# Search-based Optimal Solvers for the Multi-agent Pathfinding Problem: Summary and Challenges

**Ariel Felner**
**Roni Stern**
**Solomon Eyal Shimony**
Ben-Gurion University
Israel

**Eli Boyarski**
Bar Ilan University
Israel

**Guni Sharon**
Univ. of Texas at Austin
USA

**Glenn Wagner**
Carnegie Mellon University
USA

**Meir Goldenberg**
Jerusalem College of Technology
Israel

**Nathan Sturtevant**
University of Denver
USA

**Pavel Surynek**
(AIST), Tokyo
Japan

**Abstract**

Multi-agent pathfinding (MAPF) is an area of expanding research interest. At the core of this research area, numerous diverse search-based techniques were developed in the past 6 years for optimally solving MAPF under the sum-of-costs objective function. In this paper we survey these techniques, while placing them into the wider context of the MAPF field of research. Finally, we provide analytical and experimental comparisons that show that no algorithm dominates all others in all circumstances. We conclude by listing important future research directions.

## 1 Introduction and Overview

The *multi-agent path finding* (MAPF) problem is specified by a graph, $G = (V, E)$ and a set of $k$ agents $a_1 \dots a_k$, where each agent $a_i$ has a start position $s_i \in V$ and a goal position $g_i \in V$. Time is discretized into time steps and agent $a_i$ is in $s_i$ at time $t_0$. Between successive time steps an agent can either *move* to an adjacent empty location or *wait* in its current location. Both *move* and *wait* cost 1. A sequence of individual agent move/wait actions leading an agent from $s_i$ to $g_i$ is referred to as a *path*. The term *solution* refers to a set of $k$ paths, one for each agent. A *conflict* between two paths is a tuple $\langle a_i, a_j, v, t \rangle$ where agent $a_i$ and agent $a_j$ are planned to occupy vertex $v$ at time $t$. The task is to find a *conflict-free solution*, also called a *valid solution*.[1]

MAPF can model many real-world problems in games (Silver 2005), traffic control (Dresner and Stone 2008), robotics (Erdem et al. 2013; Yu and LaValle 2013a), aviation (Pallottino et al. 2007), robot warehouses (Wurman et al. 2008) and more. See (Yu and LaValle 2016) for a comprehensive survey.

MAPF can be categorized into two settings. In a *distributed setting* (Grady, Bekris, and Kavraki 2011; Bhattacharyat and Kumar 2011), each agent has its own computing power and its own decision making protocol which might be *self-interested* or *cooperative* (Bnaya et al. 2013). Different communication paradigms may be assumed (e.g., message passing, broadcasting etc.). The scope of this paper is limited to the *centralized setting*, where we assume that the agents are fully controlled by a single decision-maker.

In MAPF one usually aims at minimizing a global cumulative cost function. One common cost function is *sum-of-costs:* the sum, over all agents, of the number of time steps required to reach their goal, i.e., the sum of the individual path costs (Standley 2010; Standley and Korf 2011; Sharon et al. 2013; 2015). Another common MAPF cost function is *makespan* (Surynek 2010) which is the time until all agents have reached their destinations, i.e., the maximum of the individual path costs.

MAPF was shown to be NP-hard (Yu and LaValle 2013b; Surynek 2010); as the state-space grows exponentially with $k$ (# of agents). As a result, optimal solvers are worthy when the number of agents is relatively small. Nevertheless, optimal solvers can be readily modified to sacrifice optimality in return for a faster runtime. In addition, research on optimal solutions sheds light on the theoretical hardness of MAPF variants and on their attributes and characteristics.

This paper is not intended to be a thorough and balanced survey on the entire MAPF area. Instead, this paper focuses on and summarizes a line of work on *optimal MAPF solvers* and in particular on *search-based solvers* which were usually designed for the sum-of-costs variant. Nevertheless, for completeness, we will discuss the relationship between the two cost functions in different contexts and place these algorithms in the proper context of the entire area.

Section 2 examines some suboptimal solvers. In Sections 3–7 we review various existing optimal algorithms for MAPF with a focus on search-based algorithms. We then provide representative theoretical and experimental results (Sections 8–9) that show that there is no universal winner; each algorithm may be the best algorithm in different circumstances. In Section 10 we briefly discuss new research directions that solve variants of MAPF that model complex real-world scenarios. Finally, we discuss open research questions and describe the need to better understand and classify the different existing algorithms.

**Example:** Figure 1(I) shows an example 3-agent MAPF in-



[1] There are a number of ways to deal with the possibility of a chain of agents that *follow* each other. This may not be allowed, may only be allowed if the first agent of the chain moves to an unoccupied location or it may be allowed even in a cyclic chain which does not include any empty location. A special case of this is a cycle of two agents which might or might not be allowed to cross the same edge in opposite directions. Most algorithms described in this paper are applicable across all these variations.

stance that will be used throughout of the paper. Each agent (mouse) must plan a full path to its respective goal (piece of cheese). All three agents have individual paths of length 3: $\{S_1, A_1, D, G_1\}$, $\{S_2, B_1, D, G_2\}$ and $\{S_3, E, F, G_3\}$ respectively. However, the paths of agents $a_1$ and $a_2$ have a conflict, as they both include state $D$ at time point $t_2$. One of these agents must wait one time step. Therefore, the optimal solution cost, $C^*$, is 10 in this example. Throughout this paper we refer to this example with all 3 agents as EX3, and refer to the same example with $a_1$ and $a_2$ only as EX2.

## 2 Suboptimal MAPF Solvers

Since MAPF is NP-hard (Yu and LaValle 2013b; Surynek 2010) suboptimal solvers are needed, especially when $k$ is large. Most suboptimal solvers aim at quickly finding paths for all agents. These solvers can be roughly classified into *search-based* and *rule-based*.

