UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUILHERME NOVAK MOTTA DAUDT

k-Robust CBS with Continuous Time for Multi-robot Coordination

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Renan Maffei

Porto Alegre
July 2024

CIP — CATALOGING-IN-PUBLICATION

Daudt, Guilherme Novak Motta

k-Robust CBS with Continuous Time for Multi-robot Coordination / Guilherme Novak Motta Daudt. – Porto Alegre: PPGC da UFRGS, 2024.

70 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2024. Advisor: Renan Maffei.

I. Maffei, Renan. II. Título.

“Highly organized research is guaranteed to produce nothing new.”

— Frank Herbert, Dune

AGRADECIMENTOS

Agradeço a compreensão de todos pela minha falta de paciência para escrever o agradecimento.

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 MAPF algorithms exhibit shortcomings when handling high-level abstractions, neglecting real-life aspects, consequently leading to failures in live executions. In this paper, we propose k-Robust CCBS, a novel algorithm that overcomes some of these limitations. Our approach offers path planning with continuous time, leading to more precise routes. Additionally, we ensure safety through the incorporation of k-robustness, enabling the system to adapt to agent failures and minimize collision risks. Comparative evaluations demonstrate that k-Robust CCBS outperforms similar works in terms of effectiveness while maintaining reasonable costs, making it a promising solution for real-world multi-agent coordination scenarios.
Keywords: Multi-Agent Path Finding. Path Planning. Robust Planning. Artificial Intelligence.

Busca k-Robusta Baseada em Conflitos com Tempo Contínuo para Coordenação de Múltiplos Robôs

RESUMO

Coordenar múltiplos robôs é crucial em várias aplicações na vida real. Muitos algoritmos de Planejamento de Caminhos Multi-Agente (MAPF) provaram-se bem-sucedidos abordando este desafio. No entanto, alguns algoritmos MAPF possuem limitações quando precisam lidar abstrações de alto-nível, negligenciando aspectos da realidade, consequentemente levando a falhas em execuções reais. Nesta dissertação, proponho o k-Robust CCBS, um algoritmo novo que supera algumas destas limitações. Esta abordagem possui um planejamento de caminhos com tempo contínuo, gerando rotas mais precisas. Adicionalmente, enforça-se segurança através da incorporação de k-Robustez, habilitando a adaptação do sistema a falhas de agentes e minimizando os riscos de colisões. Avaliações comparativas demonstram que o k-Robust CCBS supera trabalhos similares em termos de efetividade, enquanto mantém custos razoáveis, fazendo com que seja uma solução promissora para cenários reais de coordenação de múltiplos agentes.

Palavras-chave: Planejamento de Caminhos Multi-Agente. Planejamento de Caminhos. Planejamento Robusto. Inteligência Artificial.

LIST OF ABBREVIATIONS AND ACRONYMS

AA-SIPP	Any-Angle Safe Interval Path Planning
ADG	Action Dependency Graph
CBS	Conflict-Based Search
CCBS	Continuous-time Conflict-Based Search
CT	Constraint Tree
k-RCCBS	k-Robust Continuous-time Conflict-Based Search
LB	Lower Bound
LNS	Large Neighborhood Search
MAPD	Multi-Agent Pickup and Delivery
MAPF	Multi-Agent Path Finding
MAPF-LNS2	MAPF with Large Neighborhood Search
PBS	Priority-Based Search
RHCR	Rolling-Horizon Collision Resolution
SIPP	Safe Interval Path Planning
SIPPS	Safe Interval Path Planning with Soft constraints
SLAM	Simultaneous Localization and Mapping
STN	Simple Temporal Network
TP	Token Passing
TP-SIPPwRT	Token Passing Safe-Interval Path Planning with Reservation Table
UB	Upper Bound
WSCaS	Walk, Stop, Count and Swap

