2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

k-Robust Conflict-Based Search with Continuous time for Multi-robot Coordination

Guilherme Daudt[†] Alleff Dymytry[†] Mariana Kolberg[†] Renan Maffei[†]

***Abstract*— Coordinating multiple robots is crucial for various real-life applications. Many Multi-Agent Path Finding (MAPF) algorithms have been proven to be successful in addressing this challenge. Nevertheless, some problems, such as unexpected delays during navigation, commonly arise when handling high-level abstractions, potentially leading to failures or collisions in live executions. This paper proposes k-Robust Continuous-time Conflict-Based Search (kR-CCBS), a novel algorithm that overcomes some of these limitations. Our approach offers path planning with continuous time, leading to more precise routes than discrete time approaches. Additionally, we increase safety by incorporating k-robustness, enabling the system to adapt to agent failures due to delays and minimize collision risks. Comparative evaluations demonstrate that kR-CCBS outperforms similar works in effectiveness while maintaining reasonable costs, making it a promising solution for real-world multi-agent coordination scenarios.**

I. INTRODUCTION

Mobile robots have been steadily increasing their presence in real-life scenarios. Alongside the usefulness of having autonomous robots in several different environments, some situations benefit even more from having multiple agents acting simultaneously in them [1], [2]. However, one of the key challenges in robotics is efficiently coordinating multiple robots in uncertain environments. A reasonable approach to solve this challenge is to use Multi-Agent Path Finding (MAPF) algorithms.

The MAPF problem involves finding collision-free paths for multiple agents in a shared environment. This problem has been extensively researched, producing algorithms that focused on different practical scenarios, such as air traffic control [1], autonomous driving [2], automated warehouses [3], and video-game AI [4]. While having wide applications, the existing solutions for these scenarios abstract some practical details to make them more efficient to process. Usually, we must disregard some of the factors involved in a live execution, such as considering dynamics, dealing with failures or uncertainties, etc. Thus, plans generated from most algorithms usually require some level of post-processing when applied to real-world robots, logically following that the more you abstract from reality, the more you need to adapt the plan to work correctly. Therefore, creating plans that need less adaptation while maintaining effectiveness can be a good way to save resources.

[†]Institute of Informatics, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil gdaudt, adpdeus, mariana.kolberg, rqmaffei@inf.ufrgs.br

This study was financed in part by the Brazilian National Council for Scientific and Technological Development (CNPq) and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Some algorithms have focused on solving the MAPF problem more realistically than simply planning discrete steps, one of them being Continuous-time Conflict-Based Search (CCBS) [5]. It plans using continuous time, producing results that have better time estimates and are more suitable for real-life applications than discrete-time algorithms. Nevertheless, even with such modifications, the possibility of agent failures or unpredictable delays may lead to the plan's failure in real-world scenarios. An alternative for that, aiming to lower this impact, is to exploit the concept of k-robustness [6], proposed for discrete planning. Such a concept determines safe intervals for places that robots have previously occupied. This would cause agents to plan their movements to account for possible delays in other agents' movements.

In this paper, we present k-Robust Continuous-time Conflict-Based Search (kR-CCBS), which is an algorithm that plans in continuous time and also translates the concept of k-robustness to continuous time. The use of continuous time to produce more accurate plans when transferred to real scenarios allows the robots' actions (and waits) not to be restricted to discrete steps, making plans more efficient. The inclusion of k-robustness ensures safer plan execution against unpredictable delays in movement, making the system resilient to agent failures and reducing the likelihood of collisions. This way, it is possible to deal in a simplified way with uncertainties in real applications that generate delays, without the need for more precise modeling of the dynamics of the robotic system. As shown in the experiments, our approach demonstrates higher effectiveness and success rates across diverse scenarios than related algorithms, indicating its reliability and robustness. Despite increased complexity, our algorithm remains reasonably resource-efficient, striking a balanced trade-off between effectiveness and computational costs compared to its counterparts.

This paper is organized as follows. Section II formally defines the MAPF problem and shows generalizations that lead to the case considered by our algorithm. Section III presents the related work focusing on relevant papers when developing our method and explains the key concepts that our approach shares with such methods. Section IV details our implementation, highlighting the differences between its predecessors and how we deal with the new restrictions created by the k-robust concept. Section V presents the experimental validation obtained in simulated and practical experiments. Finally, Section VI presents the conclusions and future work.

II. Problem definition

A. Multi-Agent Path Finding

MAPF is a navigation problem, defined by having a set of n agents $A = \{a_1, \cdots, a_n\}$, in an environment represented by an unweighted graph $G = (V; E)$, being $V = \{v_1, \cdots, v_m\}$ and $E = \{e_1, \cdots, e_k\}$. Each agent a_i has a starting position $s_i \in V$ and a goal position $g_i \in V$. An agent can perform different actions in a discrete time step t. A path P_i is a set of connected vertices going from s_i to g_i. A solution is a set of paths $\Pi = \{P_1, \cdots, P_n\}$ for every agent, where no agent collides with another. Each agent can perform two different actions in a discrete time step t, *move* and *wait*. A *move* is an action where an agent goes from one vertex v_i to another vertex v_j in a determined time step t_n, traversing through an edge $e_{i,j}$. A *wait* is an action where an agent stays in its current vertex v_i for any amount of time steps $(t, t + y)$, with y being any real value.