### 2.1 Search-based Solvers

A prominent example of a search-based sub-optimal algorithm is *Hierarchical Cooperative A** (HCA*) (Silver 2005). In HCA* the agents are planned one at a time according to some predefined order. Once a path to the goal is found for the first agent, that path (i.e., times and locations) is written (*reserved*) into a global *reservation table*. Any agent searching for a path may not occupy specific locations at specific times reserved by previous agents. Many enhancements to HCA* were introduced. Windowed-HCA* (WHCA*) (Silver 2005) only applies the reservation table within a limited time window, after which other agents are ignored. Finally, Bnaya and Felner (2014) dynamically placed the windows only around known conflicts and agents are prioritized according to an estimation of the likelihood of being involved in a conflict. A heuristic that computes the shortest single-agent paths is most often used to guide this search. However, Sturtevant and Buro (2006) abstracted the state-space to reduce the runtime cost of building the heuristics. Likewise, Ryan (2008) used abstraction to reduce the size of the state-space in order to to quickly compute an abstract plan from which a full plan is then derived. Search-based algorithms usually do not run extremely fast but the solutions they return are of relatively high quality (typically near-optimal). It is important to note that some of these algorithms (e.g. HCA*) do not guarantee completeness.

Several MAPF search-based algorithms are *bounded suboptimal solvers*. Given a bound $B$, they guarantee that the solution returned is at most $B \times C^*$, where $C^*$ is the cost of the optimal solution. Many of these algorithms are based on modifications or relaxations of optimal MAPF algorithms (Barer et al. 2014; Cohen et al. 2015; 2016).

### 2.2 Rule-based Solvers

Another class of suboptimal algorithms are *rule-based algorithms*. These algorithms include specific agent-movement rules for different scenarios and usually do not include massive search. Rule-based solvers usually guarantee finding a solution relatively fast, but the solutions they return are often far from optimal. They usually require special properties of the underlying graph in order to work or to guarantee completeness. The algorithm by Kornhauser, Miller, and Spirakis (1984) is complete under all circumstances but it is complex to implement. TASS (Khorshid, Holte, and Sturtevant 2011) is complete only for tree graphs. Push-and-Swap (Luna and Bekris 2011) and its enhanced versions Parallel Push-and-Swap (Sajid, Luna, and Bekris 2012) and Push-and-Rotate (de Wilde, ter Mors, and Witteveen 2014), are desgined to work under the condition that there are at least two unoccupied vertices in the graph. They introduce macros to reverse the location of two adjacent agents by using the two unoccupied locations. Push-and-Rotate (de Wilde, ter Mors, and Witteveen 2014) was proved to be complete on such graphs.

BIBOX (Surynek 2009a) is complete for bi-connected graphs and has a version called diBOX that is complete even for strongly bi-connected digraphs (Botea and Surynek 2015). Similar to Push-and-Rotate it needs at least two unoccupied vertices. BIBOX decomposes the vertices of the bi-connected graph into a main cycle and to a sequence of *ears* (paths of fresh vertices). It establishes macros that use these structures to solve the problem. BIBOX-$\theta$ (Surynek 2009b) is an enhancement which can work with one unoccupied vertex but it uses precomputed solutions for special rotations of small groups of agents.

### 2.3 Hybrid Approaches

Some algorithms are hybrids: they include both movement rules and massive search (Wang and Botea 2008; Jansen and Sturtevant 2008). They established flow restrictions similar to traffic laws that can simplify the problem (Jansen and Sturtevant 2008). Finally, similar laws can be applied very efficiently if the graph is a grid with a specific *slidable* property (Wang and Botea 2011).

## 3 Reduction-based Optimal Solvers

Several recently introduced optimal solvers reduce MAPF to standard known problems (SAT, CSP, etc.) that, while NP-hard, have existing high-quality solvers. Many of these solvers are designed for the makespan variant of MAPF; they are not easily modified to the sum-of-costs variant and sometimes a completely new reduction would be needed.

Yu and LaValle (2013a) modeled MAPF as a network flow problem where depths of the flow are associated with the different time steps. Then, by using Integer Linear Programming (ILP) they provided a set of equations and an objective function which yield the optimal solution. Erdem et al. (2013) used the declarative programming paradigm of Answer Set Programming (ASP) for optimally solving MAPF. They represent the path finding problem for each agent and the inter-agent constraints as a program $P$ in ASP. The answer sets of $P$ correspond to solutions of the problem. Finally, an effective way to solve MAPF is to reduce it to SAT (Surynek 2012). The structure of the graph, the locations of the agents and the constrains of the problem are all encoded into boolean variables. Then, a SAT formula is generated from these variables that answers the question of whether there is a valid solution with cost $C$. Search

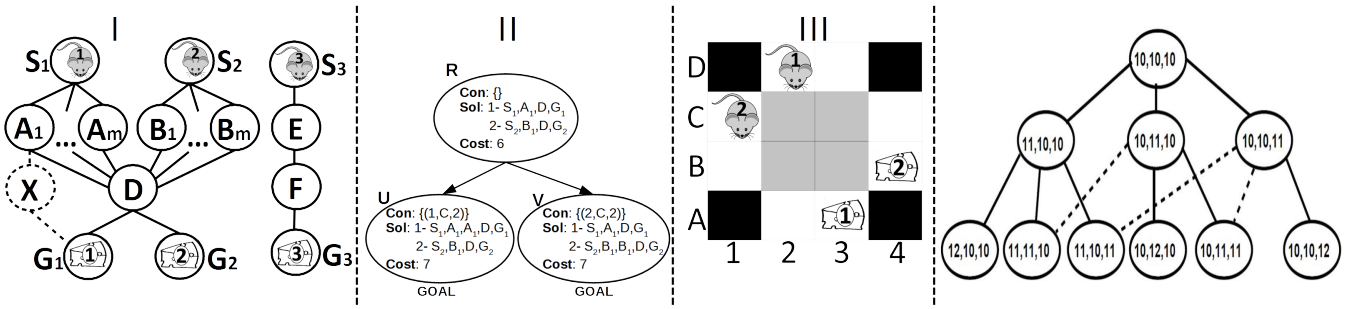



Figure 1: (I) A 3-agent MAPF instance. (II) The respective CT for CBS. (III) 2-agent MAPF. (IV) ICT for 3 agents.

over possible costs $C$ yields the optimal solution. Another reduction-based method transforms MAPF into CSP (Ryan 2010). However, this reduction is based on an initial abstraction of the graph into subgraphs of known shapes such as halls, rings and cliques. Thus, the resulting solution is not guaranteed to be optimal.