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 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 within them. However, one of the key challenges in robotics is to efficiently coordinate multiple robots in dynamic and 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. In Ho et al. (2022), an air traffic management system for Unmaned Aerial Vehicles (UAVs) is presented. The applicability of these types of systems are various, particularly in the current scenario where it is feasible to automate package delivery tasks. Effective air traffic management is desirable for an aerial robotic fleet, making it more efficient. Similarly, Wen, Liu and Li (2022) propose an algorithm that deals with a multiple Car-like robot system. Automated cars are being widely researched, with some even being used in real traffic situations as self-driving taxis. Their applicability can also be extended to office robots or unmanned surface vehicles. Applications range even into video-game AI (MA et al., 2017). Different game genres, such as turn-based strategy or real-time strategy, sometimes involve the player controlling a large number of heterogeneous agents in dynamic or congested environments. For a seamless experience, it is ideal that those agents move as a team instead of individually. To do so, a MAPF algorithm can be employed to find a collision-free path for the team and between agents individually, when going to a user-specified goal.

MAPF is also relevant in automated warehouse scenarios (LI et al., 2021). Whether it is storage or sorting products, for example, there are a number of different tasks that can be automated by using robots in a warehouse environment. One of the main difficulties in this scenario is that those environments are usually tightly packed, and maximizing efficiency means having as large of a robotic fleet as possible. Algorithms for these scenarios also should ideally deal with the constant assignment of new tasks for robots after they complete their current ones. Therefore, it is important to find a solution that accommodates fleets in large scales without losing efficiency. Figure 1.1 shows a practical scenario of an automated warehouse, as well as a representation of a map where automated mobile robots can act.

(a) Amazon automated warehouse

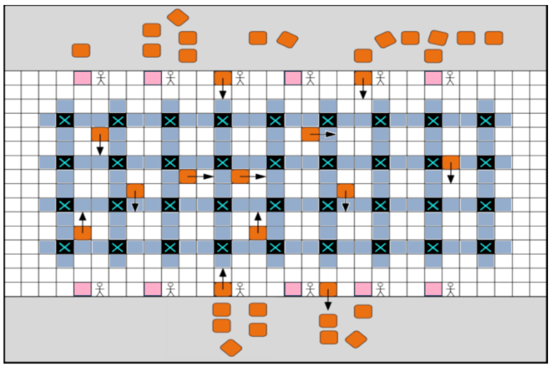

(b) Sorting center environment map

Figure 1.1 – Figure 1.1(a) shows an automated warehouse environment powered by Amazon robots. Figure 1.1(b) depicts a sorting center environment where automated robots would act on. Taken from Li et al. (2021)

While having wide applications, the existing solutions for these scenarios abstract some practical details, to make them more feasible or efficient. To make the problem more tractable, we must disregard some of the factors involved in a live execution, such as considering kinematics, dealing with failures, or considering uncertainties. This results in plans that are generated from most algorithms requiring 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, being able to create plans that need less adaptation while maintaining effectiveness can be a way to save resources. Consequently, a large number of solvers work within parameters such as discrete time and space.

1.1 Motivation

Bringing the MAPF algorithms into real-world situations can be a different challenge than solving the problem in a controlled, theoretical situation. As previously mentioned, there are a number of factors that are not taken into account for the sake of feasibility when planning paths for robots. For example, picture two robots moving from their starting positions to their goal positions in the real world. Figure 1.2 illustrates the example. Paths are purposely depicted as not perfectly straight since this would be a more accurate representation of a real-life situation. Paths are also shown in the same color as their respective robots. Consider that a MAPF algorithm has already been run for this problem and deemed that if they followed a straight line, they would come close to each other, but no collision would occur. However, this was expected to happen in an ideal