A location in the graph cannot be occupied by two agents at the same time step. This occurrence is defined as a *vertex conflict* [7], i.e. $L(a_i, t) \neq L(a_j, t)$, where $L(a_k, t_x)$ is the function that returns the vertex of the k-th agent at time t_x. Two agents also cannot traverse the same edge at the same time step. This is defined as an *edge conflict* [7], following the same inequality above.

B. MAPF generalization for practical scenarios: dealing with non-discrete actions and unpredictable delays

There are several generalizations of MAPF, each of them tailored to fit a different scenario [8]. Some of them change the scenario by having multiple goals for each agent [3], [9], or planning with kinematic constraints [10], [11]. Discrete-time steps and simultaneous movement are also an impactful simplification in real-world situations as well. From this, it stems a generalization called MAPF$_R$, as previously defined in [12], where the graph $G = (V, E)$ is weighted. Every vertex v represents a unique point in a Euclidean space, and the weights of the edges connecting those vertices are determined by the distance between the two points in a straight line. It is assumed that agents have a determined shape, in this case, determined by a circle with a center point. Each agent moves from the center of the vertex v to the center of vertex v' when performing a move action, and stays centered at a vertex when performing a wait action. We consider that agents are not affected by inertia and move at a constant speed. A collision is defined as an event in which two agents' trajectories intersect or overlap. Such an event can occur when two agents traverse the same edge but in opposite directions or when they approach each other from intersecting edges. Additionally, collisions may also occur when agents are on separate edges near the same vertex.

III. Related Work

Algorithms that solve MAPF instances usually try to minimize *flowtime*, which is the sum of the cost of the paths of every agent, or *makespan*, which is the largest cost of a path for a single agent in the set. Finding a solution with optimal values of either flowtime or makespan is considered *NP*-hard [13], [14]. However, extensive research has produced algorithms that are able to solve instances with large amounts of agents in minutes of runtime [8].

Standard solvers can be divided into different types of approaches. Reduction-based algorithms [15] reduce the MAPF problem to well-known combinatorial problems. They achieve good results on instances with a high agent density and can provide optimal, bounded-suboptimal, and suboptimal solutions. Rule-based algorithms [16] use a set of primitive operations to manage the robots. They usually cannot guarantee completeness for all problem classes but can be very efficient. Search-based algorithms [17] use heuristic search techniques to solve MAPF. They can be modified to achieve several different objectives. One of the most popular, Conflict-Based Search (CBS) [18], is explained in further detail next.

A. Conflict-Based Search (CBS)

CBS is a two-level complete algorithm designed to provide optimal solutions for the classical MAPF problem. On the high level, a best-first search is performed on a *constraint tree* (CT). Each node on the CT has a set of *constraints*, a solution (the set of paths for all agents) that satisfies these constraints, and the cost (the sum of costs of all the paths in the solution). A constraint is a tuple containing an agent, a vertex, and a time, being generated whenever a conflict is detected between paths. When a CT Node is expanded, a conflict is selected, and two constraints are created, one for each agent, prohibiting them from using the vertex at the determined time. Then, two new CT Nodes are created, one with each constraint, and the path for the conflicted agent is re-planned, since it is not consistent with the new set of constraints of the node. The low-level algorithm performs a search for the shortest single-agent plan that satisfies the constraints of the node. A CT Node is a goal node when there are no conflicts between agents.

CBS is optimal and complete, thus it became a very popular solver for MAPF instances. However, it has its shortcomings when applied to real-world scenarios, since it still deals with the abstraction of using discrete time. The Continuous-time Conflict-Based Search algorithm [5], explained next, changes the CBS algorithm to reduce this shortcoming.

B. Continuous-time Conflict-Based Search (CCBS)

As an extension of the CBS algorithm to work with Continuous time, CCBS [5] is a complete and optimal algorithm designed to solve MAPF$_R$ problems.

CCBS has the same structure as the CBS algorithm, having both a high and low-level solver, with some adaptations to fit into the MAPF$_R$ problem. Since agents have a determined geometric shape, on a high level collisions cannot be trivially detected by checking the vertex disputed between two agents. Therefore, the conflicts in CCBS are defined between actions, if two agents take actions in similar trajectories at the same determined time, they will collide. A node expansion in

CCBS will select the best node in the CT and, if it is not a goal node, select one of the conflicts between two agents detected generating two new nodes with different constraints. For each agent, the algorithm computes the respective *unsafe interval* relative to their actions in the conflict. The unsafe interval is the longest time interval during which agent a_i, while performing an action from the start to the end of its duration at time t_i, could cause a collision with agent a_j during any point in its corresponding time interval t_j. Each generated node contains a new constraint that prohibits the respective agent from performing the action that would cause a collision during the unsafe interval.

