Load Balancing in Distributed Multi-Agent Path Finder (DMAPF)

Poom Pianpak[1], Jiaoyang Li[2], and Tran Cao Son[1]

[1] New Mexico State University, Las Cruces, NM, USA
{ppianpak,tson}@cs.nmsu.edu
[2] Carnegie Mellon University, Pittsburgh, PA, USA
jiaoyangli@cmu.edu

Abstract. The Multi-Agent Path Finding (MAPF) is a problem of finding a plan for agents to reach their desired locations without colliding. Distributed Multi-Agent Path Finder (DMAPF) solves the MAPF problem by decomposing a given MAPF problem instance into smaller subproblems and solve them in parallel. DMAPF works in rounds. Between two consecutive rounds, agents may migrate between two adjacent subproblems following their abstract plans, which are pre-computed, until all of them reach the areas that contain their desired locations. Previous works on DMAPF compute an abstract plan for each agent without the knowledge of other agents' abstract plans, resulting in high congestion in some areas, especially those that act as corridors. The congestion negatively impacts the runtime of DMAPF and prevents it from being able to solve dense MAPF problems.
In this paper, we (*i*) investigate the use of Uniform-Cost Search to mitigate the congestion. Additionally, we explore the use of several other techniques including (*ii*) using timeout estimation to preemptively stop solving and relax a subproblem when it is likely to get stuck; (*iii*) allowing a solving process to manage multiple subproblems – aimed to increase concurrency; and (*iv*) integrating with MAPF solvers from the Conflict-Based Search family. Experimental results show that our new system is several times faster than the previous ones; can solve larger and denser problems that were unsolvable before; and has better runtime than PBS and EECBS, which are state-of-the-art centralized suboptimal MAPF solvers, in problems with a large number of agents.

Keywords: Multi-Agent Path Finding (MAPF) · Distributed Multi-Agent Path Finder (DMAPF) · Load Balancing · Distributed Computing.

1 Introduction

Multi-Agent Path Finding (MAPF) is a problem of finding collision-free paths for a team of agents to move from their initial locations to desired locations. It plays an important role in the field of human-robot interaction where humans and robots collaborate and has important applications in autonomous warehouse [13, 8], traffic management [7], and video games [15], etc. The problem is known to

be NP-hard to solve optimally [30]; therefore, a sacrifice on solution quality is usually made to make the MAPF solver practical.

Two main approaches to solving the MAPF problem are (*i*) search-based [5] and (*ii*) compilation-based [27]. Search-based MAPF solvers focus on developing search algorithms for MAPF problems. Some of the most prominent search-based techniques include conflict-based search [23], where conflicts between single-agent plans are detected by a high-level search on a constraint tree and resolved by a low-level search; and prioritized planning [25, 14], where agents with lower priority need to avoid conflicts with agents with higher priority. Compilation-based MAPF solvers translate the MAPF problem into another well-established formulation such as Answer Set Programming (ASP) [17, 16], Boolean Satisfiability (SAT) [2], and Constraint Satisfaction (CSP) [4], for which efficient solvers exist.

Distributed Multi-Agent Path Finder (DMAPF) [21, 19, 20] is our framework that solves the MAPF problem by applying the divide-and-conquer idea. It decomposes a given MAPF problem instance into smaller subproblems; assigns the subproblems to solving processes – which can run on a single or multiple machines; then uses an existing MAPF solver, in any approach mentioned, to solve the smaller MAPF problem instances. The partial solutions from every solving process are combined at the end to provide a solution to the original problem.

In this paper, we introduce several mechanisms to scale up DMAPF and improve its efficiency. More specifically,

1. Improved Abstract Planning – We investigate the use of Uniform-Cost Search to make abstract plans in an attempt to mitigate congestion at a high level. An *abstract plan* of an agent is a sequence of subproblems that the agent needs to traverse to reach the area that contains its desired location. This enables DMAPF to take on denser maps as it decreases the chance of being in a situation where no agent is able to progress to the next subproblem in its abstract plan because the next subproblem of every agent is overcrowded. It also reduces the runtime because the MAPF problem instances to solve tend to be less dense. See Subsection 3.1 for details.
2. Timeout Mechanism – We introduce a timeout estimation mechanism to allow DMAPF to preempt its underlying MAPF solver from solving subproblem instances that are likely to take a prohibitively long time to solve. Any subproblem instance that is stopped will be relaxed by having some of its agents' targets temporarily removed. Then, it will be solved again until either a plan is found or it cannot be relaxed further. This helps to prevent DMAPF from getting stuck on subproblem instances that would be unsolvable without the relaxation, thus, improving the success rate. In many cases, it also improves the overall runtime as it tends to be faster to avoid solving difficult subproblem instances. See Subsection 3.2 for details.
3. Multiple Subproblems Assignment – We extend our previous work on DMAPF in [20] by allowing each solving process to manage multiple subproblems instead of one. This enables DMAPF to handle MAPF problem instances of any size as it would not be restricted by the number of subproblems, which

corresponds one-to-one to the number of solving processes in the old design. This improvement has a significant impact on the applicability of DMAPF, but is purely engineering. It involved heavy re-organization of the code base; therefore, we omit the details here. Instead, its implications can be seen from the experimental results in Subsection 4.1.

4. Integration with CBS-based MAPF Solvers – We investigate the use of (*i*) CBSH2-RTC [11]; (*ii*) EECBS [12]; and (*iii*) PBS [14], as an underlying MAPF solver for DMAPF, in addition to ASP that is used in our previous works. The requirements for integrating a MAPF solver with DMAPF and modifications to the CBS-based MAPF solvers are explained in Subsection 3.3.

2 Background

2.1 Multi-Agent Path Finding

The MAPF problem can be defined as $P = (G, A, I, T)$, where $G = (V, E)$ is a graph such that V is a set of vertices corresponding to locations in the graph; $E \subseteq V \times V$ denotes pairs of locations where agents can traverse in some direction; A is a set of agents; and $I, T \subseteq A \times V$ denote start and goal locations of the agents, respectively. An agent at location v_1 can either *move* from v_1 to v_2 in one timestep if $(v_1, v_2) \in E$ or *stay* at v_1. The most common restrictions are that (*i*) each location can be occupied by at most one agent at a time; and (*ii*) two agents cannot swap locations in a single timestep. Violating any of these restrictions is said to cause a *conflict*. A solution to a MAPF problem instance is a set of movement plans (i.e., a sequence of vertices) for every agent that allows them to go to their goal locations without causing the conflict. The quality of a solution is usually measured in terms of (*i*) *makespan* – the longest length of the movement plans in the solution; and (*ii*) *sum-of-cost* – the sum of lengths of the movement plans in the solution.

