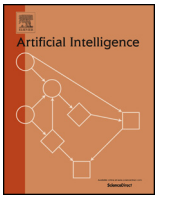
Artificial Intelligence 305 (2022) 103662

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



# Multi-agent pathfinding with continuous time

Anton Andreychuk [a,d], Konstantin Yakovlev [a,b,c,*], Pavel Surynek [g], Dor Atzmon [e], Roni Stern [e,f]

[a] Federal Research Center for Computer Science and Control of Russian Academy of Sciences, Russia
[b] HSE University, Russia
[c] Moscow Institute of Physics and Technology (MIPT), Russia
[d] Peoples' Friendship University of Russia (RUDN University), Russia
[e] Software and Information Systems Eng., Ben Gurion University of the Negev, Israel
[f] System Sciences Laboratory (SSL), Palo Alto Research Center (PARC), USA
[g] Faculty of Information Technology (FIT), Czech Technical University (ČVUT), Czechia

ARTICLE INFO

*Article history:*
Received 11 May 2020
Received in revised form 22 October 2021
Accepted 7 January 2022
Available online 11 January 2022

*Keywords:*
Multi-agent pathfinding
Conflict-based search
Safe-interval path planning
SAT modulo theory
Heuristic search

ABSTRACT

Multi-Agent Pathfinding (MAPF) is the problem of finding paths for multiple agents such that each agent reaches its goal and the agents do not collide. In recent years, variants of MAPF have risen in a wide range of real-world applications such as warehouse management and autonomous vehicles. Optimizing common MAPF objectives, such as minimizing sum-of-costs or makespan, is computationally intractable, but state-of-the-art algorithms are able to solve optimally problems with dozens of agents. However, most MAPF algorithms assume that (1) time is discretized into time steps and (2) the duration of every action is one time step. These simplifying assumptions limit the applicability of MAPF algorithms in real-world applications and raise non-trivial questions such as how to discretize time in an effective manner. We propose two novel MAPF algorithms for finding optimal solutions that do not rely on any time discretization. In particular, our algorithms do not require quantization of wait and move actions' durations, allowing these durations to take any value required to find optimal solutions. The first algorithm we propose, called Continuous-time Conflict-Based Search (CCBS), draws on ideas from Safe Interval Path Planning (SIPP), a single-agent pathfinding algorithm designed to cope with dynamic obstacles, and Conflict-Based Search (CBS), a state-of-the-art search-based MAPF algorithm. SMT-CCBS builds on similar ideas, but is based on a different state-of-the-art MAPF algorithm called SMT-CBS, which applied a SAT Modulo Theory (SMT) problem-solving procedure. CCBS guarantees to return solutions that have minimal sum-of-costs, while SMT-CCBS guarantees to return solutions that have minimal makespan. We implemented CCBS and SMT-CCBS and evaluated them on grid-based MAPF problems and general graphs (roadmaps). The results show that both algorithms can efficiently solve optimally non-trivial MAPF problems.



* Corresponding author at: Federal Research Center for Computer Science and Control of Russian Academy of Sciences, Russia.
*E-mail address:* yakovlev@isa.ru (K. Yakovlev).

https://doi.org/10.1016/j.artint.2022.103662
0004-3702/

## 1. Introduction

MAPF is the problem of finding paths for multiple agents such that each agent reaches its goal and the agents do not collide. MAPF has topical applications in warehouse management [1], airport towing [2], autonomous vehicles, robotics [3], and digital entertainment [4]. A common requirement in MAPF applications is to minimize either the *sum of costs* (SOC) of the agents' plans or their maximum, also known as their *makespan*. Finding MAPF solutions that have minimal SOC or minimal makespan are both NP hard problems [5,6]. Nevertheless, AI researchers in the past years have made substantial progress in finding solutions with minimal SOC or minimal makespan for a growing number of MAPF problems, including problems with over one hundred agents [7–12,6].

However, most prior work on solving MAPF optimally assumed that (1) time is discretized into time steps, and (2) the duration of every action is one time step. These simplifying assumptions limit the applicability of MAPF algorithms in real-world application. In this work, we address a more general type of MAPF problem called $\text{MAPF}_R$ [13].[1] In $\text{MAPF}_R$, agents occupy a defined area in a metric space at any moment of a continuous timeline, and do not rely on any time discretization.

Existing algorithms such as E-ICTS [13] or ECBS-CT [14], partially address the $\text{MAPF}_R$ problem, in that they support move actions of non-uniform duration (cost) and consider the area the agents occupy. However, these algorithms still rely on discretizing time to define the duration of the wait actions. This can have a negative effect on both solution quality and runtime. In this work, we propose two optimal algorithms that solve $\text{MAPF}_R$ problems and do not require any time discretization, allowing wait actions with arbitrary durations.

The first algorithm we propose is called CCBS. CCBS builds on two existing algorithms: SIPP [15] and CBS [7]. SIPP is a single-agent pathfinding algorithm designed to find paths that avoid dynamic obstacles. CBS is a state-of-the-art search-based MAPF algorithm with many extensions and improvements. CCBS follows the same problem-solving framework as CBS, but uses a variant of SIPP to plan paths for the individual agents, and imposes a novel type of constraints to resolve conflicts between the resulting plans.

The second algorithm we propose is called SMT-CCBS. SMT-CCBS is a SAT-based algorithm, that is, it solves a $\text{MAPF}_R$ problem by compiling parts of the problem to a Boolean Satisfiability problem (SAT) and applies a state-of-the-art SAT solver to solve it. SAT-based approaches have been applied successfully for classical MAPF and take advantage of the power of modern SAT solvers. Technically, SMT-CCBS resolves conflicts between agents using the same type of constraints as CCBS. However, it does so in the SAT Modulo Theory (SMT) problem solving framework used by the SMT-CBS algorithm [16] to solve MAPF problems. Both CCBS and SMT-CCBS are guaranteed to (1) only return solutions that are conflict-free, (2) find a solution if one exists, (3) return a solution that has a minimal SOC or makespan (for CCBS and SMT-CCBS, respectively).

We implemented CCBS and SMT-CCBS, and evaluated them on standard grid-based MAPF benchmarks as well as generated roadmaps. The results show that both algorithms can solve MAPF problems with dozens of agents optimally. While we are not the first to study MAPF beyond its basic setting [13,17,14], to the best of our knowledge we are the first to propose MAPF algorithms that can handle agents with volumes and non-uniform action durations while avoiding any form of time discretization (see Section 7 for a more detailed discussion). All our implementations are publicly available, so researchers can reproduce our results and build on them.

A preliminary version of this work was published in two conference papers [18,19]. The main contribution of this journal paper is to present these works in a unified and coherent manner. In addition, this journal paper extends these works significantly, by providing:

- A formal proof of completeness and optimality for CCBS and SMT-CCBS.
- A detailed explanation of the algorithms with a running example to illustrate their behavior.
- A detailed description of how one can implement collision and conflict detection, and unsafe interval calculation.
- A comprehensive set of experiments, including experiments on roadmaps and experimental comparison with E-ICTS [13] and ECBS-CT [14].

This paper is structured as follows. Section 2 provides relevant MAPF definitions and relevant background. Section 3 formally defines the $\text{MAPF}_R$ problem we address. Section 4 presents the CCBS algorithm and Section 5 presents SMT-CCBS. Results of the empirical evaluation of both algorithms are described in Section 6. Section 7 discusses the pros and cons of the CCBS and SMT-CCBS and how they relate to other algorithms for solving different variants of the general MAPF problem. Section 8 concludes with a list of directions for future work.

## 2. Background

A *classical MAPF* problem [20] with $k$ agents is defined by a tuple $\langle \mathcal{G}, S, G \rangle$ where $\mathcal{G} = (V, E)$ is an undirected graph, $S : [1, \ldots, k] \to V$ maps an agent to a start vertex, and $G : [1, \ldots, k] \to V$ maps an agent to a goal vertex (see Fig. 1).

Time is discretized into time steps. For every time step $t$, each agent occupies one of the graph vertices, referred to as the *location* of that agent at time $t$. An *action* in classical MAPF is a function $a : V \to V$ such that $a(v) = v'$ means the

[1] We discuss later the relation between Walker et al.'s [13] definition of $\text{MAPF}_R$ and ours.

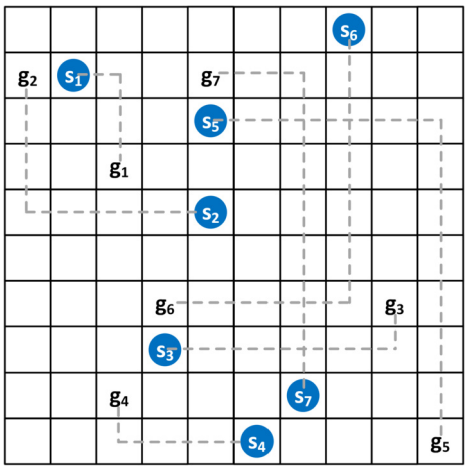



**Fig. 1.** An example of a classical MAPF problem instance on a 4-neighborhood grid.

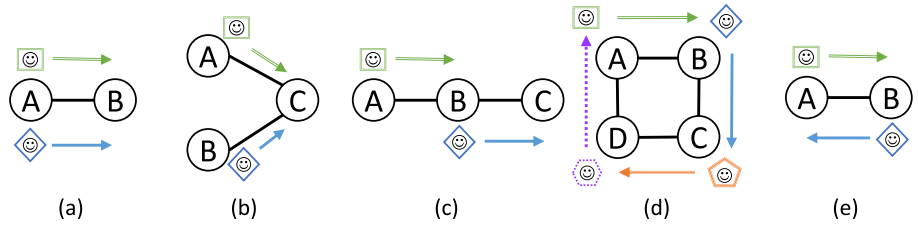



**Fig. 2.** An illustration from [20] of an edge conflict (a), a vertex conflict (b), a following conflict (c), a cycle conflict (d), and a swapping conflict (e).

agent's current location is $v$ and its location in the next time step is $v'$. In every time step each agent can choose to perform an action. There are two types of actions: a *wait* action, in which the agent stays in its location, and a *move* action, in which the agent moves to one of the vertices adjacent to its current location.

A sequence of actions $\pi_i = (a_1, \ldots, a_n)$ is a *single-agent plan* for agent $i$ if $a_n(a_{n-1}(\cdots(a_1(S(i)))\cdots)) = G(i)$, i.e., if it leads the agent from its start to its goal. The number of actions in the plan defines its cost denoted as $c(\pi_i)$. A **solution** to a classical MAPF problem is a set of $k$ single-agent plans, one for each agent: $\Pi = \{\pi_1, ..., \pi_k\}$.

A solution to a classical MAPF problem is *valid* if its constituent single-agent plans do not *conflict*. There are multiple types of conflicts. A *vertex conflict* between single-plan $\pi_i$ for agent $i$ and single-plan $\pi_j$ for agent $j$ occurs at location $v$ and time step $t$ iff according to these plans agents $i$ and $j$ plan to occupy $v$ at the same time step $t$. We represent such a conflict by the tuple $\langle i, j, v, t\rangle$. A *swapping conflict* between single-agent plans $\pi_i$ and $\pi_j$ occurs on edge $e$ at time step $t$ iff according to $\pi_i$ and $\pi_j$ both agents plan to traverse the edge $e \in E$ from opposite directions at the same time step $t$. We represent such a conflict by the tuple $\langle i, j, e, t\rangle$. Fig. 2 illustrates these conflicts as well as other types of conflict that arise in classical MAPF. See Stern et al. [20] for a deeper and more formal discussion of these conflicts.

In classical MAPF, every action incurs a unit cost, and the cost of a single-agent plan is the number of its constituent actions. *Sum-of-costs* (SOC) and *makespan* are two common ways to define the cost of a MAPF solution. The former, also known as the flowtime, is the summation over the costs of the constituent single-agent plans, $\sum_{i=1}^{k} c(\pi_i)$, and the latter is their max, $\max_{\pi \in \Pi} c(\pi)$. A solution to a MAPF problem is called SOC-optimal if it is valid and there is no valid solution that has a smaller SOC. A makespan-optimal solution is defined in a similar way. In general, the problem of finding SOC-optimal or makespan-optimal solutions for a given MAPF problem is known to be NP hard [21,5].

### 2.1. Optimal algorithms for classical MAPF

Several approaches have been proposed to find either SOC-optimal or makespan-optimal solutions for classical MAPF (e.g., see the survey by Felner et al. [22]). In this work, we build on three specific state-of-the-art classical MAPF solvers: CBS [7], MDD-SAT [23], and SMT-CBS [16]. For completeness, we provide a brief description of these algorithms here.

*2.1.1. Conflict Based Search (CBS) for finding SOC-optimal solutions*

CBS [7] is a *solution complete* algorithm for classical MAPF that is guaranteed to return a SOC-optimal solution.[2] It solves a given MAPF problem by finding a plan for each agent separately, detecting conflicts between these plans, and resolving them by replanning for the individual agents subject to specific *constraints*.

The basic CBS implementation considers two types of conflicts: vertex conflicts and swapping conflicts. Correspondingly, the basic CBS implementation considers two types of constraints. A CBS vertex-constraint is defined by a tuple $\langle i, v, t\rangle$ and means that agent $i$ is prohibited from occupying vertex $v$ at $t$. A CBS edge-constraint is defined similarly by a tuple $\langle i, e, t\rangle$, where $e \in E$. To guarantee solution completeness and optimality, CBS runs two search algorithms: a low-level search algorithm that finds plans for individual agents subject to a given set of constraints, and a high-level search algorithm that chooses which constraints to add.

***CBS: low-level search*** The task of the low-level search in CBS is to find an optimal plan for an agent that is consistent with a given set of CBS constraints. Any single-agent pathfinding algorithm that can do this can be used as the CBS low-level search. To adapt single-agent pathfinding algorithms, such as A*, to consider CBS constraints, the search space must also consider the time dimension since a CBS constraint $\langle i, v, t\rangle$ blocks location $v$ only at a specific time step $t$. This means that a state in this single-agent search space is a pair $(v, t)$, representing that the agent is in location $v$ at time step $t$. Expanding such a state generates states of the form $(v', t+1)$, where $v'$ is either equal to $v$, representing a wait action, or equal to one of the locations adjacent to $v$. States generated by actions that violate the given set of CBS constraints are pruned. Running A* on this search space returns the lowest-cost plan to the agent's goal that is consistent with the given set of CBS constraints, as required. This adaptation of textbook A* is very simple, and indeed most papers on CBS do not report it and just mention that the low-level search of CBS is A*.

***CBS: high-level search*** The high-level search algorithm in CBS works on a Constraint Tree (CT), which is a binary tree, in which each node $N$ is defined by a pair $(N.const, N.\Pi)$ where $N.const$ represents a set of CBS constraints imposed on the agents and $N.\Pi$ is a MAPF solution that satisfies these CBS constraints. A CT node $N$ is generated by first setting its constraints ($N.const$) and then computing $N.\Pi$ by running the low-level solver, which finds a plan for each agent subject to the constraints relevant to it in $N.const$. If $N.\Pi$ does not contain any conflict, then $N.\Pi$ is a valid MAPF solution and $N$ is regarded as a goal node. Else, one of the conflicts $\langle i, j, x, t\rangle$ (where $x$ is either a vertex or an edge) in $N.\Pi$ is chosen and two new CT nodes are generated $N_i$ and $N_j$. Both nodes have the same set of constraints as $N$, plus a new constraint, added to resolve the conflict: $N_i$ adds the constraint $\langle i, x, t\rangle$ and $N_j$ adds the constraint $\langle j, x, t\rangle$. CBS searches the CT in a best-first manner, expanding in every iteration the CT node $N$ with the lowest-cost joint plan. The search halts when the CT node $N$ chosen for expansion is conflict-free, i.e., when $N.\Pi$ is a valid solution.

*2.1.2. MDD-SAT for finding makespan-optimal solutions*

A Boolean Satisfiability (SAT) problem is defined by a set of Boolean variables $\{v_1, \ldots v_n\}$ and a Boolean formula $\Phi$ defined over these variables. A solution to a SAT problem is an assignment to these variables such that $\Phi$ is satisfied, or UNSAT if there is no assignment of these variables that satisfies $\Phi$. Modern SAT solvers are extremely efficient and can scale to SAT problems with over a million variables [25,26]. Thus, a successful approach for solving many combinatorial search problems is to compile them to a SAT problem and solve it with a state-of-the-art SAT solver. This approach has also been applied successfully to classical MAPF [23,27].

The SAT-based approach for finding makespan-optimal solutions works by solving a sequence of SAT problems. Each of these SAT problems represents the decision problem: "is there a valid solution to the given classical MAPF problem within a makespan of at most $\mu$?" where $\mu$ is a parameter. We call this decision problem the *bounded-cost MAPF problem*.

Each bounded-cost MAPF problem is encoded as a SAT problem by defining two types of Boolean variables. The first type, denoted $\mathcal{X}_v^t(i)$, is defined for every discrete time step $t = \{0, \ldots, \mu\}$, agent $i$, and vertex $v$ the agent may occupy at time step $t$. The second type of variables, denoted $\mathcal{E}_{u,v}^t(i)$, is defined for every time step $t$, agent $i$, and edge $(u, v)$ the agent may start traversing at time step $t$. MAPF movement rules and collision avoidance constraints are encoded on top of these variables as simple local constraints. The resulting Boolean formula is given to a SAT solver, which returns either a satisfying assignment or UNSAT. A satisfying assignment to the decision variables specifies a valid solution for the given MAPF problem. An UNSAT indicates no valid solution exists with makespan smaller than or equal to $\mu$. In this case, $\mu$ is increased by one. This process continues until the minimal $\mu$ for which a solution exists is found. This $\mu$ is guaranteed to be makespan optimal. The SATPlan algorithm [28,29] followed a similar approach for classical planning. While this SAT-based approach for classical MAPF returns makespan-optimal solutions, it is also possible, with some additional bookkeeping, to use it to return SOC-optimal solutions [23].

As expected, the size of the generated Boolean formulas has a great impact on the overall runtime. To reduce this size, Surynek et al. [28] proposed to introduce the $\mathcal{X}_v^t(i)$ and $\mathcal{E}_{u,v}^t(i)$ variables only for reachable vertices and edges at given time step $t$. This reachability analysis can be done by constructing a *Multi-Value Decision Diagram* (MDD) [30] for each agent

[2] A solution complete algorithm is an algorithm that is guaranteed to find a solution if a solution exists. But, a solution complete algorithm may not detect unsolvability. That is, given an unsolvable problem, a solution complete algorithm may run indefinitely [24].

representing all single-agent plans for that agent up to the makespan bound $\mu$. Each of these MDDs is a directed acyclic graph with a single source. A node in these MDDs represents a vertex and time pair $(v, t)$ the agent may occupy in a single agent plan that reaches the goal before the makespan bound. Edges represent moves in such a single-agent plan. The construction of $\Phi$ then relies on these MDDs rather than on the original graph, i.e., it includes only variables that have corresponding vertices and edges in an MDD. This algorithm is known as MDD-SAT.

#### 2.1.3. An SMT approach to classical MAPF

SMT-CBS [16] is a recently introduced MAPF algorithm that integrates ideas from CBS and SAT-based MAPF algorithms. Like SAT-based MAPF algorithm, SMT-CBS finds an optimal solution to the given MAPF problem by solving a sequence of bounded-cost MAPF problems. However, SMT-CBS solves each of these bounded-cost problems using a SAT Modulo Theory (SMT) [31–33] problem-solving framework. For completeness, we provide a brief background on SMT.

SMT is a problem-solving framework designed to leverage the power of modern SAT solvers while applying them to a larger set of problems. The basic use of SMT divides a given decision problem $\Gamma$ into two parts. The first, called the Propositional Skeleton (PS), is an abstraction of $\Gamma$ that keeps only its Boolean structure. The second, called $DECIDE_T$, is a decision procedure that accepts an assignment that satisfies the PS and outputs *true* if this assignment corresponds to a solution of the original problem $\Gamma$. If $DECIDE_T$ returns *false*, i.e., if it is given a solution to the PS that cannot be mapped to a solution for $\Gamma$, then $DECIDE_T$ returns also a *conflict* (often called a *lemma*) that explains why the solution to the PS is not valid.

The standard SMT solving procedure is iterative. First, find a satisfying assignment of the PS. Then, call $DECIDE_T$ with this assignment. If $DECIDE_T$ returns *true*, the satisfying assignment is returned and the SMT solving procedure is finished. Otherwise, the PS is extended with new constraints designed to resolve the conflict returned by $DECIDE_T$. In more general cases, not only new constraints are added to resolve a conflict but also new propositional variables. Appendix A lists a simple example of this problem-solving procedure.