Finally, a number of search-based optimal MAPF solvers were recently introduced. Most of them are designed for the *sum-of-costs* variant of MAPF and they are the main focus of our paper. Most of these algorithms can be easily modified to the makespan variant by slightly modifying the definition of the underlying search space. We now turn to cover these search-based solvers and begin with A*-based solvers.

## 4 A*-based Optimal Solvers

A* is a general-purpose algorithm that is well suited to solve MAPF. A common straight-forward derived state-space is denoted as the *k-agent state-space* where the states are the different ways to place $k$ agents into $|V|$ vertices, one agent per vertex. 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.

Let $b_{base}$ denote the branching factor of a single agent. For example, in a 4-connected grid $b_{base} = 5$; an agent can either move in four cardinal directions or wait at its current location. The *effective branching factor* for $k$ agents, denoted by $b$, is roughly ${b_{base}}^k$; though usually a bit smaller than ${b_{base}}^k$ because many possible combinations of moves result in immediate conflicts, especially when the environment is dense with agents.

**Admissible Heuristics for MAPF.** A simple admissible heuristic is to sum the individual heuristics of the single agents such as Manhattan distance for 4-connected grids or Euclidean distance for Euclidean graphs (Ryan 2008). A more-informed heuristic is called the *sum of individual costs* heuristic ($h_{SIC}$). For each agent $a_i$ we calculate its optimal path cost from its current state to $goal_i$ assuming that other agents do not exist. Then, we sum these costs over all agents.[2] For EX3 $h_{SIC} = 3 + 3 + 3 = 9$. The SIC heuristic can be calculated lazily on the fly (Silver 2005) or can be fully computed in a pre-processing phase (Standley 2010; Sharon et al. 2013). More-informed heuristics by using forms of pattern-databases (Goldenberg et al. 2013; 2014) had limited effectiveness due to the large overhead of creating these PDBs for each instance-specific goal state.

**Drawbacks of A* for MAPF.** The main cycle of A* *expands* a state from the open-list (denoted OPEN) and adds its $b$ successors to OPEN. Consequently, A* for MAPF suffers from two drawbacks. First, the size of the state-space is exponential in the number of agents ($k$), meaning that OPEN cannot be maintained in memory for large problems. Second, the branching factor $b$ of a given state may be exponential in $k$. Consider a state with 20 agents on a 4-connected grid. Fully generating all the $5^{20} = 9.53 \times 10^{14}$ neighbors of even the start state could be computationally infeasible. In the remainder of this section we examine enhancements to A* attempting to overcome these drawbacks.

### 4.1 Independence Detection (ID)

An exponential speedup can be obtained by reducing the number of agents in the problem by using the *Independence Detection* framework (ID) (Standley 2010). Two groups of agents are *independent* if there is an optimal solution for each group such that the two solutions do not conflict. The basic idea of ID is to divide the agents into *independent* groups, and solve these groups separately. First, every agent is placed in its own group. A* is executed separately for each group, returning an optimal solution for that group. The paths of all groups are then checked for validity with respect to each other. If a conflict is found, the conflicting groups are merged into one group and solved optimally using A*. This process of replanning and merging groups is repeated until there are no conflicts between the plans of all groups.[3] Let $k'$ denote the number of agents in the largest independent subproblem ($k' \leq k$). The runtime of solving MAPF with ID is dominated by the runtime of solving the largest independent subproblem (Standley 2010). Therefore, the runtime is reduced from $O(|V|^k)$ to $O(|V|^{k'})$.

A*+ID on EX3 works as follows. The individual optimal paths (detailed above) are found. When validating the paths of agents $a_1$ and $a_2$, a conflict occurs at state $D$ and agents

[2] Similarly, the *maximum* of the individual costs is an admissible heuristic for the makespan variant

[3] A number of technical enhancements to ID were also introduced by Standley (2010) and are not covered here.

$a_1$ and $a_2$ are merged into one group. A* is executed on this group and returns a solution of cost 7. This solution is now validated with the solution of agent $a_3$. No conflict is found and the algorithm halts. The largest group was of size 2. Without ID, A* would have to solve a 3-agent problem.

It is important to note that ID can be implemented on top of any optimal MAPF solver. Therefore, ID can be viewed as a general framework that utilizes an optimal MAPF solver.

### 4.2 Avoiding Surplus Nodes

To guarantee optimality A* must expand all states with $f = g + h < C^*$ as well as some states with $f = C^*$. Nodes generated by A* with $f > C^*$ are denoted as *surplus nodes* (Goldenberg et al. 2014). They will never be expanded as they are not needed to find an optimal solution. The number of generated nodes is the number of expanded nodes times the branching factor $b$. Thus, in MAPF, where $b$ is exponential in $k$, the number of surplus nodes is potentially huge, and avoiding generating them can yield a substantial speedup (Goldenberg et al. 2014; 2012). Next, we describe existing techniques that attempt to avoid generating surplus nodes for MAPF.

**Operator Decomposition (OD).** OD (Standley 2010) applies an arbitrary but fixed order to the agents. When a regular A* node is expanded, OD considers and applies only the moves of the first agent. This introduces an *intermediate node*. At intermediate nodes, only the moves of the next agent without currently assigned moves are considered, thus generating further intermediate nodes. When an operator is applied to the last agent, a regular node is generated. Once the solution is found, intermediate nodes in OPEN are not developed further into regular nodes, so that the number of non-intermediate surplus nodes is significantly reduced.

**Enhanced Partial Expansion (EPEA*).** EPEA* (Goldenberg et al. 2014) avoids the generation of surplus nodes by using *a priori* domain knowledge. When expanding a node $n$ EPEA* generates only the children $n_c$ with $f(n_c) = f(n)$. The other children of $n$ (with $f(n_c) \neq f(n)$) 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 with the desired $f$-value (i.e., $f(n)$). Node $n$ is then re-inserted into OPEN setting $f(n)$ to the $f$-value of the next best child of $n$. Node $n$ may be re-expanded later, when the new $f(n)$ becomes the best in OPEN. EPEA* avoids the generation of surplus nodes and dramatically reduces the number of generated nodes. An OSF for MAPF using $h_{SIC}$ 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 (Goldenberg et al. 2014).