There are several variants of the MAPF problem [26]. DMAPF follows the mentioned restrictions and assumes that every agent has unique start and goal locations; and they need to stay at their goals at the end of the solution.

2.2 Distributed Multi-Agent Path Finder

Distributed Multi-Agent Path Finder (DMAPF) applies the divide-and-conquer idea to solve the MAPF problem. Given a MAPF problem instance P, DMAPF partitions P into a set of smaller subproblems $S = \{S_1, \ldots, S_n\}$. A subproblem S_i is defined as $((V_i, E_i), A_i, I_i, T_i)$ where $V_i \subseteq V$, $E_i \subseteq E$, $A_i \subseteq A$, $I_i \subseteq I$, and $T_i \subseteq T$. Pairs of locations in E_i are only between vertices in V_i; agents in A_i are only those that have their start location in V_i; and start and goal locations respectively in I_i and T_i are only for agents in A_i. In our previous works, each solving process is only assigned one different subproblem in S. In this work, we extend the system to allow assigning multiple subproblems to each solving process, provided that every subproblem is only assigned to one solving process.

Solving processes work together in parallel. Every solving process has full knowledge of adjacency between all the subproblems. Subproblems S_1 and S_2 are adjacent and are called *neighbors* iff there exists vertices $v_1 \in S_1$ and $v_2 \in S_2$ such that v_1 and v_2 are adjacent (i.e., $(v_1, v_2) \in E$). In addition, each solving process knows every vertex in its assigned subproblems that is adjacent to a neighboring subproblem. DMAPF allows subproblems to contain sets of disconnected vertices called *areas* and operates on them, but for simplicity, we will use the term subproblem throughout the paper unless a clear distinction is required.

Every solving process starts by creating an abstract plan for each agent residing in any of its assigned subproblems. Figure 1 shows an example of a MAPF problem instance decomposed into 4 subproblems: S_1, S_2, S_3, and S_4. Suppose that subproblem S_1 is assigned to a solving process s, then s has the responsibility to create abstract plans for agents a_1 and a_2 to reach subproblem S_4 that contains their goal locations g_1 and g_2, respectively. Possible abstract plans for agents a_1 and a_2 are $\langle S_1, S_2, S_4 \rangle$, and $\langle S_1, S_3, S_4 \rangle$.

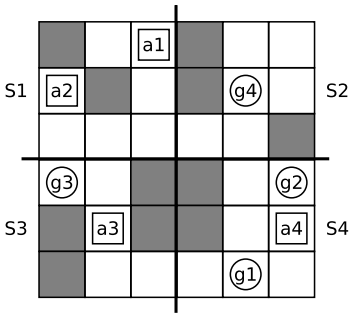


Fig. 1. An example of how a MAPF problem instance is decomposed into 4 subproblems: S_1, S_2, S_3, and S_4. Start and goal locations of agents are denoted by small squares and circles with corresponding numbers, respectively.

After an abstract plan has been created for each agent, solving processes work together round-by-round, following the protocol described in [19, 20]. Let N contain pairs of solving processes that have adjacent subproblems (i.e., they are called *neighbors*). The protocol consists of 3 phases: (*i*) *negotiation* – every pair in N decides which agents to *migrate* (i.e., progress to the next subproblem in the abstract plan) and to which border locations. Priority is given to agents with the longest remaining steps in the abstract plan and border locations are chosen such that the aggregate distance between agents and their assigned border locations is minimized; (*ii*) *rejection* – every pair in N detects which previously-agreed migrations will result in collision and rejects them. This ensures a collision-free migration agreement across all subproblems; and (*iii*) *confirmation* – every pair in N confirms agents that can successfully move to their assigned border locations. The agreed adjacent border locations, which are in their next subproblems, will be used as their start locations in the next round. The protocol allows solv-

ing processes to either solve or relax (see Subsection 3.2) their own subproblem instances **in parallel** between the *rejection* and the *confirmation* phases.

The algorithm terminates when either (*i*) a plan is found where at the end every agent stays at its goal location – the solution is then reported; or (*ii*) there is a subproblem instance that cannot be solved nor relaxed further – the system then reports that it cannot find a solution.

2.3 The CBS Family

CBSH2-RTC [11] is a state-of-the-art version of Conflict-Based Search (CBS) [23]. CBS is an optimal search-based MAPF solver where a path for each agent is individually planned from its start to goal location using a space-time A* search [25] at a low level. Conflicts between agent plans are detected in a high-level search on a constraint tree. They are resolved in a low-level search by making new plans for a subset of conflicting agents that avoid the imposed constraints. CBSH2-RTC introduces several improvements that make CBS smarter in determining which conflict to resolve first, and how, using various heuristics. CBSH2-RTC is well known for its performance compared to other optimal MAPF solvers.

EECBS [12] improves on the idea of ECBS [1], which is a bounded-suboptimal variant of CBS, by replacing focal search that acts as a high-level search in ECBS with Explicit Estimation Search [28]. It uses online learning to guide the search and employs various techniques that have been used to improve CBS. It has recently been improved by replacing the space-time A* that is used as a low-level search in ECBS with SIPPS [10], allowing EECBS to be even more efficient.

PBS [14] is a suboptimal MAPF solver that uses the idea of prioritized planning [25] where agents are given different priorities and those with lower priority need to avoid higher-priority agents. Instead of planning based on some fixed priority ordering, PBS is able to (lazily) explore all total priority orderings. PBS is not complete, but very efficient, and able to find solutions for many MAPF instances where standard prioritized MAPF algorithms cannot.

3 Methodology

3.1 Abstract Planning Methods

In addition to the ASP encoding used for creating abstract plans in our previous works, we introduce 4 new abstract planning methods to DMAPF using (*i*) Breadth-First Search (BFS); (*ii*) Random Search (RAND); (*iii*) Uniform-Cost Search (UCS); and (*iv*) Centralized Uniform-Cost Search (UCSC).