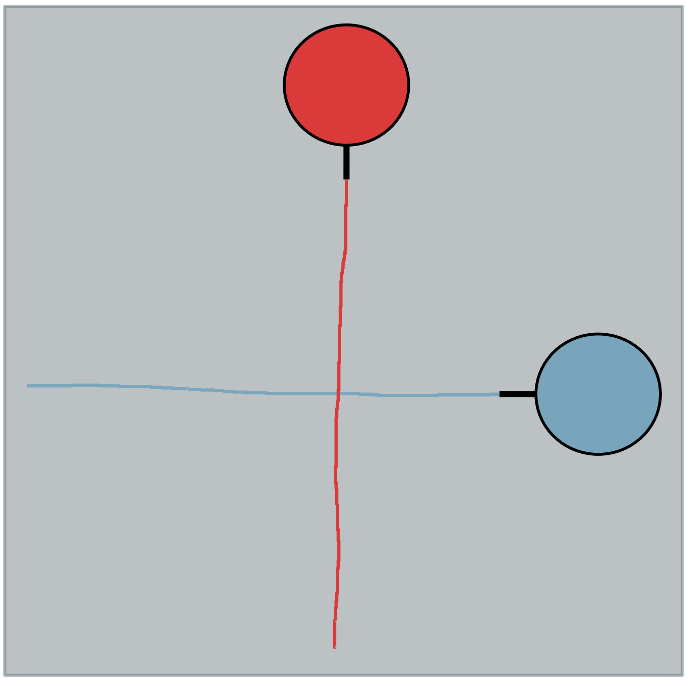


Figure 1.2 – Example of a real scenario with two disk-shaped robots, with their paths crossing over each other. The black lines coming out from the disks show the direction each robot is facing.

scenario with perfect movement from both robots. Several different things could happen, but in this case, suppose that the blue robot drifted for a short amount of time, due to a small lack of friction from its wheels to the ground, causing it to delay its arrival at the goal point. With this small delay, what would be a close passing between the two robots becomes an unforeseen collision. Now, the successful MAPF plan is turned into a failure in a real-world scenario due to the selection of a certain degree of abstraction during the planning phase.

1.1.1 Planning with continuous time

Attempting to approximate the results to reality, some algorithms have focused on solving the MAPF problem resembling a more lifelike situation than simply planning in discrete time steps. A very common abstraction used is **discretizing time**, since it is very easy for us to track and visualize scenarios by counting time as Natural numbers (as we do in our own heads as well). Time and its role in planning will be further explained in section 2.1, but for now it is important to note that it can largely impact the planning time. Even though this is a sensible abstraction, it ends up generating less reliable plans in more complex situations. While dealing with time in a continuous manner might seem unfeasible, there are ways of using time in a non-discrete manner that keep the problem tractable.

One of the algorithms that do this is Continuous Time Conflict-Based Search (CCBS) (ANDREYCHUK et al., 2019). This algorithm plans using **continuous time**, producing results that have better time estimates and are more suitable for real-life applications than discrete-time algorithms. This will be explained in more detail in section 4.2, and for now, it is highlighted as a way to work around the time abstraction while still maintaining reasonable processing times.

1.1.2 Planning with an extra layer of safety

A similarly common abstraction used in algorithms regards the robot's motion. It is usual to assume that the agents will have constant movement, disregarding acceleration and deceleration or direction changes and turn rate. Real-world movement is very complex and sometimes depends on external factors unknown by traditional planners, such as friction or drag. Without mentioning every different aspect of robotic movement, it is not difficult to see how having to take into account all of them can lead you to an intractable problem. An alternative to that would be observing the robots' movements as they occur and adjusting to those responses, but this requires constant replanning and processing of data from different components. This is possible, and as a matter of fact, this type of system is used in real-life applications.

Another aspect is that algorithms tend to focus on optimizing results, meaning that they are planning for the shortest routes for the robots to arrive at their destination as soon as possible. In a multi-agent scenario, planning optimal routes can mean that agents' paths come very close to each other. Considering the high degree of difficulty involved in robotic movement, having near-misses can be crucial when executing a plan in a realistic scenario. Post-processing abstract plans are an option, MAPF-POST (HOENIG et al., 2017) is independent of the solver algorithm, and prepares plans for practical execution. However, we are interested in techniques that can deal with some of the issues caused by the non-precise factors of robotic movement without replanning and without drastically expanding our search space by adding many more movement-related conditions. The concept of k-robustness (ATZMON et al., 2021) can lower the impact of abstracting these factors. It determines safe intervals for places that have been previously occupied by robots. This means that near-misses during routes are less likely to happen, since the safe intervals prevent that a robot would plan motions that overlap with the other robots' paths for a set period of time. k-Robust CBS, the algorithm that explores this concept, will be further explained in section 4.3. Planning with robustness causes agents to more often complete their paths without collisions despite the imperfections in other agents' movements. During our experiments, we intend to show the performance of the compared algorithms in an open space, with no obstacles in order to focus on handling collisions exclusively between robots. We have also performed tests in maps that have different configurations with obstacles, tighter passages and opportunity of movement. However, those tests weren't made with enough consistency or quantity in order to be included in our analysis.