### 4.3 M*

M* (Wagner and Choset 2015) and its enhanced recursive variant (rM*) are important A*-based algorithms related to ID. M* dynamically changes the *dimensionality* and branching factor based on conflicts. 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. 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. On EX2 a single branch is generated until the conflict at $D$ occurs. Then, M* re-inserts all previously expanded nodes back into OPEN and the dimensionally is increased to 2 for agents $a_1$ and $a_2$. Recursive M* (rM*) identifies disjoint subsets of conflicting agents and solves each of the resulting subproblems recursively, effectively running a more fine-grained version of ID. An enhanced variant of M* called ODrM* (Ferner, Wagner, and Choset 2013) builds rM* on top of Standley's OD rather than plain A*. Finally, Wagner and Choset (2015) report that using the MA-CBS framework (described below) with rM* as the low-level search yields a very strong solver.

All the above algorithms execute A* on the $k$-agent state-space. We now turn to recent algorithms which search a different search space.

## 5 The Increasing Cost Tree Search

The *increasing cost tree search* algorithm (ICTS) (Sharon et al. 2013) is a two-level MAPF solver which is conceptually different from A*. ICTS works as follows.

**High level:** At its *high level*, ICTS searches the *increasing cost tree* (ICT). Every node in the ICT consists of a $k$-ary vector $[C_1, \ldots C_k]$ which represents *all* possible solutions in which the individual path cost of agent $a_i$ is exactly $C_i$. The root of the ICT is $[opt_1, ..., opt_k]$, where $opt_i$ is the optimal individual path cost for agent $a_i$ ignoring other agents, i.e., it is the length of the shortest path from $s_i$ to $g_i$ in $G$. A child in the ICT is generated by increasing the cost for one of the agents by 1. An ICT node $[C_1, .., C_k]$ 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 $C_i$. Figure 1(IV) 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 $C_1 + \ldots + C_k$. For the root this is exactly $h_{SIC}(start) = opt_1 + opt_2 + \ldots opt_k$. We use $\Delta$ to denote the depth of the lowest cost ICT goal node. The size of the ICT tree is exponential in $\Delta$. Since all nodes at the same height have the same total cost, a breadth-first search of the ICT will find the optimal solution.

**Low level:** The low level acts as a goal test for the high level. For each ICT node $[C_1, .., C_k]$ 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 $C_i$. For each agent $a_i$, ICTS stores *all* single-agent paths of cost $C_i$ in a special compact data-structure called a *multi-value decision diagram* (MDD) (Srinivasan et al. 1990). 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). For EX2, $[3, 3]$ is the root of the ICT. The low level returned *false* for this node and the next node is $[4, 3]$. Now the low-level returns *true* and the search halts.

**Pruning rules:** A number of pruning techniques were introduced for high-level ICT nodes (Sharon et al. 2013). These techniques search for a sub-solution for $i$ agents, where $i < k$. If there exists a sub-group for which no valid solution exists, there cannot exist a valid solution for all $k$ agents. Thus, the high-level node can be declared as a non-goal without searching for a solution in the $k$-agent path space. A full study of these pruning rules and their connection to CSP is provided in (Sharon et al. 2013).

## 6 Conflict Based Search (CBS)

Another optimal MAPF solver not based on A* is *Conflict-based search* (CBS) (Sharon et al. 2015). Numerous algorithms for more sophisticated, real-world scenarios (described in Section 10) are also based on CBS, therefore CBS is examined in greater detail below.

In CBS, agents are associated with constraints. A *constraint* for agent $a_i$ is a tuple $\langle a_i, v, t\rangle$ 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* is a solution composed of only consistent paths. Note that a consistent solution can be *invalid* if despite the fact that the paths are consistent with the individual agent constraints, they still have inter-agent conflicts.

The high-level of CBS searches the *constraint tree* (CT). The CT is a binary tree, in which each node $N$ contains: **(1)** A set of constraints imposed on the agents ($N.constraints$), **(2)** A single solution ($N.solution$) consistent with these constraints, **(3)** The cost of $N.solution$ ($N.cost$).

The root of the CT contains an empty set of constraints. A successor of a node in the CT inherits the constraints of the parent and adds a single new constraint for one agent. $N.solution$ is found by the low-level search described below. A CT node $N$ is a goal node when $N.solution$ is valid, i.e., the set of paths for all agents has no conflicts. The high-level of CBS performs a best-first search on the CT where nodes are ordered by their costs ($N.cost$).

**Processing a node in the CT:** Given a CT node $N$, the low-level search is invoked for individual agents to return an optimal path that is consistent with their constraints in $N$. Any optimal single-agent path-finding algorithm can be used by the low level of CBS. A simple and effective low-level solver used in (Sharon et al. 2015) is A* with the true shortest distance heuristic (ignoring constraints). Ties between low-level nodes were broken by preferring paths with fewer conflicts with known paths of other agents.

Once a consistent path has been found (by the low level) for each agent, these paths are *validated* with respect to the other agents by simulating the movement of the agents along their planned paths ($N.solution$). If all agents reach their goal without any conflict, $N$ is declared as the goal node, and $N.solution$ is returned. If, however, while performing the validation, a conflict is found for two (or more) agents, the validation halts and the node is declared as non-goal.

**Algorithm 1:** High-level of ICBS

```
1  Main(MAPF problem instance)
2      Init R with low-level paths for the individual agents
3      insert R into OPEN
4      while OPEN not empty do
5          N ← best node from OPEN // lowest solution cost
6          Simulate the paths in N and find all conflicts.
7          if N has no conflict then
8              return N.solution // N is goal
9          C ← find-cardinal/semi-cardinal-conflict(N) // (PC)
10         if C is semi- or non-cardinal then
11             if Find-bypass(N, C) then // (BP)
12                 Continue
13         if should-merge(a_i, a_j) then // Optional, MA-CBS:
14             a_ij = merge(a_i,a_j)
15             if MR active then // (MR)
16                 Restart search
17             Update N.constraints()
18             Update N.solution by invoking low-level(a_ij)
19             Insert N back into OPEN
20             continue // go back to the while statement
21         foreach agent a_i in C do
22             A ← Generate Child(N, (a_i, s, t))
23             Insert A into OPEN
24 Generate Child(Node N, Constraint C = (a_i, s, t))
25     A.constraints ← N.constraints + (a_i, s, t)
26     A.solution ← N.solution
27     Update A.solution by invoking low level(a_i)
28     A.cost ← SIC(A.solution)
29     return A
```