Let F be a frontier containing sequences of areas that have not yet been explored. To find an abstract plan for an agent a, an initial plan containing only the area where agent a starts from is added to F. Then, one of the plans in F is removed and checked whether the last area in the plan contains the goal location of agent a. If not, new plans are created and added to F by extending the plan with every one of the adjacent areas that have not been explored. This process repeats until either an abstract plan is found or F is exhausted.

In BFS, F acts as a queue, so plans are selected in the order when they were added to F, resulting in the shortest abstract plan. RAND only differs from BFS in that F acts as a set instead of a queue, so a plan is randomly selected in each iteration, resulting in an abstract plan that may not be the shortest.

In UCS and UCSC, F acts as a priority queue where the ordering (lowest-first) is based on the congestion within each area in a plan. We define *congestion* within an area a at an abstract step t as n/v, where n is the number of agents in a at abstract step t; and v is the number of vertices in a. The overall congestion is tracked using a *congestion matrix* which contains congestion within every area at every abstract step. In UCS, every solving process makes an abstract plan for each agent in its assigned subproblems and uses the plan to update its congestion matrix locally in each iteration. In UCSC, only one solving process is designated to make an abstract plan for each agent in the original problem P and use the plan to update a single congestion matrix in each iteration. At the end, the plans are distributed to each responsible solving process. Resulting abstract plans from UCSC create less overall congestion than UCS since only a single congestion matrix is consulted and updated.

An example of how a congestion matrix is updated in each iteration by UCSC is shown in Figure 2 where it takes the problem from Figure 1 and sequentially updates it with abstract plans of agents $a1$, $a2$, $a3$, and $a4$, which are $\langle S1, S2, S4\rangle$, $\langle S1, S3, S4\rangle$, $\langle S3\rangle$, and $\langle S4, S2\rangle$, respectively.

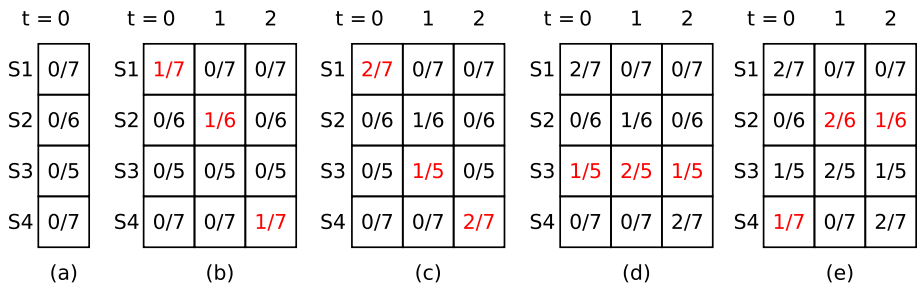


Fig. 2. An illustration on how a congestion matrix is updated over time by UCSC. (a) initial. (b) with $\langle S1, S2, S4\rangle$. (c) with $\langle S1, S3, S4\rangle$. (d) with $\langle S3\rangle$. (e) with $\langle S4, S2\rangle$. Updated values are highlighted in red.

3.2 Timeout Mechanism

For a MAPF subproblem instance S_i to be processed, there must exist some agent with a *target*, either its original goal or an assigned border location in S_i. If there is some agent with an assigned border location in S_i, then the other agents without an assigned border location will be considered as having no goal

Algorithm 1 Solving a MAPF problem instance with timeout

Input: S_i – MAPF subproblem instance; n – #agents in S_i
Parameter: t_a – Approximate timeout per agent
f – Timeout penalty factor; ϵ – Timeout tolerance factor
Output: *sol* – Solution of S_i

```
1: while true do
2:   if S_i is solved within n · t_a · ε then
3:     if Some agent in S_i has a goal or border location assigned then
4:       t_a ← t_s/n where t_s is the time used to solve S_i
5:     else
6:       t_a ← f · t_a
7:     return sol
8:   else
9:     Stop solving S_i
10:    if Some agent in S_i has a border location assigned then
11:      Remove an assigned border location from one agent in S_i
12:    else
13:      terminate
```

(in the current round) to create the least constraint for the agents with assigned border locations to reach their targets.

Algorithm 1 shows the timeout estimation mechanism added to the subproblem solving procedure. Line 2 tries to solve S_i within the time limit of $n \cdot t_a \cdot \epsilon$, where n, t_a, and ϵ are the number of agents, an approximate timeout per agent, and a timeout tolerance factor, respectively. The value of ϵ is a multiplicative constant that accommodates errors from the approximation. If S_i is solved where some agent has a target, then t_a is re-estimated to the time used to solve S_i per agent (Line 4). However, if S_i is solved but there is no agent with a target, it means that S_i is has been relaxed too much. It then will be tried to solve again in the next round with a higher timeout limit of $f \cdot t_a$ where f is another multiplicative constant greater than 1 (Line 6). If S_i cannot be solved within the time limit, then the MAPF solver is stopped (Line 9) and S_i is checked whether it is relaxable (Line 10). S_i can be relaxed if it has some agent that needs to migrate and is assigned with a border location. Line 11 relaxes S_i by removing an assigned border location from one of the migrating agents. The heuristic is to select an agent with the least number of steps left in its abstract plan. Otherwise, DMAPF terminates at Line 13 and reports that it cannot find a solution.

3.3 Integration with CBS-based MAPF Solvers

To integrate a MAPF solver with DMAPF, it needs to satisfy the requirements that: (*i*) agents without a goal location are allowed in the problem; and (*ii*) agents need to be able to avoid being in a set of certain vertices V_P at the end of the plan unless they need to go to a location in V_P. The second condition accommodates the design of DMAPF that improves its success rate by making sure there are unoccupied vertices for migrating agents to move in.

CBSH2-RTC, EECBS, PBS all use an A*-style algorithm (i.e., space-time A* [25], SIPP [18], or SIPPS [10]) to plan paths for individual agents. We modify the heuristic function and the goal test function for agents without goals as follows. For the heuristic function, the h-value of a node at a vertex that is in the prohibited set V_P is 1, and the h-value of a node at a vertex that is not in V_P is 0. For the goal test function, we claim a node at vertex v at timestep t as a goal node iff vertex v is not in V_P and there are no vertex constraints that prohibit this agent from being at vertex v at any timestep after timestep t.

