Multi-Agent Path Finding – An Overview

Roni Stern(✉)

Ben Gurion University of the Negev, Be'er Sheva, Israel
sternron@post.bgu.ac.il

Abstract. Multi-Agent Pathfinding (MAPF) is the problem of finding paths for multiple agents such that every agent reaches its goal and the agents do not collide. In recent years, there has been a growing interest in MAPF in the Artificial Intelligence (AI) research community. This interest is partially because real-world MAPF applications, such as warehouse management, multi-robot teams, and aircraft management, are becoming more prevalent. In this overview, we discuss several possible definitions of the MAPF problem. Then, we survey MAPF algorithms, starting with fast but incomplete algorithms, then fast, complete but not optimal algorithms, and finally optimal algorithms. Then, we describe approximately optimal algorithms and conclude with non-classical MAPF and pointers for future reading and future work.

Keywords: Multi-Agent Pathfinding · Heuristic search

1 Introduction

MAPF is the problem of finding paths for multiple agents such that every agent reaches its desired destination and the agents do not conflict. MAPF has real-world applications in warehouse management [50], airport towing [27], autonomous vehicles, robotics [45], and digital entertainment [26].

Research on MAPF has been developing rapidly in the past decade. In this paper, we provide an overview of MAPF research in the Artificial Intelligence (AI) community. The purpose of this overview is to help researchers and practitioners that are less familiar with MAPF research better understand the problem and current approaches for solving. It is not to intended to serve as a comprehensive survey on MAPF research.

This overview paper is structured as follows. In Sect. 2, we define the problem formally, and discuss several of its notable variants. Then, a simple analysis of the problem is given to illustrate its difficulty. Section 3 starts by describing *prioritized planning* [34], which is still the most common approach in practice to solve MAPF problems. We discuss the limitation of this approach, in particular, the lack of completeness or optimality. Then, we mention several MAPF algorithms that are fast and complete, but may return solutions that are not optimal.

Supported by ISF grant 210/17 to Roni Stern.

G. S. Osipov et al. (Eds.): Artificial Intelligence, LNAI 11866, pp. 96–115, 2019.
https://doi.org/10.1007/978-3-030-33274-7_6

Section 4 surveys several families of MAPF algorithms that are guaranteed to return an optimal solution. Section 5 covers approximately optimal algorithms, i.e., algorithms that guarantee the solution they return is at most a constant factor more costly than an optimal solution. Finally, the paper concludes with a partial list of MAPF extensions (Sect. 6), and pointers to further reading and resources (Sect. 7). In addition, throughout this paper, we point to interesting directions for future work.

2 Problem Definition

The literature includes multiple definitions of the MAPF problem. In this paper, we mostly focus on what is called *classical MAPF* [37]. Section 6 discusses other variants of MAPF. A classical MAPF problem with k agents is defined by a tuple $\langle G, s, t \rangle$ where:

- $G = (V, E)$ is an undirected graph whose vertices are the possible locations agents may occupy and every edge $(n, n') \in E$ represents that an agent can move from n to n' without passing through any other vertex.
- s is a function that maps an agent to its initial location.
- t is a function that maps an agent to its desired destination location.

Time is discretized into time steps. In every time step, each agent can perform a single *action*. There are two types of actions: *wait* and *move*. An agent performing a *wait* action stays in its current location for one time step. A *move* action moves an agent from its location to some other location. Move action takes exactly one time step, and can only move an agent from its current location to one of its adjacent locations. A *valid solution* to a MAPF problem is a joint plan that moves all agents to their goals, in a way that agents do not collide. Next, we define the terms valid solution, joint plan, and collision, in a formal way.

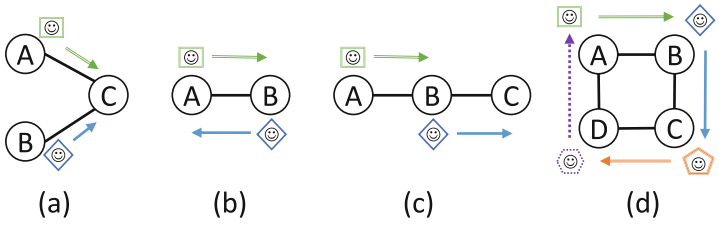

Fig. 1. Illustration of different types of conflicts, taken from Stern et al. [37]: (a) a vertex conflict, (b) a swapping conflict, (c) a following conflict, and (d) a cycle conflict.

A *single-agent plan* for an agent i is a sequence of actions that if agent i performs these actions in location $s(i)$ it will end up in location $t(i)$. Formally, a single-agent plan for agent i is sequence of actions $\pi = (a_1, \ldots a_n)$ such that

$$a_n(\cdots a_2(a_1(s(i)))\cdots) = t(i) \tag{1}$$

A *joint plan* is a set of single-agent plans, one for each of the k agents. For a joint plan Π, we denote by Π_i its constituent single-agent plan for agent i. A pair of agents i and j have a *vertex conflict* in a joint plan Π if according to their respective single-agent plans Π_i and Π_j both agents are planned to occupy the same vertex at the same time. Similarly, agents have a *swapping conflict* in a joint plan if they are planned to swap locations over the same edge at the same time. A *valid* solution to a MAPF problem is a joint plan that has none of these conflicts.

Some MAPF applications have stricter requirements from a valid solution, prohibiting other types of conflicts. Two notable types of conflicts are *following* conflicts and *cycle* conflicts. A *following conflict* occurs if an agent plans to occupy at time step $t+1$ a location that was occupied by some other agent at time step t. A *cycle conflict* occurs if a set of agents $i, i+1, \ldots j$ plan to move in the same time step t in a circular pattern, i.e., agent i plans to move in time step $t+1$ to agent's $i+1$ location at time step t, agent $i+1$ plans to move in time step $t+1$ to agent's $i+2$ at time step t, and so on, while agent j plans to move in time step $t+1$ to agent's i location at time step t. Figure 1 illustrates all these different types of conflicts. See Stern et al. [37] for a comprehensive discussion on different types of conflicts and the relationships between them.