**Resolving a conflict - the *split* action:** Given a non-goal CT node, $N$, whose solution ($N.solution$) includes a *conflict*, $\langle a_i, a_j, v, t\rangle$, we know that in any valid solution at most one of the conflicting agents, $a_i$ or $a_j$, may occupy vertex $v$ at time $t$. Therefore, at least one of the constraints, $\langle a_i, v, t\rangle$ or $\langle a_j, v, t\rangle$, must be satisfied. Consequently, CBS *splits* $N$ and generates two new CT nodes as children of $N$, each adding one of these constraints to the previous set of constraints, $N.constraints$. Note that for each (non-root) CT node the low-level search is activated only for one agent – the agent for which the new constraint was added.

**CBS Example:** Pseudo-code for CBS is shown in Algorithm 1. It is explained on EX2 while also counting the number of low-level nodes (for each CT node) that are expanded as we aim to compare CBS to A* below. The shaded lines (9-20) are all optional and include the enhancements discussed below. The corresponding CT is shown in Figure 1(II). The root ($R$) contains an empty set of constraints and the low-level (A* for individual agents) returns paths $P_1 = \langle S_1, A, C, G_1\rangle$ for $a_1$ and path $P_2 = \langle S_2, B, C, G_2\rangle$ for $a_2$ (line 2). Thus, $R.cost = 6$. Since the length of both $P_1$ and $P_2$ is 3, a total of 8 low-level nodes were generated for $R$. $R$ is then inserted into OPEN and will be expanded next. When validating the two-agents solution (line 6), a conflict $\langle a_1, a_2, C, 2\rangle$ is found. As a result, $R$ is declared as non-goal.

$R$ is split and two children are generated (via the *generate-child()* function, also shown in Algorithm 1) to resolve the conflict (line 22). The left (right) child $U(V)$ adds the constraint $\langle a_1, C, 2\rangle$ ($\langle a_2, C, 2\rangle$). The low-level search is now invoked (line 27) for $U$ to find an optimal path for agent $a_1$ that also satisfies the new constraint. At $U$, $S_1$ plus all $m$ states $A_1, \ldots, A_m$ are expanded by the low-level search with $f = 3$. Then, $D$ and $G_1$ are expanded with $f = 4$ and the search halts and returns the path $\{S_1, A_1, A_1, D, G_1\}$. Thus, at CT node $U$ a total of $m + 3$ low-level nodes are expanded. The path for $a_2$, $\langle S_2, B, C, G_2\rangle$, remains unchanged in $U$. Since the cost of $a_1$ increased from 3 to 4 the cost of $U$ is now 7. In a similar way, the right child $V$ is generated, also with cost 7, and $m + 3$ low-level nodes are expanded for $V$. Both children are added to OPEN (line 23). Next $U$ is chosen for expansion at the high level, and the underlying paths are validated. Since no conflicts exist, $U$ is declared as a goal node (lines 6-8) and its solution is returned. In total, $2m + 14$ low-level states are expanded for EX2.

## 6.1 Improvements to CBS

Basic CBS arbitrarily chooses conflicts to split and arbitrarily chooses paths in the low-level. CBS is very sensitive to these choices and poor choices might significantly increase the size of the high-level search tree (the CT). Four orthogonal improvements to CBS were introduced to improve these choices. A significant speedup is achieved when all improvements are combined. The latter variant is called *Improved CBS* (ICBS) (Boyarski et al. 2015b).

**Improvement 1: Meta-agent CBS.** MA-CBS (Sharon et al. 2012) generalizes CBS by adding the option to *merge* the conflicting agents $a_i$ and $a_j$ into a *meta-agent* (Lines 13-20) instead of the *split* action. A meta-agent is logically treated as a single composite agent, whose state consists of a vector of locations, one for each individual agent. A meta-agent is never split in the subtree of the CT below the current node; it may, however, be merged with other (meta-)agents into new meta-agents. A merge action first unifies the constraints from the two merged agents (Line 23). Then, the low-level search is re-invoked for this new meta-agent only (Line 18), as nothing has changed for the other agents that were not merged. In fact, the low-level search for a meta-agent of size $M$ faces an optimal MAPF problem for $M$ agents, and may be solved with any MAPF solver (e.g., A*).

**Merge policy:** The optional *merge* action is performed only if the *should-merge()* function returns *True* (Line 13). Sharon et al. (2015) presented a simple, experimentally-effective merge policy. Two agents $a_i$ and $a_j$ are merged into a meta-agent $a_{ij}$ if the number of conflicts between $a_i$ and $a_j$ seen so far during the search exceeds a predefined parameter $B$. Otherwise, a regular split is performed. This merge policy is denoted by MA-CBS($B$). MA-CBS was shown to outperform CBS (without the merge option).

Boyarski et al. (2015a; 2015b) further added the following three improvements to CBS.

**Improvement 2: Merge and Restart.** Had we known that a pair of agents are due to be merged, significant computational effort would have been saved by performing the merge ab-initio, at the root node of the CT. To adopt this observation in a simple manner, once a merge decision has been reached for a group of agents $G$ inside a CT node $N$, we discard the current CT and *restart* the search from a new root node, where these agents in $G$ are merged into a meta agent at the beginning of the search. This is called the *merge and restart* (MR) scheme (Boyarski et al. 2015b). It is optionally applied in lines 15-16 of Algorithm 1. MR is simple to implement, and saves much computational effort which otherwise may have been duplicated in multiple CT nodes.