In addition, CBSH2-RTC and EECBS use some speedup techniques that rely on the assumption that the agents have unique goal locations. We therefore turn off those techniques when the involved agents do not have goals. Specifically, they both build MDDs [24], i.e., a direct acyclic graph that consists of all shortest paths from the start vertex to the goal vertex of an agent, for individual agents, which are used for finding cardinal conflicts [3] and rectangle conflicts [11]. We do not build MDDs for agents without goals. Thus, if such agents are involved in a conflict, we classify this conflict as semi-cardinal or non-cardinal (depending on how the MDD of the other agents involved in the conflict looks like), and do not perform rectangle reasoning for it. Moreover, target reasoning [11] happens when an agent runs into another agent that has already reached its goal location and sat there, so we perform target reasoning only if the second agent has been assigned a goal vertex.

4 Experiments

We conduct experiments in Subsections 4.1-4.4 sequentially to determine the best parameters for DMAPF on our machine. Subsection 4.1 determines the optimal number of solving processes to be executed in parallel. Subsection 4.2 determines the optimal size of subproblems that gives the best tradeoff between performance and success rate. Subsection 4.3 determines the optimal sensitivity of timeout that allows DMAPF to appropriately stop its underlying MAPF solver. Subsection 4.4 determines the abstract planning method that computes abstract plans with the least overall congestion. Finally, Subsection 4.5 uses the best parameters obtained from the previous subsections to compare variations of DMAPF with CBSH2-RTC [11], EECBS [12], and PBS [14].

The experiments are performed on a Dell Precision 3630 Tower with an Intel Core i9-9900K @3.60 GHz and 64 GB of RAM. The software used includes Ubuntu 20.04.5 LTS, ROS Noetic Ninjemys [22], and Clingo 5.6.2 [6]. We use maps and random scenarios from the MAPF benchmark[3] [26]. The scenarios have at most 1000 agents (limit). Each agent in the scenarios has unique start and goal locations. The maps used in our experiments are *den312d*, *random-64-64-20*, *maze-128-128-2*, *lak303d*, and *warehouse-20-40-10-2-2*, which contain 2445, 3270, 10858, 14784, and 38756 vertices, respectively. Unless stated otherwise, we use 20 solving processes, subproblems that contain roughly 40 vertices, timeout

[3] https://movingai.com/benchmarks/mapf/index.html

penalty factor (f) of 2, timeout tolerance factor (ϵ) of 10, centralized Uniform-Cost-Search to make abstract plans, and ASP as an underlying MAPF solver in DMAPF. To automatically decompose a map into subproblems that contain roughly 40 vertices, for example, we would specify $\lfloor v/40 \rfloor$ where v is the number of vertices in the map, as the number of desired subproblems to the problem divider which is implemented using balanced k-means with real distance [20].

For the reproducibility of our results, the experiments in the following subsections also state seed values used by the problem divider. Because the performance of DMAPF greatly depends on how the input map is decomposed, the seed values used to decompose the maps in Subsections 4.2-4.5 are chosen from 101 to 110 for the one that gives the best runtime in the first scenario. Then, the maps decomposed with the chosen seed values will be used throughout the whole experiments. The reported values come from an average of running each random scenario from 1-10 once in the same (decomposed) map, under the time limit of 5 minutes, for the total of 10 times.

4.1 The Numbers of Solving Processes

Table 1 attempts to determine the optimal number of solving processes by comparing runtimes of DMAPF using 4, 8, 12, ..., 32 solving processes on lak303d map with 200, 400 and 600 agents from the first random scenario. The map is decomposed into 240 subproblems using the seed value of 2. Every reported runtime is averaged from solving the first random scenario 10 times.

Table 1. Comparing runtimes of DMAPF using p solving processes on lak303d map with n agents.

n	Runtime (s)							
	$p = 4$	$p = 8$	$p = 12$	$p = 16$	$p = 20$	$p = 24$	$p = 28$	$p = 32$
200	32.1	25.1	21.8	20.0	19.4	**18.9**	20.5	19.6
400	97.0	75.1	63.1	66.2	**52.8**	56.2	53.8	63.5
600	214.2	158.7	129.3	127.5	**110.7**	113.3	116.0	120.0

On our machine that is equipped with a CPU that has 8 cores and 16 hardware threads, the results suggest that using 20 - 24 solving processes, or 125% - 150% of the number of hardware threads provides the best runtime. Using too few solving processes underutilizes the computational resources and using too many solving processes introduces too much competition for the resources, which are both detrimental to the performance.

4.2 The Size of Subproblems

Table 2 attempts to determine the optimal size of subproblems by comparing runtimes, makespan, sum-of-cost, and success rate of DMAPF on random-64-64-20 map that has been decomposed into subproblems of different sizes: 30, 40,

Table 2. Comparing runtimes, makespan, sum-of-cost, and success rates of DMAPF on random-64-64-20 map, decomposed into subproblems that contain roughly v vertices, with 1000 agents.

v	Runtime (s)	Makespan	SoC ($\times 1k$)	Success Rate
30	**39.9**	**864**	**558.4**	0.4
40	46.6	906	598.5	**1.0**
50	88.2	1041	627.9	0.8
60	80.0	1107	676.6	**1.0**
70	86.8	1070	633.6	**1.0**

50, 60, and 70 vertices, using the seed values of 107, 105, 101, 109, and 105, respectively. DMAPF with ASP as an underlying MAPF solver is optimized for makespan; therefore, makespan is a better indicator of solution quality than sum-of-cost.

The results show that, with small subproblems, DMAPF tends to run faster and give better solution quality, but have a lower success rate. DMAPF runs faster in small subproblems because ASP, which its runtime is known to be very sensitive to the number of vertices, is less affected by the sizes of the subproblems since they are small. It also gives better solution quality because as the subproblems become more fine-grained, it results in less agents waiting to move between consecutive rounds. However, the success rate is now lower because there is more chance that some subproblem instance becomes unsolvable as the ratio b/v, where b is the number of border vertices (i.e., vertices that are adjacent to vertices in another subproblem) and v is the number of vertices in the subproblem, increases. In DMAPF, agents follow their abstract plans to move into the next subproblems between two consecutive rounds. The greater the ratio b/v is, the more agents can enter (or leave) subproblems while the subproblems may contain only a few vertices, making it difficult (or impossible) to find a movement plan. Our results are consistent with the original work on DMAPF [21] that suggests that the size of subproblems around 40 - 60 vertices provide the best performance and solution quality.