2.1 Optimization

MAPF problems can have more than one valid solution. In many MAPF applications, one would like to find a valid solution that optimizes some objective function. The two most common objective functions used for evaluating a MAPF solution are *makespan* and *sum of costs*. The makespan of a joint plan Π, denoted $\mathrm{M}(\Pi)$ is the number of time steps until all agents reach their goal.

$$\mathrm{M}(\Pi) = \max_{1 \leq i \leq k} |\pi_i| \tag{2}$$

The sum of costs of a joint plan Π, denoted $\mathrm{SOC}(\Pi)$ is the sum of actions performed until all agents reach their goal.

$$\mathrm{SOC}(\Pi) = \sum_{1 \leq i \leq k} |\pi_i| \tag{3}$$

Following most prior work, we assume that when an agent waits in its destination then it also increase the SOC of the overall joint plan, unless that agent is not planned to move later from its destination location. For example, consider the case where agent i reaches its destination at time step t, leaves it at time step t', arrives back to its destination at time step t'', and stays there until all agents reach their destinations. Then this single-agent plan contributes t'' to the SOC of the corresponding joint plan.

2.2 From Single-Agent Pathfinding to MAPF

A single-agent shortest-path problem (SPP) is the problem of finding the shortest path in a graph $G = (V, E)$ from a given source vertex $s \in V$ to a given target

vertex $t \in V$. MAPF can be reduced to a shortest-path problem in a graph known as the k-agent search space. This graph, denoted G_k, is different from the single-agent graph G. A vertex in G represents a location that an agent may occupy in a particular time step. A vertex in G_k represents a set of locations, one per agent, that the agents can occupy in a particular time step. Thus, a vertex in G_k is a vector of k vertices in G. An edge in G_k represents a *joint action* of all agents, that is, a set of k actions, one per agent, that the agents can perform *simultaneously* in a particular time step. Joint actions that result in a conflict, will not have a corresponding edge in G_k. The cost of an edge in G_k corresponds to the cost of the corresponding joint action.

Observation 1. *A lowest-cost path in G_k from $\big(s(1), \ldots, s(k)\big)$ to $\big(t(1), \ldots t(k)\big)$ is an optimal solution to the MAPF problem $\langle G, s, t\rangle$ and vice versa.*

Heuristic Search and the A* Algorithm. Heuristic search in general and the A* algorithm in particular [18] are commonly used to solve shortest-path problems. For completeness, we provide a brief background on A*.

A* is a best-first search algorithm. It maintains a list of vertices called OPEN. Initially, OPEN contains the source vertex. In every iteration, a single vertex is removed from OPEN and *expanded*. To expand a vertex means to go over each of its neighbors and *generate it*. To generate a vertex means creating it and adding it to OPEN, unless it has already been generated before. For every generated vertex n, A* maintains several values.

- $g(n)$ is the cost of the lowest-cost path found so far from the source vertex to n.
- *parent*(n) is the vertex before n on that path.
- $h(n)$ is a *heuristic* estimate of the cost of the lowest-cost path from n to the target vertex.

Let $h^*(n)$ be a *perfect* heuristic estimate for n, that is, the cost of the lowest-cost path from n to a goal. If $h^*(n)$ is known for all nodes, then one can find the shortest path from the source vertex to the target by choosing to go to the vertex with the smallest h value. A heuristic function h is called *admissible* iff for every vertex n it holds that $h(n) \leq h^*(n)$. The A* algorithm chooses to expand the vertex n in OPEN that has the smallest $g(n) + h(n)$ value.

Theorem 1 (Optimality of A* [18]). *Given an admissible heuristic, A* is guaranteed to return an optimal solution, i.e., a shortest path from the source vertex to its target.*

Observation 1 and Theorem 1 mean that one can solve a given MAPF problem by running A* on the k-agent search space. A simple way to obtain an admissible heuristic for the k-agent search space is by considering the cost of the shortest path in G from every vertex $v \in V$ to every target vertex $t(1), \ldots t(k)$. This is done as follows. Let $d(v, t(i))$ be the cost of the shortest path from v to $t(i)$.

Computing $d(v, t(i))$ for every $v \in V$ and $i \in \{1, \ldots, k\}$ can be done in time that is polynomial in $|V|$ and k, in the beginning of the search. Then, the following is an admissible heuristic when optimizing for sum of costs

$$h\big((v_1, \ldots v_k)\big) = \sum_{i \in \{1,\ldots,k\}} d(v_i, t(i)) \tag{4}$$

and the following is an admissible heuristic when optimizing makespan

$$h\big((v_1, \ldots v_k)\big) = \max_{i \in \{1,\ldots,k\}} d(v_i, t(i)) \tag{5}$$

Challenges in Solving MAPF with A*. A very rough way to estimate the hardness of solving a shortest path problem, with A* and other algorithms, is by considering the *size* of the search space and its *branching factor*, which in our case corresponds to the number of vertices in G_k and its average outgoing degree. Thus, in the worst case, the size of the search space is $|V|^k$ and the branching factor is $\left(\frac{|E|}{|V|}\right)^k$. As can be seen, both values are exponential in the number of agents.

To get an estimate of these numbers, consider a MAPF problem with 20 agents on a 4-connected grid with 500×500 cells. In this case, the size of the search space is $25{,}000^{20} \approx 9.09 \cdot 10^{87}$ and the branching factor is $4^{20} \approx 1.1 \cdot 10^{12}$ $|V| \approx 25{,}000$. The exponential branching factor is especially problematic for A*, since A* must at least expand all vertices along an optimal path. The computational cost of expanding a vertex, however, is at least linear in the branching factor. Thus, textbook A* cannot be used to solve a MAPF problem with a large number of agents, even with a perfect heuristic function.

3 Fast MAPF Algorithms

A fundamental approach to address this combinatorial explosion is to try to decouple the MAPF task to k single-agent pathfinding problems with as minimal interaction as possible. Perhaps one of the most popular approaches to do so is *prioritized planning*.

3.1 Prioritized Planning

The first step in prioritized planning is to assign each agent a unique number from $\{1, \ldots k\}$. Then, a single-agent plan is found for each agent in order of their priority. When an agent searches for a plan, it is required to find a plan that avoids creating a conflict with plans already found for agents with higher priority.

A fundamental difference between a textbook shortest-path problem and the problem of finding a plan for the agent with the i^{th} priority is that in the latter an optimal solution may require an agent to wait in its location. Thus, to find

a plan for the i^{th} agent, is, in fact, a shortest path problem in a *time-expansion* graph [34]. In a time-expansion graph, every vertex represents a pair (v, t), where v is a vertex in the underlying graph G and t is a time step. There is an edge between vertices (v, t) and (v', t') in the time-expansion graph iff $t' = t+1$ and v' is either equal to v or it is one of its neighbors. The size and branching factor of the corresponding search space is manageable: the number of vertices is $|V| \times T$, where T is an upper bound on the solution makespan, and the branching factor is $\frac{|E|}{|V|} + 1$. For example, in a MAPF problem with 20 agents on a 4-connected grid with 500×500 cells, assuming $T = 1{,}000$, we have a search space size of 25,000,000 and a branching factor of 5. A* has been successfully applied to much larger search spaces.