The low-level solver runs an adapted version of Safe Interval Path Planning (SIPP) [19], an algorithm created for planning in dynamic environments. It introduced the concept of safe intervals, defined as a contiguous period of time, during which there is no collision, and it is in collision immediately before and after the period. An A* search is performed on a graph where each node is formed by a pair (location, safe interval) and it prioritizes arriving at the desired location at the earliest new safe interval time. The SIPP framework is not necessarily tied to discrete time steps, and it was modified by the authors in [5] to handle CCBS constraints.

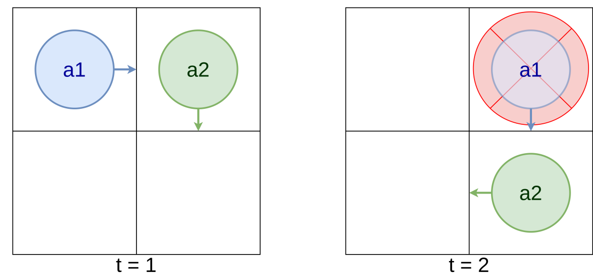
C. k-Robust MAPF

In order to produce paths that can be followed in the occurrence of unpredictable delays, a new form of robustness was introduced in [6]. Different from the traditional MAPF, k-Robust MAPF (or kR-MAPF) plans are only considered valid if it is conflict-free and if it does not contain k-delay conflicts. A k-delay conflict between two agents occurs if and only if there exists a $\Delta \in [0, k]$ where both agents are located in the same location in time steps t and $t + \Delta$. Having plans that are robust to delays means that the agents will still be able to follow their plan if it suffers a delay that is lower or equal to k. It is noted that the standard MAPF is a case of kR-MAPF with $k = 0$. Besides the original algorithms, k-Robust CBS (kR-CBS) and Improved k-Robust CBS (IkR-CBS), that were proposed to generate robust plans, a probabilistic approach was also later presented [20].

IV. kR-CCBS

Our algorithm is an extension of the CBS strategy considering not only continuous time, like in CCBS, but generating plans that are robust to k delays in such continuous time. Satisfying the k-Robust condition of planned paths requires considering new types of conflicts that did not exist in the regular CCBS algorithm. One way to visualize those new conflicts is to picture that an agent, while moving through the map, leaves an afterimage through its path that lasts for k seconds. This prohibits the agents from performing certain patterns, such as train-like motions in close-quarters[1].

[1]Note that agents aren't totally restricted to moving in a line, otherwise movement through narrow hallways would be impossible for our algorithm. A train-like motion would still be possible, but the distance between the agents would be proportionate to the k value set by the algorithm, in other words, as large as the distance of the afterimage of the robot in front.



(a) Instance with two agents at time $t = 1s$, with a_1 at position $(0, 0)$ and a_2 at position $(0, 1)$.

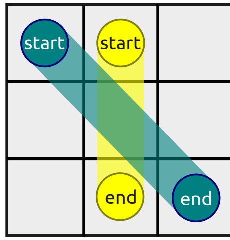
(b) Instance with two agents at time $t = 2s$, with a_1 at position $(0, 1)$, which is a k-conflict, and a_2 at position $(1, 1)$.

Fig. 1: Example of a k-conflict occurrence in a configuration with $k = 1.2s$.

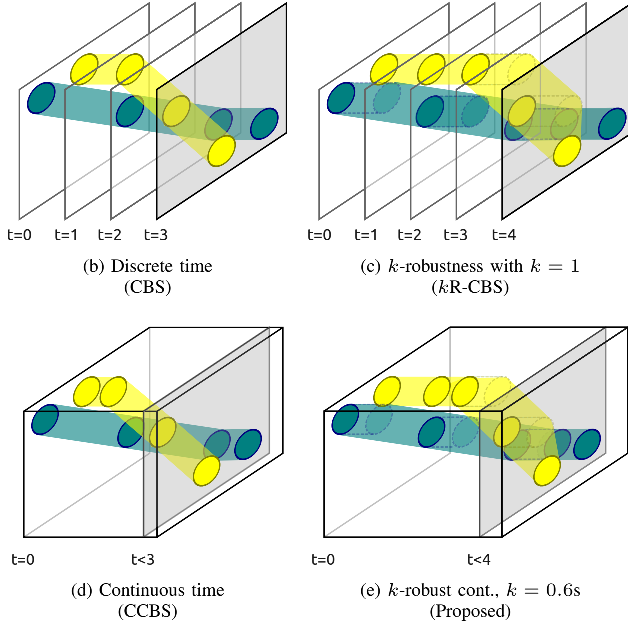
Fig. 1 shows an example of this type of new conflict, considering, for instance, a value of $k = 1.2s$. At time $t = 1s$, agent a_1 is at position $(0, 0)$, and wants to move to position $(0, 1)$ at $t = 2s$, where agent a_2 is currently located. Agent a_2 ideally moves down to an empty location at $(1, 1)$ at $t = 2s$. In this case, with a k-value $= 1.2s$, agent a_1 occupying position $(0, 1)$ at $t = 2s$ is considered a k-conflict. Due to the presence of a_2 in $(0, 1)$ at $t = 1s$, a_1 would only be able to safely occupy $(0, 1)$ after time $t = 1+k = 2.2s$. In our strategy, we maintain the closed-loop collision detection [21], but add another layer of collision checking for k-Robust conflicts.