4.3 Timeout Sensitivity

Table 3 attempts to determine the optimal value of the timeout tolerance factor ϵ by comparing runtimes, makespan, sum-of-cost, and the number of times that DMAPF preemptively stops its underlying MAPF solver because it exceeds the estimated timeout limit, under different values of ϵ. The greater the value of ϵ is, the longer DMAPF allows each subproblem instance to be solved. We decompose random-64-64-20 map with the seed value of 105 to use in this experiment.

The results show that setting the value of ϵ too small (i.e., $\epsilon < 10$ in Table 3) causes DMAPF to be too sensitive and stops its MAPF solver too early, resulted in worse performance. However, when the value of ϵ is too big such as when $\epsilon = 14$, DMAPF waits too long to stop its MAPF solver from solving problem instances that are likely to be too difficult, which also resulted in

Table 3. Comparing runtimes, makespan, sum-of-cost, and the number of times DMAPF stops its an underlying MAPF solver under different timeout tolerance factors ϵ, on random-64-64-20 map with 1000 agents.

ϵ	Runtime (s)	Makespan	SoC ($\times 1k$)	#Stops
4	51.9	906	611.9	25
6	49.2	906	600.1	13
8	48.6	**903**	607.3	7
10	46.6	906	**598.5**	3
12	**46.3**	906	603.2	**1**
14	49.1	908	606.8	2

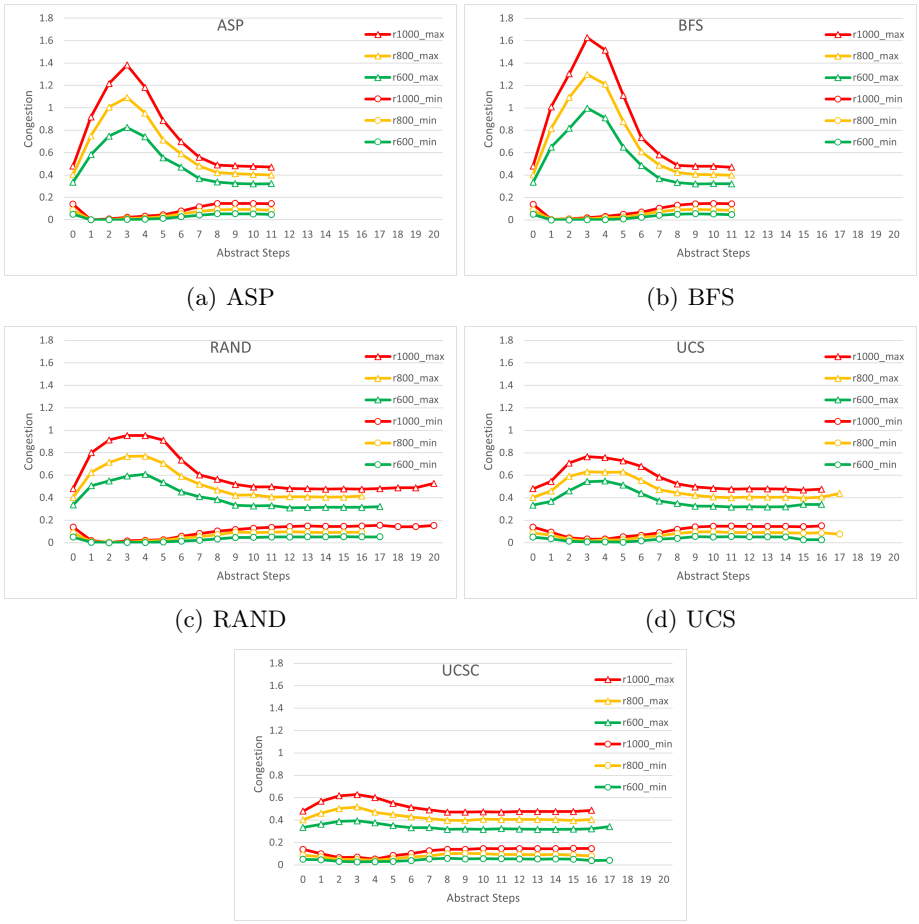
worse performance. The number of times that DMAPF stops its MAPF solver increases when the value of ϵ increases from 12 to 14. This shows that in practice there is a chance, especially in dense maps, that DMAPF will have to face a few difficult subproblem instances. Without the timeout mechanism (i.e., $\epsilon = +\infty$) like in our previous works, DMAPF would likely get stuck or take a very long time to solve those subproblem instances. In these situations, it would be more efficient to stop the MAPF solver early, relax the subproblem instance, and retry, which the timeout mechanism allows DMAPF to do. The results also suggest the optimal value of ϵ to be around 10 - 12, and there is no significant deviation of solution qualities between the different values of ϵ overall.

4.4 Congestion

Figure 3 compares congestion resulting from abstract plans created by different methods: ASP, BFS, RAND, UCS, and UCSC, on random-64-64-20 map with 600, 800, and 1000 agents. The map is decomposed with the seed value of 105. The charts depict the trend of the congestion (min and max) in each abstract step. We are mainly concerned with the max congestion as that is usually when some subproblem instance becomes too difficult or unsolvable. The max congestion is the highest congestion across all areas at particular abstract steps. The opposite is true for the min congestion.

In Figure 3, both ASP and BFS produce the shortest abstract plans among all the plans from all the methods; however, their plans also create the highest congestion. In the case of 800 and 1000 agents, their plans result in the value of congestion greater than 1 at abstract step 3. This means that if every agent is able to follow its abstract plan until abstract step 2, there must be at least one area at abstract step 3 where the number of agents who want to be there is greater than the number of vertices in the area!

Abstract plans from both RAND and UCS show significantly lower congestion compared to those from ASP and BFS; however, the length of abstract plans from RAND is quite random (but would still be less than the total number of areas) as it selects nodes to expand randomly; and the plans from UCS, which uses the knowledge of congestion, are only slightly better than the plans from RAND. This is because the knowledge is incomplete when abstract plans are



(a) ASP (b) BFS

(c) RAND (d) UCS

(e) UCSC

Fig. 3. Comparing the trend of congestion from abstract plans created by using Answer Set Programming (ASP), Breadth-First Search (BFS), Random Search (RAND), Uniform-Cost Search (UCS), and Centralized Uniform-Cost Search (UCSC). Each chart depicts the min and max congestion from a particular method on random-64-64-20 map with 600, 800, and 1000 agents.

made independently by different solving processes. It results in solving processes unknowingly create abstract plans that still have high congestion collectively. Instead of having solving processes independently create abstract plans for agents within their responsible subproblems, UCSC uses the same technique as UCS, but designates one of the solving processes to create abstract plans for all agents in the problem. This results in a collection of abstract plans with significantly lower congestion among all the other methods.