1.2 Objectives

Our primary goal is to generate plans that can be executed safely and reliably, even when confronted with unforeseen circumstances, all the while maintaining their effectiveness. In this dissertation, we present k-robust CCBS, an algorithm that uses continuous time, in order to produce plans that are more accurate when transferred to real scenarios, while also including the concept of k-robustness, making the plans safer against unpredictable delays in movement.

1.3 Contribution

Our algorithm offers several advantages over existing approaches. The inclusion of k-robustness ensures safer plan execution, making the system resilient to agent failures and reducing the likelihood of collisions. 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. In summary, we propose an algorithm that plans by using continuous time and k-robust features. We prove through empirical simulations and real-world tests that k-Robust CCBS is more successful in producing collision-free executions than CCBS. We also show that the added safety feature in our algorithm does not drastically increase planning costs before and after execution. To the best of our understanding, k-Robust CCBS is a new algorithm, being optimal (bounded by the k factor of robustness) and complete.

1.4 Organization

This dissertation is organized as follows. Chapter 2 states the theoretical background in different aspects related to planning. Chapter 3 describes the MAPF problem and shows some generalizations that lead to the case considered by our algorithm. Then, in Chapter 4, we present the related work focusing on papers that were relevant when developing our method, as well as explaining in more detail the key concepts that we share with them. In Chapter 5, we detail our implementation, highlighting the differences between its predecessors and showing how we deal with the new restrictions created by the k-robust concept. Chapter 6 shows our experiments and the high success rate achieved by our algorithm in high-fidelity simulations. The chapter also shows a successful execution with two real robots in a specific environment, bringing out the usefulness of using k-robust CCBS. Finally, we conclude with a brief summary and suggest future work.

2 THEORETICAL BACKGROUND

2.1 Planning

When performing, whether in simulations or in the real world, robots need several different capabilities to be able to accomplish their goals. Those capabilities are usually categorized into three base aspects, and the subcategories being the combination of them. The three base categories are Mapping, Localization, and Planning. Mapping can be defined as building a representation or diagram of a determined location or environment. Localization can be defined as estimating the robot's position in the environment based on the information captured by sensors (i.e., light, vision, sound). This dissertation falls mainly into the third category: **Planning**.

There are a few basic ingredients that are part of virtually every topic related to planning, as described in Lavalle (2006).

State is typically referred to a determined configuration or condition in the planning system at a certain point in *time*. Therefore, planning systems work on finding solutions in a *state space*, which is the complete set of states in the problem, and they can be discrete (finite or countably infinite) or continuous (uncountably infinite). A state represents the current situation of a system, including the values of relevant variables and parameters. This could be a robot's position defined by a set of coordinates in a plane. It could also be the configuration of a puzzle, like which rings are on the rods in the Tower of Hanoi. States can be more or less complex, and it is important to notice that increasing state complexity can also increase the size of your *state space*.

Time is a factor that is part of every planning problem, often crucial, and influences the dynamics of a system and the sequencing of *actions*. It can be modeled explicitly in scenarios such as when a car has to race towards its goal as early as possible. On the other hand, it can be represented implicitly in problems where speed isn't necessarily relevant, representing, in this case, the order of actions taken, i.e., the succession of steps necessary to solve a puzzle. Time can be viewed as a finite resource, meaning that it is common for planning systems to have as an objective the efficient management of this resource. As with state spaces, it can be either discrete or continuous. Incorporating time into planning models allows for a more realistic representation of dynamic systems where actions take time to execute, and the timing of actions can significantly impact the outcome. Temporal reasoning in planning is essential for addressing real-world scenarios where timing constraints and dependencies exist. Planning algorithms that consider time help generate plans that not only achieve the desired goals but also adhere to temporal constraints and requirements.