An illustration of how the proposed method differs from the original CBS with discrete time [18], the CBS version with k-robustness [6] and the continuous time version [5] can be seen in Fig. 2. In (a), the scenario shows two robots that will cross each other in the middle of their paths. In (b), the CBS computes that three time steps are necessary for the agents to complete their routes (the green agent arrives at the goal at $t = 2$, but the yellow agent needs to wait for one step and therefore only arrives at $t = 3$). In (c), the k-robust CBS considers $k = 1$ step. Therefore, now the yellow agent must wait two steps and only arrive at the final goal at $t = 4$. In (d), the continuous time strategy allows the waiting and moving intervals not to be limited by discrete values, but whatever arbitrary time is found during the search process. This tends to reduce the makespan. In contrast, it makes trajectories more tightly aligned, which can increase the risk of robot collisions in real applications. In (e), our proposed method considers conflicts of up to $k = 0.6s$ (note that k can be any real value). Naturally, the makespan tends to increase compared to CCBS, but it depends on the value of k chosen. As an advantage, this strategy spaces the robots' actions by at least k seconds in each location, increasing safety.

Although our goal is to generate plans that remain safely executable in the occurrence of unforeseen delays, we still want to maintain effectiveness and efficiency. Since we are dealing with continuous time, it would be intractable to perform a check on every instant of the Real interval duration between two agents' action, thus, we limited the collision checks to be during the intervals when the agents are located in the middle of the cell. This has to be taken into



(a) Scenario with a conflict between 2 agents



(b) Discrete time (CBS)

(c) k-robustness with $k = 1$ (kR-CBS)

(d) Continuous time (CCBS)

(e) k-robust cont., $k = 0.6$s (Proposed)

Fig. 2: Illustration of the approaches considering discrete or continuous time, and the use or absence of k-robustness. Green and yellow circles represent two agents. In (b) and (c), each vertical plane shows the current state of the agents' configurations at each step. In (d) and (e), time is continuous, and only a few moments are sampled (at arbitrary times when the agents are over the cell centers). The trail of circles along time shows the areas occupied by agents or k-conflicts, which cannot be occupied by other agents.

consideration when choosing the k value in the algorithm since, while it could be any $k \in [0, \mathbb{R}^+]$ number, setting it to a value that is lower than the time an agent takes to travel from the middle of a cell to another, the safety factor will be negated. Naturally, this loses optimality, but considering that we are interested on real-time execution of the generated plans, we figured it was a worthy trade-off.

A. High-level solver

We describe the High-level solver in Algorithm 1. After initializing the root node with the paths for every agent, the algorithm starts picking nodes from the Constraint Tree, choosing the one with the lowest flowtime or makespan value. Then, it checks for conflicts, and if there are none, returns the set of paths of the agents as the solution. Otherwise, it selects the first conflict of the node. From the conflict, one constraint is generated for each agent involved. The low-level algorithm is executed once per agent, with the constraints from the node plus the new one created from the conflict. After replanning, the algorithm searches for new conflicts that might have occurred and adds the new nodes to the Constraint Tree. This process repeats while there are nodes in the tree or a solution is found.

One of the main differences from CCBS is that our algorithm builds a reservation table for each agent before running the Low-level algorithm. This helps pruning nodes involved in k-robust conflicts, explained in more detail in the next section. The other main difference is the need to check for k-robust conflicts alongside regular conflicts. This check returns the first conflict found, which can be of either type.

Algorithm 1: High-Level Algorithm

1 Initialization;
2 Create root node;
3 **while** *there are nodes in tree* **do**
4   **if** *there are no conflicts* **then**
5   **return** solution
6   **end if**
7   **get first conflict**;
8   **for** *each agent in conflict* **do**
9   create_constraint(agent);
10   **LowLevelAlgorithm**(**agent**);
11   Find new conflicts (regular and k-robust);
12   creates node with paths;
13   add node to CCBS tree;
14   **end for**
15 **end while**
16 **return** "No solution";