The computational efficiency and simplicity of prioritized planning algorithms is the main reason for their widespread adoption by practitioners. Implementing prioritized planning includes many design choices. For example, several methods have been proposed for setting the agents' priorities [1,7]. The Windowed Hierarchical Cooperative A* algorithm (WHCA*) [34] also allowed interleaving planning and execution in a prioritized planning framework. In WHCA*, the agents plans to avoid conflicts only for the next X time steps (the "window"). After performing these X steps, the agents can re-plan the next X steps in the same manner.

Prioritized planning is a *sound* approach for MAPF, in the sense that it returns valid solutions. However, it is neither *complete* nor *optimal*. That is,

- **Not complete.** A prioritized planning algorithm may not find any solution to a solvable MAPF problem.
- **Not optimal.** The solution returned by a prioritized planning algorithm may not be optimal, w.r.t. to a given objective function (e.g., sum of costs or makespan).

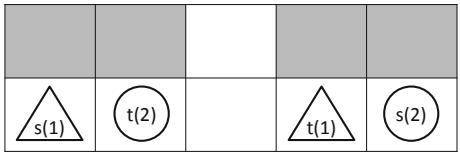

Fig. 2. A MAPF problem in which prioritized planners will not find any solution.

As an example of these prioritized planning limitations, see the MAPF problem depicted in Fig. 2. In this example, any prioritized planning algorithm will fail to find a solution, regardless of which agent has a higher priority. The problem, however, is clearly solvable, by having agent 1 move to the middle grid cell in the upper row, allowing agent 2 to move to its target ($t(2)$), and then moving to its own target ($t(1)$).

3.2 Complete MAPF Solvers

We say that a MAPF algorithm is *fast* if its worst-case time complexity is polynomial in the size of the graph G, and not exponential in the number of agents. Surprisingly, there are fast and complete algorithms for solving MAPF problems. The most general of those is Kornhauser's algorithm [20], which is complete and runs in a worst case time complexity of $O(|V|^3)$. This algorithm is regarded as complicated to implement. Thus, a variety of algorithms have been proposed that are also fast and complete, at least for some restricted classes of MAPF problems. Below, we provide a partial list of such algorithms and classes of MAPF problems.

The Push-and-Swap algorithm [24] and its extensions Parallel Push-and-Swap [31] and Push-and-Rotate [11], are fast MAPF algorithms that are complete for any MAPF problem in which there are at least two unoccupied vertices in the graph. Very roughly, these algorithms work by executing a set of macro-operators that move an agent towards its goal (push) and swap the location of two agents (swap).

A MAPF problem is *well-formed* if, for any pair of agent i and j, there exists a path from $s(i)$ to $t(i)$ that does not pass through $s(j)$ and $t(j)$. Čáp et al. [9] proved that prioritized planning algorithms that compulsory avoid start locations are complete for well-formed MAPF problems.

A MAPF problem is *slidable* if for any triple of locations v_1, v_2, and v_3, there exists a path from v_1 to v_3 that does not go through v_2.[1] Wang and Botea [49] proposed a fast algorithm called MAPP that is complete for slidable MAPF problems. The BIBOX algorithm is also fast and complete under these conditions [38].

While all the above algorithms are fast and, under certain conditions, complete, they do not provide any guarantee regarding the *quality* of the solution they return. In particular, they do not guarantee that the resulting solution is optimal, either w.r.t. sum-of-costs or makespan. In fact, finding a solution that has the smallest makespan or the smallest sum of costs, is NP hard [39,53]. Nevertheless, solution quality is important in many applications, e.g., saving operational costs in an automated warehouse. Also, modern MAPF algorithms can find provably optimal solutions in a few minutes to problems with more than a hundred agents [14,21,32].

In the next section, we present the state-of-the-art in MAPF algorithms that are guaranteed to return a solution that is optimal with respect to a given objective function. Such algorithms are referred to as *optimal MAPF algorithms*.

4 Optimal MAPF Solvers

It is possible to classify optimal MAPF algorithms to four high-level approaches:

[1] The exact definition of slidable is slightly more involved. The interested reader can see the exact definition in Wang and Botea's paper [49].

- **Extensions of A*.** These are algorithms that search the k-agent search space using a variant of the A* algorithm.
- **The Increasing Cost Tree Search** [33]. This algorithm splits the MAPF problem into two problems: finding the cost added by each agent, and finding a valid solution with these costs.
- **Conflict-Based Search** [32]. This algorithmic family solves MAPF by solving multiple single-agent pathfinding problems. To achieve coordination, specific constraints are added incrementally to the single-agent pathfinding problems, in a way that verifies soundness, completeness, and optimality.
- **Constraints programming** [6,39]. This approach compiles MAPF to a set of constraints and solves them with a general purpose constraints solver.

4.1 Extensions of A*

Standley [36] proposed two very effective extensions to A* for solving MAPF problems.

Operator Decomposition. The first extension is called *Operator Decomposition* (OD). OD is designed to cope with the exponential branching factor of the k-agent search space. In OD, the agents are sorted according to some arbitrary order. When expanding the source vertex $\big(s(1),\ldots,s(k)\big)$, only the actions of one agent are considered. This generates a set of vertices that represent a possible location for the first agent in time step 1, and the locations all other agents are occupying at time step 0. These vertices are added to OPEN. When expanding one of these vertices, only the actions of the second agent are considered, generating a new set of vertices. These vertices represent a possible location for the first and second agents in time step 1, and the locations of all other agents are occupying at time step 0. The search continues in this way. Only the k^{th} descendent of the start vertex is a vertex that represents a possible location of all agents at time step 1. Vertices that represent the location of all agents at the same time step are called *full vertices*, while all other vertices are called *intermediate vertices*. The search continues until reaching a full vertex that represents the target $\big(t(1),\ldots,t(k)\big)$.

The obvious advantage of A* with OD compared to A* without OD is the branching factor. With OD, the branching factor is that of a single agent, while without OD, it is exponential in the number of agents. However, the solution is k times deeper when using OD, since there are k vertices between any pair of full states. In the case of MAPF, this tradeoff is usually beneficial due to the heuristic function. A high heuristic value for an intermediate vertex can help avoid expanding the entire subtree beneath that vertex.