SMT-CBS follows the same SMT solving procedure for solving bounded-cost MAPF. The PS initially created in SMT-CBS is the same as the SAT problem created by SAT-based MAPF algorithms, except that it does not encode constraints to avoid conflicts between the agents' plans. Instead, these conflicts are detected after a solution to this PS is found, in a MAPF-specific $DECIDE_T$ procedure that we denote by $DECIDE_{MAPF}$. That is, whenever the SAT solver outputs a solution, this solution is checked for conflicts. If no conflicts exist, the solution is returned. Otherwise, additional constraints are added to all subsequent PS to ensure that every previously detected conflict will not occur again. These constraints are exactly the constraints CBS would impose to resolve the found conflicts. Theoretically, SMT-CBS can converge to the same formula as MDD-SAT but it often finishes with a smaller formula due to the iterative/lazy construction. A similar approach has been used in the Lazy CBS algorithm [34]. SMT-CBS showed impressive performance on standard MAPF benchmarks.

### 2.2. Limitations of classical MAPF and prior work on general MAPF

Moving a group of mobile robots safely in a shared environment is commonly considered as a primary application for a MAPF research. An important question is therefore how the definition of a classical MAPF problem relates to real-world MAPF applications in robotics. In particular, consider the following intrinsic simplifying assumptions of classical MAPF and their implications in robotics.

- **A1. The duration of every move action is one time step.** This means either all agents move in exactly the same speed and graph edges represent transitions of the same length, or that graph edges represent transitions of different lengths and the agents adapt their velocity and acceleration profile so that all moves take one time step.
- **A2. The duration of a wait action is one time step.** This means an agent may wait any discrete number of time steps, as opposed to any real valued duration.

In this work, we lift these assumptions. Since we are not the first to do this, we first discuss prior works and the relation between them. Walker et al. [13] introduced the $\text{MAPF}_R$ problem, which lifts the first classical MAPF assumption (A1). In $\text{MAPF}_R$, every edge $e = (v, v')$ in the underlying graph $\mathcal{G}$ is associated with a positive weight $w(e) \in \mathbb{R}_{>0}$ that represents the duration it takes an agent to move from $v$ to $v'$. Every location $v \in V$ is associated with a unique coordinate in a metric space, denoted $coord(v)$. Agents occupy non-zero volume in this space. When the location of an agent is $v$, it means that the reference point of the agent is located at $coord(v)$. When an agent moves along an edge $(v, v')$, it means it moves along a straight line from $coord(v)$ to $coord(v')$ in a constant velocity motion. There is a conflict between two single-agent plans iff the volumes of two or more agents "overlap at the same instant in time" [13]. This can be detected using standard collision-detection techniques [35,36].

The original definition of $\text{MAPF}_R$ only states that action durations are non-uniform, and does not differentiate between move actions and wait actions. However, the algorithm Walker et al. proposed in that paper – Extended Increasing Cost Tree Search (E-ICTS) – relies on wait actions having a fixed, pre-determined duration. So, while E-ICTS does not lift A2, the definition of $\text{MAPF}_R$ there is ambiguous.

Cohen et al. [14] proposed an extension of classical MAPF called *multi-agent motion planning* (MAMP). In MAMP, each agent is associated with a graph $\mathcal{G}_i = (V_i, E_i)$. A vertex in $V_i$ represents a possible *state* of agent $i$, where a state of an agent

represents its location as well as other relevant features such as orientation and steering angle. An edge $e = (v, v') \in E_i$ represents a kinodynamically feasible motion of agent $i$ from state $v$ to state $v'$, and the weight of an edge is the duration of performing this motion. The agents in MAMP move in an *environment* represented by a set of grid cells $\mathcal{C}$. Every state $v \in V_r$ of an agent $i$ is associated with a set of cells in $\mathcal{C}$, representing the cells occupied by that agent when in state $v$. Every edge $e = (v, v') \in E_i$ is associated with a multiset of cells in $\mathcal{C}$. Each cell in this multiset is associated with a time interval indicating the time interval in which this cell is occupied when agent $i$ moves from $v$ to $v'$. Like E-ICTS, ECBS-CT lifts A2 only partially, as the duration of the wait actions is tied to the given time discretization (which is done by diving it into the timesteps of $\epsilon$ duration).[3]

### 2.3. SIPP

The MAPF$_R$ algorithms we propose in this paper are based on the SIPP algorithm. SIPP [15] is a powerful algorithm for building a plan for a single agent moving among static and dynamic obstacles [15]. It has also been used within prioritized MAPF solvers [38] and for solving multi-agent pickup and delivery (MAPD) problems [39]. SIPP accepts as input a graph, a start and goal vertices in that graph, and trajectories specifying the motion of the dynamic obstacles over time. The algorithm pre-processes these trajectories to compute *safe intervals* for each vertex in the graph. A safe interval is a contiguous period of time for a vertex, during which if the agent occupies that vertex then it will not collide with any dynamic obstacle. Safe intervals are assumed to be maximal, i.e., extending a safe interval is not possible.

For example, consider a case where only one dynamic obstacle is present and both an agent and the obstacle are disks of radius $r$. Now, consider a vertex $v$ such that the distance between this obstacle and $v$ is less than $2r$ between time moments $t_1$ and $t_2$ (excluding $t_1$ and $t_2$ themselves). The corresponding safe intervals for $v$ are $[0, t_1]$ and $[t_2, +\infty)$. Note that in this example, we assume that (1) a collision happens only when the distance between two disks is less than the sum of their radii, when the distance is equal to it – no collision happens; and (2) the collision does not occur at the specific points $t_1$ and $t_2$.

A vertex may have multiple, non-overlapping, safe intervals. In general, the number of safe intervals is proportional to the number of obstacles that pass nearby the vertex. The chronologically last safe interval for a vertex might not end with $\infty$. For example, an obstacle may come to that vertex and stay in it. The safe interval might be an $\emptyset$ as well, e.g., in cases where an obstacle constantly moves back-and-forth in the vicinity of the vertex.

SIPP performs an A*-like search over the search space in which each node is a pair of graph vertex and one of its safe intervals. This means there may be multiple search nodes for the same vertex but with different safe time intervals. In the example above, nodes $n_1 = (v, [0, t_1])$ and $n_2 = (v, [t_2, +\infty))$ correspond to the same vertex $v$, but have different safe intervals. When expanding a node $n = (v, T_v)$, SIPP generates a node for every pair $(v', T'_v)$ where $v'$ is a neighbor of $v$ and $T'_v$ is a safe interval of $v'$ in which the agent can safely arrive starting at a time in $T_v$. SIPP also considers collisions with obstacles that occur when the agent moves along an edge. This is done when computing the time it takes to traverse an edge. When there exists an expected collision on some edge, then the time it takes to traverse that edge at that time will include staying in the vertex until it is possible to move along that edge safely (i.e., without colliding). Guided by a consistent heuristic, SIPP is guaranteed to return optimal solutions. Sub-optimal and anytime variations of the algorithm are also known [40,41].

## 3. Problem statement

We define a MAPF$_R$ problem by the tuple $\langle \mathcal{G}, \mathcal{M}, S, G, coord, \mathcal{A} \rangle$ where $\mathcal{G} = (V, E)$ is a graph, $\mathcal{M}$ is a metric space, $S$ and $G$ are the start and goal functions, *coord* maps every vertex in $\mathcal{G}$ to a coordinate in $\mathcal{M}$, and $\mathcal{A}$ is a finite set of possible *move actions*.

Every action $a \in \mathcal{A}$ in MAPF$_R$ is defined by a duration $a_D$ and a *motion function* $a_\varphi$. A motion function $a_\varphi$ is a function $a_\varphi : [0, a_D] \to \mathcal{M}$, that maps time to metric space. Here $a_\varphi(t)$ is the coordinate of an agent (in $\mathcal{M}$) at the time $t$ while executing an action $a$. Observe that the definition of these motion functions allows us to model agents that move with a non-constant speed and follow an arbitrary geometric curve.

There are two types of actions in MAPF$_R$: move actions and wait actions. For a move action $a \in \mathcal{A}$, we restrict $a_\varphi$ so that it starts in some vertex $v$, ends in some other vertex $v'$, and $(v, v')$ is an edge in $E$. That is, there exists $v$ and $v'$ such that $a_\varphi(0) = coord(v)$, $a_\varphi(a_D) = coord(v')$, and $(v, v') \in E$. We denote these vertices by $from(a)$ and $to(a)$. Note that by assuming a finite set of move actions, we limit the completeness and optimality of the algorithms we propose to be only with respect to the chosen set of move actions. We say that an algorithm is *complete with respect to a set of move actions* $\mathcal{A}$ if it is guaranteed to return a valid solution if one exists in which all move actions the agents perform are from $\mathcal{A}$. Similarly, we say that an algorithm is *optimal with respect to a set of move actions* $\mathcal{A}$ if the solution it returns is guaranteed to have the lowest cost among all other valid solutions in which all move actions the agents perform are from $\mathcal{A}$. Solutions that are optimal or even valid with respect to one set of move actions can be non-optimal or even invalid under a different set of move actions.

[3] These details are listed in Cohen's Ph.D. thesis [37].

A *wait action* $a$ is an action for which there exists a vertex $v \in V$ such that for every $t \in [0, a_D]$ we have that $a_\varphi(t) = coord(v)$. For completeness, we define $from(a)$ and $to(a)$ for wait actions to be this vertex. Note that while the set of move actions is given as input ($\mathcal{A}$), the set of wait actions is implicitly defined for every vertex $v \in V$ and *any* positive real number $a_D$. Thus, the set of wait actions is infinitely large.

In MAPF$_R$, when an agent is at a vertex $v$ it can choose to perform any action — move or wait — that starts at $v$, i.e., $from(a) = v$. A *collision* between the agents occurs if their shapes overlap. To detect such an overlap, we assume a *collision-detection* method IsCollision : $\{1, \dots, k\} \times \{1, \dots, k\} \times \mathcal{M} \times \mathcal{M} \rightarrow \{true, false\}$ is available where IsCollision $(i, j, m_i, m_j)$=*true* iff when agents $i$ and $j$ occupy locations $m_i$ and $m_j$, respectively, then their shapes overlap. For example, if the agents are 2D disk-shaped with a radius of $r$, then a collision occurs if the distance between the centers of the agents is less than $2r$. That is, in this setting,

$$\text{IsCollision}(i, j, m_i, m_j) = \begin{cases} true & ||m_i - m_j||_2 < 2r \\ false & \text{Otherwise} \end{cases} \tag{1}$$

Note that our problem definition and the algorithms we propose later are not restricted to disk-shaped agents and this particular type of IsCollision implementation.

For a sequence of actions $\pi = (a_1, \dots, a_n)$, we denote by $\pi[:j]$ the prefix of the first $j$ actions, i.e., $\pi[:j] = (a_1, \dots a_j)$. The duration and motion function of $\pi$, denoted $\pi_D$ and $\pi_\varphi$, respectively, are defined as follows:

$$\pi_D = \sum_{a \in \pi} a_D \tag{2}$$

$$\pi_\varphi(t) = \begin{cases} a_{1\varphi}(t) & t \leq a_{1D} \\ \dots & \dots \\ a_{j\varphi}(t - (\pi[:j-1])_D) & (\pi[:j-1])_D < t \leq (\pi[:j])_D \\ \dots & \dots \\ a_{n\varphi}(t - (\pi[:n-1])_D) & (\pi[:n-1])_D < t \leq (\pi[:n])_D \\ a_{n\varphi}(a_{nD}) & (\pi[:n])_D < t \end{cases} \tag{3}$$

To explain Equation (3), which computes the location at each time moment while executing $\pi$, observe that the motion functions are not defined with respect to when their respective actions are applied. In other words, motion functions are defined with respect to relative time and not absolute time. For instance, for any action $a$ that moves the agent from $v$ to $v'$, by definition $a_\varphi(0) = v$ and $a_\varphi(a_D) = v'$. Therefore, to compute $\pi_\varphi(t)$ we need to first identify the action planned to be executed at time $t$. This can be computed by observing that the $i^{th}$ action in $\pi$ starts at time $(\pi[:i-1])_D$ and ends at time $(\pi[:i])_D$ for every $i > 1$. Then, we "correct" $t$ by the starting time of that action, to obtain the location of the agent during the execution of that action. The last line in Equation (3) defines that the agent is assumed to stay in its last location after the plan ends.[4]

As in classical MAPF, we define a single-agent plan for an agent $i$ to be a sequence of actions $\pi_i = (a_1, \dots, a_n)$ such that executing it moves agent $i$ from $S(i)$ to $G(i)$. A *conflict* between two single-agent plans is naturally defined as the case where if the agents execute their respective plans starting at the same time then there exists a point in time in which a collision occurs.

**Definition 1** *(Conflict in MAPF$_R$).* Two single-agent plans $\pi_i$ and $\pi_j$ have a conflict iff

$$\exists t \in [0, \max(\pi_{iD}, \pi_{jD})] \ \text{IsCollision}\big(i, j, \pi_{i\varphi}(t), \pi_{j\varphi}(t)\big) \tag{4}$$

A solution to a MAPF$_R$ is valid if all its constituent single-agent plans do not conflict. The cost of a single-agent plan is its duration. Like in classical MAPF, the sum-of-costs of a solution is the sum-of-costs (SOC) of its constituent single-agent plans, and the makespan of a solution is the maximum over these costs. Correspondingly, we define two problems: the problem of finding a SOC-optimal solution to a MAPF$_R$ problem and the problem of finding a makespan-optimal solution to a MAPF$_R$ problem. In this work, we consider both problems and propose algorithms for solving them.

### 3.1. MAPF$_R$ examples

To highlight the differences between MAPF and MAPF$_R$, we provide here two examples of the MAPF$_R$ problems.

[4] In this work we assume that an agent does not disappear when it reaches its goal. Other variants also exist [20,42].

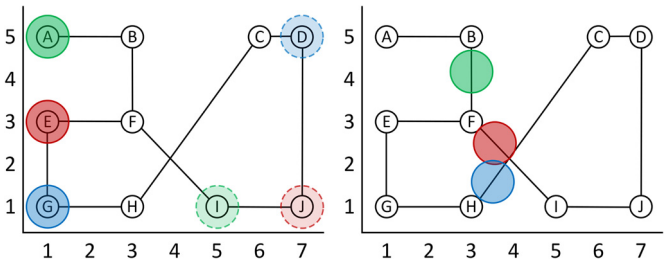



**Fig. 3.** A MAPF$_R$ problem instance with 3 agents. (left) The dashed circle mark the goal location of each agent. (right) The positions of the agents at time moment $t = 2.8$ when following their individual plans. Red and blue agents are in a collision. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

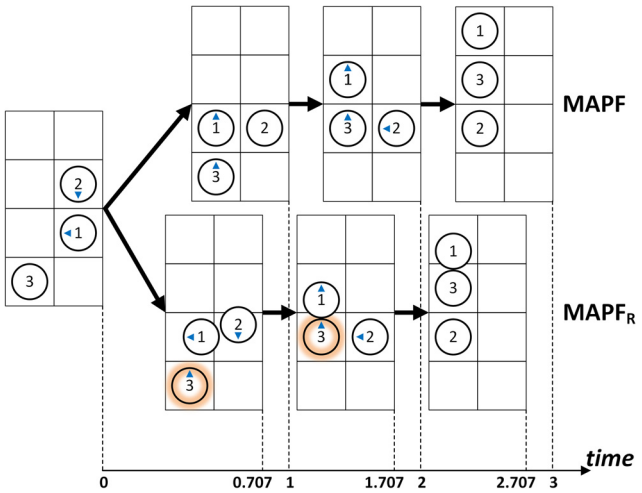



**Fig. 4.** A MAPF$_R$ problem instance with 3 agents where non-unit wait actions are needed to find a SOC-optimal solution.

**Example 1** *(Fig. 3).* The graph vertices and edges are represented by small circles with letters inside and the straight line segments between them, respectively. Agents are shown as colored circles. Each agent is a disk with a radius of 0.5. Initially, the green agent occupies vertex $A$, the red agent occupies vertex $E$, and the blue agent occupies vertex $G$. Their respective goals are $I$ (green agent), $J$ (red agent), and $D$ (blue agent). There are 20 move actions in this example, corresponding to 10 undirected graph edges. We assume the agents move in a constant speed of one unit; they start and stop instantaneously; and move from one vertex to the other following the straight line segment connecting them. Thus, the duration of every action equals the distance between the vertices that define that move action. E.g. the duration of the action "move from $A$ to $B$", denoted as $A \rightarrow B$, is 2. The motion function that describes this action is: $\overrightarrow{OA} + \frac{1}{||AB||} \cdot \overrightarrow{AB} \cdot t$, which is an analytical way to define a constant-velocity straightforward motion between $A$ and $B$. The least-cost individual plans to the agents' respective goals are: $\pi_{green} = \{A \rightarrow B, B \rightarrow F, F \rightarrow I\}$ for the green agent; $\pi_{red} = \{E \rightarrow F, F \rightarrow I, I \rightarrow J\}$ for the red agent; $\pi_{blue} = \{G \rightarrow H, H \rightarrow C, C \rightarrow D\}$ for the blue one. If the agents start executing them simultaneously, then at $t = 2.8$ their positions will be $(3, 4.2)$, $(3 + 0.4 \cdot \sqrt{2}, 3 - 0.4 \cdot \sqrt{2})$, $(3.48, 1.64)$ respectively (see Fig. 3 (right)). The distance between the positions of red and blue agents is less than the sum of their radii, thus, they are colliding and their plans are in conflict.

**Example 2** *(Fig. 4).* In this MAPF$_R$ problem there are 3 disk-shaped agents of size (diameter) $\sqrt{2}/2$ located in a 4-neighborhood $2 \times 4$ grid. Thus, the move actions correspond to moving along the four cardinal directions. The duration of every such move is 1 time unit. The start locations of the agents are illustrated in the grid on the left. The goal locations of the agents are shown on the grid that is on the top-right. The 3 top right grids show the solution found by a SOC-optimal MAPF solver which assumes that minimal wait time equals 1 time unit. The 3 bottom right grids show key steps in the solution found by a SOC-optimal MAPF$_R$ solver. As can be seen, agents 2 and 3 reach their goals earlier in the SOC-optimal MAPF$_R$ solution, and therefore its overall SOC is lower than the SOC-optimal MAPF solution (8.414 vs. 9). This highlights the advantage of allowing arbitrary wait durations. An animation of this example is given in https://github.com/PathPlanning/Continuous-CBS/blob/master/Demos/CCBS_vs_CBS.mp4.

The first example illustrates that in MAPF$_R$ agents may collide even if they do not occupy the same edge or vertex. The second example illustrates that allowing arbitrary wait times enables finding solutions with a lower cost. In particular, consider applying ECBS-CT or E-ICTS on this example. Recall that both algorithms require determining a-priori the duration

of wait actions. However, since the underlying graph here is based on a simple 4-neighborhood grid, it is not obvious how to determine the appropriate wait action duration so as to result in the SOC-optimal MAPF$_R$ solution shown in this example. The algorithms we propose in this work are able to find this solution.

## 4. Conflict-based search with continuous time

In this section, we introduce CCBS – an algorithm that finds SOC-optimal solutions to MAPF$_R$. It is based on two algorithms: CBS [7] and SIPP [15].

### 4.1. From CBS to CCBS

CCBS follows the CBS framework. The main differences between CCBS and CBS are:

- To detect conflicts, CCBS uses the given conflict detection mechanism InConflict, which, in turn, uses the given geometry-aware collision detection mechanism IsCollision.
- To resolve conflicts, CCBS uses a geometry-aware *unsafe-interval detection mechanism*.
- CCBS adds constraints over pairs of actions and time ranges, instead of location-time pairs.
- For the low-level search, CCBS uses a version of SIPP adapted to handle CCBS constraints.

Next, we explain these differences in detail.

#### 4.1.1. Conflict detection in CCBS

CCBS is designed for MAPF$_R$, where agents can have any geometric shape, agents' actions can have any duration, and agents move continuously in time following some motion function. Thus, conflicts can occur between agents traversing different edges, as well as between agents where one is traversing an edge and the other is waiting at a vertex [43,17]. Also, an action $a$ initiated at some time $t$ may conflict with an action $a'$ initiated at some other time $t'$, as long as there is some overlap in their execution time, i.e., as long as $[t, t+a_D] \cap [t', t'+a'_D] \neq \emptyset$. To this end, we define a CCBS conflict with respect to a pair of *timed actions*.