B. Low-level solver

On the low level of our proposed algorithm, the SIPP structure originally used in CCBS [5] was adapted to work within the framework of k-robust CCBS. The agent selected in the Constraint Tree node is set for replanning using SIPP, while avoiding the conflicts detected by the high-level algorithm. In the original CCBS implementation, the low-level algorithm would calculate the safe intervals by iterating through all the intervals in a desired location. The replanned agent was only aware of the constraints contained in the node created by the high-level algorithm, which are a pair of time and location that cannot be occupied when planning a new path. However, every other path already planned for the other agents in the node was disregarded. Knowing the other agents' paths during replanning helps avoid some extra conflicts, that would generate more nodes in the Constraint Tree down the line. To work with that efficiently, we incorporated a *reservation table* [9] to our SIPP algorithm. A reservation table is a feature which stores the location and time steps to calculate the safe intervals of a cell. We stored the continuous time intervals of every cell in the table, making it easier to know the earliest safe interval of a given location, while also avoiding collisions with other agents' previously planned paths.

The reservation table is indexed by a cell and stores the associated time intervals when those cells are occupied by

the other agents. When inserted to the reservation table, the time intervals are adjusted to satisfy the k-Robust condition, meaning that a reserved interval that would normally be $[t_{begin}, t_{end}]$ is changed to $[t_{begin} - k, t_{end} + k]$. This allows the SIPP algorithm to find the earliest possible time to arrive at the desired cell, whilst maintaining k-Robustness.

The resulting algorithm produces plans with a slight increase in cost (shown in Section V), but with an increased degree of safety, while still being precise due to the avoidance of some abstractions that are part of the standard MAPF problem.

C. Theoretical properties

Our algorithm upholds the same standards of soundness and completeness as CCBS, as we have preserved the fundamental structure that imbues CCBS with these properties. Through this structure, kR-CCBS generates pairs of range constraints that are sound, defined in [6] as being sound iff all plans satisfy at least one of the constraints. Soundness is maintained since we still detect conflicts within every joint plan, as in [5], with the added check of k-Robust conflicts.

Completeness comes from generating sound pairs of constraints and the kR-CBS proof from [6]. Similar to CCBS, consider a CT node denoted as N, which generates two children, N_1 and N_2, each governed by a sound pair of constraints, C_1 and C_2, respectively. Here, $\pi(N)$ represents the set of all joint plans compliant with N's constraints. Given that C_1 and C_2 form a sound pair of constraints, it logically follows that $\pi(N)$ encompasses the union of joint plans satisfying N_1 and N_2 individually. Thus, the splitting of CT nodes does not result in the loss of any joint plan. Due to the same characteristic shown above, it follows that kR-CCBS generates optimal k-Robust plans[2], meaning that they are optimal given the value of k.

V. Experiments

A. Configurations

In this section, we present the experiments conducted to evaluate our method in relation to other continuous time approaches. Our algorithm is compared in simulated and practical scenarios with the standard implementation of CCBS [22] and also varying the k-robustness parameter. Although some other algorithms have made progress in bringing abstract plans to higher-fidelity simulations, we did not select them for comparisons. MAPF-POST [23] is an algorithm that accounts for the robots' kinematics, but only by post-processing MAPF plans received as input, which is a different problem than what is dealt with in our work. Another algorithm, ICTS [12], works with non-unit costs, and it is already compared to CCBS in its presentation, so the comparison was also omitted since CCBS outperforms ICTS [22].

In the experiments, we used standard Turtlebot3 robots from ROBOTIS[3] and conducted the simulations on Gazebo robot simulator. Table I shows the parameters of the simulated experiments, such as the grid size, cell size (in relation to the robot diameter), etc. Ten random configurations of start and goal positions were generated for each different group of (6 to 13) agents. The starting configuration of all instances had each agent in the center of the cell corresponding to their start position and facing towards their final goal cell. For each algorithm, we set a time limit to find a valid plan. If an algorithm failed to produce a plan within this time limit, we considered it a failure in execution. The calculated plans for each instance were then fed to robots, which executed them with no post-processing.

grid size:	10×10	num. agents in group:	6 to 13
cell size:	$2\times$ robot diam.	num. runs per group:	10
movement:	2^3 neighbor.	time limit for planning:	$60s$

TABLE I: Parameters of the experiments

We tested our proposed algorithm with two different k-robustness parameters, $k = 2s$ and $k = 3s$. It is noted that the k-value's effectiveness may vary according to the configuration of each environment, regarding cell size and/or agent speed. In our case, the k-values represent the estimated time for a robot to travel two and three cells, respectively, in a non-diagonal direction.