OD can be viewed as a special case of the Enhanced Partial Expansion A* (EPEA*) algorithm [17]. EPEA* is a variant of A* that can avoid generating some of the vertices A* would generate when expanding a vertex. For details on EPEA* and how it relates to OD, see Goldenberg et al. [17].

Independence Detection. The second A* extension proposed by Standley [36] is called *Independence Detection* (ID). ID attempts to decouple a MAPF problem with k agents to smaller MAPF problems with fewer agents. It works as follows. First, each agent finds an optimal single-agent plan for itself while ignoring all other agents. If there is a conflict between the plans of a pair of agents, these agents are *merged* to a single *meta-agent*. Then, A* +OD is used to find an optimal solution for the two agents in this meta-agent, ignoring all other agents. This process continues iteratively: in every iteration a single conflict is detected, the conflicting (meta-)agents are merged, and then solved optimally with A* +OD. The process stops where there are no conflicts between the agents' plans.[2]

In the worst case, ID will end up merging all agents to a single meta-agent and solving the resulting k-agents MAPF problem. However, in other cases, an optimal solution can be returned and guaranteed by only solving smaller MAPF problems with fewer agents. This can have a dramatic impact on runtime. ID is a very general framework for MAPF solvers, as one can replace A* +OD with any other complete and sound MAPF solver.

M*. The M* algorithm [47] also search the k-agent search space like A*. To handle the exponential branching factor, M* dynamically changes the branching factor of the search space, as follows. Initially, whenever a vertex is expanded, it generates only a single vertex that corresponds to all agents moving one step in their own, individual, optimal path. This generates a single path in the k-agent search space. Since the agents are following their individual optimal path, a vertex n may be generated that represents a conflict between a pair of agents i and j. If this occurs, all the vertices along the path from the start vertex to n are re-expanded, this time generating vertices for all combinations of actions agents i and j may perform. In general, a vertex in M* stores a *conflict set*, which is a set of agents for which it will generate all combinations of actions. For agents not in the conflict set, M* only considers a single action – the one on their individual optimal path. Recursive M* (rM*) is a notable improved version of M*. rM* attempts to identify sets of agents in the conflict set that can be solved in a decoupled manner.

M* is similar to OD in that it limits the branching factor of some vertices. rM* also bears some similarity to ID, in that it attempts to identify which sets of agents can be solved separately. Nevertheless, rM*, OD, and ID, can be used together: rM* can be used by ID to find optimal solutions to conflicting meta-agents, and rM* can search the k-agent search space with A* with OD instead of plain A*. The latter is referred to as ODrM* and was shown to be effective in some scenarios [47].

[2] This is actually a description of the *simple ID* algorithm. In the full ID algorithm, the conflicting agents attempt to individually avoid the conflict while maintaining their original solution cost.

4.2 The Increasing Cost Tree Search (ICTS)

The Increasing Cost Tree Search (ICTS) [33] algorithm does not search the k-agent search space directly. Instead, it interleaves two search processes. The first, referred to as the *high-level search*, aims to find the sizes of the agents' single-agent plans in an optimal solution for the given MAPF problem. The second, referred to as the *low-level search*, accepts a vector of plan sizes $(c_1, \ldots, c_k)$, and verifies if there exists a valid solution $(\pi_1, \ldots, \pi_k)$ to the given MAPF problem in which the size of every single agent plan π_i is exactly c_i.

The high-level search of ICTS is implemented as a search over the *increasing cost tree* (ICT). The ICT is a tree in which each node is a k-dimensional vector of non-negative values. The root of the ICT is a vector $(c_1, \ldots, c_k)$ where for every agent i, the value c_i is the size of its individual optimal path. The children of a node n in this tree are all vectors that result from adding one to one of the k elements in n. The high-level of ICTS searches the ICT in a breadth-first manner. This is done to verify that the first valid solution found by the low-level search is an optimal solution.

As mentioned above, the low-level search of ICTS accepts an ICT node $(c_1, \ldots c_k)$ from the high-level search, and searches for a valid solution $(\pi_1, \ldots, \pi_k)$ in which $\forall i : |\pi_i| = c_i$. To do so efficiently, ICTS computes for each agent i all single-agent plans of size c_i. Generating these set of plans is done with a simple breadth-first search, and they are stored compactly in a Multi-valued Decision Diagram (MDD) [35]. The cross product of the agents' MDDs is a subgraph of the k-agent search space that contains all joint plans that correspond to the given ICT node. Observe that this cross product is a subgraph of the k-agent search space. ICTS searches this cross product of MDDs for a valid solution. Since this search solves a satisfaction problem and not an optimization problem, a simple depth-first branch-and-bound is commonly used.

An effective way to speedup ICTS is to prune the ICT by quickly identifying subsets of single-agent plan costs for which there is no valid solution [33]. For example, assume an ICT node $(c_1, \ldots c_k)$ given to the low-level search. One can check if there is a pair of single-agent plans for agents 1 and 2 such that their costs is c_1 and c_2, respectively, and they do not conflict. If no such pair of plans exists, then the low-level search can safely return that there is no valid solution for the corresponding ICT node. While this technique for pruning the ICT is highly effective in practice, there is no current theory about how to choose which subsets of costs to check. This is an open question for future research.

4.3 Conflict-Based Search

Conflict-Based Search (CBS) [32] is an optimal MAPF algorithm. It is unique in that it solves a MAPF problem by solving a sequence of single-agent pathfinding problems.

In more detail, CBS, similar to ICTS, runs two interleaving search processes: a *low-level* search and a *high-level* search. The CBS *low-level* search accepts as input an agent i and a set of constraints of the form $\langle i, v, t \rangle$, representing that

agent i must not be at vertex v in time step t. The task of the CBS low-level search is to find the lowest-cost single-agent plan for agent i that does not violate the given set of constraints. Existing single-agent pathfinding algorithms, such as A*, can be easily adapted to serve as the CBS low-level search.

The CBS *high-level* searches a set of constraints to impose on the low-level search so that the resulting joint plan is a cost-optimal valid solution. This search is performed over the Constraint Tree (CT). The CT is a binary tree in which each node n is a pair $(n.cont, n.\Pi)$ where $n.cont$ is a set of CBS constraints and $n.\Pi$ is a joint plan consistent with these constraints. A CT node n is generated by first setting its constraints and then using the CBS low-level search to find a single-agent plan for each agent that satisfies its constraints. The root of the CT is a CT node with an empty set of constraints. The objective of the high-level search is to find node n in the CT in which $n.\Pi$ is a cost-optimal valid solution.