Table 4 shows that runtime and solution quality of DMAPF significantly improve when UCSC is used to make abstract plans. It also shows a close inverse relationship between congestion and runtime in DMAPF, following the trend in Figure 3. The runtimes spent on abstract planning are also shown to confirm that UCSC does not incur a significant overhead. In fact, the ASP encoding used in the previous works is even slower than UCSC. According to the success rates, our previous works which do not have the congestion avoidance mechanism would only be able to solve about 60% of the problem instances with 800 agents and unable solve any problem instance with 1000 agents. The congestion avoidance mechanism allows DMAPF to perform at least 3 times faster, reduce the makespan by almost half, and be able to solve all the problem instances.

Table 4. Comparing runtimes used in abstract planning, the total runtimes, makespan, sum-of-cost, and success rates of DMAPF on random-64-64-20 map with n agents using different abstract planning methods: ASP, BFS, RAND, UCS, and UCSC.

n	Method	Runtime (s)		Makespan	SoC ($\times 1k$)	Success Rate
		Abs.	Total			
600	ASP	0.3	48.4	914	247.3	**1.0**
	BFS	**0.0**	110.2	1185	265.6	0.8
	RAND	**0.0**	19.8	673	236.8	**1.0**
	UCS	**0.0**	28.2	779	258.4	**1.0**
	UCSC	**0.0**	**16.1**	**564**	**207.2**	**1.0**
800	ASP	0.4	106.6	1241	511.7	0.6
	BFS	**0.0**	162.1	1367	525.3	0.1
	RAND	**0.0**	74.2	893	424.8	0.8
	UCS	**0.0**	56.1	934	451.0	0.4
	UCSC	**0.0**	**26.1**	**757**	**379.1**	**1.0**
1000	ASP	0.5	-	-	-	0.0
	BFS	**0.0**	-	-	-	0.0
	RAND	**0.0**	204.6	1058	703.9	0.1
	UCS	**0.0**	103.8	1109	740.2	0.1
	UCSC	**0.0**	**46.6**	**906**	**598.5**	**1.0**

4.5 Comparison Between MAPF Solvers

Table 5 compares runtimes, solution quality (indicated by makespan and sum-of-cost), and success rate of DMAPF that has been integrated with 4 different

Table 5. Comparing runtimes, makespan, sum-of-cost, and success rate between MAPF solvers: (*i*) DMAPF w/ASP (DMAPF-A); (*ii*) DMAPF w/CBSH2-RTC (DMAPF-C); (*iii*) DMAPF w/EECBS (DMAPF-E); (*iv*) DMAPF w/PBS (DMAPF-P); (*v*) EECBS; and (*vi*) PBS, on different maps: (*i*) den312d; (*ii*) random-64-64-20 (random); (*iii*) maze-128-128-2 (maze); (*iv*) lak303d; and (*v*) warehouse-20-40-10-2-2 (warehouse), and different number of agents (shown under the map names). The number of vertices in each map is shown on the right hand side of its name.

Solver	— Runtime (seconds) —														
	den312d (2445)			*random* (3270)			*maze* (10858)			*lak303d* (14784)			*warehouse* (38756)		
	200	300	400	600	800	1000	100	200	300	200	400	600	600	800	1000
DMAPF-A	6.9	18.4	39.3	16.1	26.0	**46.6**	20.7	**40.7**	-	15.6	37.9	75.9	32.5	42.4	52.3
DMAPF-C	-	-	-	-	-	-	8.5	-	-	35.7	-	-	14.2	19.8	35.2
DMAPF-E	170.6	-	-	-	-	-	9.8	-	-	10.7	-	-	14.8	20.4	62.9
DMAPF-P	3.1	-	-	-	-	-	8.1	-	-	7.5	44.2	-	13.1	17.0	**20.8**
EECBS	**0.4**	**1.4**	**6.4**	**2.5**	**21.4**	141.7	**2.6**	135.8	**279.6**	**1.2**	**6.5**	**42.6**	**5.8**	**13.1**	22.0
PBS	15.1	217.5	-	-	-	-	50.5	-	-	17.2	266.9	-	9.4	26.1	57.2
	— Makespan —														
DMAPF-A	475	722	980	564	757	906	3075	3704	-	1014	1794	2774	748	774	795
DMAPF-C	-	-	-	-	-	-	3091	-	-	1017	-	-	753	781	804
DMAPF-E	643	-	-	-	-	-	3072	-	-	1033	-	-	780	828	893
DMAPF-P	477	-	-	-	-	-	3069	-	-	1016	1690	-	761	779	803
EECBS	180	288	**377**	**145**	**218**	**302**	**1474**	**1571**	**1702**	483	511	**583**	**451**	**455**	**457**
PBS	**132**	**158**	-	-	-	-	1475	-	-	**482**	**479**	-	**451**	**455**	**457**
	— Sum-of-Cost (×1000) —														
DMAPF-A	51.8	117.3	248.9	207.2	379.1	598.5	181.4	511.8	-	112.9	362.2	794.9	233.1	331.3	448.3
DMAPF-C	-	-	-	-	-	-	187.2	-	-	110.7	-	-	230.7	335.8	443.2
DMAPF-E	56.4	-	-	-	-	-	175.4	-	-	110.4	-	-	236.7	356.6	498.8
DMAPF-P	54.1	-	-	-	-	-	176.2	-	-	111.8	358.7	-	231.8	334.8	449.1
EECBS	13.8	28.0	**46.9**	**34.8**	**60.6**	**101.6**	**56.1**	**119.7**	**191.4**	38.2	78.6	**131.1**	109.6	146.4	181.1
PBS	**11.6**	**19.1**	-	-	-	-	56.4	-	-	**37.9**	**74.1**	-	**109.5**	**146.2**	**180.9**
	— Success Rate —														
DMAPF-A	**1.0**	**1.0**	0.7	**1.0**	**1.0**	**1.0**	**1.0**	0.8	0.0	**1.0**	**1.0**	0.6	**1.0**	**1.0**	**1.0**
DMAPF-C	0.0	0.0	0.0	0.0	0.0	0.0	0.8	0.0	0.0	**1.0**	0.0	0.0	**1.0**	**1.0**	0.7
DMAPF-E	0.4	0.0	0.0	0.0	0.0	0.0	0.9	0.0	0.0	**1.0**	0.0	0.0	**1.0**	0.9	0.6
DMAPF-P	0.6	0.0	0.0	0.0	0.0	0.0	0.9	0.0	0.0	**1.0**	0.4	0.0	0.6	0.5	0.3
EECBS	**1.0**	**1.0**	**1.0**	**1.0**	0.9	**1.0**	0.9	**0.9**	**0.1**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
PBS	**1.0**	0.8	0.0	0.0	0.0	0.0	**1.0**	0.0	0.0	**1.0**	0.3	0.0	**1.0**	**1.0**	**1.0**