B. Real experiments

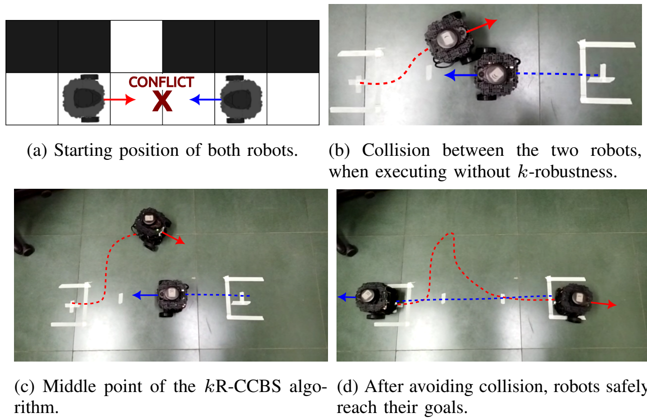
We had two Turtlebot3 burger robots available for practical experiments, the exact same model used in our Gazebo experiments. Since the amount of robots was not large enough to reproduce similar scenarios as the ones presented in the previous section, we opted to create specific conditions that would be able to replicate a case that would be very common in large spaces with multiple robots. Though not physically there, the abstract map where the algorithm planned the path for both agents had only one row of cells, meaning that they had no option but to cross each others paths. There was also a single cell at the top row, meaning that in order to avoid collisions, one agent had to use that cell to maneuver around the other.

Fig. 3 shows the drawing of the grid and screenshots taken from the video of the experiment. As expected, the two robots successfully avoided each other and completed their paths without collisions. In contrast, executing the same experiment without the k-robust parameter caused a collision between the two Turtlebots (seen in Fig. 3b). The collision occurred when the robot on the left was moving diagonally "upwards" to occupy the free cell in the top row, and the robot on the right was moving "left" towards its goal position. This happened because, as previously stated, the generated plans of CCBS expected a perfect execution,

[2]The higher the value of k, the longer the cell occupation time for each agent, and thus the longer the plan obtained will be, which clearly is not optimal. But the result is optimal considering the restriction of such occupation time k. If $k = 0$, the algorithm falls into the (optimal) CCBS.

[3]The Turtlebot3 burger has a roughly cylindrical shape with dimensions (L x W x H): 138 mm x 178 mm x 192 mm. Its maximum translational speed is 0.22 m/s, but our experiments limited it to 0.2 m/s. The maximum rotational speed is 2.84 rad/s, and we limited to 2.5 rad/s.

but a small delay of the left robot caused an unexpected conflict.



(a) Starting position of both robots.

(b) Collision between the two robots, when executing without k-robustness.

(c) Middle point of the kR-CCBS algorithm.

(d) After avoiding collision, robots safely reach their goals.

Fig. 3: Configuration and execution of both CCBS and kR-CCBS algorithms, taken from videos of practical executions with two Turtlebot3 robots. Without considering unexpected delays, collisions may occur, as in (b). With kR-CCBS, in (c) and (d), safety regions related to k are respected to avoid collision. Complete videos are available at `https://youtube.com/playlist?list=PLAst3_ReOaDwqgDiNFzVL6f3Lh9IdbMEW&si=24dCdFwAlpOmgUDJ`

C. Simulated experiments

Simulated experiments with a larger amount of robots were performed to assess the scalability of our method. We selected Gazebo as the simulation tool, which accurately simulates rigid bodies considering factors such as friction, which may cause stuttering or different velocities on both wheels, resulting in unbalanced movement and unpredicted delays. Fig. 4 shows the start and goal configuration of a successful execution with ten robots.

For our evaluation, we compared the success rate of executions and the planned flowtime and makespan values with the values obtained in the simulations. Fig. 5 displays our data in all three aspects. Fig. 5a shows the results of the plans executed in Gazebo simulations. We can see that plans generated with k-robustness have a higher success rate when executed in such scenarios, achieving an average success rate of 92.5% with the k-robustness parameter set to

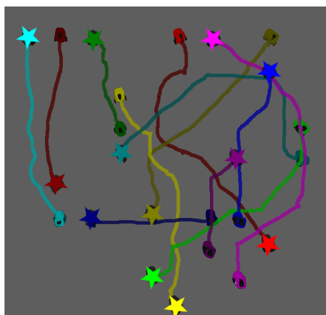
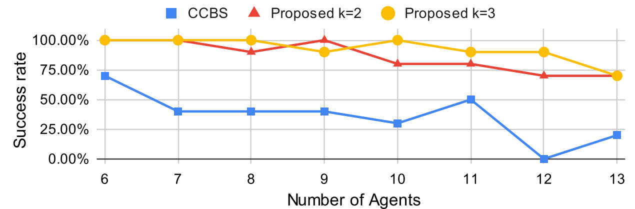
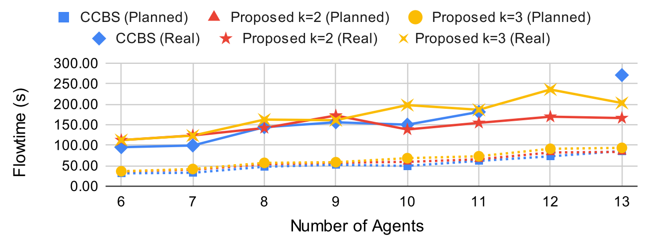


Fig. 4: Example of a successful execution, where robots reach their goal positions without collisions. Instance completed using kR-CCBS with k-value = 2. Robots were colored for ease of visualization. The starting position is highlighted, with the path in the same color leading to the goal position. The goal position for each robot is represented by a star in their corresponding colors.