The high-level search achieves this objective by searching the CT as follows. First, the root of the CT is generated. If the joint plan for the root has no conflict, meaning it is a valid solution, then the search returns it. Otherwise, one of the conflicts in the joint plan is chosen. Let i, j, x, and t be the pair of agents, location, and time steps for which this conflict has occurred. Two new CT nodes, n_i and n_j, are generated and added as children to the root node. The CT node n_i is generated with the constraint $\langle i, x, t\rangle$ and the CT node n_j is generated with the constraint $\langle j, x, t\rangle$. The *cost* of a CT node is the cost of the joint plan it represents. The high-level search continues to search the CT in a best-first manner, choosing in every iteration to expand a CT node with the lowest cost. Expanding a CT node means choosing one of its conflicts, and resolving them by generating two new CT nodes with an additional constraint as shown above. The search halts when a CT node n is found in which $n.\Pi$ has no conflicts. Then, $n.\Pi$ is returned, and is guaranteed to be optimal.

CBS has many extensions and improvements. Meta-agent CBS [32] is a generalization of CBS in which instead of adding new constraints to resolve a conflict between two agents, the algorithm may choose to merge the conflicting agent to a single meta-agent. Improved CBS [8] attempts to reduce the size of the CT by intelligently choosing which conflict to resolve in every iteration. HCBS [14] adds an admissible heuristic to the high-level search to prune more nodes from the CT. Recent work suggested a different scheme for resolving conflicts. For a conflict in location x at time t between agents i and j, they proposed to generate three CT nodes: one with a constraint that agent i must occupy x at time t, one with a constraint that agent j must occupy x at time t, and one with a constraint that neither agent i nor agent j can occupy x at time t. The benefit of this three-way split is that the sets of solutions that satisfy them is disjoint.

4.4 Constraint Programming

Constraint Programming (CP) is a problem-solving paradigm in which one models a given problem as a *Constraints Satisfaction Problem* (CSP) or a *Constraint Optimization Problems* (COP), and then use a general-purpose *constraints solver* to find a solution. A notable special case of CP is to model a problem as a

Boolean Satisfiability (SAT) problem, which is a special case of CSP, and use a general-purpose SAT solver.

CP is a very general paradigm because many problems, including MAPF, can be modeled as a CSP or a COP. The major benefit of using CP is that current general-purpose constraints solver are very efficient and are constantly getting better. In particular, modern SAT solvers are extremely efficient, solving SAT problems with over a million variables.

A common approach for finding a solution to a given MAPF problem with optimal makespan with CP is by splitting the problem to two problems: (1) finding a valid solution whose makespan is equal to or smaller than a given bound T, and (2) finding a value of T that is equal to the optimal makespan. Next, we provide a brief description of this approach.

Finding a Valid Solution for a Given Makespan Bound. For every triplet of agent a, vertex $v \in V$, and time step t, we define a Boolean variable $\mathcal{X}_{a,v,t}$. Setting $\mathcal{X}_{a,v,t}$ to true means that a is planned to occupy v at time t. The constraints imposed on these variables ensure that:

1. **Agent occupies one vertex in each time step.** For every time step and agent there is exactly one variable $\mathcal{X}_{a,v,t}$ that is assigned true. that is assigned a true value.
2. **No conflicts.** For every time step and location, there is at most one variables $\mathcal{X}_{a,v,t}$ that is assigned true.[3]
3. **Agents start and ends in the desired locations.** For every agent i, $\mathcal{X}_{i,s(i),1}$ and $\mathcal{X}_{i,t(i),C}$.
4. **Agents move along edges.** For every time t before T, agent i, and pair of vertices v and v', if the variables $\mathcal{X}_{i,t,v}$ and $\mathcal{X}_{i,t,v'}$ are both true then there is an edge $(v, v') \in E$.

Any assignment of values to the variables $\mathcal{X}_{a,v,t}$ corresponds to a valid solution for our MAPF problem whose makespan is at most T.

Finding the Optimal Makespan. To find the optimal makespan, we start by setting T to be a lower bound on the optimal makespan. Such a lower bound can be easily obtained by taking the maximum over the agents' individual shortest path to their goal. Then, a constraints solver is used to search for a solution to the CSP defined above. If a solution has been found, we have found an optimal solution. If not, T is incremented by one, and the constraints solver is used again to solve the new CSP. This process continues until an optimal solution is found. Finding a solution with optimal sum-of-costs is also possible with CP, but it requires some additional constraints and changes to the process [6,42].

[3] Actually, this constraint only prevents vertex conflicts. To prevent swapping conflicts, an additional constraint is needed, in which for every time step t before T, pair of agents a and a', and pair of locations v and v', if the variables $\mathcal{X}_{i,t,v}$ and $\mathcal{X}_{i,t',v'}$ are both true then the variables $\mathcal{X}_{j,t,v'}$ and $\mathcal{X}_{j,t',v}$ must not be both true.

It is important to note that the above is not the only way to solve MAPF with CP. Surynek explored five different ways to model MAPF using SAT, showing how different modeling choices impact the SAT solver's runtime [40]. Barták et al. [6] modeled several variants of MAPF using Picat [54], a higher-level CP language. A CP written in Picat can be automatically compiled and solved with either SAT, a CP solver, or a Mixed Integer-Linear Program (MILP) solver [44]. They showed that different modelings and solvers are effective for different MAPF variants and problems. Still, how to choose the best model and solver for a given MAPF problem is, to-date, an open question.

It is worth noting that solving MAPF with CP is, in it self, a special case of a more general approach for solving MAPF in which one compiles MAPF to a different problem, solves it with an algorithm designed for that problem. Prominent examples are MAPF compilation to Answer Set Programming (ASP) [13], to SAT Modulu Theory (SMT) [41], and to multi-commodity network flow [52]. Such MAPF algorithms are sometimes referred to as *reduction-based* MAPF solvers [15].

4.5 Summary of Optimal Solvers

Unfortunately, there are no clear guidelines to predict which of the MAPF algorithms detailed above would work best for a given MAPF problem. Prior work suggested the following rules-of-thumb:

- A*-based and CP approaches are effective for small graphs that are dense with agents.
- CBS and ICTS are effective for large graphs.

However, this rules-of-thumb has not been grounded theoretically and its empirical support is weak. We expect that future work will explore automated methods to select the best solver to use for a given problem. Another appealing direction for future work is to create hybrid algorithms that enjoy the complementary benefits of different MAPF solvers.

