1) while other solvers ran out of time. The right part of the table illustrates the cumulative size of the formulae generated during the solving process. Although the map is much larger than the square grids, the size of formulae is comparable to the densely occupied grid. This is because $\xi_0$ is a good lower bound of the optimal cost in the sparse maps.

The observation from this experiments is that the large underlaying graph does not necessarily imply generating of large Boolean formulae in the MDD-SAT solving process. Though in harder scenarios (where start and goals are far apart) large formulae are eventually generated
| Solved instances | Sorted Runtimes |
|------------------|-----------------|
| Ost003d|16 agents | Ost003d|16 agents |
| Number of instances | Runtime (seconds) |
| 0| 1| 10| 100 |
| 0| 0| 0.0001| 0.001| 0.01| 0.1| 1| 10| 100| 100 |
| Solved instances | Sorted Runtimes |
|------------------|-----------------|
| Ost003d|32 agents | Ost003d|32 agents |
| Number of instances | Runtime (seconds) |
| 0| 1| 10| 100 |
| 0| 0| 0.0001| 0.001| 0.01| 0.1| 1| 10| 100| 100 |

Figure 13: Results for dragon age map ost003d with 16 and 32 agents. Although MDD-SAT performs as best with 16 agents, it gets outperformed in the case with 32 agents by ICTS. This case shows that there is no universal winner among the tested algorithms.

but still do not represent any significant disadvantage for the MDD-SAT solver according to presented measurements. We observed that generating large formulae takes considerable portion of the total runtime (up to 10%-30%) within the MDD-SAT solver. Hence efficient implementation of this part of the solver has significant impact on the overall performance.

6. Summary and Conclusion

We summarized how to migrate techniques from search-based optimal MAPF solvers to SAT-based method. The outcome is the first SAT-based solver for the sum-of-costs variant of MAPF. The new solver was experimentally compared to the state-of-the-art search-based solvers over a variety of domains - we tested 4-connected grids with random obstacles and large maps from computer games. We have seen that the SAT-based solver is a better option in hard scenarios while the search-based solvers may perform better in easier cases.

Nevertheless, as previous authors mentioned [31, 7] there is no universal winner and each of the approaches has pros and cons and thus might work best in different circumstances. For example, ICTS was best on ost003d with 32 agents. This calls for a deeper study of various classes of MAPF instances and their characteristics and how the different algorithms behave across them. Not too much is known at present to the MAPF community on these aspects.
Table 1: Runtime for 10 instances (left) and the average size of the MDD-SAT formulae for ost003d (right).

There are several factors behind the performance of the SAT-based approach: clause learning, constraint propagation, good implementation of the SAT solver. On the other hand, the SAT solver does not understand the structure of the encoded problem which may downgrade the performance. Hence, we consider that implementing techniques such as learning directly into the dedicated MAPF solver may be a future direction. Finally, migrating of other ideas from both classes of approaches might further improve the performance.

Another interesting future direction is to consider how additional techniques from search-based MAPF solvers can be used to further improve our SAT-based search, e.g., incorporating our SAT-based solver in the meta-agent conflict-based search framework as a low-level solver [29]. This includes techniques for decomposing the problem to subproblems [39] and incorporating our SAT-based solver in the meta-agent conflict-based search framework as a low-level solver [29].

7. Acknowledgments

This research has been supported by the Czech Science Foundation (grant application number 19-17966S).

8. References

[1] G. Audemard, J. Lagniez, and L. Simon. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In Theory and Applications of Satisfiability Testing - SAT 2013, pages 309–317, 2013.
[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In IJCAI, pages 399–404, 2009.
[3] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In CP, pages 108–122, 2003.
[4] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014), pages 961–962, 2014.
[5] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.
[36] D. Silver. Cooperative pathfinding. In AIIDE, pages 117–122, 2005.
[37] C. Szin. Towards an optimal CNF encoding of boolean cardinality constraints. In CP, pages 827–831, 2005.
[38] A. Srinivasan, T. Ham, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In ICCAD, pages 92–95, 1990.
[39] T. Standley. Finding optimal solutions to cooperative pathfinding problems. In AAAI, pages 173–178, 2010.
[40] Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. Computational Intelligence and AI in Games, 4(2):144–148, 2012.
[41] P. Surynek. An optimization variant of multi-robot path planning is intractable. In AAAI, 2010.
[42] P. Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. In PRICAI, pages 564–576, 2012.
[43] P. Surynek. A simple approach to solving cooperative path-finding as propositional satisfiability works well. In PRICAI, pages 827–833, 2014.
[44] P. Surynek. Reduced time-expansion graphs and goal decomposition for solving cooperative path finding sub-optimally. In IJCAI, pages 1916–1922, 2015.
[45] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In 2009 IEEE International Conference on Robotics and Automation, ICRA 2009, Kobe, Japan, May 12-17, 2009, pages 3613–3619, 2009.
[46] Pavel Surynek. An alternative eager encoding of the all-different constraint over bit-vectors. In ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31, 2012, pages 927–928, 2012.
[47] Pavel Surynek. On propositional encodings of cooperative path-finding. In IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012, pages 524–531, 2012.
[48] Pavel Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. In PRICAI 2012: Trends in Artificial Intelligence - 12th Pacific Rim International Conference on Artificial Intelligence, Kuching, Malaysia, September 3-7, 2012. Proceedings, pages 564–576, 2012.
[49] Pavel Surynek. Mutex reasoning in cooperative path finding modeled as propositional satisfiability. In 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013, pages 4326–4331, 2013.
[50] Pavel Surynek. Optimal cooperative path-finding with generalized goals in difficult cases. In Proceedings of the Tenth Symposium on Abstraction, Reformulation, and Approximation, SARA 2013, 11-12 July 2013, Leavenworth, Washington, USA., 2013.
[51] Pavel Surynek. Compact representations of cooperative path-finding as SAT based on matchings in bipartite graphs. In 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014, pages 875–882, 2014.
[52] Pavel Surynek. Simple direct propositional encoding of cooperative path finding simplified yet more. In Nature-Inspired Computation and Machine Learning - 13th Mexican International Conference on Artificial Intelligence, MICAI 2014, Tuxtla Gutiérrez, Mexico, November 16-22, 2014. Proceedings, Part II, pages 410–425, 2014.
[53] Pavel Surynek. Solving abstract cooperative path-finding in densely populated environments. Computational Intelligence, 30(2):402–450, 2014.
[54] Pavel Surynek. On the complexity of optimal parallel cooperative path-finding. Fundam. Inform., 137(4):517–548, 2015.
[55] Guido Tack. Constraint Propagation - Models, Techniques, Implementation. phdthesis, Saarland University, Germany, 2009.
[56] G. Wagner and H. Choset. Subdimensional expansion for multirobot path planning. Artif. Intell., 219:1–24, 2015.
[57] Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In Intelligent Robots and Systems (IROS), pages 3260–3267, 2011.
[58] K. Wang and A. Botea. MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. JAIR, 42:55–90, 2011.
[59] J. Yu and S. LaValle. Planning optimal paths for multiple robots on graphs. In ICRA, pages 3612–3617, 2013.
[60] Jingjin Yu. Intractability of optimal multirobot path planning on planar graphs. IEEE Robotics and Automation Letters, 1(1):33–40, 2016.
[61] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. In 2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013, pages 3612–3617, 2013.
[62] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA., 2013.