Action is a basic unit of change that can be performed on a system to transition it from one state to another. Actions are the building blocks of plans, and the planning process involves determining a sequence of actions to achieve a specific goal or reach a desired state. Actions possess preconditions, denoting the conditions that must be fulfilled for an action to be applicable; only when these preconditions are met can the action be executed. Furthermore, actions have effects that articulate how the system's state transforms after the action is carried out, outlining the resultant new state. In addition, actions may be associated with costs, signifying the time or resources required for their execution. Planning algorithms often aim to minimize the overall cost of a plan. Actions can also be characterized by durations, representing the time required for completion. Some actions may be instantaneous, while others may extend over a period. Consideration of concurrency is another aspect, wherein certain planning domains allow actions to be performed concurrently, allowing for overlap in execution. Moreover, actions may be subject to various constraints, such as resource constraints or temporal constraints, influencing their applicability or timing.

Initial and goal states are usually present in every planning system. Plans are a sequence of actions taken to attempt to reach a certain goal state, starting from a determined initial state.

A criterion encodes a plan's desired outcome. Generally, there are two different planning concerns based on the criterion of *feasibility* and *optimality*. Feasibility means finding a plan that arrives at the goal state, regardless of its efficiency. Optimality means finding a plan that arrives at the goal state, while also optimizing performance in some determined matter. While feasibility ensures that a solution is viable and can be executed, optimality aims to identify the most desirable solution among the feasible alternatives. It is common for planners and decision-makers to seek not only feasible solutions but also optimal ones based on specific criteria or objectives. The trade-off between feasibility and optimality often depends on the goals and constraints of the particular problem at hand.

A plan is a strategic arrangement of actions designed to guide the transition from an initial state to a desired end state. Plans articulate a sequence of actions, each governed by specific preconditions and leading to defined effects on the system. The order in which actions are executed is critical, as it determines the trajectory toward achieving a particular goal. Ultimately, the purpose of a plan is to outline a coherent and effective strategy for navigating a system or solving a problem. Planning algorithms are commonly employed to generate plans, considering factors such as feasibility, optimality, and adherence to specified constraints.

Next, we show an example of planning robot movement in a 2D Grid, taken from Lavalle (2006). Suppose that a robot moves on a grid where each grid point has integer coordinates of the form (i, j). The robot executes discrete movements in one of four directions: up, down, left, or right, modifying one coordinate per step. Visualize this as the robot traversing a vast grid, like stepping from tile to tile on an infinite tile floor, as depicted in Figure 2.1. Let X represent the set of all integer pairs in the form (i, j), where $i, j \in \mathbb{Z}$ ($\mathbb{Z}$ denoting all integers). Define U as the set $(0, 1), (0, -1), (1, 0), (-1, 0)$, representing the possible movements. For any x in X, let $U(x)$ be equivalent to U. The state transition equation is defined as $f(x, u) = x + u$, with $x \in X$ and $u \in U$ treated as two-dimensional vectors for addition purposes. For instance, if $x = (3, 4)$ and $u = (0, 1)$, then $f(x, u) = (3, 5)$. Assuming an initial state of $x_I = (0, 0)$ for convenience, diverse goal sets can be defined. Let's consider $X_G = (100, 100)$ as an example. Finding a sequence of actions to transform the state from $(0, 0)$ to $(100, 100)$ is straightforward. To introduce complexity, obstacles are introduced by shading specific square tiles, as illustrated in Figure 2.2. Tiles that are shaded have their corresponding vertices and associated edges removed from the state transition graph. Additionally, an outer boundary can enclose a finite region, rendering X finite. This introduces the possibility of constructing intricate mazes, resulting on interesting navigation problems.