**Improvement 3: Preferring Cardinal Conflicts.** We classify conflicts into three types. A conflict $C = \langle a_1, a_2, v, t\rangle$ is *cardinal* for a CT node $N$ if adding any of the two constraints derived from $C$ ($\langle a_1, v, t\rangle$, $\langle a_2, v, t\rangle$) to $N$ and invoking the low-level on the constrained agent, increases the cost of its path compared to its cost in $N$. $C$ is *semi-cardinal* if adding one of the two constraints derived from $C$ increases $N.cost$ but adding the other leaves $N.cost$ unchanged. Finally, $C$ is *non cardinal* if adding any of the two constraints derived from $C$ does not increase $N.cost$. In EX2 the conflict $\langle a_1, a_2, D, 2\rangle$ is cardinal. If the dotted lines in figure 1(I) are added then agent $a_1$ has a bypass via node $X$. This makes the conflict $\langle a_1, a_2, D, 2\rangle$ semi-cardinal because agent $a_2$ is still forced to go via node $D$.

Based on these definitions, when a CT node $N$ with $N.cost = c$ is chosen for expansion by CBS, we examine all its conflicts (Line 9). If one of them is cardinal, it is immediately chosen for splitting (Line 21). This generates two children with cost $> c$. This is very beneficial if another node $M$ in OPEN exists with cost $c$. $M$ will be chosen for expansion next without further developing nodes below $N$.

**Improvement 4: bypassing Conflicts (BP).** When a semi-cardinal or non-cardinal conflicts are chosen, it is sometimes possible to prevent a split action and bypass the conflict by modifying the chosen path of one of the agents. For a given CT node $N$, BP peeks at either of the immediate children of $N$ in the CT. If the path of a child includes an alternative path with the same cost but without the conflict this path is *adopted* by $N$ without the need to split $N$ and add new nodes to the CT. This can potentially save a significant amount of search due to a smaller size CT. BP is optionally added to the CBS pseudo code (lines 11-12 in Algorithm 1).

# 7 Sum-of-Costs SAT Solver

A first reduction-based SAT solver for the sum-of-costs variant was recently introduced (Surynek et al. 2016). Let $\Delta$ be again the difference between $h_{SIC}$ and the cost of the optimal sum-of-costs ($\Delta = C^* - h_{SIC}$), and let $\mu_0$ be the length of the *longest path* among the shortest individual paths. The main idea is based on the proposition that a solution with sum-of-costs $C^*$ must be achievable within at most $\mu = \mu_0 + \Delta$ time steps because in the worst-case, all $\Delta$ moves belong to the agent whose shortest individual path was $\mu_0$. Based on this, a SAT formula is built that answers whether there is a solution with exactly $\Delta$ extra edges more than $\mu_0$, up to time point $\mu = \mu_0 + \Delta$. An outer loop iterated over different possible values of $\Delta$. Surynek et al. (2016) also showed that the number of propositional vari-

ables that are used within the SAT formula can be significantly reduced with the help of the same MDDs used by the ICTS algorithm. Their best variant is called MDD-SAT.

## 8 Analysis on Example Graphs

As we have seen, many novel algorithms and approaches were introduced recently. Which algorithm is the best? Naturally, every paper on a new algorithm provides experimental results that support its own approach. Nevertheless, our experience suggests that there is no universal winner. The different algorithms have pros and cons and they behave differently in different circumstances. To exemplify this we compare CBS and A* on two example graphs.

First, consider EX2. As detailed above, $2m+14$ low-level states are expanded by CBS here. Now, consider A* which is running in a 2-agent state-space. The root $(S_1, S_2)$ has $f = 6$. It generates $m^2$ nodes, all in the form of $(A_i, B_j)$ for $(1 \leq i, j \leq m)$. All these nodes are expanded with $f = 6$. Now, node $(A_1, D)$ with $f = 7$ is expanded (agent $a_1$ waits at $A_1$). Then, nodes $(D, G_2)$ and $(G_1, G_2)$ are expanded and the solution is returned. So, a total of $m^2 + 3$ nodes are expanded by A*. For $m \geq 5$ this is larger than $2m + 14$ and consequently, CBS will expand fewer nodes. A* must expand the Cartesian product of the single agent paths with $f = 3$. By contrast, CBS only tried two such paths to realize that no solution of cost 6 is valid.

Now consider figure 1(III). Both agents must cross the open area in the middle (colored gray). For each agent there are four optimal paths of length 4 and thus $h_{SIC}(start) = 8$. However, each of the 16 combinations of these paths have a conflict in one of the gray cells. Consequently, $C^* = 9$ as one agent must wait at least one step to avoid collision. For this problem A* will expand 5 nodes with $f = 8$: $((D2, C1), (D3, C2), (D3, B1), (C2, B1), (C3, B2))$ and 3 nodes with $f = 9$ $((B3, B2), (A3, B3), (A3, B4))$ until the goal is found, a total of 8 nodes. By contrast, CBS will build a CT (not shown) which consists of 5 non-goal CT nodes with cost 8, each of them adds a new constraint on one of the conflicts. Only then it would generate 6 CT goal nodes with cost 9. The root CT node will run the low-level search for each agent to a total of 8 low-level expansions. Each of the internal non-goal CT nodes will run the low-level search for a single agent to a total of 4 low-level expansions. Each goal CT node will expand 5 low-level nodes. In total, CBS will expand $8 + 4 \cdot 4 + 6 \cdot 5 = 54$ low-level nodes.

## 9 Experimental Results

We experimentally compared different leading algorithms that were designed for the sum-of-costs variant. We do not intend here to provide a full systematic comparison between all existing algorithms and all their variants (a direction for future work, see below). The aim of this section is to provide experimental evidence that different behavior is obtained (with our implementation) by the different algorithms on different domains. For each family of algorithms the best variant available to us is reported: EPEA* for the A* family, ODrM* for the M* family, ICBS(25)[4] for the CBS family, ICTS+p for the ICTS family and MDD-SAT for the SAT-based solvers designed for the sum-of-costs objective function. All the algorithms were executed within the ID framework except ICBS which uses a similar internal mechanism.