A timed action is a pair $(a, t)$ where $a$ is an action and $t$ is a point in time. To execute a timed action $(a, t)$ means to execute action $a$ starting from time $t$. A CCBS conflict is a tuple $\langle i, j, (a_i, t_i), (a_j, t_j)\rangle$, representing that if agent $i$ executes the timed action $(a_i, t_i)$ and agent $j$ executes the timed action $(a_j, t_j)$ then they will collide. Formally:

**Definition 2** *(CCBS conflict).* $\langle i, j, (a_i, t_i), (a_j, t_j)\rangle$ is a CCBS conflict, denoted $\textsc{InConflict}\Big(i, j, (a_i, t_i), (a_j, t_j)\Big)$, iff

$$\exists t \in [t_i, t_i + a_{i\,D}] \cap [t_j, t_j + a_{j\,D}] : \textsc{IsCollision}\Big(i, j, a_{i\varphi}(t - t_i), a_{j_\varphi}(t - t_j)\Big) \tag{5}$$

Whenever $i$ and $j$ are clear from the context, we omit them and use $\textsc{IsCollision}(m_i, m_j)$ and $\textsc{InConflict}\Big((a_i, t_i), (a_j, t_j)\Big)$. Observe that a CCBS conflict is agnostic to the absolute time the actions are performed, and only considers their relative time, that is, for any constant number $\Delta > 0$,

$$\textsc{InConflict}\Big((a_i, t_i), (a_j, t_j)\Big) \rightarrow \textsc{InConflict}\Big((a_i, t_i + \Delta), (a_j, t_j + \Delta)\Big) \tag{6}$$

The complexity of computing InConflict depends on the shapes of the agents $i$ and $j$ and the motion functions of the actions $a_i$ and $a_j$. For the setting used in our experiments – disk-shaped agents moving in constant speed on straight lines – we used a fast closed-loop collision detection mechanism [35] that runs in $O(1)$. In general, computing InConflict for arbitrary agents' shapes and motion functions translates to the task of collision detection for arbitrary-shaped moving objects, which is a non-trivial problem extensively studied in computer graphics, computational geometry, and robotics [36].

Any single-agent plan $\pi = (a_1, \ldots, a_n)$ can also be viewed as a set of timed actions $\{(a_1, t_1), \ldots, (a_n, t_n), (a_{n+1}, t_{n+1})\}$, where $t_i$ is the time in which $a_i$ is planned to be executed according to $\pi$. $t_1$ is equal to zero, and for all other values of $i = 2, \ldots, n$ it is the duration of the plan up to action $i$, that is, $t_i = (\pi[: (i-1)])_D$. The last timed action $(a_{n+1}, t_{n+1})$ is a "dummy" wait action whose duration is infinite. The purpose of this last timed action is to detect conflicts between an agent that finished its plan and other agents. In CCBS, we say that single-agent plans $\pi_i$ and $\pi_j$ for agents $i$ and $j$ have a CCBS conflict if there exists a pair of timed actions $(a_i, t_i) \in \pi_i$ and $(a_j, t_j) \in \pi_j$ such that $\textsc{InConflict}\Big((a_i, t_i), (a_j, t_j)\Big)$ is true. It is straightforward to see that a pair of single-agent plans have a conflict, as defined in Definition 1, if they have a CCBS conflict.

In our running example from Fig. 3, consider the following single-agent plans for the green, red, and blue agents, represented as sequences of timed actions: $\pi_{green} = \{(A \rightarrow B, 0), (B \rightarrow F, 2), (F \rightarrow I, 4)\}$, $\pi_{red} = \{(E \rightarrow F, 0), (F \rightarrow I, 2), (I \rightarrow J, 2 + 2\sqrt{2})\}$, $\pi_{blue} = \{(G \rightarrow H, 0), (H \rightarrow C, 2), (C \rightarrow D, 7\}$. The plans of the red and blue agents do, indeed, conflict (as was shown on Fig. 3 (right)). CCBS conflict here is: $\langle red, blue, (F \rightarrow I, 2), (H \rightarrow C, 2)\rangle$.

*4.1.2. Resolving conflicts in CCBS*

The high-level search in CCBS runs a best-first search like CBS for classical MAPF. In every iteration, a leaf node $N$ in the CT is selected that has the solution with the smallest cost. The InConflict function is used to check if $N.\Pi$ has a CCBS conflict. If no CCBS conflicts were found, $N$ is declared a goal node and the search halts. Otherwise, the high-level search expands $N$ by choosing one of the CCBS conflicts detected in $N.\Pi$. Let $\langle (a_i, t_i), (a_j, t_j) \rangle$ be that chosen CCBS conflict. CCBS resolves this conflict by generating two new CT nodes, $N_i$ and $N_j$. To compute the constraints to add to $N_i$ and $N_j$, CCBS computes for each timed action its *unsafe intervals* w.r.t the other timed action. The unsafe interval of $(a_i, t_i)$ w.r.t. $(a_j, t_j)$ is the maximal contiguous time interval starting from $t_i$ in which if agent $i$ will perform $a_i$ then it will conflict with the timed action $(a_j, t_j)$. Formally:

**Definition 3** *(CCBS unsafe interval).* $[t_i, t_i^u)$ is the unsafe interval for $(a_i, t_i)$ with respect to $(a_j, t_j)$, where

$$t_i^u = \text{argmin}_{t \in [t_i, t_j + a_{j_D}]} \{\text{InConflict}((a_i, t), (a_j, t_j)) = \mathit{false}\} \tag{7}$$

and $[t_j, t_j^u)$ is the unsafe interval for $(a_j, t_j)$ with respect to $(a_i, t_i)$, where

$$t_j^u = \text{argmin}_{t \in [t_j, t_i + a_{i_D}]} \{\text{InConflict}((a_i, t_i), (a_j, t)) = \mathit{false}\} \tag{8}$$

If there is no time $t \in [t_i, t_j + a_{j_D}]$ for which $\text{InConflict}((a_i, t), (a_j, t_j)) = \mathit{false}$ then $t_i^u$ is not defined mathematically. For such cases, we set $t_i^u$ to be $t_j + a_{j_D}$, indicating that $a_i$ must start after $a_j$ has already finished. Similarly, $t_j^u$ can be undefined when there is no $t \in [t_j, t_i + a_{i_D}]$ for which $\text{InConflict}((a_i, t_i), (a_j, t_j)) = \mathit{false}$, and we set $t_j^u$ to $t_i + a_{i_D}$ in such cases.

A constraint in CCBS is of the form $\langle i, a_i, [t_i, t_i^u) \rangle$, saying that agent $i$ cannot perform $a_i$ in the range $[t_i, t_i^u)$. When the agent's index is clear from the context, we omit it from the definition of the constraint, i.e., just write $\langle a, [t, t^u) \rangle$. Constraints over time intervals, also known as "range constraints", have already been introduced in the context of robust MAPF [44].

For a CCBS conflict $\langle (a_i, t_i), (a_j, t_j) \rangle$, CCBS adds to $N_i$ the constraint $\langle i, a_i, [t_i, t_i^u) \rangle$ and adds to $N_j$ the constraint $\langle j, a_j, [t_j, t_j^u) \rangle$. Section 4.3 discusses methods for computing the unsafe intervals.

For example, assume that we are running CCBS over the $\text{MAPF}_R$ problem depicted in Fig. 3. The high-level search expands a root CT node that contains individual plans of the agents that were planned agnostic to each other. As mentioned above, the plans of the red and blue agents conflict. This CCBS conflict is $\langle red, blue, (F \rightarrow I, 2), (H \rightarrow C, 2) \rangle$. The unsafe interval of the action $(F \rightarrow I, 2)$ w.r.t. action $(H \rightarrow C, 2)$ is $[2.000, 3.743)$, i.e. the first moment of time the red agent might safely start moving from $F$ to $I$ after time 2 is 3.743. The unsafe interval of the action $(H \rightarrow C, 2)$ w.r.t. action $(F \rightarrow I, 2)$ is $[2.000, 3.310)$.

*4.1.3. SIPP for the low-level of CCBS*

The low-level solver of CCBS is based on SIPP [15]. Originally, SIPP uses the trajectories of the dynamic obstacles to compute the safe intervals of the graph vertices. In our cases, we do not have dynamic obstacles. Instead, the safe intervals for every vertex is computed by reasoning over the CCBS constraints passed to SIPP, as follows. A CCBS constraint $\langle i, a_i, [t_i, t_i^u) \rangle$ imposed over a wait action $a_i$ translates to two safe intervals: one that ends at $t_i$ and another that starts at $t_i^u$.

Consider the following example: two CCBS constraints imposed over the wait actions associated with the same graph vertex $v$ are passed to SIPP: $\langle i, a_1, [t_1, t_1^u) \rangle$ and $\langle i, a_2, [t_2, t_2^u) \rangle$ $(t_1 < t_1^u < t_2 < t_2^u)$. Then, the safe intervals for $v$ are the following: $[0; t_1]$, $[t_1^u; t_2]$, $[t_u^2, +\infty)$.[5]

CCBS constraints imposed over move actions are incorporated into SIPP in a different way. Let $\langle i, a_i, [t_i, t_i^u) \rangle$ be a CCBS constraint imposed over a move action that is defined by the graph edge $(v, v')$ and a SIPP search node associated with the vertex $v$. For this search node, we replace the action $a_i$ with action $a_i'$ that starts by waiting in $v$ for duration $t_i^u - t_i$ before moving to $v'$. Formally, $a_i'$ is defined by a duration $a_{i_D}' = a_{i_D} + t_i^u - t_i$ and a motion function

$$a_{i_\varphi}'(t) = \begin{cases} coord(v) & t \leq t_i^u \\ a_{i_\varphi}(t + t_i^u) & t > t_i^u \end{cases} \tag{9}$$

We denote this version of SIPP, which accepts a set of CCBS constraints, as Constrained Safe Interval Path Planning (CSIPP). CSIPP is similar to Soft Conflict Interval Path Planning (SCIPP) by Cohen et al. [14], which is also a variant of SIPP designed to be a low-level search for CBS. CSIPP and SCIPP are fundamentally different in several aspects. First, SCIPP is designed for the multi-agent motion planning (MAMP) problem addressed by Cohen et al, which is different from $\text{MAPF}_R$ as explained in Section 2.2. Second, SCIPP considers constraints over points in time, while CSIPP considers constraints of time intervals. Third, SCIPP is designed to consider soft-constraints so as to allow bounded-suboptimal solutions, while CSIPP is currently designed to return optimal solutions.

[5] Note that the safe intervals include the boundary time moments $t_1$, $t_2$. The rationale behind this is the following. CCBS constrains wait actions that have certain durations, that is – no wait action is possible that starts at $t_1$ or $t_2$. However, the agent can start moving at $t_1$ (or $t_2$) and thus leave the vertex immediately after that time moment, without violating the CCBS constraint.

**Algorithm 1:** CCBS pseudo code.

```
   input: G = (V, E), S, G)
1  foreach agent i do
2      π_i ← A* (G, S(i), G(i))
3  N ← (∅, (π_1, ..., π_k))
4  Create OPEN; Add N to OPEN
5  while OPEN is not empty do
6      N ← pop N from OPEN such that cost(N.Π) = min_{N'∈OPEN} cost(N'.Π)
7      if N.Π has no conflicts then
8          return N.Π
9      ⟨i, j, (a_i, t_i), (a_j, t_j)⟩ ← FindConflict(N.Π)
10     for l ∈ {i, j} do
11         [t_l, t_l^u) ← compute unsafe interval for agent l
12         const ← N.const ∪ {⟨l, a_l, [t_l, t_l^u)⟩}
13         π_l' ← CSIPP(G, S(l), G(l), const)
14         Π' ← (N.Π \ {N.π_l}) ∪ {π_l'}
15         N_l ← (const, Π')
16         Add N_l to OPEN
```

#### 4.1.4. CCBS pseudo-code

Algorithm 1 lists the complete pseudo-code of CCBS. First, a simple A* search is used to find a single-agent plan for each agent ignoring all other agents (line 2). Note that SIPP is not needed as this stage, because the optimal plan for each agent at this stage includes only move actions. This set of single-agent plans is used to create the root of the CT, which is added to the open list (OPEN). Then, in every iteration of the algorithm, we extract the node $N$ from OPEN that represents a solution with a minimal cost, compared to solutions in the other nodes in OPEN (line 6). If $N$ has no CCBS conflicts, we return $N.\Pi$. Otherwise, we choose one of the CCBS conflicts $C = \langle i, j, (a_i, t_i), (a_j, t_j)\rangle$ detected for $N.\Pi$. For each agent in the conflict, i.e., $i$ and $j$, we compute the unsafe interval for its action (line 11), and create a new CT node with the corresponding constraint and a new single-agent plan for that agent. Note that in the pseudo-code we use $N.\pi_l$ to refer to the single-agent plan of agent $l$ in $N.\Pi$. This CT node is added to OPEN, so that it may be chosen for expansion in future iterations.

### 4.2. Theoretical properties

Next, we prove that CCBS is sound, solution complete, and optimal. Our analysis is based on the notion of a *sound* pair of constraints, established by Atzmon et al. [44].

**Definition 4** *(Sound pair of constraints).* For a given MAPF$_R$ problem, a pair of constraints is sound iff in every optimal valid solution it holds that at least one of these constraints hold.

**Lemma 1.** *For any CCBS conflict $\langle (a_i, t_i), (a_j, t_j)\rangle$ and corresponding unsafe intervals $[t_i, t_i^u)$ and $[t_j, t_j^u)$, the pair of CCBS constraints $\langle i, a_i, [t_i, t_i^u)\rangle$ and $\left\langle j, a_j, [t_j, t_j^u)\right\rangle$ is a sound pair of constraints.*

**Proof.** By contradiction, assume that there exists a valid solution to the corresponding MAPF$_R$ problem in which action $a_i$ is performed at time $t_i + \Delta_i \in [t_i, t_i^u)$ and action $a_j$ is performed at time $t_j + \Delta_j \in [t_j, t_j^u)$. This means that

$$\text{InConflict}((a_i, t_i + \Delta_i), (a_j, t_j + \Delta_j)) = \mathit{false} \tag{10}$$

**Case #1:** $\Delta_i > \Delta_j$**.** By definition, $t_i + \Delta_i - \Delta_j$ is in the unsafe interval $[t_i, t_i^u)$ and therefore:

$$\text{InConflict}((a_i, t_i + \Delta_i - \Delta_j), (a_j, t_j)) \tag{11}$$

Due to Equation (6), this means InConflict$((a_i, t_i + \Delta_i), (a_j, t_j + \Delta_j))$, contradicting Equation (10).

**Case #2:** $\Delta_j \geq \Delta_i$**.** Similarly, $t_j + \Delta_j - \Delta_i$ is in the unsafe interval $[t_j, t_j + t_j^u)$ in this case, and thus

$$\text{InConflict}((a_i, t_i), (a_j, t_j + \Delta_j - \Delta_i)) \tag{12}$$

Again, using Equation (6) results in InConflict$((a_i, t_i + \Delta_i), (a_j, t_j + \Delta_j))$ which contradicts Equation (10). □

**Theorem 1.** *CCBS is sound, solution complete, and is guaranteed to return an optimal solution if one exists.*

**Proof.** Soundness follows from the fact that CCBS stops only when the $N.\Pi$ has no conflicts. Solution completeness and optimality proof is similar to the completeness and optimality proof for $k$-robust CBS [44], as follows.

For a CT node $N$, let $\pi(N)$ be all valid MAPF$_R$ solutions that satisfy $N.constraints$, and let $N_1$, and $N_2$ be the children of $N$. For any $N$ that is not a goal node, the following two conditions hold.

1. $\pi(N) = \pi(N_1) \cup \pi(N_2)$
2. $cost(N) \leq \min(cost(N_1.\Pi), cost(N_2, \Pi))$

The first condition holds because $N_1$ and $N_2$ are constrained by a sound pair of constraints (Lemma 1 and Definition 4). The second condition holds because $N.\Pi$ by construction is the lowest cost solution that satisfies the constraints in $N$, and the constraints in $N_1$ and $N_2$ are a superset of $N.constraints$.

Now let $N$ be the root of the CT. $\pi(N)$ is the set of all valid solutions, and thus any valid solution will be either in $\pi(N_1)$ or $\pi(N_2)$. Applying this reasoning recursively yields that every valid solution is always reachable via one of the un-expanded CT nodes. By performing a best-first search over the CT, exploring CT nodes with minimal cost first, CCBS is guaranteed to find an optimal MAPF$_R$ solution. □

CCBS, like CBS, is solution-complete but it is not complete. That is, if a valid solution exists, CCBS will find it, but if a valid solution does not exist then CCBS will not detect this. In an unsolvable MAPF$_R$ problem, CCBS will add more and more constraints indefinitely, eventually generating single-agent plans in which the agents go back and forth between locations they already occupied. In classical MAPF this can be partially overcome by computing an upper bound on the solution cost of a given MAPF problem and pruning CT nodes whose cost exceed this bound. Obtaining an upper bound on the solution cost of a given MAPF problem can be done in polynomial time using Kornhauser et al.'s algorithm [45], as long as the underlying graph is undirected [46]. However, such an upper bound is not known for MAPF$_R$ problems.

### 4.3. Conflict and unsafe interval detection methods

The soundness, solution completeness, and optimality of CCBS rely on having accurate collision detection, conflict detection, and unsafe interval detection mechanisms. That is, we require (1) the collision detection mechanism (IsCollision) to detect a collision iff one exists, (2) the conflict detection mechanism to detect a conflict iff one exists (InConflict), (3) and the unsafe interval detection mechanism returns the maximal unsafe interval for every given pair of actions. Constructing such accurate mechanisms for agents with arbitrary shapes and arbitrary motion-functions is not trivial, however for many individual cases fast and exact solutions exist.

#### 4.3.1. Implementing collision detection

If the agents are modeled as spheres (or disks in 2D), IsCollision is trivially implemented in $O(1)$ by computing the distance between the centers of the agents and comparing this distance to the sum of their radii. When the agents are convex polyhedrons, IsCollision can be implemented in $O(\log(n)\log(m))$, where $m$, $n$ is the number of vertices comprising the polyhedrons [47]. General polyhedra are more difficult to handle. In this case the agents' regions or their surfaces are typically decomposed into convex parts and then collision detection is applied to these parts in a systematic fashion, see [48] for example.

#### 4.3.2. Implementing conflict detection

A general approach to detect conflicts is to sample the agents' motion functions and to apply IsCollision to the sampled positions of the agents. This approach is widespread in robotics, see [49] for example. However, this approach may result in missing conflicts due to an inappropriate sampling strategy. To mitigate this issue, more advanced approaches to conflict detection have been proposed – see Jiménez et al. [36] for an overview. Tang et al. [50] proposed a particular mechanism that accurately solves conflict-detection queries when the agents are represented as triangle meshes (i.e. their bounding surfaces are composed of triangles whose coordinates are known). These InConflict detectors are non-trivial to implement and computationally expensive. Fortunately, in a number of cases exact and fast InConflict implementations can be proposed. For example, when agents are represented as disks that move along straight lines with constant speed, conflict detection can be done in $O(1)$ using a closed-loop formula [35]. Walker and Sturtevant [51] proposed an extension of this formula that is able to handle accelerated movements.

#### 4.3.3. Implementing unsafe interval detection

Computing the unsafe interval of an action w.r.t another action also requires analyzing the kinematics and the geometry of the agents. However, unlike collision and conflict detection, which have been studied for many years and can be computed with closed-loop formulas in some settings, the problem of computing an unsafe interval is less investigated. A naïve general method for computing the unsafe interval for an action $a_i$ is to apply the conflict detection mechanism multiple times, starting from $t = t_i$ and incrementing $t$ by some small $\Delta > 0$ until InConflict returns $false$, meaning the unsafe interval is done. This approach is limited in that the resulting unsafe interval may be larger than the real one. One can extend this approach to get a more accurate solution, as follows. Suppose that InConflict returns $true$ when the start moment

of the action $a_i$ is $t_i + (k-1)\Delta$ and returns *false* if it is $t_i + k\Delta$. Obviously, the true endpoint of the unsafe interval lies in $(t_i + (k-1)\Delta, t_i + k\Delta]$. One can now apply binary search over this interval to identify the unsafe interval endpoint. Theoretically, this search may not converge due to the unlimited number of time moments comprising the interval. However, in practice, these moments are represented as the floating-point approximations thus the search will, indeed, converge. Moreover, the resultant endpoint will be exact in the sense that a computer program is not able to present this endpoint with more accuracy. In our implementation of CCBS we used this approach for unsafe interval computation. Finally, a recent investigation of Walker and Sturtevant [51] describes an approach to compute the "exact minimum delay for collision avoidance" in case agents are disks moving with constant velocities. This can be straightforwardly transformed to the exact computation of the CCBS unsafe intervals.

Overall, when agents are represented as disks that move from one location to the other with constant velocities along the straight lines, there exist fast and exact mechanisms to implement IsCollision, InConflict, and to compute the endpoints of unsafe intervals.