2.2 Space representation

In the previous section, we defined Planning and its most relevant components. As we showed in the previous example, plans are generated to navigate a system or solve

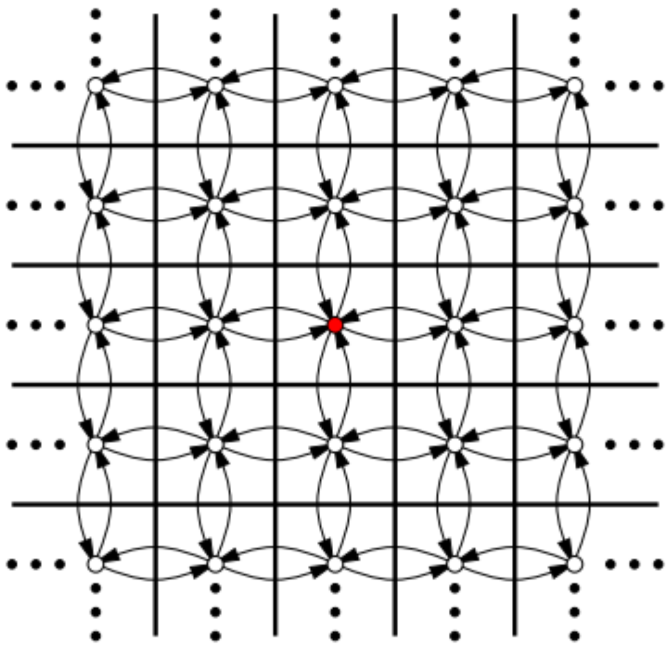


Figure 2.1 – A state transition graph for the example problem of walking on an infinite tile floor. This depicts the case of a robot that can only move in four cardinal directions. Image extracted from Lavalle (2006)

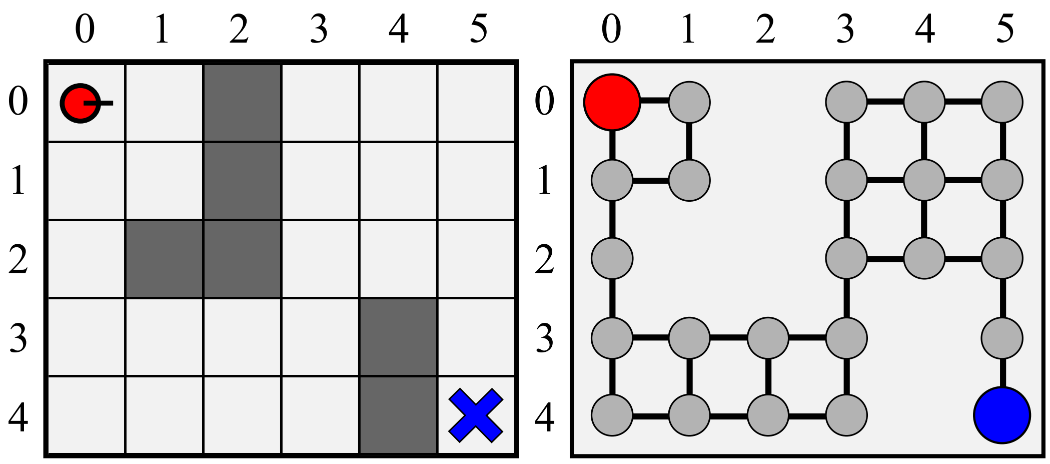


Figure 2.2 – A 2D grid with free/occupied cells can be viewed as an unweighted graph, with edges connecting free cells. An outer boundary was added, and shaded squares are considered obstacles. The initial position is shown in red, and the goal one in blue.

a problem. These problems occur in a space, and it is important to have a way to depict this space to convey and capture this spatial information. There are plenty of different approaches and techniques, and choosing which ones to use to model the space depends on the difficulty of the problem or on the application (LAVALLE, 2006).

Models where the planning algorithm acts on are commonly called **worlds**, and generally have two (2D) or three (3D) dimensions. Worlds usually contain at least two basic elements, the **robots** or **agents**, which are controlled via the plan, and the **obstacles**, which are areas in the world that are permanently occupied.