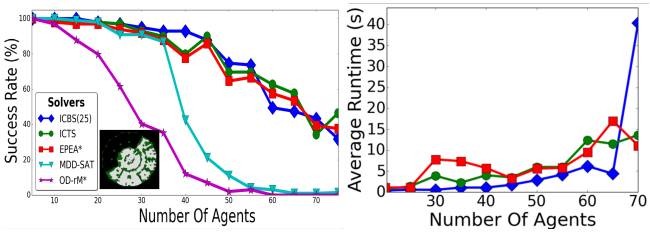



Figure 2: Results on DAO map ost003d

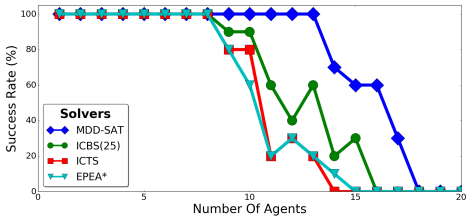
Figure 2(left) presents the *success rate* (= % instances solved) when given 5 minutes for a representative benchmark map (ost003d) of the game *Dragon Age: Origins* from the movingai repository (Sturtevant 2012). It can be easily seen that for $\leq$55 agents ICBS had the best success rate while for $\geq$60 agents ICTS was the best. Nevertheless, Figure 2(right) presents the CPU time averaged over all instances that can be solved by all algorithms (excluding MDD-SAT and ODrM* which were very weak here). On this set of problems, ICBS was almost always the fastest. Next, Figure 3(left) shows results on an 8x8 grid with 10% random obstacles. Here, MDD-SAT significantly outperformed all other algorithms. Finally, figure 3(right) presents results on mazes *512-1-*{*2,6,9*} and *512-2-*{*2,5,9*} from the repository (Sturtevant 2012) when varying the number of cells in the width ($W$) of the corridors of the mazes. For $W = 1$ ODrM* and EPEA* are the best. But, for $W = 2$ ICBS was the best. These representative results demonstrate that there is no universal winner, and that a different algorithm is best in different circumstances.

While it is hard to predict the performance of the algorithms in actual domains, based on our experience and on the above analysis we can give some guidance. If there are more bottlenecks, CBS will outperform A* as it will quickly rule out the $f$-value of the conflicts in the bottleneck and then find solutions which bypass the bottlenecks. A* variants have the advantage over CBS in areas dense with agents, as CBS needs to handle many conflicts. In addition, ICTS is effective if $\Delta$ is rather small and $k$ is large. MDD-SAT is strong in hard problems where the SAT-solver is able to gather enough information on dependencies between its internal variables. Nevertheless, this improvement should be attributed to the strength of modern SAT solvers.

## 10 Real World Scenarios for MAPF

Throughout this paper we treated the MAPF problem under the basic formal definition. However, many real-world

[4]ICBS is usually robust to the low-level solver. We used EPEA*. Other low-level solvers (e.g., variants of M*, ICTS, or even a recursive call to ICBS) were not tried here and are a matter for future work.




| | Success rate | | | | | Runtime (ms) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| W | EPEA* | ICTS | ICBS | SAT | M* | EPEA* | ICTS | ICBS | SAT | M* |
| 1 | 84% | 43% | 51% | 7% | **87%** | 3,016 | 23,535 | 7,778 | >111,805 | **2,974** |
| 2 | **100%** | **100%** | **100%** | 59% | 99% | 2,033 | 2,012 | **239** | >243,045 | 1,935 |

Figure 3: Success rate on open grids (left). Results on mazes with varying widths. 50 agents. (right)

scenarios include more sophisticated settings. We briefly overview preliminary embarking lines of research addressing such real-world issues.

Ma and Koenig (2016) addressed the case where agents are divided into teams and that there is a set of goal locations for each team. However, the agents within each team are indistinguishable. A set of non-conflicting paths should be found such that each agent reaches one of the goal locations affiliated with its team. Another line of research addressed the fact that agents operating in continuous environments may have kinematic constraints. In addition, uncertainty (delay probabilities) might be involved regarding the time needed to traverse certain edges. Such cases were addressed by proposing a two-level framework. First, a MAPF plan is generated. Then, an execution policy of that plan is activated online (Ma et al. 2016b; Ma, Kumar, and Koenig 2017) to address these settings. Another setting is *MAPF with Payload Transfers* (Ma et al. 2016b) where agents are allowed to switch locations by traversing the same edge in opposite directions, simulating a payload transfer between two adjacent agents.

The work described thus far assumed that all agents were seeking to optimize a global commutative objective function. However, in many cases agents may be *self interested* and thus seek paths that minimize their own cost. This was addressed by introducing a set of incentives such that the optimal, single-agent, selfish routes would align with a centralized solution (Bnaya et al. 2013). In addition, a strategy proof mechanism for self interested agents has been proposed (Amir, Sharon, and Stern 2015). Finally, in the *convoy movement problem* (Thomas, Deodhare, and Murty 2015) agents are viewed as convoys of varying length where each convoy moves in a long line and convoys cannot cross each other. A survey of several of these directions is provided in (Ma et al. 2016a).

## 11 Future Challenges

Despite the diverse work in the field, there is much more to be done. The following research directions are suggested to advance the study on the MAPF problem.

**(1:)** Theoretical research is needed to better understand MAPF and explore the parameters that influence the difficulty of the problem such as the number of agents, the size of the graph, the density of the agents, and the way conflicts are distributed. It is not yet known how each of these parameters influence the hardness of the problem.

**(2:)** New algorithms can be developed but many optimizations can also be performed on the algorithms described above. In particular, hybrid algorithms that migrate different sub-procedures from other algorithms or that intelligently (e.g., via meta-reasoning) alternate between algorithms can be created. In addition, further study is needed on *admissible heuristics* for the A* family on a wide range of problem variants. Finally, optimal MAPF algorithms can be modified to better trade runtime for solution quality.

**(3:)** A deep systematic comparison between the various algorithms should be done in order to better understand the pros and cons of the different algorithms and which algorithm works best under what circumstances. A useful set of benchmark instances widely used by all researchers will be greatly welcomed.

**(4:)** Currently, there is fragmentation of works between sum-of-cost variants and makespan variants. A deep unifying research should be done to migrate and compare algorithms for the two cost functions.

**(5:)** New directions were established recently to handle more complex real-world scenarios as detailed in Section 10. But, more research should be done to apply existing algorithms to perform well in real-world domains. New domain properties may include: directed edges, non-unit edge costs, uncertainty in actions, indistinguishable agents and agents that dynamically appear and disappear.

## 12 Acknowledgments

This research was supported by Israel Science Foundation (ISF) grant #417/13 and by a joint grant of the Israel and Czech Ministries of Science #8G15027.