### 4.4. Design choices and implementation details

One of the major design choices when implementing an algorithm from the CBS family is which conflict to choose when processing a high-level node. In classical MAPF, choosing conflicts in an intelligent manner can significantly reduce the number of the expanded CT nodes and speed up the search by several orders of magnitude [52,53]. A particularly effective method for choosing conflicts is to prefer *cardinal* conflicts over *semi-cardinal* conflicts, and *semi-cardinal* conflicts over *non-cardinal* conflicts. A CBS conflict is called cardinal *iff* resolving it using either of the corresponding CBS constraints results in increasing the solution cost. The conflict is semi-cardinal if the solution cost increases only when imposing the corresponding CCBS constraint on only one of the agents involved in the conflict. In all other cases, the conflict is non-cardinal.

Effectively implementing this prioritization of cardinal and non-cardinal conflicts in CCBS is not trivial, since conflict detection in MAPF$_R$ is more costly than in classical MAPF. In our CCBS implementation, we store all detected conflicts with their types (cardinal, semi-cardinal, or non-cardinal) in the nodes of the CCBS constraint tree. When a child CT node is generated, it immediately copies all the conflicts from its parent, except those that were resolved. Then, we detect conflicts only with the newly constructed plan and identify their type (cardinal, semi-cardinal, or non-cardinal). This allowed choosing cardinal and semi-cardinal conflicts first in an effective manner, which in turn proved to be very effective for speeding up CCBS. Note that in a preliminary version of this work [18], we proposed a heuristic for choosing when to detect cardinal conflicts and when to avoid doing so. This heuristic does not yield significant improvement when using the described-above implementation for detecting conflict types.

An additional CCBS design choice that is worth mentioning is the tie-breaking strategy used to choose which CT node to expand from all CT nodes with minimal cost. We used the following tie-breaking strategy. If two or more high-level nodes with the same (minimal) cost exist in the CT tree we prefer the one with the lower number of conflicts. A secondary tie-breaking criterion we used is the number of constraints, preferring to expand first nodes with more constraints. Similar tie-breaking techniques for CT nodes were proposed by Barer et al. [54]. Our implementation of CCBS is in C++ and is available at github.com/PathPlanning/Continuous-CBS.

## 5. An SMT-based approach for makespan-optimal MAPF$_R$

In this section, we present SMT-CCBS, an algorithm that finds makespan-optimal solutions to MAPF$_R$ problems. As it name suggests, SMT-CCBS builds on the SMT-CBS algorithm [16] for classical MAPF. SMT-CCBS breaks the problem of finding a valid makespan-optimal solution in MAPF$_R$ into a sequence of *bounded-cost MAPF$_R$ problems*. A bounded-cost MAPF$_R$ problem is a problem of finding a valid solution to a given MAPF$_R$ problem with makespan lower than a given bound $\mu$. Each of the bounded-cost MAPF$_R$ problems created by SMT-CCBS is solved using an SMT problem-solving procedure. Section 5.1 describes in detail how SMT-CCBS solves each of these bounded-cost problems and Section 5.2 provides a complete pseudo-code for SMT-CCBS. The relation between CCBS and SMT-CCBS is discussed later, in Section 7.1.

### 5.1. Solving bounded-cost MAPF$_R$ problems with SMT

We refer to the SMT-based bounded-cost MAPF$_R$ algorithm we propose as $\mu$SMT-CCBS. Some aspects of $\mu$SMT-CCBS are similar to the SMT-based bounded-cost MAPF algorithm used by SMT-CBS. The propositional skeleton (PS) in $\mu$SMT-CCBS is constructed such that a satisfying assignment to this PS defines a solution to $P$ with makespan at most $\mu$, where $P$ is the MAPF$_R$ problem we are solving. The $DECIDE_T$ procedure in $\mu$SMT-CCBS checks if this solution is valid, i.e., if it contains any CCBS conflicts. We refer to this procedure as to $DECIDE_{MAPF_R}$. If the solution is not valid, $DECIDE_{MAPF_R}$ returns one of the detected CCBS conflicts. Then, the PS is updated so that assignments that satisfy it define a solution in which all the CCBS conflicts returned so far by $DECIDE_{MAPF_R}$ are avoided.

Algorithm 2 lists a high-level pseudo-code for $\mu$SMT-CCBS. $\mu$SMT-CCBS maintains a set $\Psi$ that contains all the CCBS conflicts returned by $DECIDE_{MAPF_R}$ so far. Initially, $\Psi$ is empty (line 1). In every iteration of $\mu$SMT-CCBS, the PS is created for the current set of conflicts (CreatePS($\cdot$) in line 3). A SAT solver is used to search for a satisfying assignment to this PS

**Algorithm 2:** The $\mu$SMT-CCBS algorithm.

```
   Input: P, the MAPF_R problem; μ, the makespan bound
1  Ψ ← ∅
2  while True do
3      PS ← CreatePS(Ψ, P, μ)
4      Π ← Solve(PS)
5      if No solution found (i.e., Π is null) then
6          return null
7      Con ← DECIDE_{MAPF_R} (Π, P, μ)
8      if No conflict found (i.e., Con is null) then
9          return Π
10     Add Con to Ψ
```

(line 4). If no solution exists, the given decision problem is unsolvable, i.e., there are no solutions to $P$ with makespan equal to or smaller than $\mu$. Otherwise, we apply $DECIDE_{MAPF_R}$ to check if the found solution is a valid $\text{MAPF}_R$ solution (line 7). If it is a valid solution, we return it. Otherwise, $DECIDE_{MAPF_R}$ returns one of the CCBS conflicts that exist in this solution. The returned CCBS conflict is added to the set of conflicts $\Psi$. This process continues until either a valid solution is found (line 9) or we establish that no solution exists (line 6).

Our $DECIDE_{MAPF_R}$ procedure is exactly the same as the conflict-detection step in CCBS. It applies a conflict-detection mechanism to check for collisions between the agents' plans in the given solution. The process of generating our PS for a given set of CCBS conflicts, $\text{MAPF}_R$ problem, and makespan bound – CreatePS($\Psi$, P, $\mu$) – is more involved and we describe it next.

*5.1.1. Generating the propositional skeleton*

CreatePS consists of the following steps.

1. Compute a set of CCBS constraints for the set of conflicts in $\Psi$.
2. Compute for each agent a set of single-agent plans that satisfy these constraints.
3. Create a PS for choosing a solution from these sets of single-agent plans.
4. Add constraints to verify the chosen solution avoids all conflicts in $\Psi$.

Next, we describe the details of each step of CreatePS. As we will see, these steps are designed to obtain a behavior similar to CCBS. In particular, in the first step, CreatePS computes all the constraints CCBS would impose to resolve the conflicts in $\Psi$, and in the second step, CreatePS computes efficiently all single-agent plans that CSIPP would generate to satisfy every subset of these constraints.[6]

**Step 1. Compute the set of CCBS constraints for $\Psi$.** For every CCBS conflict $Con$ in $\Psi$, we generate the pair of CCBS constraints CCBS would create to resolve $Con$. This pair of CCBS constraints is specified in Lemma 1. Note that to compute such a pair of constraints, we need to compute the unsafe intervals for the respective conflict. By $const(Con)$ we denote the pair of CCBS constraints for a conflict $Con \in \Psi$, and by $const(\Psi)$ – the set of all pairs of CCBS constraints were generated this way, i.e.,

$$const(Con) = const(\langle (a_i, t_i), (a_j, t_j) \rangle) = \{\langle i, a_i, [t_i, t_i^u) \rangle, \langle j, a_j, [t_j, t_j^u) \rangle\} \tag{13}$$

$$const(\Psi) = \{const(Con) \mid Con \in \Psi\} \tag{14}$$

**Step 2. Compute the set of relevant single-agent plans.** For each agent, we compute all single-agent plans CSIPP would generate for that agent given any subset of constraints in $const(\Psi)$ that involve that agent. To compute and store this set of single-agent plans efficiently, we create for each agent a specific *Multi-Value Decision Diagram* (MDD) [30] that we refer to as $MDD_R$. An MDD is a direct acyclic graph with a single source and sink.[7] MDDs have been used in the context of MAPF before, e.g., in the ICTS and E-ICTS algorithms [8,13]. Every node in our $MDD_R$ is a pair $(u, t)$ where $u$ is a vertex in $\mathcal{G}$ and $t$ is a point in time. An edge $((u, t)(v, t'))$ corresponds to a timed action from $u$ to $v$ that starts at $t$ and ends in $t'$. We generate our $MDD_R$ by performing a variant of Dijkstra's algorithm that considers all possible wait actions CSIPP may include to avoid any constraint in $const(\Psi)$. Algorithm 3 describes in detail how this is done.

The input to Algorithm 3 is a $\text{MAPF}_R$ problem $\Pi$, a set of CCBS constraints $const(\Psi)$, a makespan bound $\mu$, and an agent $i$. $X^i$ and $E^i$ in Algorithm 3 are the nodes and edges of the $MDD_R$ it generates for agent $i$. The root node of this $MDD_R$ is $(S(i), t)$. Initially, $X^i$ contains only the root node and $E^i$ is empty. OPEN is a collection of $MDD_R$ nodes ordered by their time points, which is initialized with the $MDD_R$ root node. In every iteration, the best (minimal time) node $(u, t)$ in OPEN

[6] Note that CreatePS does so without explicitly running CSIPP an exponential number of times.

[7] Technically, an MDD can have multiple sinks, where each sink is labeled as either true or false. This is equivalent to having a single sink that gathers all sinks labeled as true, and removing all branches that only end up in sinks labeled as false.

**Algorithm 3:** Generate $MDD_R$ for agent $i$.

```
1  GenerateMdd_R (Π, const(Ψ), μ, i)
2      X^i ← {(S(i), 0)}; E^i ← ∅; OPEN ← ∅
3      insert (S(i), 0) into OPEN
4      while OPEN ≠ ∅ do
5          (u, t) ← pop (u, t) from OPEN where t = min_t(OPEN)
6          if t ≤ μ then
7              foreach a ∈ A such that from(a) = u and t + a_D ≤ μ do
8                  insert (to(a), t + a_D) into OPEN
9                  X^i ← X^i ∪ {(to(a), t + a_D)}
10                 E^i ← E^i ∪ {[(u, t); (to(a), t + a_D)]}
11                 foreach ⟨i, a, [t', t'^u)⟩ ∈ const(Ψ) do
12                     if t ∈ [t', t'^u) and t'^u + a_D ≤ μ then
13                         insert (u, t'^u) into OPEN
14                         X^i ← X^i ∪ {(u, t'^u)}
15                         E^i ← E^i ∪ {[(u, t); (u, t'^u)]}
16                 foreach ⟨i, a', [t', t'^u)⟩ ∈ const(Ψ) where a' is wait at to(a) do
17                     if t < t'^u − a_D and t'^u ≤ μ then
18                         insert (u, t'^u − a_D) into OPEN
19                         X^i ← X^i ∪ {(u, t'^u − a_D)}
20                         E^i ← E^i ∪ {[(u, t); (u, t'^u − a_D)]}
21     return (X^i, E^i)
```

is removed from OPEN and expanded. The children of $(u, t)$ are generated as follows. For every move action $a \in \mathcal{A}$ that starts with $u$ and ends before the makespan bound (i.e., $t + a_D \leq \mu$), we generate a child node $(to(a), t + a_D)$, which represents performing action $a$ at time $t$ (line 9-10). If there exists a CCBS constraint that prohibits agent $i$ from performing $a$ at time $t$, we create an *additional* node in the $MDD_R$ that represents the option of waiting at $u$ until $a$ can be performed without conflicting with this constraint (line 11-15). If there is a CCBS constraint that prohibits waiting at $to(a)$ then we create an additional node in the $MDD_R$ to allow the agent to stay at $from(a)$ until it can apply $a$ to reach $to(a)$ immediately after the constraint on waiting at $to(a)$ ends (line 16-20). Note that the generated $MDD_R$ includes single-agent plans that violate some of the given CCBS constraints. For example, for a move action $a \in \mathcal{A}$ and an $MDD_R$ node $(u, t)$, Algorithm 3 will create a child node $(to(a), t + a_D)$ even if there is a CCBS constraint that prohibits performing $a$ at that time. This is done so that the generated $MDD_R$ represents all single-agent plans CSIPP would return given *any* subset of the given CCBS constraints. The create PS described below will ensure that the set of single-agent plans chosen will comprise a valid solution if possible.

**Example 3.** Fig. 5 shows the $MDD_R$ structures for the agents in our running example (Fig. 3). The upper part shows the $MDD_R$ structures created for each agent w.r.t. an empty set of constraints. In our example, each agent has a single shortest single-agent plan, and so the $MDD_R$ created for an empty set of constraints is a line. The red and blue agents' plans have a conflict between the timed actions $(F \rightarrow I, 2.000)$ and $(H \rightarrow C, 2.000)$. The unsafe intervals for these actions are $[2.000, 3.310)$ and $[2.000, 3.743)$, respectively. So, the CCBS constraints generated for this conflict are

$$\{\langle red, (F \rightarrow I, 2.000), [2.000, 3.310)\rangle, \langle blue, (H \rightarrow C, 2.000), [2.000, 3.743)\rangle\} \tag{15}$$

The lower part of Fig. 5 shows the $MDD_R$ generated for each agent when considering the pair of CCBS constraints in Equation (15).

**Step 3. Create a PS.** We create in this step a PS such that every satisfying assignment to it corresponds to a MAPF$_R$ solution in which each agent follows a single-agent plan from its $MDD_R$. To create such a PS, we create Boolean variables $\mathcal{E}^{t,t'}_{v,v'}(i)$ and $\mathcal{X}^t_v(i)$ for every edge and node in $E^i$ and $X^i$ from $MDD_R$ respectively. Setting $\mathcal{X}^t_v(i)$ to true represents that agent $i$ plans to occupy location $v$ in time $t$. Setting $\mathcal{E}^{t,t'}_{v,v'}(i)$ to true represents that agent $i$ plans performs an action that starts at time $t$, ends at time $t'$, and moves the agent from $v$ to $v'$. Wait actions from the $MDD_R$ are represented in our PS by $\mathcal{E}^{t,t'}_{v,v'}(i)$ variables with $v = v'$. To ensure that an assignment to all these variables constitutes a solution to the given MAPF$_R$ problem, we add the following restrictions over these Boolean variables to the PS.

$$\mathcal{X}^t_v(i) \Rightarrow \bigvee_{(v',t') \mid [(v,t);(v',t')] \in E^i} \mathcal{E}^{t,t'}_{v,v'}(i), \tag{16}$$

$$\sum_{(v',t') \mid [(v,t);(v',t')] \in E^i} \mathcal{E}^{t,t'}_{v,v'}(i) \leq 1 \tag{17}$$

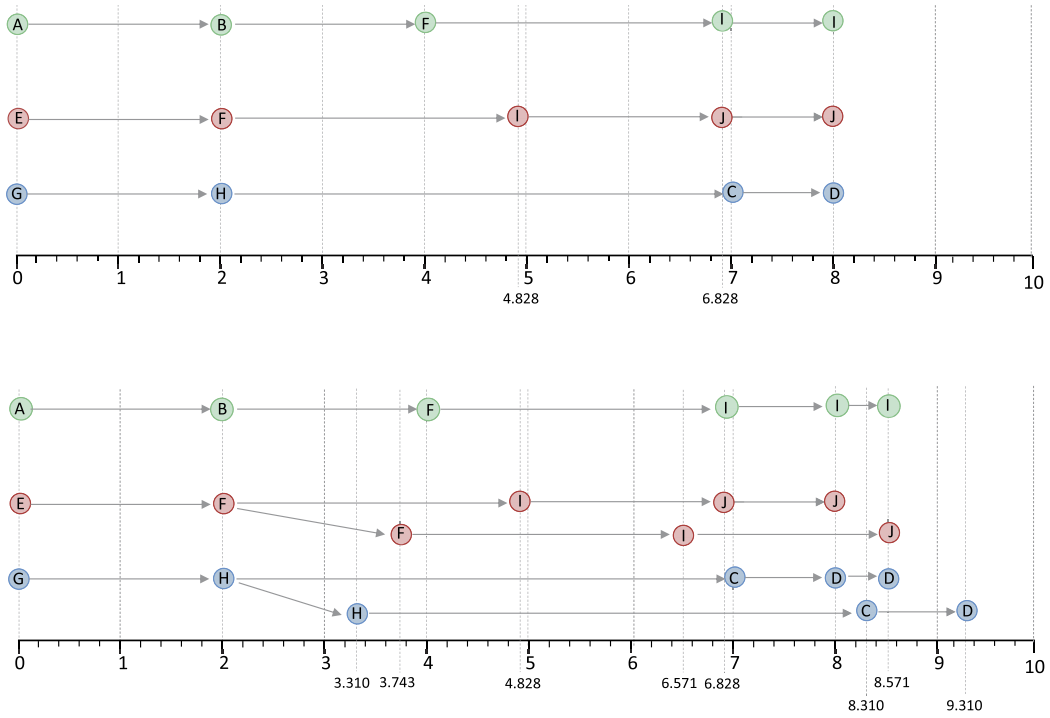



**Fig. 5.** An illustration of $MDD_R$ s. The upper part shows an initial $MDD_R$ for the MAPF$_R$ problem in Fig. 3. The lower part shows the $MDD_R$ for the same problem after resolving a conflict between the red and blue agents traversing $(F, I)$ and $(H, C)$ respectively.

$$\mathcal{E}^{t,t'}_{v,v'}(i) \Rightarrow \mathcal{X}^{t'}_{v'}(i) \tag{18}$$

Equations (16) and (17) state that if agent $i$ appears in vertex $v$ at time $t$ then it has to leave through exactly one edge connected to $v$. Equation (18) establishes that once an agent enters an edge in the $MDD_R$ it has to leave it at its endpoint. We also add to the PS clauses that verify every agent starts in its start location and ends in its goal. Thus, a satisfying assignment to this formula represents a solution to the given MAPF$_R$ problem.

**Step 4. Add constraints to verify all conflicts are avoided.** To avoid all the CCBS conflicts in $\Psi$, we verify that for every conflict $Con \in \Psi$ one of the CCBS constraints in $const(Con)$ is satisfied. This is done by adding to the PS created in step 3 an appropriate disjunction. That is, for a conflict $\langle (a_i, t_i), (a_j, t_j) \rangle$, where $a_i$ is the action that moves the agent $i$ from $v$ to $v'$ and $a_j$ – moves agent $j$ from $u$ to $u'$, we add the following clause:

$$\neg\mathcal{E}^{t_i, t_i + a_{iD}}_{v,v'}(i) \vee \neg\mathcal{E}^{t_j, t_j + a_{jD}}_{u,u'}(j) \tag{19}$$

This clause represents mutual exclusion between the two conflicting actions. Thus, having this clause in the PS ensures at most one of these timed actions will be applied in any satisfying solution.

**Example 4.** Consider again our running example from Fig. 3, and the three $MDD_R$ structures shown in the upper part of Fig. 5. The variables created for the corresponding PS, are:

$$\mathcal{X}^{0.000}_{A}(1), \mathcal{X}^{2.000}_{B}(1), \mathcal{X}^{4.000}_{F}(1), \mathcal{X}^{6.828}_{I}(1), \mathcal{X}^{8.000}_{I}(1), \mathcal{E}^{0.000,2.000}_{A,B}(1), \mathcal{E}^{2.000,4.000}_{B,F}(1), \mathcal{E}^{4.000,6.828}_{F,I}(1), \mathcal{E}^{6.828,8.000}_{I,I}(1)$$

for agent 1 (green)

$$\mathcal{X}^{0.000}_{E}(2), \mathcal{X}^{2.000}_{F}(2), \mathcal{X}^{4.828}_{I}(2), \mathcal{X}^{6.828}_{J}(2), \mathcal{X}^{8.000}_{J}(2)\mathcal{E}^{0.000,2.000}_{E,F}(2), \mathcal{E}^{2.000,4.828}_{F,I}(2), \mathcal{E}^{4.828,6.828}_{I,J}(2), \mathcal{E}^{6.828,8.000}_{J,J}(2)$$

for agent 2 (red), and

$$\mathcal{X}^{0.000}_{G}(3), \mathcal{X}^{2.000}_{H}(3), \mathcal{X}^{7.000}_{C}(3), \mathcal{X}^{8.000}_{D}(3)\mathcal{E}^{0.000,2.000}_{G,H}(3), \mathcal{E}^{2.000,7.000}_{H,C}(3), \mathcal{E}^{7.000,8.000}_{C,D}(3)$$

for agent 3 (blue). These variables are used in the constraints defined by Equations (16)-(19). In addition, we establish that each agent starts and ends in its start and goal location by setting the variables $\mathcal{X}^{0.000}_{A}(1)$, $\mathcal{X}^{0.000}_{E}(2)$, $\mathcal{X}^{0.000}_{G}(3)$ (start positions) and $\mathcal{X}^{8.000}_{I}(1)$, $\mathcal{X}^{8.000}_{J}(2)$, $\mathcal{X}^{8.000}_{D}(3)$ (goal positions) to *true*. The resulting formula can be solved easily by setting all variables to *true*, which corresponds to the solution in which the agents do not wait in any position and go directly to

their goals: agent 1 (green) goes from A to B to F to I, agent 2 (red) goes from E to F to I to J, and agent 3 (blue) goes from G to H to C to D. As noted earlier, this solution has a conflict, and the $MDD_R$ in the lower part of Fig. 5 is created by considering the CCBS constraints designed to resolve this conflict. The PS created for the three $MDD_R$s in the lower part of Fig. 5 includes additional variables representing new timed actions and locations the agent may now occupy. These variables are:

$$\text{Agent 1 (green)}: \mathcal{X}_I^{8.571}(1)\mathcal{E}_{I,I}^{8.000,8.571}(1)$$

$$\text{Agent 2 (red)}: \mathcal{X}_F^{3.743}(2), \mathcal{X}_I^{6.571}(2), \mathcal{X}_J^{8.571}(2), \mathcal{E}_{F,F}^{2.000,3.743}(2), \mathcal{E}_{F,I}^{3.743,6.571}(2), \mathcal{E}_{I,J}^{6.571,8.571}(2)$$

$$\text{Agent 3 (blue)}: \mathcal{X}_H^{3.310}(3), \mathcal{X}_C^{8.310}(3), \mathcal{X}_D^{9.310}(3), \mathcal{E}_{H,H}^{2.000,3.310}(3), \mathcal{E}_{H,C}^{3.310,8.310}(3), \mathcal{E}_{C,D}^{8.310,9.310}(3)$$

Additional clauses are added for these variables following Equations (16)–(18). In addition, the following clause it added as specified in Equation (19)

$$\neg\mathcal{E}_{F,I}^{2.000,4.828}(2) \vee \neg\mathcal{E}_{H,C}^{2.000,7.000}(3), \tag{20}$$

to verify that the conflict identified in the previous solution will be avoided.