5 Approximately Optimal Solvers

While modern optimal MAPF algorithms have pushed the state of the art impressively, there are still many MAPF problems for which current algorithms cannot solve optimally in reasonable time. In such cases, one can always use one of the fast MAPF algorithms described in Sect. 3, but that would mean the solution returned may be very costly.

Approximately optimal MAPF algorithms, also known as *bounded-suboptimal* algorithms, lie in the range between these algorithms and optimal algorithms. An *approximately optimal* algorithm is an algorithm that accepts a parameter $\epsilon > 0$ and returns a solution whose cost is at most $1 + \epsilon$ times the cost of an optimal solution. Ideally, an approximately optimal algorithm would return solutions faster when increasing ϵ, thus providing a controlled trade-off between

runtime and solution quality. Approximately optimal MAPF algorithms have been proposed based on each of the optimal MAPF approaches described in the previous section. We describe them briefly below.

5.1 A*-based

Creating an approximately optimal version of an A*-based MAPF algorithm is straightforward, since there are many approximately optimal A*-based algorithm in the heuristic search literature. Perhaps the most well-known approximately optimal A*-based algorithm is *Weighted A** [30], which is a best-first search that uses the $g + (1 + \epsilon)h$ evaluation function to choose which node to expand in every iteration. All A*-based MAPF algorithms can use the same evaluation function and obtain the guarantee: that the solution cost is at most $1 + \epsilon$ times the cost of an optimal solution. Such a variant was mentioned explicitly for M* [47], under the name *inflated M**.

An interesting direction for future work is to use more modern A*-based approximately optimal algorithms to improve the performance of approximately optimal A*-based MAPF algorithms. Explicit Estimation Search (EES) [43] and Dynamic Potential Search (DPS) [16] are some examples of such approximately optimal A*-based algorithms.

5.2 ICTS

To the best of our knowledge, there is no approximately optimal ICTS-based algorithm for classical MAPF. The challenge in creating such an algorithm is that the ICTS high-level search is done in a breadth-first manner. Thus there is no heuristic to inflate, preventing the clear application of Weighted A* and other approximately optimal search algorithms.

However, there is an approximately optimal variant of ICTS for MAPF problems in which moving an agent across different edges can have different costs. This algorithm is based on the Extended ICTS (eICTS) algorithm [48], which is an ICTS-based algorithm designed for this type of MAPF problems. In eICTS, each ICT node is associated with a lower and upper bound. The high-level search in this case becomes a best-first search on the lower bound, and low-level search looks for optimal solutions within these bounds. This allows creating an approximately optimal version of eICTS called wICTS, in which suboptimality is added to both high-level and low-level search.

5.3 CBS

Enhanced CBS [4] is an approximately optimal MAPF algorithm that is based on CBS. It introduces suboptimality in the low-level search and in the high-level search. The low-level search in CBS can be any optimal shortest path algorithm, such as A*. As noted above, there are several approximately optimal algorithms that are based on A*, including Weighted A* [30], EES [43], and DPS [16].

Thus, introducing suboptimality to the low-level search can be done by simply using one of these approximately optimal algorithms.

Introducing suboptimality to the high-level search is slightly more involved. To do so, ECBS uses a *focal search* framework for its high-level search. Focal search is a heuristic search framework introduced by Pearl and Kim [28] in which the node expanded in every iteration is chosen from a subset of nodes called FOCAL. FOCAL contains all nodes in OPEN that may lead to a solution that may be approximately optimal. To choose which node to expand From FOCAL, a secondary heuristic can be used. Importantly, this heuristic can be inadmissible and domain-dependent. ECBS uses the focal search framework, and uses a MAPF-specific secondary heuristic that prioritizes CT node with fewer conflicts. For details, see Barer et al. [4]. Later work proposed an extension to ECBS in which user-defined paths called *highways* are prioritized to further improve runtime [10].

5.4 Constraint Programming

EMDD-SAT is a recently proposed approximately optimal MAPF algorithm from the CP family. This algorithm models MAPF as a SAT problem. It follows the high-level approach we described in Sect. 4.4, except that it is designed for (approximately) optimizing SOC and not makespan.

In a very high-level manner, EMDD-SAT works by creating a SAT model that allows solutions with longer makespan and larger SOC. The suboptimality is controlled by high much larger is the SOC from a computed SOC lower-bound. To the best of our knowledge, there is no approximately optimal MAPF algorithm from this family that is designed for finding solutions with approximately optimal makespan.

In general, significantly less efforts have been dedicated, to date, to develop approximately optimal MAPF algorithms. However, existing approximately optimal MAPF algorithms demonstrate that adding even a very small amount of suboptimality can allow solving much larger problems. For example, ECBS with at most 1% suboptimality is able to solve MAPF problems with 250 agents on large maps [4].

6 Beyond Classical MAPF

The scope of this overview is mostly limited to what is referred to as *classical MAPF* [37]. Classical MAPF assumes that (1) every action takes exactly one time step, (2) time is discretized into time steps, as oppose to continuous, and (3) each agent occupies exactly one vertex. These assumptions do not necessarily hold in real-world MAPF applications. With the maturity of classical MAPF algorithms, recent years have also begun to explore MAPF problems that relax these assumptions. Below, we provide a partial overview of these efforts.

6.1 Beyond One-Time Step Actions

The eICTS algorithm [48] mentioned above is designed for actions that may require more than one time step. Such a setting is sometimes called MAPF with non-unit edge cost. Adapting the CBS algorithm to non-unit edge cost settings is straightforward, as it only requires changing the conflict-detection step.

Barták et al. [5] proposed a CP-based algorithm for MAPF with non-unit edge costs. Their model uses scheduling constraints to support actions with different duration.

6.2 Beyond Discrete Time Steps

Time is continuous, and thus every time step discretization is, by definition an abstraction of the real-world. This abstraction in the context of MAPF may lead to suboptimality and even incompleteness.

As long as the agents do not need to wait, there is no need to directly deal with this problem: the duration of move actions depend on the time required to traverse the corresponding edge. However, when an agent needs to wait and time is not discretized, then each agent has an infinite number of possible wait actions in each vertex.