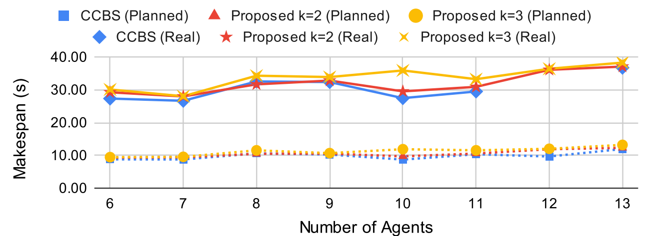
$3s$. We observed that the plans generated with k-Robustness also achieve a higher percentage of success as the number of simultaneous robots increases, maintaining consistency even with larger numbers. Note that the more robots in the environment, the greater the chance that their paths will cross and the greater the chance that unforeseen delays will generate collisions. This is why all methods, but especially CCBS, fail more as the number of robots increases. The proposed method minimizes this by giving larger safe intervals to occupy each region. Obviously, success depends on the delays not exceeding the value of k (or only slightly exceeding it). Otherwise, the method may also fail.



(a) Success rate of kR-CCBS with two different k and CCBS.



(b) Average planned flowtime and average flowtime measured after execution.



(c) Average planned makespan and average makespan measured after execution.

Fig. 5: Comparison of kR-CCBS with $k = 2s$ and $k = 3s$ and standard CCBS in terms of (a) success rate, (b) flowtime, and (c) makespan varying the number of agents, along with the differences between the planned and obtained times in the simulated executions.

Fig. 5b compares the average flowtime values for the CCBS algorithm and the kR-CCBS algorithm. Fig. 5c compares the average makespan values (both originally planned and after execution) for the two algorithms as well. The *Planned* category in both Figures 5b and 5c represent the values measured in the abstract plan. The *Real* category in the same figures represent the values measured in the simulated executions. We note that there are no flowtime or makespan values for CCBS with 12 agents, since all executions failed. As expected, higher k values result in higher flowtime and makespan values, while the CCBS algorithm yields slightly lower values. Nevertheless, when seeing the difference in values alongside the difference in success rate, we can see

that there are advantages when using our algorithm.

We further analyze this by looking at Table II. It shows the flowtime and makespan increase when executing the generated plans, and the average success rate for the two configurations of kR-CCBS and for standard CCBS.

	Flowtime Increase	**Makespan Increase**	**Success Rate**
$k = 2s$	196.89%	204.74%	86.25%
$k = 3s$	190.77%	202.54%	92.50%
CCBS	201.30%	207.50%	36.25%

TABLE II: Average flowtime increase and makespan increase, and average success rate for all instances. The increase is calculated by comparing the flowtime and makespan values from the abstract plan with the values from the executed plan in simulation, then averaging the difference between all completed instances for each algorithm.

As expected, there is a significant increase in values, due to the physics involved in realistic movement, but with low variation between both algorithms, with average increases of around 200%. The overall success rate of k-robust plans was 89.38%, as opposed to CCBS's overall success rate of 36.25%, meaning that with roughly similar values of executed flowtime and makespan, we managed to provide safer plans for groups of robots. Additionally, as the number of agents increases, the success rate for the proposed algorithm remains practically consistent, while the success rate for CCBS drops.

VI. Discussion

In this paper, we presented kR-CCBS, an algorithm that implements a safety layer using k-robustness for multi-agent path planning with continuous time. We showed the effectiveness of our algorithm when executing the plans on simulations of multi-robot systems, and on real robots on a small scale. Our results, compared to competing methods, indicated that the cost increase in plans using kR-CCBS, especially in a larger amount of agents, is a reasonable trade-off considering the higher rate of success of our executions.

The method's main advantage is that by combining continuous time with the concept of k-robustness, the kR-CCBS presents itself as a simplified way of applying MAPF in realistic multi-robot systems. Unlike discrete planning algorithms, kR-CCBS is not limited to the application of discrete move and waiting actions, allowing different robots to move at and for different periods. Furthermore, applying the k-robust checking mechanism, allows a layer of tolerance to be added to the execution of actions, being robust to delays that fall within the predicted range, without the need for explicit modeling of the factors that lead to such delays.

As limitations, a collision-free operation is guaranteed if the delays do not exceed k seconds. Furthermore, as k increases, the longer it takes for the paths to complete since each region can only be occupied by one agent during the entire period of k, thus there is this trade-off between excessive precaution and greater lack of protection against collisions. For future work, we intend to improve the algorithm's scalability by adding landmarks and disjoint splitting, as in [22]. Adapting the algorithm to work on roadmaps could also generate more organic paths.