**Theorem 2.** *$\mu$SMT-CCBS is a sound and complete algorithm for bounded-cost MAPF$_R$, i.e., every solution returned by $\mu$SMT-CCBS is indeed a solution with makespan equal to or smaller than the makespan bound, and if such a solution exists then it will be found.*

Soundness is established by the fact that $DECIDE_{MAPF_R}$ verifies that the returned solution is valid. Establishing completeness, however, is less trivial. The key understanding is that performing wait actions is only beneficial if it eventually enables performing a move action. Therefore, it is sufficient to only consider the wait actions' duration specified in lines 12 and 17. A formal proof is given in Appendix B.

### 5.2. SMT-CCBS

**Algorithm 4:** Pseudo code for SMT-CCBS.

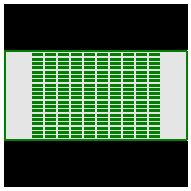
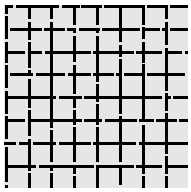
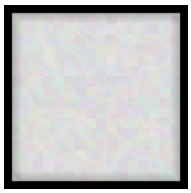
```
   Input: P, the MAPF_R problem
1  Π ← {π_1, ..., π_k}, where π_i is the shortest single-agent plan for agent a_i
2  μ ← max_{i=1}^{k} μ(π_i)
3  while True do
4      Π ← μSMT-CCBS (P, μ)
5      if Π is not null then
6          return Π
7      μ ← Compute next μ
```

Finally, we can present our SMT-CCBS algorithm. Pseudo code is listed in Algorithm 4. It starts by setting the makespan-bound $\mu$ to the single-agent plan with the longest duration. Then, it runs $\mu$SMT-CCBS to try to find a valid solution whose makespan is $\mu$. If it succeeded, it returns that solution. Otherwise, $\mu$ is increased to the next relevant makespan (line 7). The next relevant makespan is set by adding to $\mu$ the minimal amount that enables adding at least one more node to one of the $MDD_R$ structures. We computed this minimal amount when generating the $MDD_R$ for each agent (Algorithm 3). There, whenever an MDD node is pruned for exceeding the makespan bound $\mu$ (lines 7, 12, and 17), we keep track of the gap between $\mu$ and the minimal makespan bound in which that node would not have been pruned. The minimal gap over all pruned nodes and agents is exactly the minimal increase in $\mu$ that would allow an additional action. Appendix C lists the complete pseudo code for generating an $MDD_R$ for an agent and computing the next makespan bound.

**Theorem 3.** *SMT-CCBS is sound, solution complete, and is guaranteed to return a makespan-optimal solution.*

**Proof.** Since every solution returned by SMT-CCBS was also returned by $\mu$SMT-CCBS, and $\mu$SMT-CCBS is sound (Theorem 2), then SMT-CCBS is sound as well. The initial value of $\mu$ is set as a lower bound on the optimal makespan, since no agent can reach faster to its goal than when it ignores all other agents. In every subsequent iteration of SMT-CCBS, $\mu$ is incremented by the minimal amount required to allow Algorithm 3 to add at least a single node to one of the $MDD_R$ structures. Let $\Delta_\mu$ be this minimal amount. By definition, increasing $\mu$ by any amount smaller than $\Delta_\mu$ will result in exactly the same $MDD_R$ structures as created for $\mu$. Consequently, $\mu$SMT-CCBS will not find a valid solution for any makespan bound smaller than $\mu + \Delta_\mu$. Thus, solution completeness and makespan-optimality of SMT-CCBS directly follows from the completeness of $\mu$SMT-CCBS (Theorem 2). □

**Table 1**
Details about the grid graphs used in our experiments, including their source files in the grid MAPF benchmark, their dimensions, and the numbers of free cells.

| **warehouse** | **rooms** | **den520d** | **empty16x16** |
|---|---|---|---|
| `warehouse-10-20-10-2-2` | `room-64-64-8` | `den520d` | `empty-16-16` |
|  |  |  |  |
| $161 \times 63$<br>9,776 free cells | $64 \times 64$<br>3,232 free cells | $256 \times 257$<br>28,176 free cells | $16 \times 16$<br>256 free cells |

SMT-CCBS, like CCBS, is also only solution-complete as opposed to complete. Given an unsolvable MAPF$_R$ problem, SMT-CCBS will continue to increase its makespan bound indefinitely. However, if an upper bound on the makespan of the given MAPF$_R$ problem is available, then SMT-CCBS can be modified to be complete by halting when $\mu$ reaches that bound.

Moreover, $\mu$SMT-CCBS is complete for any input parameter $\mu$, hence we can easily modify SMT-CCBS to a bounded sub-optimal planner by changing the $\mu$ increasing strategy at the high level. Instead of the minimum increase of $\mu$ guaranteeing the optimality, a larger amount of increase can be used.

### 5.3. Design choices and implementation details

For our implementation of SMT-CCBS we used Glucose 3.0 as the underlying SAT solver [55]. Glucose 3.0, as well as other modern SAT solvers, can be run in an *incremental* manner [25]. This means that the SAT solver re-uses information from its previous call to speedup its next call, namely learnt clauses. This is particularly useful in SMT-CCBS, because it calls the SAT solver multiple times when solving a given MAPF$_R$ problem, adding more clauses. Moreover, the instances being sequentially solved by the SAT solver are very similar to each other, differing only in relatively few new binary clauses which empowers the role of clause learning from previous calls.

Preliminary experiments with SMT-CBS, the previous SMT-based solver for the discrete variant of MAPF, showed that it is better to collect and reflect all conflicts discovered after each plan validation step rather than choosing a subset of them according to any preference [56]. It also turned out to be good to transfer a set of conflicts to the next value of the objective in the iterative scheme. We follow the analogous design choice in SMT-CCBS too. That is, we collect and maintain the set of conflicts discovered by $DECIDE_{MAPF_R}$ throughout the entire course of the SMT-CCBS algorithm. Our implementation of SMT-CCBS is written in C++ and available at github.com/surynek/boOX.

## 6. Experimental evaluation

We implemented CCBS and SMT-CCBS, and evaluated their performance on a range of MAPF$_R$ problems. In this section, we report the results of this evaluation.

### 6.1. Experimental setup

In our experiments, we used two types of graphs: $2^k$-neighborhood grids [57] and graphs that represent roadmaps.

#### 6.1.1. $2^k$-Neighborhood grids

For the $2^k$-neighborhood grid graphs, we used four grids from the recently introduced grid-based MAPF benchmark [20] in the Moving AI repository [58].[8] Table 1 provides a visual representation of these grids, as well as other statistics such as the dimension of each grid and the number of non-obstacle cells. We chose these specific grids as they represent different settings in which MAPF$_R$ problems arise: video-games (**den520d**), warehouse logistics (**warehouse**), indoor robotics (**rooms**), and field robotics (**empty16**$\times$**16**). The corridors on the Warehouse grid are 2 cells in width, the width of the entrances on the Rooms grid is 1 cell.

For each grid we created a graph whose vertices are the grid cells, and edges connect every grid cell to the $2^k$ grid cells closest to it, where $k$ varied from 2 to 5 (see Fig. 6). We ignored inertial effects and assumed the moving speed of each agent is one grid cell per one time unit. That is, in one time unit an agent covers the segment whose length is equal to the distance between two adjacent grid cells. This does not mean the duration of all move actions is unit. For example, the duration of a move action from grid cell $(x, y)$ to $(x+1, y+1)$ is $\sqrt{2}$.

[8] movingai.com/benchmarks/mapf.html.

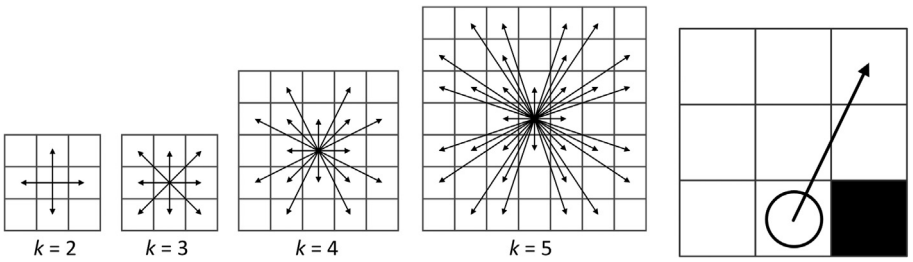
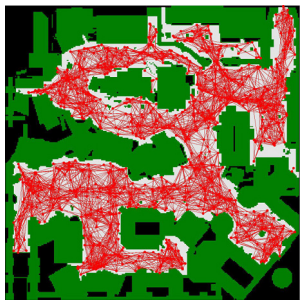



**Fig. 6.** (*Left*) Illustration of the $2^k$ neighborhood for $k = 2, 3, 4$, and 5. (*Right*) An invalid move on a 16-neighborhood grid. Although the source and target cells are unblocked and the line connecting them does not intersect the blocked cell, a disk-shaped agent of radius $\sqrt{2}/4$ will run into the latter in case the move is executed.

**Table 2**
Details about the roadmap graphs used in our experiments, including the number of vertices and edges in each graph.

| sparse | dense | mega-dense |
|---|---|---|
|  |  |  |
| 169 vertices, 349 edges | 878 vertices, 7,341 edges | 11,342 vertices, 263,533 edges |

Agents' shapes were disks of radius $\sqrt{2}/4$. The shape of an agent was considered when checking collisions with static obstacles as well as other agents. Thus, if a move action starts and ends in empty grid cells but the disk representing the agent intersects with a blocked cell while performing this move action, then this action is prohibited. An example of this is given in Fig. 6 (right).

This grid-based MAPF benchmark [20] also provides *scenario* files for each grid. Each scenario file lists start and goal locations for the agents. We used this list to create MAPF$_R$ problem instances as suggested by the maintainers of the benchmark [20]. That is, we created a MAPF$_R$ problem instance with two agents whose start and goal locations are given in the first two lines of the scenario file. Then, we created a new MAPF$_R$ problem instance by adding a third agent whose start and goal locations are given in the third line of the scenario file. This process is repeated to create a sequence of MAPF$_R$ problem instances with more and more agents. To evaluate an algorithm on this sequence of MAPF$_R$ problem instances, we run it until a MAPF$_R$ problem instance is reached that the evaluated algorithm cannot solve in the given time limit. In our experiments, we generated such sequences of MAPF$_R$ problem instances for all the 25 `random` scenario files available for each grid. Overall, for each grid and a specific number of agents we ran the evaluated algorithms on 25 different MAPF$_R$ problem instances.

### 6.1.2. Roadmap graphs

The second type of graphs we used in our experiments represents *roadmaps*. These graphs were generated by processing the `den520d` grid graph with a roadmap-generation tool from the Open Motion Planning Library (OMPL) [59], which is a widely used tool in the robotics community. Specifically, we created three graphs which we refer to as the `sparse`, `dense`, and `mega-dense` roadmap graphs. As their names suggest, these graphs differ in their sizes and density. Table 2 lists the exact number of vertices and edges in each graph and shows them visually.[9] The finite set of move actions $\mathcal{A}$ for each MAPF$_R$ problem instances created for these road graphs comprise two move actions for each edge in the roadmap graph, corresponding to crossing that edge in each direction at a fixed speed. We created a suite of such MAPF$_R$ problem instances in a similar way as described above for the grid-based graphs, creating 25 scenario files for each roadmap graph such that each scenario file contains non-overlapping start-goal pairs chosen randomly out of the graph vertices. All the scenario files used in our experiments are publicly available at: github.com/PathPlanning/Continuous-CBS/tree/master/Instances.

[9] The `mega-dense` graph contains so many edges that they visually overlap.

#### 6.1.3. Evaluation metrics and time limit

We considered three main metrics in our evaluation. The first metric is the *success rate*, which is the ratio of problem instances solved under a given time limit. The second and third metrics are the SOC and makespan, respectively, of the returned solution. These metrics are common in the MAPF literature. Unless stated otherwise, we set the time limit to 30 seconds. We chose this timeout to study the applicability of the suggested approaches in applications where the time allocated to pathfinding is very minimal. Such applications are prevalent in robotics, digital entertainment, and many other real-work problems. However, we also performed dedicated experiments with different time limits, to investigate how the evaluated algorithms scale with time.

### 6.2. Grid results

Fig. 7 depict the success rates of CCBS and SMT-CCBS under the 30 second time limit for different number of agents ($x$-axis) in our $2^k$-neighborhood grids. Plots of different colors correspond to results for different values of $k$ (neighborhood size).

The results show that CCBS and SMT-CCBS can find SOC-optimal and makespan-optimal solutions, respectively, under the 30 second time limit for MAPF$_R$ problem instances with several dozens of agents. For example, both CCBS and SMT-CCBS solved 80% of problem instances for 24 agents on the `warehouse` grid for all evaluated values of $k$ (2, 3, 4, and 5). Note that state-of-the-art solvers for classical MAPF can find SOC/makespan-optimal solutions on similar grids with many more agents. This is expected as these solvers solve an easier problem (MAPF as opposed to MAPF$_R$), do not perform time-consuming conflict detection and unsafe-intervals estimation procedures, and their constraints over vertices/edges are more restrictive.

Consider now the impact of increasing the neighborhood size, i.e., increasing $k$. For both algorithms, in general, increasing $k$ leads to lower success rates. This is an expected effect resulting from the difference in branching factor, which is 5 for $k = 2$ and goes up to 32 for $k = 5$. However, for some settings, the difference in algorithms' performance for $k = 2$ and $k = 3$ is negligible, and setting $k$ to 3 can actually lead to better results. Consider for example results of CCBS on `den520` or the results of SMT-CCBS on `warehouse`, when the number of agents is below 30. In these examples, setting $k = 3$ improves the success rate in most cases. A possible explanation is that increasing $k$ also allows the agents to find shorter single-agent plans. This can result in a faster low-level search for CCBS, and smaller $MDD_R$ structures for SMT-CCBS, both of which may reduce the overall runtime. In other words, increasing $k$ means a search space with a larger branching factor, but also having a potentially smaller depth.

Relating the SMT-CCBS results to the CCBS results is somewhat problematic, since the former aims to optimize makespan while the latter aims to optimize SOC. Nevertheless, one can see that there is no universal winner, where CCBS is able to solve problem instances with more agents in `den520d` and `warehouse` grids while SMT-CCBS solves problem instances with more agents in `empty` $16 \times 16$ grids. These results are consistent with the common observation that SAT-based methods work well for small and dense graphs while CBS-based methods excel in larger and sparser maps [56].

To better understand the scalability of CCBS and SMT-CCBS, we repeated the above experiments with $k = 3$ and timeouts of 1, 10, 30, 60, 150, and 300 seconds. The results are depicted on Fig. 8. The $y$-axis shows the total number of solved instances for each grid across the different number of agents. As expected, increasing the timeout allows both algorithms to solve more instances. However, CCBS quickly reaches a plateau in which extending the timeout does not allow solving significantly more instances. The most notable increase in the performance of CCBS is visible only for going from a 1 s timeout to 30 s. On the other hand, SMT-CCBS is able to gain more by increasing the timeout in all maps except `empty`, where the difference between the number of tasks solved under 30 s timeout and 300 s timeout is considerable. For example, this increase in timeout results in SMT-CCBS solving approximately 1.5 times more problem instances in the `den520` and `warehouse` grids.

Allowing for longer runtime is especially beneficial for SMT-CCBS because more time can be spent in the SAT solving phase and this time can be utilized by the SAT solver to fully employ its learning mechanism to prune the search space. During long runs of the SMT-CCBS solver, the high-level phases that construct the formula become relatively less time-consuming while the SAT solving phase is increasingly dominant with respect to the overall runtime. Hence the efficiency of the SAT solver in pruning the search space represented by formulae modeling the MAPF$_R$ problem becomes more pronounced.

Overall, these results suggest that when the time budget is low (1-30 s) CCBS should be preferred, when this budget is high (more than 3 minutes) than the preference should be given to SMT-CCBS.

#### 6.2.1. Solution cost results

Next, we analyze the costs — makespan and SOC — of the solutions returned by CCBS and SMT-CCBS. To allow comparing these costs for both algorithms and between values of $k$, we considered only MAPF$_R$ problem instances that were successfully solved by both algorithms for all values of $k$. For each problem instance, algorithm, and value of $k$ we computed two additional metrics: (1) *SOC gain ratio*, which is the ratio between the SOC of the returned solution and SOC of the solution returned by CCBS ($k = 2$); and (2) *makespan gain ratio*, which is the ratio between the makespan of the returned solution and makespan of the solution returned by SMT-CCBS ($k = 2$). For example, if the SOC gain ratio for an algorithm for a

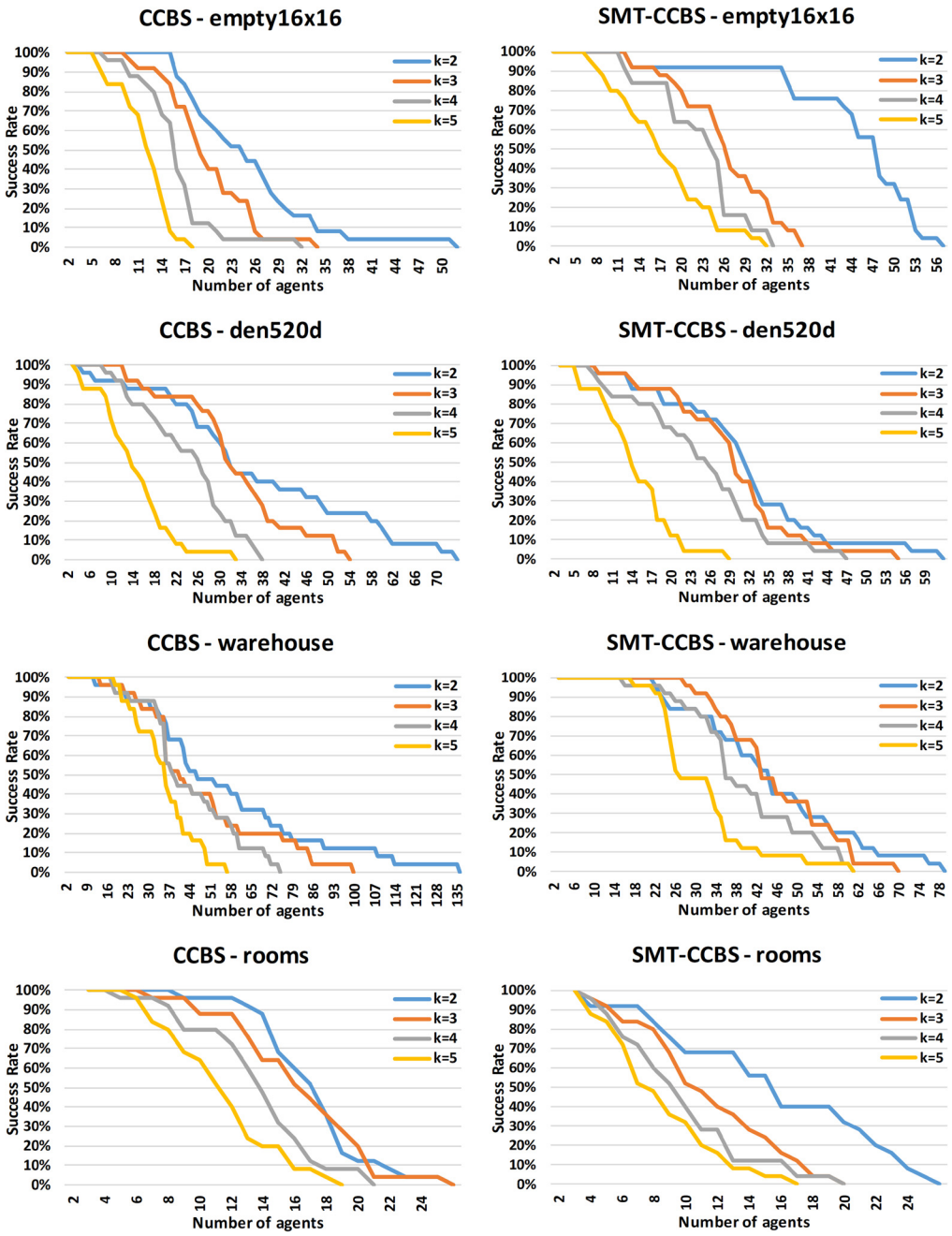



**Fig. 7.** Success rate for CCBS (left plots) and SMT-CCBS (right plots) on grid maps. Please note that CCBS and SMT-CCBS optimize different objective functions.

particular $\text{MAPF}_R$ problem instance is 0.75, it means that the SOC of the solution returned by this algorithm is smaller by 25% compared to the SOC of the solution obtained by CCBS with $k = 2$.