MAPF solvers : (*i*) ASP; (*ii*) CBSH2-RTC[4]; (*iii*) EECBS[5]; (*iv*) PBS[6], denoted as DMAPF-A, DMAPF-C, DMAPF-E, and DMAPF-P, respectively; and EECBS and PBS, representing state-of-the-art bounded-suboptimal and optimal MAPF solvers, respectively. We enable SIPPS in EECBS and PBS (including the ones integrated with DMAPF) and set the suboptimality factor of EECBS to 5 to ensure it gives the best runtime without caring for optimality guarantee [10]. We also compared with CBSH2-RTC, but it was not able to solve any problem instance, so we do not include it in the table. The maps used in the comparison are den312d, random-64-64-20 (random), maze-128-128-2 (maze), lak303d, and warehouse-20-40-10-2-2 (warehouse). For DMAPF, they have been decomposed

[4] https://github.com/Jiaoyang-Li/CBSH2-RTC
[5] https://github.com/Jiaoyang-Li/EECBS
[6] https://github.com/Jiaoyang-Li/PBS

into sub-problems, each containing (roughly) 40 vertices, with the seed values of 107, 105, 110, 102, and 108, respectively.

In terms of runtime, EECBS typically outperforms the other solvers, but its speed deteriorates much quicker than DMAPF as the number of agents increases. This is shown when DMAPF-A is able to outperform EECBS in the random map with 1000 agents and in the maze map with 200 agents. DMAPF-P also outperforms EECBS in the warehouse map with 1000 agents.

In terms of solution quality, DMAPF returns solutions with makespan and sum-of-cost about 2-6 times higher than those returned by EECBS and PBS. However, they are comparable in the number of movements agents need to make to reach the goals – the results are omitted due to space limitation. This suggests that agents planned by DMAPF spend about the same number of movements as those planned by EECBS and PBS, but they waste a lot of time in waiting to move from one subproblem to the next between subsequent rounds.

DMAPF-C, DMAPF-E, and DMAPF-P are about twice as fast as DMAPF-A in sparse maps (i.e., maps where the number of agents is low compared to the number of vertices) such as in the warehouse map. However, they are only able to solve a few problem instances in dense maps, especially after the original map has been decomposed into smaller subproblems which introduces more conflicts. On the other hand, DMAPF-A is less affected by the number of conflicts, allowing it to solve significantly more problems instances.

The issue that hinders DMAPF-A is not the conflicts, but rather about how the problem is decomposed. Figure 4 shows subproblem instances that can easily prevent DMAPF from finding the solutions. Figure 4a typically happens in maps with narrow corridors such as the maze map – agent $a1$ needs to go to location $g1$ but is blocked by agent $a2$ that does not need to go anywhere. Figure 4b depicts a similar problem, but it is caused by a mixture of congestion and bad problem decomposition, so an improvement in either area should help to prevent this scenario.

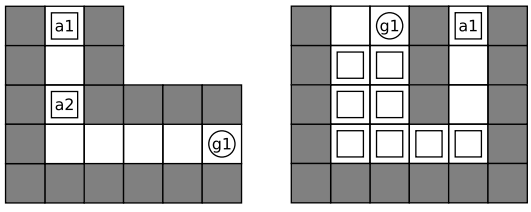


Fig. 4. Issues from bad problem decomposition. Start and goal locations of agents are denoted by small squares and circles with corresponding numbers, respectively.

5 Related Work

There are very few works on MAPF that share the idea of spatially decomposing the problem. To our knowledge, other works with a similar idea includes (*i*) Spatially Distributed Multiagent Planner (SDP) [29] separates a given MAPF problem into low-contention and high-contention areas. Special searching rules are enforced in high-contention areas to speed up the search and agents are not allowed to have a goal location in those areas; (*ii*) Hierarchical Multi-Agent Path Planner (HMAPP) [31] shares a very similar approach with DMAPF. The main difference is that it limits the direction of border vertices between adjacent subproblems, whereas DMAPF does not; and (*iii*) the shard system [9] designates special areas, connecting subproblems, to be used as buffers to improve the solution quality. In the current implementation, agents are not allowed to have a goal location in the buffer areas.

The only recent related works are HMAPP and the shard system. Their source codes are not readily available, so in our experiment we decided to compare DMAPF with EECBS and PBS instead. This design choice serves two purposes: (*i*) our results can be indirectly compared with the two systems – HMAPP has been compared with ECBS [1], the baseline of EECBS, and the shard system has been compared with EECBS; and (*ii*) it shows the behavior of DMAPF when EECBS and PBS are used as its underlying MAPF solver compared to their standalone versions.

6 Summary

We introduce several techniques to improve DMAPF, including (*i*) allowing each solving process to manage multiple subproblems; (*ii*) timeout estimation mechanism; (*iii*) congestion avoidance in abstract plans; and (*iv*) integration with other MAPF solvers. Allowing each solving process to manage multiple subproblems enables DMAPF to work with maps of any size – not limited by the number of subproblems like in our previous works. The combination of timeout estimation mechanism and congestion avoidance in abstract plans enables DMAPF to solve dense maps more efficiently and also increases the success rate. The integration with MAPF solvers from the CBS family provides an insight on the kinds of MAPF solvers that will be suitable with DMAPF for different situations. Even though the improvements we introduce may seem simple, they provide significant improvement over our previous works (as shown in Table 4). Moreover, they can serve as the basis for future improvements.