The key technique used so far to address this problem is to use the Safe Interval Path Planning (SIPP) [29] algorithm. SIPP is a single-agent pathfinding algorithm that is designed to avoid moving obstacles. Since obstacles are moving, the single agent may choose to wait in its location, which raises again the challenge of dealing with continuous time. SIPP addresses this challenge by identifying *safe intervals* in which the agent can occupy each vertex, and runs an A* search on (vertex, safe interval) pairs. Andreychuck et al. showed how to use SIPP to solve MAPF problems with continuous time, in a prioritized planning framework [51] and in a CBS framework [2].

Surynek [41] recently proposed to use a CP-related approach for continuous time. Instead of modeling the problem as a CSP or SAT problem, Surynek proposed to model it as a SAT Modulu Theory (SMT) problem, and then apply an SMT solver.

6.3 Beyond One-Agent per Vertex

The graph G of possible location in classical MAPF is an abstraction of the real world the agents are moving in. Arguably, in most real-world MAPF applications the agents are moving in Euclidean space and have some geometric shape. Thus, an agent may conflict if they stop in different areas, because their geometric shapes overlap. Li et al. [22] referred to this as *MAPF with large agents*. In such settings, an agent may "occupy" multiple vertices and a move action may create a conflict with agents occupying multiple vertices.

Li et al. [22] proposed a CBS-based algorithm for addressing this setting. They showed how to design suitable constraints for large agents and proposed an admissible heuristic to speedup the search. They also described an A*-based

algorithm and a SAT-based algorithm for this setting. Atzmon et al. [12] proposed another CBS-based algorithm that can consider agents of arbitrary shape, even without a reference point that is stable to rotations.

Robustness and Kinematic Constraints. Even if an agent only occupies a single vertex, it is still desirable in many scenarios to add a buffer around each agent to further minimize the chance of collisions. Such a buffer can be either spatial or temporal. A prime motivation for having such a buffer is to account for the inherent uncertainty during the executing of the solution. That is, to have the agents' joint plan be valid and executable even if some agents do not fully follow it.

The MAPF-POST [19] algorithm was designed to address such requirements. MAPF-POST accepts as input a solution for a classical MAPF problem and adapts it to consider safety and kinematic constraints. A limitation of MAPF-POST is that it does not retain any guarantee on solution quality. For adding robustness to temporal delays during execution, Atzmon et al. [3] proposed an optimal CBS-based algorithm and CP-based algorithm.

6.4 Beyond One-Shot MAPF

In addition, classical MAPF is a one-shot, offline problem. In some MAPF applications, there is a sequence of related MAPF problems that are being solved sequentially. Some recent work also addresses several types of *online* MAPF settings. This includes settings where there is a fixed set of agents and a stream of pathfinding tasks [25], as well as a setting where new agents appear over time but each agent has a single navigation task [46]. The former setting is referred to as the *MAPF warehouse model* and the latter as the *MAPF intersection model*.

Also, so far we assumed the allocation of agents to goals is given. In the *Multi-Agent Pickup-and-Delivery* (MAPD) problem, this is not the case [23]. In MAPD, there is a fixed set of agents that need to solve a batch of pickup and delivery of tasks. A MAPD algorithm needs to plan paths without conflicts, and also to allocate which agent should go to which destination.

7 Conclusion

This paper provides an overview of the current research on Multi-Agent Path Finding (MAPF). After providing several definitions of MAPF were given, we presented polynomial-time algorithms for solving the problem. Then, a range of algorithms was described that return optimal solutions. These algorithms can be split into four families: A*-based, ICTS, CBS, and CP. Following, we described how to transform several of these optimal algorithms to be approximately optimal algorithms, allowing trading solution quality for runtime. Finally, we presented some extensions of classical MAPF, including non-unit edge costs, continuous time, large agents, and online MAPF. Throughout this paper, we suggested several directions for future work.

It is our hope that this paper will be useful to both researchers and practitioners looking for a brief introduction to MAPF. For formal definitions of MAPF variants and benchmarks, see [37]. For additional MAPF-related resources, including pointers to publications and additional tutorials, see the http://mapf.info web site, created by Sven Koenig's group.

References

1. Andreychuk, A., Yakovlev, K.: Two techniques that enhance the performance of multi-robot prioritized path planning. In: International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), pp. 2177–2179 (2018)
2. Andreychuk, A., Yakovlev, K., Atzmon, D., Stern, R.: Multi-agent pathfinding with continuous time. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 39–45 (2019)
3. Atzmon, D., Stern, R., Felner, A., Wagner, G., Barták, R., Zhou, N.F.: Robust multi-agent path finding. In: International Conference on Autonomous Agents and Multi Agent Systems (AAMAS), pp. 1862–1864 (2018)
4. Barer, M., Sharon, G., Stern, R., Felner, A.: Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In: Symposium on Combinatorial Search (SoCS) (2014)
5. Barták, R., Švancara, J., Vlk, M., et al.: A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs. In: International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), pp. 748–756. International Foundation for Autonomous Agents and Multiagent Systems (AAMAS) (2018)
6. Barták, R., Zhou, N., Stern, R., Boyarski, E., Surynek, P.: Modeling and solving the multi-agent pathfinding problem in picat. In: IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pp. 959–966 (2017)
7. Bnaya, Z., Felner, A.: Conflict-oriented windowed hierarchical cooperative A. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 3743–3748 (2014)
8. Boyarski, E., et al.: ICBS: improved conflict-based search algorithm for multi-agent pathfinding. In: International Joint Conference on Artificial Intelligence (IJCAI) (2015)
9. Čáp, M., Vokřínek, J., Kleiner, A.: Complete decentralized method for on-line multi-robot trajectory planning in well-formed infrastructures. In: International Conference on Automated Planning and Scheduling (ICAPS) (2015)
10. Cohen, L., Uras, T., Koenig, S.: Feasibility study: using highways for bounded-suboptimal multi-agent path finding. In: Symposium on Combinatorial Search (SoCS) (2015)
11. De Wilde, B., Ter Mors, A.W., Witteveen, C.: Push and rotate: a complete multi-agent pathfinding algorithm. J. Artif. Intell. Res. **51**, 443–492 (2014)
12. Atzmon, D., Diei, A., Rave, D.: Multi-train path finding. In: Symposium on Combinatorial Search (SoCS) (2019)
13. Erdem, E., Kisa, D.G., Oztok, U., Schüller, P.: A general formal framework for pathfinding problems with multiple agents. In: AAAI Conference on Artificial Intelligence (2013)
14. Felner, A., et al.: Adding heuristics to conflict-based search for multi-agent path finding. In: International Conference on Automated Planning and Scheduling (ICAPS) (2018)