Fig. 9 shows the SOC- and makespan-gain ratios as box-and-whisker plots, for different values of $k$. Each plot shows the following statistics: minimum (lower whisker), maximum (upper whisker), median (horizontal line inside the box), mean (cross sign inside the box), first and third quartiles (box) and also outliers (bold dots above/below whiskers).

As one can see, increasing $k$, indeed, decreases both SOC and makespan. This effect is most notable when going from $k = 2$ to $k = 3$. Consider, `den520d` map for example. Setting $k$ to 3 reduces both makespan and SOC by about 15-17% on average. However, increasing $k$ beyond 3 had a much smaller effect on cost. In fact, in all our experiments going from $k = 4$ to $k = 5$ yielded a marginal improvement of at most 1% approximately in terms of makespan and SOC, for both CCBS and SMT-CCBS. It is noteworthy, that a similar result was previously observed in $2^k$ single-agent pathfinding [57].

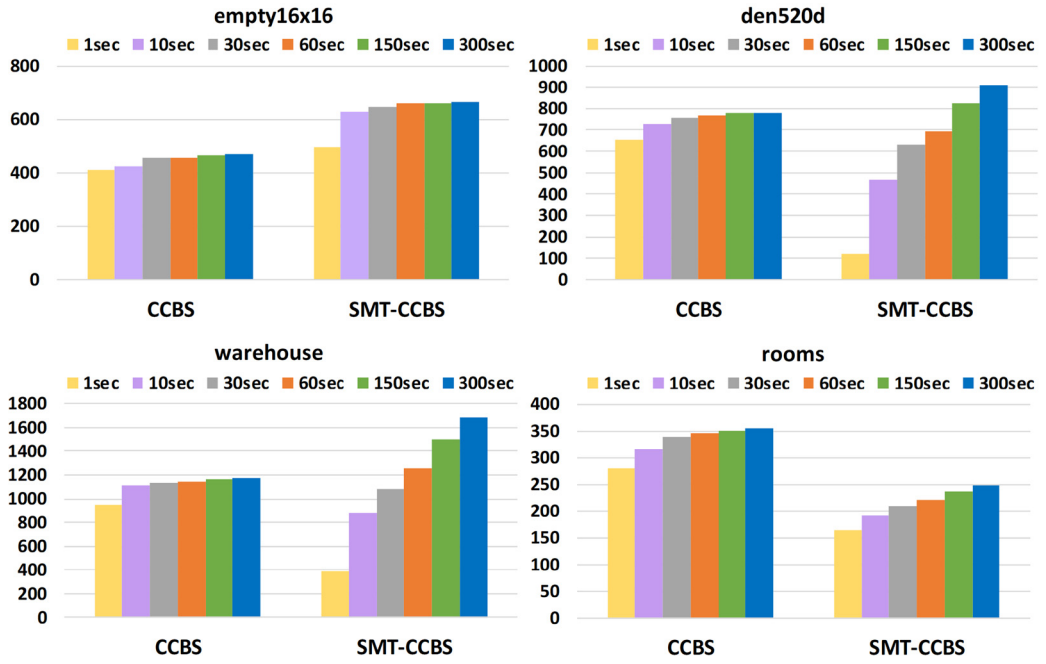



**Fig. 8.** Amounts of totally solved instances by CCBS and SMT-CCBS on different 8-neighborhood grids depending on the time-limit.

Next, consider the makespan results for CCBS. While CCBS is designed to optimize SOC and not makespan, our results show that very often the solution returned by CCBS is also optimal w.r.t. makespan. Similarly, while SMT-CCBS is designed to optimize makespan, it often returns solutions that are SOC-optimal. To see this, consider the flat bars at 1.0 on both *SOC gain* and *makespan gain* plots for CCBS (k=2) and SMT-CCBS (k=2) on all grids. In some cases, CCBS returns a solution with suboptimal makespan, and, similarly, SMT-CCBS returns a solution with suboptimal SOC. This can be seen by observing the outliers in the `room` and `empty` grids. But in general, the shapes of the CCBS and SMT-CCBS boxes look very similar. This suggests that for our MAPF$_R$ problems a SOC-optimal solution is often makespan-optimal and vice versa. Moreover, the high-level behavior of both algorithms is similar – starting with the optimal single-agent plans, identifying conflicts and resolving them in a similar way. Therefore, the eventual solutions they return tend to be similar.

### 6.3. Roadmaps results

Fig. 10 depicts the success rates for CCBS and SMT-CCBS on our roadmap graphs. Consider first the impact of the different roadmap types on the success rates of CCBS and SMT-CCBS. For CCBS, the success rates of the `sparse` and `mega-dense` roadmaps are similar and notably lower than the success rate for the `dense` roadmap. This can be explained as follows. In the `sparse` roadmap, the number of edges is small and thus agents are likely to conflict by planning to use the same edges. In the `mega-dense` roadmap, the edges densely populate the metric space in the map, and thus agents are likely to conflict by planning to use different edges that are too close to each other. In other words, in the `sparse` roadmap the agents have fewer alternative paths to choose from to avoid conflicts, while in the `mega-dense` roadmap agents have many alternative paths to choose from but a large number of them conflict. In both cases, numerous conflicts arise, making the problem harder for CCBS. The `dense` roadmap provides a reasonable trade-off between the number of alternative paths the agents have to choose from and the likelihood that such paths conflict, yielding a notably higher success rate.

The impact of the different roadmaps on success rates is different for SMT-CCBS. There, the success rates monotonically decrease for all number of agents when increasing the density of the roadmap graph. This occurs because significantly larger and denser graphs require creating and solving SAT formulae with more variables and constraints. The impact of increasing the density of edges here is multiplied by the absence of compression at the $MDD_R$ level, as in the roadmaps there is scarce symmetry in the neighborhood of a position. Therefore, the possibility of generating a single node in $MDD_R$ by different paths is eliminated. This is also shown when comparing the success rates of CCBS and SMT-CCBS. The latter performs better on the `sparse` roadmap while the former is superior on the moderately sized and larger roadmaps (`dense` and `mega-dense`).

Next, consider the SOC and makespan of CCBS and SMT-CCBS in the different roadmaps. Since the results show SOC and makespan over only the solved instances, there are some cases where the average SOC or makespan for the denser roadmaps is slightly higher. However, in general, denser roadmaps allow finding solutions with lower costs by both algorithms, as expected. That being said, the actual difference in makespan between `dense` and `mega-dense` is very small, suggesting that for finding optimal solutions, the `dense` roadmap provide sufficient discretization of the continuous space to find high

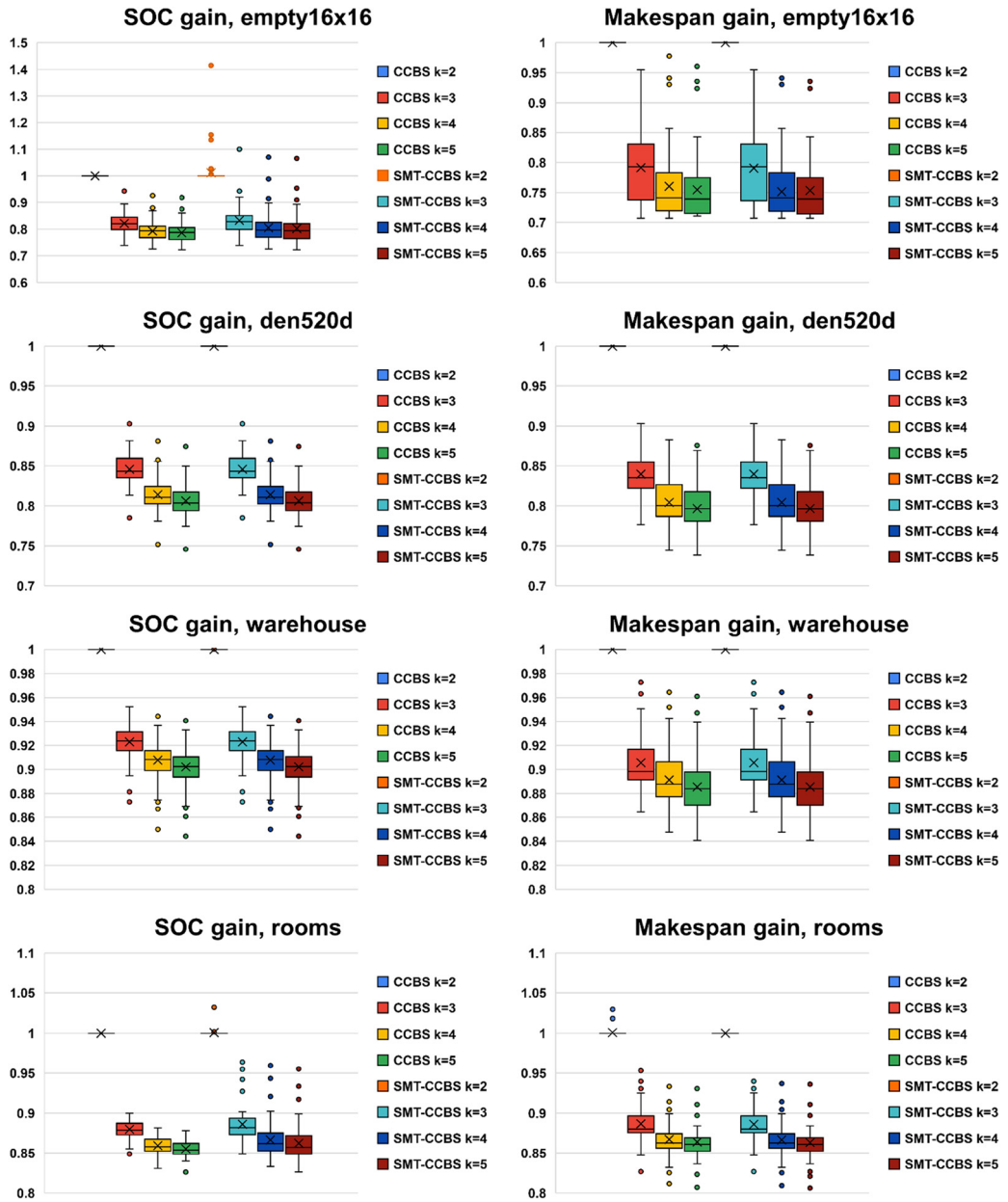



**Fig. 9.** The SOC- and makespan-gain ratios for CCBS and SMT-CCBS on grid maps.

quality solutions. However, finding the suitable density for a roadmap of a given terrain is a topic that is beyond the scope of this work.

### 6.4. Comparison to other solvers

Next, we compare the performance of CCBS and SMT-CCBS to other solvers that also address versions of the MAPF$_R$ problem as explained in Section 2.2. Namely, we compared our algorithms with E-ICTS [13] and ECBS-CT [14]. Both E-ICTS and ECBS-CT are bounded-suboptimal algorithms — they accept a parameter $w \geq 1$ that bounds the suboptimality of the solution they return. In our experiments, we set $w = 1$ to ensure that the returned solution is optimal.[10] We refer to ECBS-

[10] Note that since E-ICTS and ECBS-CT discretize time, the cost of the solutions they return may still be higher than those returned by CCBS and SMT-CCBS. An example where discretizing time yields suboptimal solution is shown in Fig. 4. However, in our experiments we did not observe any notable difference in solution costs.

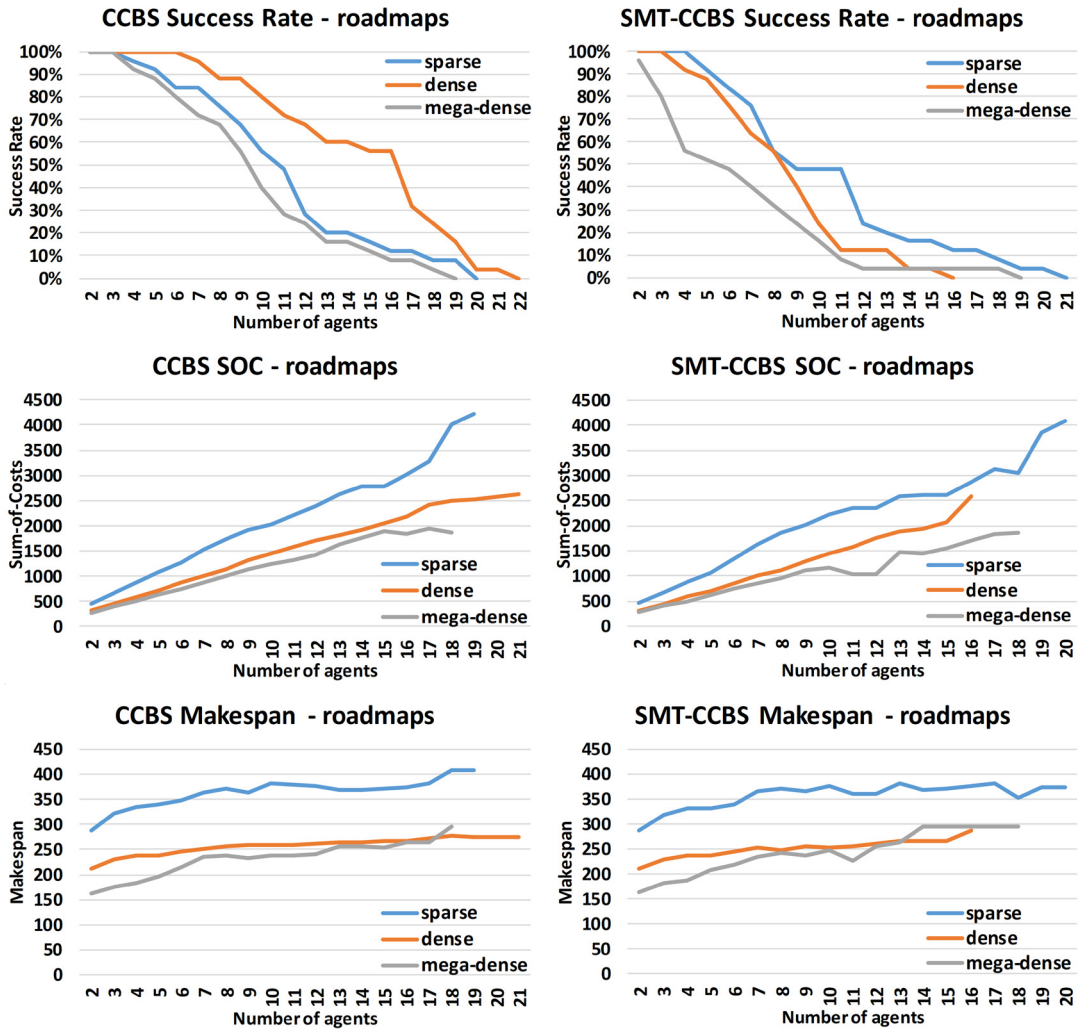



**Fig. 10.** Results of CCBS (left) and SMT-CCBS (right) for different roadmaps.

CT with $w = 1.0$ as CBS-CT. We used the authors' implementations of these algorithms, which are either freely available on Github (E-ICTS)[11] or were shared with us by the ECBS-CT authors. Note that these implementations may not exactly match those used in prior publications about these algorithms, and thus the results we report may differ.

E-ICTS and CBS-CT implementations handle continuous time by discretizing it according to a minimal wait time parameter $\Delta$. We set it to be 1/1000 in our experiments. Since these implementations do not support general graphs, we ran our comparison on $2^k$-neighborhood grids only. E-ICTS supports these grids by default. For CBS-CT we designed motion primitives that correspond to moves along the edges of $2^k$-neighborhood grid.

Fig. 11 shows the number of solved instances for all algorithms under a time limit of 30 s. Each plot corresponds to a particular map from our dataset. On the `empty-16-16` grid SMT-CCBS clearly outperforms the competitors, while the performance of CCBS is similar to the one of E-ICTS. On the `warehouse` map, CCBS dominates for all $k$, however, SMT-CCBS is very close for $k = 3, 4, 5$. On `rooms` for $k = 2$ E-ICTS is a winner, for $k = 2, 3$ – CCBS is and for $k = 5$ CBS-CT solves the most instances. Finally, for the largest `den520d` map CCBS and SMT-CCBS evidently outperform the other algorithms for $k = 3, 4, 5$. For $k = 2$ CBS-CT is very close to SMT-CCBS. Overall, our experiments show that there is currently no universal winner. This is aligned with the current state of the art in classical MAPF algorithms, where identifying which MAPF algorithm to use in which domain is an open question.

[11] We used the version dated 30 August 2019 from https://github.com/thaynewalker/hog2. Newer versions were unstable in our tests.

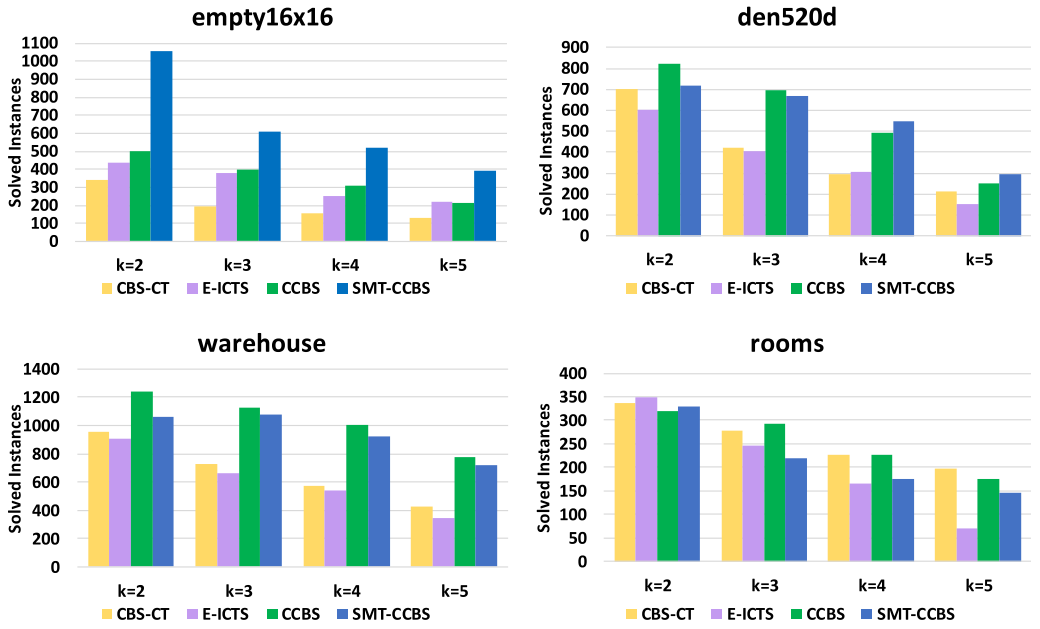



**Fig. 11.** Amounts of totally solved instances.

## 7. Related work and discussion

In this section, we discuss the pros and cons of the two algorithms we proposed (CCBS and SMT-CCBS), and how they are related to algorithms for solving other variants of the general MAPF problem.