DMAPF still has many rooms for future improvements. From the experiments, we found that its performance is sensitive to how the problem is decomposed. Having a tool [20] that automatically decomposes a given MAPF problem is convenient, but it still does not guarantee good results. Improvements on problem decomposition technique is still much desirable. This may include developing a visualizing tool to aid the user in manually decomposing the problem. We also expect DMAPF to be able to solve more problems and has better performance with the use of more advanced load balancing mechanisms.

References

1. Barer, M., Sharon, G., Stern, R., Felner, A.: Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In: Seventh Annual Symposium on Combinatorial Search (2014)
2. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)
3. Boyarski, E., Felner, A., Stern, R., Sharon, G., Tolpin, D., Betzalel, O., Shimony, E.: Icbs: Improved conflict-based search algorithm for multi-agent pathfinding. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)
4. Dechter, R., Cohen, D., et al.: Constraint processing. Morgan Kaufmann (2003)
5. Felner, A., Stern, R., Shimony, S., Boyarski, E., Goldenberg, M., Sharon, G., Sturtevant, N., Wagner, G., Surynek, P.: Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In: International Symposium on Combinatorial Search. vol. 8 (2017)
6. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with clingo 5. In: Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
7. Ho, F., Geraldes, R., Gonçalves, A., Rigault, B., Sportich, B., Kubo, D., Cavazza, M., Prendinger, H.: Decentralized multi-agent path finding for uav traffic management. IEEE Transactions on Intelligent Transportation Systems (2020)
8. Hönig, W., Kiesel, S., Tinka, A., Durham, J.W., Ayanian, N.: Persistent and robust execution of mapf schedules in warehouses. IEEE Robotics and Automation Letters **4**(2), 1125–1131 (2019)
9. Leet, C., Li, J., Koenig, S.: Shard systems: Scalable, robust and persistent multi-agent path finding with performance guarantees. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 36, pp. 9386–9395 (2022)
10. Li, J., Chen, Z., Harabor, D., Stuckey, P.J., Koenig, S.: Mapf-lns2: Fast repairing for multi-agent path finding via large neighborhood search. In: Proceedings of the AAAI Conference on Artificial Intelligence (2022)
11. Li, J., Harabor, D., Stuckey, P.J., Ma, H., Gange, G., Koenig, S.: Pairwise symmetry reasoning for multi-agent path finding search. Artificial Intelligence **301**, 103574 (2021)
12. Li, J., Ruml, W., Koenig, S.: EECBS: A bounded-suboptimal search for multi-agent path finding. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 12353–12362 (2021). https://doi.org/10.1609/aaai.v35i14.17466
13. Li, J., Tinka, A., Kiesel, S., Durham, J.W., Kumar, T.S., Koenig, S.: Lifelong multi-agent path finding in large-scale warehouses. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 11272–11281 (2021)
14. Ma, H., Harabor, D., Stuckey, P.J., Li, J., Koenig, S.: Searching with consistent prioritization for multi-agent path finding. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 7643–7650 (2019). https://doi.org/10.1609/aaai.v33i01.33017643
15. Ma, H., Yang, J., Cohen, L., Kumar, T.S., Koenig, S.: Feasibility study: Moving non-homogeneous teams in congested video game environments. In: Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference (2017)
16. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm, pp. 375–398. Springer (1999)

17. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of mathematics and Artificial Intelligence **25**(3), 241–273 (1999)
18. Phillips, M., Likhachev, M.: Sipp: Safe interval path planning for dynamic environments. In: 2011 IEEE International Conference on Robotics and Automation. pp. 5628–5635. IEEE (2011)
19. Pianpak, P., Son, T.C.: DMAPF: A decentralized and distributed solver for multi-agent path finding problem with obstacles. Electronic Proceedings in Theoretical Computer Science (EPTCS) **345**, 99–112 (Sep 2021). https://doi.org/10.4204/eptcs.345.24
20. Pianpak, P., Son, T.C.: Improving problem decomposition and regulation in distributed multi-agent path finder (dmapf). In: PRIMA 2022: Principles and Practice of Multi-Agent Systems. pp. 156–172 (2023). https://doi.org/10.1007/978-3-031-21203-1_10
21. Pianpak, P., Son, T.C., Toups, Z.O., Yeoh, W.: A distributed solver for multi-agent path finding problems. In: Proceedings of the First International Conference on Distributed Artificial Intelligence (DAI). pp. 1–7 (2019). https://doi.org/10.1145/3356464.3357702
22. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., et al.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)
23. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence **219**, 40–66 (2015)
24. Sharon, G., Stern, R., Goldenberg, M., Felner, A.: The increasing cost tree search for optimal multi-agent pathfinding. Artificial intelligence **195**, 470–495 (2013)
25. Silver, D.: Cooperative pathfinding. In: Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment. vol. 1, pp. 117–122 (2005)
26. Stern, R., Sturtevant, N.R., Felner, A., Koenig, S., Ma, H., Walker, T.T., Li, J., Atzmon, D., Cohen, L., Kumar, T.K.S., Boyarski, E., Bartak, R.: Multi-agent pathfinding: Definitions, variants, and benchmarks. Symposium on Combinatorial Search (SoCS) pp. 151–158 (2019)
27. Surynek, P.: Compilation-based solvers for multi-agent path finding: a survey, discussion, and future opportunities. arXiv preprint arXiv:2104.11809 (2021)
28. Thayer, J.T., Ruml, W.: Bounded suboptimal search: A direct approach using inadmissible estimates. In: Twenty-Second International Joint Conference on Artificial Intelligence (2011)
29. Wilt, C., Botea, A.: Spatially distributed multiagent path planning. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 24, pp. 332–340 (2014)
30. Yu, J., LaValle, S.M.: Structure and intractability of optimal multi-robot path planning on graphs. In: Twenty-Seventh AAAI Conference on Artificial Intelligence (2013)
31. Zhang, H., Yao, M., Liu, Z., Li, J., Terr, L., Chan, S.H., Kumar, T.S., Koenig, S.: A hierarchical approach to multi-agent path finding. In: Proceedings of the International Symposium on Combinatorial Search. vol. 12, pp. 209–211 (2021)