15. Felner, A., et al.: Search-based optimal solvers for the multi-agent pathfinding problem: summary and challenges. In: Symposium on Combinatorial Search (SoCS), pp. 29–37 (2017)
16. Gilon, D., Felner, A., Stern, R.: Dynamic potential search-a new bounded suboptimal search. In: Symposium on Combinatorial Search (SoCS) (2016)
17. Goldenberg, M., Felner, A., Sturtevant, N.R., Holte, R.C., Schaeffer, J.: Optimal-generation variants of EPEA. In: SoCS (2013)
18. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **SSC-4**(2), 100–107 (1968)
19. Hönig, W., et al.: Summary: multi-agent path finding with kinematic constraints. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 4869–4873 (2017)
20. Kornhauser, D., Miller, G., Spirakis, P.: Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In: Symposium on Foundations of Computer Science, pp. 241–250. IEEE (1984)
21. Li, J., Harabor, D., Stuckey, P., Felner, A., Ma, H., Koenig, S.: Disjoint splitting for multi-agent path finding with conflict-based search. In: International Conference on Automated Planning and Scheduling (ICAPS) (2019)
22. Li, J., Surynek, P., Felner, A., Ma., H., Kumar, T.K.S., Koenig, S.: Multi-agent path finding for large agents. In: AAAI Conference on Artificial Intelligence (2019)
23. Liu, M., Ma, H., Li, J., Koenig, S.: Task and path planning for multi-agent pickup and delivery. In: International Conference on Autonomous Agents and MultiAgent Systems (AAMAS), pp. 1152–1160 (2019)
24. Luna, R., Bekris, K.E.: Efficient and complete centralized multi-robot path planning. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3268–3275 (2011)
25. Ma, H., Li, J., Kumar, T., Koenig, S.: Lifelong multi-agent path finding for online pickup and delivery tasks. In: Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 837–845 (2017)
26. Ma, H., Yang, J., Cohen, L., Kumar, T.K.S., Koenig, S.: Feasibility study: moving non-homogeneous teams in congested video game environments. In: Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), pp. 270–272 (2017)
27. Morris, R., et al.: Planning, scheduling and monitoring for airport surface operations. In: AAAI Workshop: Planning for Hybrid Systems (2016)
28. Pearl, J., Kim, J.H.: Studies in semi-admissible heuristics. IEEE Trans. Pattern Anal. Mach. Intell. **PAMI-4**, 392–399 (1982)
29. Phillips, M., Likhachev, M.: SIPP: safe interval path planning for dynamic environments. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 5628–5635 (2011)
30. Pohl, I.: Heuristic search viewed as path finding in a graph. Artif. Intell. **1**(3–4), 193–204 (1970)
31. Sajid, Q., Luna, R., Bekris, K.E.: Multi-agent pathfinding with simultaneous execution of single-agent primitives. In: SoCS (2012)
32. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artif. Intell. **219**, 40–66 (2015)
33. Sharon, G., Stern, R., Goldenberg, M., Felner, A.: The increasing cost tree search for optimal multi-agent pathfinding. Artif. Intell. **195**, 470–495 (2013)
34. Silver, D.: Cooperative pathfinding. In: AIIDE, vol. 1, pp. 117–122 (2005)

35. Srinivasan, A., Ham, T., Malik, S., Brayton, R.K.: Algorithms for discrete function manipulation. In: IEEE International Conference on Computer-Aided Design, pp. 92–95 (1990)
36. Standley, T.S.: Finding optimal solutions to cooperative pathfinding problems. In: AAAI Conference on Artificial Intelligence, pp. 173–178 (2010)
37. Stern, R., et al.: Multi-agent pathfinding: definitions, variants, and benchmarks. In: Symposium on Combinatorial Search (SoCS) (2019)
38. Surynek, P.: A novel approach to path planning for multiple robots in bi-connected graphs. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 3613–3619 (2009)
39. Surynek, P.: An optimization variant of multi-robot path planning is intractable. In: AAAI (2010)
40. Surynek, P.: Makespan optimal solving of cooperative path-finding via reductions to propositional satisfiability. arXiv preprint arXiv:1610.05452 (2016)
41. Surynek, P.: Multi-agent path finding with continuous time viewed through satisfiability modulo theories (SMT). arXiv preprint arXiv:1903.09820 (2019)
42. Surynek, P., Felner, A., Stern, R., Boyarski, E.: Efficient sat approach to multi-agent path finding under the sum of costs objective. In: European Conference on Artificial Intelligence (ECAI), pp. 810–818 (2016)
43. Thayer, J.T., Ruml, W.: Bounded suboptimal search: a direct approach using inadmissible estimates. In: International Joint Conference on Artificial Intelligence (IJCAI) (2011)
44. Van Roy, T.J., Wolsey, L.A.: Solving mixed integer programming problems using automatic reformulation. Oper. Res. **35**(1), 45–57 (1987)
45. Veloso, M.M., Biswas, J., Coltin, B., Rosenthal, S.: CoBots: robust symbiotic autonomous mobile service robots. In: IJCAI, p. 4423 (2015)
46. Švancara, J., Vlk, M., Stern, R., Atzmon, D., Barták, R.: Online multi-agent pathfinding. In: AAAI Conference on Artificial Intelligence (2019)
47. Wagner, G., Choset, H.: Subdimensional expansion for multirobot path planning. Artif. Intell. **219**, 1–24 (2015)
48. Walker, T.T., Sturtevant, N.R., Felner, A.: Extended increasing cost tree search for non-unit cost domains. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 534–540 (2018)
49. Wang, K.H.C., Botea, A.: MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. J. Artif. Intell. Res. **42**, 55–90 (2011)
50. Wurman, P.R., D'Andrea, R., Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. AI Mag. **29**(1), 9 (2008)
51. Yakovlev, K., Andreychuk, A.: Any-angle pathfinding for multiple agents based on SIPP algorithm. In: International Conference on Automated Planning and Scheduling (ICAPS), pp. 586–593 (2017)
52. Yu, J., LaValle, S.M.: Multi-agent path planning and network flow. In: Frazzoli, E., Lozano-Perez, T., Roy, N., Rus, D. (eds.) Algorithmic Foundations of Robotics X. STAR, vol. 86, pp. 157–173. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36279-8_10
53. Yu, J., LaValle, S.M.: Structure and intractability of optimal multi-robot path planning on graphs. In: AAAI (2013)
54. Zhou, N.-F., Kjellerstrand, H., Fruhman, J.: Constraint Solving and Planning with Picat. Springer Briefs in Intelligent Systems. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25883-6