### *7.1. Discussion: CCBS vs. SMT-CCBS*

In this paper, we defined CCBS to optimize SOC and SMT-CCBS to optimize makespan. But, it is possible to adapt them to optimize other objective functions. Adapting SMT-CCBS to optimize SOC requires bounding both SOC and makespan, such that SOC-optimal solutions are not missed. Surynek [60,61] has recently proposed a method to do so. Adapting CCBS to optimize makespan is simpler, and Appendix D reports on limited comparison between such a makespan-optimal version of CCBS and SMT-CCBS.

Our current results show that CCBS often outperforms SMT-CCBS. In addition, there are many enhancements to the basic CBS algorithm that can be migrated to CCBS and further improve its performance [62]. Nevertheless, a major benefit for using SMT-CCBS is that improvements in the underlying SAT/SMT solver may automatically result in an improved MAPF solver. We have observed this in a different MAPF-related study. Thus, SMT-CCBS may still yield better results than CCBS in the future, with the advancement of SAT solver technology.

More generally, CCBS and SMT-CCBS embody two standard approaches for solving combinatorial search problems – heuristic search and compilation to SAT. Both approaches have been used to solve different types of planning problems, including classical planning [63,28,64] and classical MAPF. A comparison between MAPF algorithms from these two approaches have been reported in many papers [12,65], including a recent paper [66] that included a comprehensive evaluation over a wide range or benchmark domains. The results, in general, show that there is no single approach that can solve all problems. Some classical MAPF problems can only be solved with a SAT-based algorithm while others can only be solved by a search-based algorithm. This motivated us to develop both a search-based algorithm (CCBS) and a SAT-based algorithm (SMT-CCBS) for solving $\text{MAPF}_R$ problems.

### *7.2. Discussion: solvers for different MAPF variants*

There exists a vast body of other works that study MAPF beyond its basic, classical, setting. Table 3 provides a differential overview of related work on MAPF beyond its basic setting. Rows correspond to different algorithms or family of algorithms. Columns specify algorithm properties that the listed algorithms have or not. These properties are whether the listed algorithm (1) supports non-uniform durations for move actions ("Non-uniform move durations" column), (2) requires discretizing the durations of wait actions ("Wait is not discretized"), (3) allows moving from one graph vertex to the other via the straight line even if there is no corresponding edge ("Any-angle moves"), (4) considers the agents' geometric shapes ("Agents' volume"), (5) guarantees solution completeness ("Complete"), (6) guarantees optimal solution cost ("Opt."), and (7)

**Table 3**
Overview: MAPF research beyond the basic setting.

| | Non-uniform move duration | Wait is not discretized | Any-angle moves | Agents' volumes | Complete | Opt. | Dist. |
|---|---|---|---|---|---|---|---|
| CBS-CT [14] | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| E-ICTS [13] | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| AA-SIPP($m$) [38] | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| MCCBS [17] | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| CBS-CL [67] | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ |
| MAPF-POST [68,69] | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ |
| ORCA [70], ALAN [71] | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ |
| dRRT* [72] | ✓ | ✗ | ✗ | ✓ | ✓ | ✗? | ✗ |
| CCBS [18], SMT-CCBS [19] | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✗ |

whether the algorithm is distributed or not ("Dist."). Completeness and optimality here are with respect to a given movement and time resolution. In the rest of this section, we discuss the related works listed in this table and others, in more detail.

In Section 2.2, we discussed several algorithms that are closest to our work, including E-ICTS [13] and CBS-CT [14]. As explained there, both E-ICTS and CBS-CT require a minimal wait duration to be given as the input parameter, and the duration of all wait actions are multiplicatives of that parameter. As shown in Example 2, discretizing wait actions in such a way can lead to finding suboptimal solutions. So, while E-ICTS and CBS-CT are optimal with respect to the chosen discretization, the solutions they return may have a higher cost than the solutions returned by CCBS and SMT-CCBS, which do not discretize time.

Yakovlev and Andreychuk [38] proposed AA-SIPP($m$), an any-angle MAPF algorithm. AA-SIPP($m$) is based on SIPP and adopts a prioritized planning approach. It does not guarantee completeness or optimality. Li et al. [17] proposed Multi-Constraint CBS (MCCBS), a CBS-based algorithm for agents with a geometrical shape that may have different configuration spaces. However, they assumed all actions have a unit duration and did not address continuous time. Walker et al. [67] proposed CBS-CL, a CBS-based algorithm designed to handle non-unit edge costs and hierarchy of movement abstractions. CBS-CL is solution complete. However, CBS-CL does not allow reasoning about continuous time and does not return provably optimal solutions. Hönig et al. [68,69] proposed MAPF-POST, which is a post-processing step that adapts a solution to a classical MAPF such that it respects a given set of kinematic constraints over the agents' motions. They prove that MAPF-POST can always find a feasible schedule that satisfies the given kinematic constraints, and thus we view MAPF-POST as a complete algorithm. It does not, however, guarantee optimality.

dRRT* is a hybrid between a tree-search algorithm and a sampling-based algorithm [72]. It runs a tree search over locations sampled in the configuration space. dRRT* is solution complete – given enough time it will find a solution if one exists. Regarding solution quality, dRRT* provides a probabilistic type of guarantee: given enough time it will return an optimal solution. However, it does not provide a mechanism for identifying when this occurs. Thus, we do not consider dRRT* as an optimal MAPF algorithm, since the cost the solution it returns when halted may be far from optimal. Also, it is not clear how the duration of wait actions are chosen by dRRT*.

ORCA [73,70] is a fast and distributed collision avoidance mechanism that has been used to navigate multiple agents in continuous space [74]. It works by computing for each agent in every time step the direction and velocity it should use to avoid the other agents. While fast, using ORCA to navigate multiple agents does not provide any completeness or optimality guarantees. In addition, ORCA requires time to be discretized. ALAN [71] is a multi-agent navigation algorithm that integrates multi-armed bandits with collision avoidance using ORCA that yields lower cost solutions. Like ORCA, ALAN is not complete or optimal, and requires time to be discretized.

All the algorithms mentioned above are designed to solve some variants of the MAPF problem. The Multi-Agent Pickup and Delivery (MAPD) problem is a problem that is closely related to MAPF, in which agents need to pick up and deliver packages from one location to another. Techniques proposed to solve MAPF problems have also been adapted to solve MAPD problems [75,76,39]. In particular, Ma et al. [39] proposed a MAPD algorithm called TP-SIPPwRT that also uses SIPP to handle continuous movement of non-holonomic agents in this setting. TP-SIPPwRT is not optimal and is only complete for well-formed MAPD problems. Future work may consider integrating ideas from CCBS and SMT-CCBS and applying them to solve MAPD problems as well.

### *7.3. Discussion: improvements for CCBS and SMT-CCBS*

Since CCBS has been introduced, a number of enhancements were suggested that have the potential to improve its performance. Some of these improvements were designed explicitly for CCBS or SMT-CCBS. Others are more general and can be applied to both MAPF and MAPF$_R$.

Andreychuk et al. [62] transferred different techniques designed for solving classical MAPF with CBS to MAPF$_R$ and CCBS. These enhancements included disjoint splitting [77], prioritizing conflicts based on their cost impacts, high-level heuristics [11]. The resulting solver, named Improved CCBS (ICCBS), was reported to notably outperform CCBS on both $2^k$ grids and roadmap graphs.

Walker et al. [24] proposed another powerful CBS enhancement as part of their CBS+TAB algorithm. The key idea of CBS+TAB is to add multiple time-range constraints (similar to the ones proposed in this work) per single CT node of CBS in order to prune the CT. The reported increase in algorithm's performance is significant. The original implementation of CBS+TAB allows wait actions only of the fixed duration. That being said, one may be able to implement CBS+TAB such that it handles non-discretized wait actions, and one may incorporate the action pruning rules from CBS+TAB in our algorithms.

This spatial symmetry-breaking technique of CBS+TAB was later augmented with a cost-based symmetry-breaking [78] inspired by techniques used in E-ICTS. The resulting hybrid algorithm, called Conflict-Based Increasing Cost Search (CBICS), was reported to outperform the competitors. Still, their evaluation was carried out with the fixed duration of the wait action equal to one time step. We believe that implementing CBICS that supports wait actions of arbitrary duration is a promising direction of future work.

Recent development in SMT-CCBS focused on reducing the size of $MDD_R$ structures being generated by the algorithm [79]. The $MDD_R$ size reduction method is based on including promising paths into $MDD_R$ first. An additional loop is needed in the corresponding modification of SMT-CCBS that gradually extends the set of paths represented in $MDD_R$ structures starting with promising ones and continuing towards full $MDD_R$ structures representing all paths. The preference of paths for inclusion in $MDD_R$ structures represents a room for integrating domains specific heuristics in SMT-CCBS.

## 8. Conclusion

We proposed two novel algorithms for solving MAPF$_R$, which is a form of MAPF in which time is continuous, actions can have an non-uniform duration, and agents, as well as objects, have geometric shapes. The first algorithm we presented is called CCBS. It follows the Conflict Based Search (CBS) framework [7], but uses an adapted version of Safe Interval Path Planning (SIPP) [15] for the low-level search, and a unique type of conflicts and constraints for the high-level search. We prove that CCBS is a sound, solution complete, and returns SOC-optimal solutions. The second algorithm we presented is called SMT-CCBS. It follows the same approach as SMT-CBS [16] by breaking the given MAPF$_R$ problem into a sequence of bounded-cost MAPF$_R$ problems. Each of these bounded-cost problems is solved by applying a SAT modulo Theory (SMT) problem-solving framework. That is, a propositional skeleton (PS) is generated and continuously updated until a solution to this PS represents a valid solution to the given bounded-cost problem. We prove that SMT-CCBS is also sound and solution complete, and guarantees a makespan-optimal solution is returned. To the best of our knowledge, CCBS and SMT-CCBS are the first algorithms to provide optimality guarantees for such a general version of MAPF.

We implemented both algorithms and evaluated them experimentally on a set of benchmarks, including grid-based graphs [20] and roadmaps. Our experimental results showed that while MAPF$_R$ is, in general, more difficult than classical MAPF, using either CCBS or SMT-CCBS enables finding optimal solutions to problems with dozens of agents under a strict time limit of 30 seconds.

Nevertheless, there is much room for improvement, and scaling to even larger problems is still an open challenge. One of the bottlenecks we observed in our experimental evaluation is conflict detection, which is more challenging in MAPF$_R$ than in classical MAPF. Future work may apply meta-reasoning techniques to decide when and how much to invest in conflict detection throughout the search. Another direction for future work is to integrate in CCBS the many extensions and improvements that have been proposed over the years. These improvements include disjoint splitting of CBS constraints [77], adding admissible heuristic to the high-level search [11,80], and novel forms of constraints symmetry-breaking [81]. Some of these improvements may be easy to incorporate in CCBS while incorporating others is more challenging. An initial step in this direction has already been taken [62]. Finally, incorporating recently proposed CBS enhancements, such as those mentioned in Section 7.3, into CCBS and SMT-CCBS, is another prominent prospect of future work.

One possible future research direction for SMT-CCBS is to study its conversion to DPLL(MAPF$_R$), a solver that would integrate MAPF$_R$ solving and the SAT solver in a more sophisticated way similarly as it is done in DPLL(T) solvers [31]. In contrast to SMT-CCBS which waits until the SAT solver finds a complete satisfying truth-value assignment and then tests it for collisions, DPLL(MAPF$_R$) would test for collisions even partial truth-value assignments that arise during the search in the SAT solver. This could prune the search made by the SAT solver using the high-level knowledge of MAPF$_R$.

Finally, both CCBS and SMT-CCBS rely on a given finite set of move actions that are given as input. This limits the completeness of both algorithms, in the sense that a MAPF$_R$ problem may be solvable with one set of move actions and unsolvable in another. Similarly, it limits our claims for optimality — CCBS and SMT-CCBS are both only optimal with respect to the given set of move actions. Future research may investigate developing algorithms that do not accept the set of allowed move actions as input, and instead automatically computes the necessary move actions to find an optimal solution. This research direction is particularly challenging, as there are an infinite number of move actions, which include stopping for an arbitrary amount of time at any point in space and varying the speed at which the agents move between vertices.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

**Acknowledgements**

This research was partially funded by the Israel Science Foundation (ISF) grant #210/17 to Roni Stern. Konstantin Yakovlev and Anton Andreychuk were supported by the Russian Science Foundation (RSF) grant #16-11-00048. Anton Andreychuk was also supported by the "RUDN University Program 5-100". Pavel Surynek was supported by the Czech Science Foundation (GAČR) grant #19-17966S.

## Appendix A. An example of the SMT problem-solving procedure

Consider the following simple example of problem-solving with SMT. The decision problem is to assign integer values to the variables $a, b, c, d$ such that the following formula is true:

$$\Gamma := ((a = b) \wedge (a = c)) \vee (\neg(b = c) \wedge \neg(a = d)) \tag{A.1}$$

Note that in any solution to $\Gamma$, the common axioms of equality, such as transitivity, must hold. In an SMT approach to solve $\Gamma$, we can define the PS

$$(X_{ab} \wedge X_{ac}) \vee (\neg X_{bc} \wedge \neg X_{ad}) \tag{A.2}$$

where $X_{ab}$, $X_{ac}$, $X_{bc}$, and $X_{ad}$ are propositional variables such that $x_{ij}$ represents that $i = j$ for $i, j = \{a, b, c\}$. The SAT solver can set $X_{ab} = true$, $X_{ac} = true$, $X_{bc} = false$, and $X_{ad} = false$ to satisfy this PS. The corresponding solution is inconsistent with the semantic of equality (EQ) theory, due to the transitivity of equality. Namely, if $x_{ab}$ and $x_{ac}$ are both true, then $x_{ac}$ must also be true. Note that the SAT solver does not know this, as it is not represented in the PS. A suitable $DECIDE_T$ is needed to check that equality theory holds and suggest a conflict otherwise. We denote by $\text{DECIDE}_{EQ}$ this implementation of $DECIDE_T$. In our case, $\text{DECIDE}_{EQ}$ can suggest the conflict $X_{ab} \wedge X_{ac} \rightarrow X_{bc}$. This conflict is added as to PS, so the SAT solver can give a new assignment. In this way, knowledge from the underlying theory is propagated by the conflict to the PS level.

## Appendix B. Completeness of $\mu$SMT-CCBS

In this section, we provide a formal proof for Theorem 2, which states that $\mu$SMT-CCBS is sound and complete. Soundness in this context means that every solution returned by $\mu$SMT-CCBS is indeed a valid $\text{MAPF}_R$ solution with makespan equal to or smaller than the makespan bound. Completeness in this context means that if such a solution exists then it will be found, and if a solution does not exist then $\mu$SMT-CCBS will return that no solution exists. Soundness is established by the fact that $DECIDE_{MAPF_R}$ verifies that the returned solution is valid. Establishing completeness, however, is less trivial and is done below.

For $\mu$SMT-CCBS to be complete, we require that the set of single-agent plans it considers contains every single-agent plan that CSIPP would return given every subset of the set of all pairs of CCBS constraints ($const(\Psi)$). To state this more formally, let $MDD_R(\mu, const(\Psi))$ be the set of single-agent plans represented by the $MDD_R$ created for $\mu$ and $const(\Psi)$, and let $CSIPP(Const')$ be the single-agent plan created by CSIPP given the set of constraints $Const'$.

**Lemma 2.** *For every set of CCBS constraints $Const' \subseteq const(\Psi)$ it holds that $CSIPP(Const') \in MDD_R(\mu, const(\Psi))$.*

**Proof.** Let $\pi = (a_1, \dots, a_n))$ be the single-agent plan returned by CSIPP given a set of constraints $Const' \subseteq const(\Psi)$. Let $\{(a_1, t_1), \dots, (a_n, t_n)\}$ be the corresponding set of timed-actions, i.e., $t_j = ([\pi[: (j-1)])_D$. To prove Lemma 2, we proof by induction over $j$ that for every timed action $(t_j, a_j)$ in this set there exists an edge $((from(a_j), t_j), (to(a_j), t_j + (a_j)_D))$ in our $MDD_R$.

**Base case.** For $j = 0$, we need to show that $((from(a_1), 0), (to(a_1), a_{1D}))$ is in our $MDD_R$. There are two cases: either $a_1$ is a wait action or a move action. In case $a_1$ is a move action, the $((from(a_1), 0), (to(a_1), (a_1)_D))$ trivially exists in our $MDD_R$ by construction (lines 9-10 in Algorithm 3). Now consider the case where $a_1$ is a wait action. There are only two cases in which the plan returned by CSIPP starts with a wait action:

1. There is a move action $a'$ that starts at $S(i)$, and there is CCBS constraint $\langle i, a', [0, t^u)\rangle$.
2. There is a move action $a'$ that starts at $S(i)$ and ends in some vertex $v$ which has safe interval that starts at some time $t$ where $t > a'_D$. In this case, CSIPP may choose to wait at $S(i)$ until it can arrive at $v$ at time $t$.

In the first case, $(a_1)_D = t^u$. In the second case, $(a_1)_D = t - a'_D$. Our $MDD_R$ covers both cases. The first case is explicitly mentioned in lines 11-15. To see why the second case is also covered in our $MDD_R$, recall that CSIPP only creates a new safe interval in a vertex $v$ only when there is constraint over a wait action at that vertex. Let $\langle i, a', [t, t^u)\rangle$ be this constraint. The new safe interval for CSIPP is designed such that it starts exactly when the unsafe interval of this constraint ends, i.e., at $t^u$. Therefore, to reach $v$ at time $t^u$ with some action $a_{vu}$ we need to wait at $v$ exactly $t^u - a_{vu}$. An edge corresponding to such a wait action exists in our $MDD_R$, as specified in lines 16-20.

**Induction step.** Now, assume the induction statement holds for $j < m$, and consider the $m^{th}$ timed action $(a_m, t_m)$ in the plan CSIPP returned. Let $(v, t)$ be the location and time reached after performing the first $m-1$ actions in $\pi$. By the induction assumption, the node $(v, t)$ exists in our $MDD_R$. The same argument used in the base step hold here as well. If $a_m$ is a move action, the edge $((from(a_m), t), (to(a_m), t + (a_m)_D))$ exists in our $MDD_R$ (lines 9-10). If $a_m$ is a wait action, it was created either to avoid a conflict with a subsequent move action or to allow a subsequent move action to reach the start of some safe interval. Both options are covered in our $MDD_R$ generation algorithm in lines 11-15 lines 16-20. □

**Theorem 4.** *$\mu$SMT-CCBS is a complete algorithm for bounded-cost MAPF$_R$.*

**Proof.** There are three outcomes for every iteration of $\mu$SMT-CCBS: (1) the current PS is not solvable, (2) the solution returned for the current PS has a conflict, and (3) the solution returned for the current PS has no conflict. If the first outcome occurs, then due to Lemma 2 we can conclude that no solution indeed exists. If the second outcome occurs, the added conflict will be avoided in future iteration by adding its corresponding pair of CCBS constraints. No potential solution is lost by this since the CCBS constraints are a sound pair of constraints. If the third outcome occurs, the solution is returned, as required. Thus, in every iteration of $\mu$SMT-CCBS we do not lose any possible solutions.