## References

Amir, O.; Sharon, G.; and Stern, R. 2015. Multi-agent pathfinding as a combinatorial auction. In *AAAI*, 2003–2009.

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *ECAI*, 961–962.

Bhattacharyat, S., and Kumar, V. 2011. Distributed optimization with pairwise constraints and its application to multi-robot path planning. *Robotics: Science and Systems VI* 177.

Bnaya, Z., and Felner, A. 2014. Conflict-oriented windowed hierarchical cooperative A*. In *(ICRA)*.

Bnaya, Z.; Stern, R.; Felner, A.; Zivan, R.; and Okamoto, S. 2013. Multi-agent path finding for self interested agents. In *SoCS*.

Botea, A., and Surynek, P. 2015. Multi-agent path finding on strongly biconnected digraphs. In *AAAI*, 2024–2030.

Boyarski, E.; Felner, A.; Sharon, G.; and Stern, R. 2015a. Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In *ICAPS*, 47–51.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Shimony, E.; Bezalel, O.; and Tolpin, D. 2015b. Improved conflict-based search for optimal multi-agent path finding. In *IJCAI-2015*.

Cohen, L.; Uras, T.; Koenig, S.; and Koenig, S. 2015. Feasibility study: Using highways for bounded-suboptimal mapf. In *SOCS*, 2–8.

Cohen, L.; Uras, T.; K., T.; Kumar, S.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Improved solvers for bounded-suboptimal multi-agent path finding. In *IJCAI*, 3067–3074.

de Wilde, B.; ter Mors, A.; and Witteveen, C. 2014. Push and rotate: a complete multi-agent pathfinding algorithm. *J. Artif. Intell. Res. (JAIR)* 51:443–492.

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

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

Ferner, C.; Wagner, G.; and Choset, H. 2013. ODrM* optimal multirobot path planning in low dimensional search spaces. In *ICRA*, 3854–3859.

Goldenberg, M.; Felner, A.; Stern, R.; and Schaeffer, J. 2012. A* Variants for Optimal Multi-Agent Pathfinding. In *SOCS*.

Goldenberg, M.; Felner, A.; Sturtevant, N. R.; Holte, R. C.; and Schaeffer, J. 2013. Optimal-generation variants of EPEA. In *SoCS*.

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

Grady, D. K.; Bekris, K. E.; and Kavraki, L. E. 2011. Asynchronous distributed motion planning with safety guarantees under second-order dynamics. In *Algorithmic foundations of robotics IX*, 53–70. Springer.

Jansen, R., and Sturtevant, N. R. 2008. A new approach to cooperative pathfinding. In *AAMAS*, 1401–1404.

Khorshid, M. M.; Holte, R. C.; and Sturtevant, N. R. 2011. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *SoCS*.

Kornhauser, D.; Miller, G.; and Spirakis, P. 1984. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *Symposium on Foundations of Computer Science*, 241–250. IEEE.

Luna, R., and Bekris, K. 2011. Efficient and complete centralized multi-robot path planning. In *IROS*, 3268–3275.

Ma, H., and Koenig, S. 2016. Optimal target assignment and path finding for teams of agents. In *AAMAS*, 1144–1152.

Ma, H.; Koenig, S.; Ayanian, N.; Cohen, L.; Honig, W.; Kumar, T. K. S.; Uras, T.; and Xu, H. 2016a. Overview: Generalizations of multi-agent path finding to real-world scenarios. In *WOMP Workshop at IJCAI, Available at http://idm-lab.org/publications.html*.

Ma, H.; Tovey, C. A.; Guni, S.; Kumar, T. K. S.; and Koenig, S. 2016b. Multi-agent path finding with payload transfers and the package-exchange robot-routing problem. In *AAAI*, 3166–3173.

Ma, H.; Kumar, T. K. S.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *AAAI*, 3605–3612.

Pallottino, L.; Scordio, V. G.; Bicchi, A.; and Frazzoli, E. 2007. Decentralized cooperative policy for conflict resolution in multivehicle systems. *Robotics* 23(6):1170–1183.

Ryan, M. 2008. Exploiting subgraph structure in multi-robot path planning. *JAIR* 31:497–542.

Ryan, M. 2010. Constraint-based multi-robot path planning. In *ICRA*, 922–928.

Sajid, Q.; Luna, R.; and Bekris, K. 2012. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *SOCS*.

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

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence* 195:470–495.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.* 219:40–66.

Silver, D. 2005. Cooperative pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 117–122.

Srinivasan, A.; Ham, T.; Malik, S.; and Brayton, R. K. 1990. Algorithms for discrete function manipulation. In *(ICCAD*, 92–95.

Standley, T. S., and Korf, R. E. 2011. Complete algorithms for cooperative pathfinding problems. In *IJCAI*, 668–673.

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

Sturtevant, N. R., and Buro, M. 2006. Improving collaborative pathfinding using map abstraction. In *AIIDE*, 80–85.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games* 4(2):144–148.

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

Surynek, P. 2009a. A novel approach to path planning for multiple robots in bi-connected graphs. In *ICRA*, 3613–3619.

Surynek, P. 2009b. Towards shorter solutions for problems of path planning for multiple robots in theta-like environments. In *FLAIRS*.

Surynek, P. 2010. An optimization variant of multi-robot path planning is intractable. In *AAAI*.

Surynek, P. 2012. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *PRICAI*. 564–576.

Thomas, S.; Deodhare, D.; and Murty, M. N. 2015. Extended conflict-based search for the convoy movement problem. *IEEE Intelligent Systems* 30(6):60–70.

Wagner, G., and Choset, H. 2015. Subdimensional expansion for multirobot path planning. *Artif. Intell.* 219:1–24.

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

Wang, K.-H. C., and Botea, A. 2011. Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *JJAIR* 42(1):55–90.

Wurman, P. R.; D'Andrea, R.; Mountz, M.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* 29(1):9–20.

Yu, J., and LaValle, S. M. 2013a. Planning optimal paths for multiple robots on graphs. In *ICRA*, 3612–3617.

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

Yu, J., and LaValle, S. M. 2016. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics* PP(99):1–15.