References

[1] F. Ho, R. Geraldes, A. Gonçalves, B. Rigault, B. Sportich, D. Kubo, M. Cavazza, and H. Prendinger, "Decentralized multi-agent path finding for uav traffic management," *Trans. Intell. Transport. Sys.*, vol. 23, no. 2, p. 997–1008, feb 2022.

[2] L. Wen, Y. Liu, and H. Li, "Cl-mapf: Multi-agent path finding for car-like robots with kinematic and spatiotemporal constraints," *Robotics and Autonomous Systems*, vol. 150, p. 103997, 2022.

[3] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. K. S. Kumar, and S. Koenig, "Lifelong Multi-Agent Path Finding in Large-Scale Warehouses," *arXiv:2005.07371 [cs]*, Mar. 2021, arXiv: 2005.07371.

[4] H. Ma, J. Yang, L. Cohen, T. K. S. Kumar, and S. Koenig, "Feasibility Study: Moving Non-Homogeneous Teams in Congested Video Game Environments," *arXiv:1710.01447 [cs]*, Oct. 2017, arXiv: 1710.01447.

[5] A. Andreychuk, K. Yakovlev, D. Atzmon, and R. Stern, "Multi-agent pathfinding with continuous time," in *Proc. of 28th Int. Joint Conference on Artificial Intelligence, IJCAI-19*, 7 2019, pp. 39–45.

[6] D. Atzmon, R. Stern, A. Felner, G. Wagner, R. Bartak, and N.-F. Zhou, "Robust Multi-Agent Path Finding," *Proc. of International Symposium on Combinatorial Search*, vol. 9, no. 1, pp. 2–9, Sept. 2021.

[7] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak, "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks," p. 8.

[8] H. Ma, "Graph-Based Multi-Robot Path Finding and Planning," *Current Robotics Reports*, vol. 3, no. 3, pp. 77–84, June 2022, arXiv:2206.11319 [cs].

[9] H. Ma, W. Hönig, T. K. S. Kumar, N. Ayanian, and S. Koenig, "Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery," *AAAI 2019*, vol. 33, pp. 7651–7658, July 2019.

[10] K. Yakovlev and A. Andreychuk, "Any-Angle Pathfinding for Multiple Agents Based on SIPP Algorithm," Mar. 2017, arXiv:1703.04159 [cs].

[11] K. Yakovlev, A. Andreychuk, and V. Vorobyev, "Prioritized Multi-agent Path Finding for Differential Drive Robots," *2019 European Conference on Mobile Robots (ECMR)*, pp. 1–6, Sept. 2019.

[12] T. T. Walker, N. R. Sturtevant, and A. Felner, "Extended Increasing Cost Tree Search for Non-Unit Cost Domains," in *Proc. of 27th IJCAI*, Stockholm, Sweden, July 2018, pp. 534–540.

[13] P. Surynek, "An Optimization Variant of Multi-Robot Path Planning Is Intractable," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, pp. 1261–1263, July 2010.

[14] J. Yu and S. LaValle, "Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs," *Proc. of the AAAI Conference on Artificial Intelligence*, vol. 27, no. 1, pp. 1443–1449, June 2013.

[15] S. D. Han and J. Yu, "Integer programming as a general solution methodology for path-based optimization in robotics: Principles, best practices, and applications," in *IEEE/RSJ IROS*, 2019, pp. 1890–1897.

[16] B. de Wilde, A. W. ter Mors, and C. Witteveen, "Push and rotate: Co-operative multi-agent path planning," in *Proc. of 2013 Int. Conference on Autonomous Agents and Multi-Agent Systems*, 2013, p. 87–94.

[17] J. Li, Z. Chen, D. Harabor, P. J. Stuckey, and S. Koenig, "MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search," *Proc. of AAAI Conference on Artificial Intelligence*, vol. 36, no. 9, pp. 10 256–10 265, June 2022.

[18] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent path finding." in *MAPF@AAAI*, ser. AAAI Workshops, F. et al., Ed., vol. WS-12-10, 2012.

[19] M. Phillips and M. Likhachev, "SIPP: Safe interval path planning for dynamic environments," in *2011 IEEE ICRA*. Shanghai, China: IEEE, May 2011, pp. 5628–5635.

[20] D. Atzmon, R. Stern, A. Felner, N. R. Sturtevant, and S. Koenig, "Probabilistic Robust Multi-Agent Path Finding," *Proc. of Int. Conf. on Automated Planning and Scheduling*, vol. 30, pp. 29–37, June 2020.

[21] S. Guy and I. Karamouzas, *Guide to anticipatory collision avoidance*. CRC Press, Jan. 2015, pp. 195–208.

[22] A. Andreychuk, K. Yakovlev, E. Boyarski, and R. Stern, "Improving Continuous-time Conflict Based Search," Mar. 2021, arXiv:2101.09723 [cs].

[23] W. Hoenig, T. K. S. Kumar, L. Cohen, H. Ma, H. Xu, N. Ayanian, and S. Koenig, "Multi-Agent Path Finding with Kinematic Constraints," p. 9, 2017.