Observe that a given MAPF$_R$ problem can give rise to finitely many sound pairs of constraints under the given makespan limit $\mu$. Hence the $\mu$SMT-CCBS terminates as only finitely many sound pairs of constraints can be found and resolved while pairs of constraints being resolved remain resolved in all future steps of the algorithm. Therefore, after finitely many steps the algorithm either succeeds or fails. In case of failure, the set of sound constraints cannot be satisfied for the current makespan limit $\mu$ which means there is no solution of the input MAPF$_R$ instance for this makespan limit $\mu$. □

## Appendix C. Generating an $MDD_R$ and computing the next makespan bound

**Algorithm 5:** Generate $MDD_R$ for agent $i$ and compute next makespan bound $\mu$.

```
1  GenerateMdd_R (Π, const(Ψ), μ, i)
2      X^i ← {(S(i), 0)}; E^i ← ∅; OPEN ← ∅
3      insert (S(i), 0) into OPEN
4      μ_next ← ∞
5      while OPEN ≠ ∅ do
6          (u, t) ← pop (u, t) from OPEN where t = min_t(OPEN)
7          if t ≤ μ then
8              foreach a ∈ A such that from(a) = u do
9                  if t + a_D ≤ μ then
10                     insert (to(a), t + a_D) into OPEN
11                     X^i ← X^i ∪ {(to(a), t + a_D)}
12                     E^i ← E^i ∪ {[(u, t); (to(a), t + a_D)]}
13                     foreach ⟨i, a, [t', t'^u)⟩ ∈ const(Ψ) such that t ∈ [t', t'^u] do
14                         if t'^u + a_D ≤ μ then
15                             insert (u, t'^u) into OPEN
16                             X^i ← X^i ∪ {(u, t'^u)}
17                             E^i ← E^i ∪ {[(u, t); (u, t'^u)]}
18                         else
19                             μ_next ← min(μ_next, t'^u + a_D)
20                     foreach ⟨i, a', [t', t'^u)⟩ ∈ const(Ψ) where a' is wait at to(a) do
21                         if t < t'^u − a_D then
22                             if t'^u ≤ μ then
23                                 insert (u, t'^u − a_D) into OPEN
24                                 X^i ← X^i ∪ {(u, t'^u − a_D)}
25                                 E^i ← E^i ∪ {[(u, t); (u, t'^u − a_D)]}
26                             else
27                                 μ_next ← min(μ_next, t'^u)
28             else
29                 μ_next ← min(μ_next, t + a_D)
30         else
31             μ_next ← min(μ_next, t)
32     return (X^i, E^i), μ_next
```

Here, we provide the complete pseudo-code for generating an $MDD_R$ structures, which also integrates the computation of the next makespan bound. The main difference between this pseudo-code and the one given in Algorithm 3 is that it

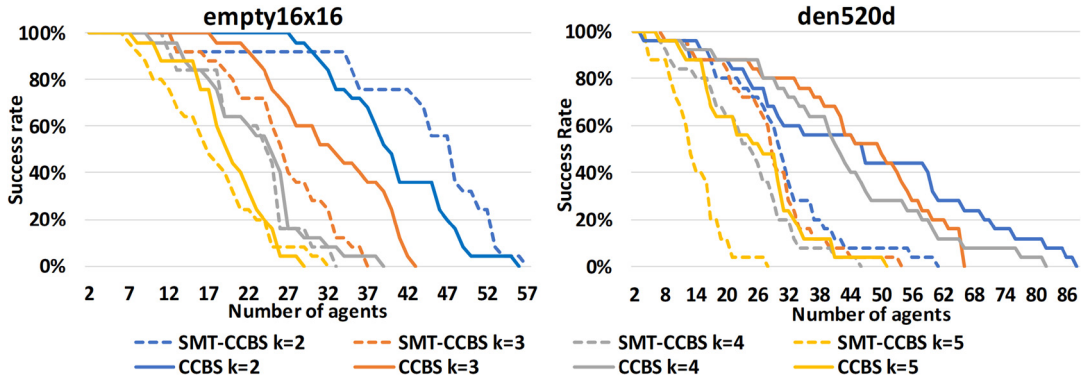



**Fig. D.12.** Comparison of CCBS and SMT-CCBS, that both optimize makespan objective.

includes the computation of the next makespan bound to consider, denoted in the pseudo code as $\mu_{next}$. This computation of $\mu_{next}$ is given in lines 4, 19, 27, 29, and 31.

## Appendix D. Makespan-optimal CCBS

The experimental results of CCBS and SMT-CCBS are not directly comparable due to the different implementations and different cost objectives. While CCBS optimizes SOC, SMT-CCBS returns makespan-optimal solutions. We have also implemented a version of CCBS that returns makespan-optimal solutions. This involves changing the cost of nodes in the CT to be the makespan of the incumbent solution rather than SOC.

We performed a limited set of experiments with this makespan-optimal version of CCBS, on two $2^k$-connected grid maps — `empty16x16` and `den520d` — for $k = 2, 3, 4, 5$. We used the same scenario files and generated instances in the same way as described in Section 6. The time limit was 30 seconds.

Fig. D.12 shows the success rate plots comparing the makespan-optimal CCBS with SMT-CCBS. As one can note, in many settings CCBS was able to find more solutions compared to SMT-CCBS. Still, in some settings (`empty-16-16`, $k$=2) the latter outperformed CCBS. Exploring when each algorithms will perform best is a topic for future research.

## References

[1] P.R. Wurman, R. D'Andrea, M. Mountz, Coordinating hundreds of cooperative, autonomous vehicles in warehouses, AI Magazine 29 (1) (2008) 9–19.
[2] R. Morris, C.S. Pasareanu, K.S. Luckow, W. Malik, H. Ma, T.S. Kumar, S. Koenig, Planning, scheduling and monitoring for airport surface operations, in: AAAI Workshop: Planning for Hybrid Systems, 2016.
[3] M.M. Veloso, J. Biswas, B. Coltin, S. Rosenthal, Cobots: robust symbiotic autonomous mobile service robots, in: The 24th International Joint Conference on Artificial Intelligence, IJCAI, 2015, pp. 4423–4429.
[4] H. Ma, J. Yang, L. Cohen, T.K.S. Kumar, S. Koenig, Feasibility study: moving non-homogeneous teams in congested video game environments, in: The 13th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE, 2017, pp. 270–272.
[5] P. Surynek, An optimization variant of multi-robot path planning is intractable, in: The 24th AAAI Conference on Artificial Intelligence, AAAI, 2010, pp. 1261–1263.
[6] J. Yu, S.M. LaValle, Structure and intractability of optimal multi-robot path planning on graphs, in: The 27th AAAI Conference on Artificial Intelligence, AAAI, 2013, pp. 1443–1449.
[7] G. Sharon, R. Stern, A. Felner, N.R. Sturtevant, Conflict-based search for optimal multi-agent pathfinding, Artif. Intell. 219 (2015) 40–66.
[8] G. Sharon, R. Stern, M. Goldenberg, A. Felner, The increasing cost tree search for optimal multi-agent pathfinding, Artif. Intell. 195 (2013) 470–495.
[9] G. Wagner, H. Choset, Subdimensional expansion for multirobot path planning, Artif. Intell. 219 (2015) 1–24.
[10] T.S. Standley, Finding optimal solutions to cooperative pathfinding problems, in: The 24th AAAI Conference on Artificial Intelligence, AAAI, 2010, pp. 173–178.
[11] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, T.S. Kumar, S. Koenig, Adding heuristics to conflict-based search for multi-agent path finding, in: The 28th International Conference on Automated Planning and Scheduling, ICAPS, 2018, pp. 83–87.
[12] R. Barták, N.-F. Zhou, R. Stern, E. Boyarski, P. Surynek, Modeling and solving the multi-agent pathfinding problem in Picat, in: International Conference on Tools with Artificial Intelligence, ICTAI, 2017, pp. 959–966.
[13] T.T. Walker, N.R. Sturtevant, A. Felner, Extended increasing cost tree search for non-unit cost domains, in: The 27th International Joint Conference on Artificial Intelligen, IJCAI, 2018, pp. 534–540.
[14] L. Cohen, T. Uras, T.S. Kumar, S. Koenig, Optimal and bounded-suboptimal multi-agent motion planning, in: The 12th International Symposium on Combinatorial Search, SoCS, 2019, pp. 44–51.
[15] M. Phillips, M. Likhachev, SIPP: Safe interval path planning for dynamic environments, in: IEEE International Conference on Robotics and Automation, ICRA, 2011, pp. 5628–5635.
[16] P. Surynek, Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories, in: The 28th International Joint Conference on Artificial Intelligence, IJCAI, 2019, pp. 1177–1183.
[17] J. Li, P. Surynek, A. Felner, H. Ma, Multi-agent path finding for large agents, in: The 33rd AAAI Conference on Artificial Intelligence, AAAI, 2019, pp. 7627–7634.
[18] A. Andreychuk, K. Yakovlev, D. Atzmon, R. Stern, Multi-agent pathfinding with continuous time, in: The 28th International Joint Conference on Artificial Intelligence, IJCAI, 2019, pp. 39–45.

[19] P. Surynek, Multi-agent path finding with continuous time and geometric agents viewed through satisfiability modulo theories (SMT), in: The 12th International Symposium on Combinatorial Search, SOCS, 2019, pp. 200–201.
[20] R. Stern, N.R. Sturtevant, A. Felner, S. Koenig, H. Ma, T.T. Walker, J. Li, D. Atzmon, L. Cohen, T.K.S. Kumar, R. Barták, E. Boyarski, Multi-agent pathfinding: definitions, variants, and benchmarks, in: The 12th International Symposium on Combinatorial Search, SOCS, 2019, pp. 151–159.
[21] J. Yu, S.M. LaValle, Multi-agent path planning and network flow, in: The 10th Workshop on the Algorithmic Foundations of Robotics, WAFR, 2012, pp. 157–173.
[22] A. Felner, R. Stern, S.E. Shimony, E. Boyarski, M. Goldenberg, G. Sharon, N.R. Sturtevant, G. Wagner, P. Surynek, Search-based optimal solvers for the multi-agent pathfinding problem: summary and challenges, in: The 10th International Symposium on Combinatorial Search, SoCS, 2017, pp. 29–37.
[23] P. Surynek, A. Felner, R. Stern, E. Boyarski, Efficient SAT approach to multi-agent path finding under the sum of costs objective, in: The 22nd European Conference on Artificial Intelligence, ECAI, 2016, pp. 810–818.
[24] T. Walker, N.R. Sturtevant, A. Felner, Generalized and sub-optimal bipartite constraints for conflict-based search, in: The 24th AAAI Conference on Artificial Intelligence, AAAI, 2020, pp. 7277–7284.
[25] G. Audemard, J. Lagniez, L. Simon, Improving glucose for incremental SAT solving with assumptions: application to MUS extraction, in: M. Järvisalo, A.V. Gelder (Eds.), International Conference Theory and Applications of Satisfiability Testing, SAT, in: Lecture Notes in Computer Science, vol. 7962, Springer, 2013, pp. 309–317.
[26] G. Audemard, L. Simon, On the glucose sat solver, Int. J. Artif. Intell. Tools 27 (01) (2018) 1840001.
[27] P. Surynek, Time-expanded graph-based propositional encodings for makespan-optimal solving of cooperative path finding problems, Ann. Math. Artif. Intell. 81 (3–4) (2017) 329–375.
[28] H.A. Kautz, B. Selman, Planning as satisfiability, in: The 10th European Conference on Artificial Intelligence, ECAI, 1992, pp. 359–363.
[29] H.A. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: The 16th International Joint Conference on Artificial Intelligence, IJCAI, 1999, pp. 318–325.
[30] A. Srinivasan, T. Ham, S. Malik, R.K. Brayton, Algorithms for discrete function manipulation, in: IEEE International Conference on Computer-Aided Design, 1990, pp. 92–95.
[31] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to dpll(T), J. ACM 53 (6) (2006) 937–977.
[32] M. Bofill, M. Palahí, J. Suy, M. Villaret, Solving constraint satisfaction problems with SAT modulo theories, Constraints 17 (3) (2012) 273–303.
[33] R. Nieuwenhuis, SAT modulo theories: getting the best of SAT and global constraint filtering, in: The 16th International Conference Principles and Practice of Constraint Programming, CP, 2010, pp. 1–2.
[34] G. Gange, D. Harabor, P.J. Stuckey, Lazy CBS: implicit conflict-based search using lazy clause generation, in: The 29th International Conference on Automated Planning and Scheduling, ICAPS, 2019, pp. 155–162.
[35] S.J. Guy, I. Karamouzas, Guide to anticipatory collision avoidance, in: S. Rabin (Ed.), Game AI Pro 2: Collected Wisdom of Game AI Professionals, 2015, pp. 195–208, Ch. 19.
[36] P. Jiménez, F. Thomas, C. Torras, 3D collision detection: a survey, Comput. Graph. 25 (2) (2001) 269–285.
[37] L. Cohen, Efficient bounded-suboptimal multi-agent path finding and motion planning via improvements to focal search, Ph.D. thesis, 2020.
[38] K. Yakovlev, A. Andreychuk, Any-angle pathfinding for multiple agents based on SIPP algorithm, in: The 27th International Conference on Automated Planning and Scheduling, ICAPS, 2017, pp. 586–593.
[39] H. Ma, W. Hönig, T.S. Kumar, N. Ayanian, S. Koenig, Lifelong path planning with kinematic constraints for multi-agent pickup and delivery, in: The 33rd AAAI Conference on Artificial Intelligence, AAAI, 2019, pp. 7651–7658.
[40] V. Narayanan, M. Phillips, M. Likhachev, Anytime safe interval path planning for dynamic environments, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, 2012, pp. 4708–4715.
[41] K. Yakovlev, A. Andreychuk, R. Stern, Revisiting bounded-suboptimal safe interval path planning, in: The 30th International Conference on Automated Planning and Scheduling, ICAPS, 2020, pp. 300–304.
[42] J. Švancara, M. Vlk, R. Stern, D. Atzmon, R. Barták, Online multi-agent pathfinding, in: The 33rd AAAI Conference on Artificial Intelligence, AAAI, 2019, pp. 7732–7739.
[43] W. Hönig, J.A. Preiss, T.S. Kumar, G.S. Sukhatme, N. Ayanian, Trajectory planning for quadrotor swarms, IEEE Trans. Robot. 34 (4) (2018) 856–869.
[44] D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Barták, N. Zhou, Robust multi-agent path finding and executing, J. Artif. Intell. Res. 67 (2020) 549–579.
[45] D. Kornhauser, G. Miller, P. Spirakis, Coordinating pebble motion on graphs, the diameter of permutation groups, and applications, in: The 25th IEEE Annual Symposium on Foundations of Computer Science, FOCS, 1984, pp. 241–250.
[46] B. Nebel, On the computational complexity of multi-agent pathfinding on directed graphs, in: The 30th International Conference on Automated Planning and Scheduling, ICAPS, 2020, pp. 212–216.
[47] D.P. Dobkin, D.G. Kirkpatrick, Determining the separation of preprocessed polyhedra – a unified approach, in: International Colloquium on Automata, Languages, and Programming, ICALP, 1990, pp. 400–413.
[48] S. Gottschalk, M.C. Lin, D. Manocha, Obbtree: a hierarchical structure for rapid interference detection, in: The 23rd Annual Conference on Computer Graphics and Interactive Techniques, 1996, pp. 171–180.
[49] S. Cameron, A study of the clash detection problem in robotics, in: IEEE International Conference on Robotics and Automation, ICRA, 1985, pp. 488–493.
[50] M. Tang, R. Tong, Z. Wang, D. Manocha, Fast and exact continuous collision detection with Bernstein sign classification, ACM Trans. Graph. 33 (6) (2014), Article 186.
[51] T.T. Walker, N.R. Sturtevant, Collision detection for agents in multi-agent pathfinding, arXiv:1908.09707, 2019.
[52] E. Boyarski, A. Felner, R. Stern, G. Sharon, O. Betzalel, D. Tolpin, E. Shimony, ICBS: the improved conflict-based search algorithm for multi-agent pathfinding, in: The 24th International Joint Conference on Artificial Intelligence, IJCAI, 2015, pp. 740–746.
[53] E. Boyraski, A. Felner, G. Sharon, R. Stern, Don't split, try to work it out: bypassing conflicts in multi-agent pathfinding, in: The 25th International Conference on Automated Planning and Scheduling, ICAPS, 2015, pp. 47–51.
[54] M. Barer, G. Sharon, R. Stern, A. Felner, Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem, in: The 7th Annual Symposium on Combinatorial Search, SoCS, 2014, pp. 19–27.
[55] G. Audemard, L. Simon, On the glucose SAT solver, Int. J. Artif. Intell. Tools 27 (1) (2018) 1840001, 25 pp.
[56] P. Surynek, Lazy compilation of variants of multi-robot path planning with satisfiability modulo theory (SMT) approach, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, 2019, pp. 3282–3287.
[57] N. Rivera, C. Hernández, J.A. Baier, Grid pathfinding on the 2k neighborhoods, in: The 31st AAAI Conference on Artificial Intelligence, AAAI, 2017, pp. 891–897.
[58] N.R. Sturtevant, Benchmarks for grid-based pathfinding, IEEE Trans. Comput. Intell. AI Games 4 (2) (2012) 144–148.
[59] I.A. Şucan, M. Moll, L.E. Kavraki, The open motion planning library, IEEE Robot. Autom. Mag. 19 (4) (2012) 72–82, http://ompl.kavrakilab.org.
[60] P. Surynek, Multi-agent path finding modulo theory with continuous movements and the sum of costs objective, in: U. Schmid, F. Klügl, D. Wolter (Eds.), KI 2020: Advances in Artificial Intelligence, in: Lecture Notes in Computer Science, vol. 12325, Springer, 2020, pp. 219–232.

[61] P. Surynek, Logic-based multi-agent path finding with continuous movements and the sum of costs objective, in: S.O. Kuznetsov, A.I. Panov, K.S. Yakovlev (Eds.), Artificial Intelligence - 18th Russian Conference, RCAI 2020, Proceedings, in: Lecture Notes in Computer Science, vol. 12412, Springer, 2020, pp. 85–99.
[62] A. Andreychuk, K. Yakovlev, E. Boyarski, R. Stern, Improving continuous-time conflict based search, in: The 35th AAAI Conference on Artificial Intelligence, AAAI, 2021, pp. 11220–11227.
[63] B. Bonet, H. Geffner, Planning as heuristic search, Artif. Intell. 129 (1–2) (2001) 5–33.
[64] J. Rintanen, K. Heljanko, I. Niemelä, Planning as satisfiability: parallel plans and algorithms for plan search, Artif. Intell. 170 (12–13) (2006) 1031–1080.
[65] P. Surynek, A. Felner, R. Stern, E. Boyarski, An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives, in: The 9th International Symposium on Combinatorial Search, SOCS, 2016, pp. 145–147.
[66] O. Kaduri, E. Boyarski, R. Stern, Experimental evaluation of classical multi agent path finding algorithms, in: The 14th International Symposium on Combinatorial Search, SOCS, 2021, pp. 126–130.
[67] T.T. Walker, D. Chan, N.R. Sturtevant, Using hierarchical constraints to avoid conflicts in multi-agent pathfinding, in: The 27th International Conference on Automated Planning and Scheduling, ICAPS, 2017, pp. 316–324.
[68] W. Hönig, T.S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, S. Koenig, Multi-agent path finding with kinematic constraints, in: The 26th International Conference on Automated Planning and Scheduling, ICAPS, 2016, pp. 477–485.
[69] W. Hönig, T. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, S. Koenig, Summary: multi-agent path finding with kinematic constraints, in: The 26th International Joint Conference on Artificial Intelligence, IJCAI, 2017, pp. 4869–4873.
[70] J. Snape, J. Van Den Berg, S.J. Guy, D. Manocha, The hybrid reciprocal velocity obstacle, IEEE Trans. Robot. 27 (4) (2011) 696–706.
[71] J. Godoy, T. Chen, S.J. Guy, I. Karamouzas, M. Gini, Alan: adaptive learning for multi-agent navigation, Auton. Robots (2018) 1–20.
[72] R. Shome, K. Solovey, A. Dobson, D. Halperin, K.E. Bekris, dRRT*: scalable and informed asymptotically-optimal multi-robot motion planning, Auton. Robots 44 (3) (2020) 443–467.
[73] J.P. Van Den Berg, M.H. Overmars, Prioritized motion planning for multiple robots, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, 2005, pp. 430–435.
[74] J. Snape, J. Van Den Berg, S.J. Guy, D. Manocha, Smooth and collision-free navigation for multiple robots under differential-drive constraints, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, 2010, pp. 4584–4589.
[75] H. Ma, J. Li, T.K.S. Kumar, S. Koenig, Lifelong multi-agent path finding for online pickup and delivery tasks, in: The 16th International Conference on Autonomous Agents and Multiagent Systems, AAMAS, 2017, pp. 837–845.
[76] M. Liu, H. Ma, J. Li, S. Koenig, Task and path planning for multi-agent pickup and delivery, in: The 18th International Conference on Autonomous Agents and Multiagent Systems, AAMAS, 2019, pp. 1152–1160.
[77] J. Li, D. Harabor, P.J. Stuckey, H. Ma, S. Koenig, Disjoint splitting for multi-agent path finding with conflict-based search, in: The 29th International Conference on Automated Planning and Scheduling, ICAPS, 2019, pp. 279–283.
[78] T.T. Walker, N.R. Sturtevant, A. Felner, H. Zhang, J. Li, T.S. Kumar, Conflict-based increasing cost search, in: The 31st International Conference on Automated Planning and Scheduling, ICAPS 2021, 2021, pp. 385–395.
[79] P. Surynek, Sparse real-time decision diagrams for continuous multi-robot path planning, in: The 33rd International Conference on Tools with Artificial Intelligence, ICTAI, IEEE, 2021, pp. 91–96.
[80] J. Li, A. Felner, E. Boyarski, H. Ma, S. Koenig, Improved heuristics for multi-agent path finding with conflict-based search, in: The 28th International Joint Conference on Artificial Intelligence, IJCAI, 2019, pp. 442–449.
[81] J. Li, G. Gange, D. Harabor, P.J. Stuckey, H. Ma, S. Koenig, New techniques for pairwise symmetry breaking in multi-agent path finding, in: The 30th International Conference on Automated Planning and Scheduling, ICAPS, 2020, pp. 193–201.