We define a **Configuration Space**, C, as a structured representation of all configurations in the world, that is used to determine the feasible states a robot can achieve. A 2D world generally has in C a configuration like $C = (x, y, \theta)$, where (x, y) represents the two coordinates of the robot's position and θ represents the orientation of the robot. Orientation can be represented as $\theta = [0, 2\pi)$. We will also use C_{free} and C_{obs} as notations to define free and obstacle areas in the configuration space, respectively.

The **free area** is a subset of the whole configuration space, expressed as $C_{free} \subseteq C$, that does not intersect with any of the obstacle areas. C_{free} will be the space where the planning algorithms search for paths to reach their destinations. In a similar manner, the **obstacle area**, C_{obs}, is another subset of the configuration space, expressing areas where the robot cannot go through. The subset $C_{obs} \subseteq C$ expresses the invalid configurations where the robot's position overlaps with an obstacle.

Configuration spaces can be either discrete or continuous. In a continuous configuration space, coordinates can take any real value along the limits of the space. Discrete configuration spaces have a finite set of values.

Following these definitions, we can define the robot motion planning problem formally. We show a formulation from Lavalle (2006) that presents the problem as the Piano Mover's Problem.

2.2.1 Formulation: Piano Mover's Problem

1. A world $\mathbb{W}$ in which $\mathbb{W} = \mathbb{R}^2$ or $\mathbb{W} = \mathbb{R}^3$.
2. A configuration space C determined by specifying the set of all possible transformations that can be applied to the robot. From this, derive C_{free} and C_{obs}.
3. A configuration $q_I \in C_{free}$ designated as the *initial configuration*.

4. A configuration $q_G \in C_{free}$ is designated as the *goal configuration*. The initial and goal configurations are often called a *query pair* (or *query*) and designated as (q_I, q_G).
5. A complete algorithm must compute a (continuous) *path*, $\tau : [0, 1] \to C_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, or correctly report that such a path does not exist.

Due to the dimension of C being unbounded, it was shown that this problem is PSPACE-hard, which implies NP-hard (LAVALLE, 2006). We illustrate the problem and how solutions are deemed valid or invalid in Figure 2.3.

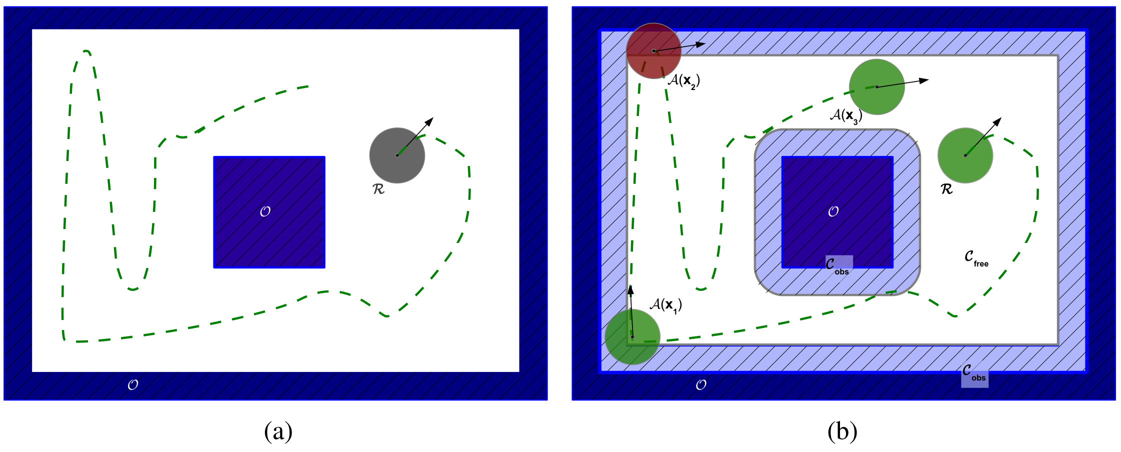


Figure 2.3 – Figure 2.3(a) shows a world W, with obstacles, free area, and a robot. The obstacles (O) are represented by the hatched blue regions, the free space by the white regions, and robot R occupies the area denoted by the grey circle, with its orientation shown by the black arrow. In a configuration space problem, the robot must find a path from its starting position to its goal position in C, navigating through C_{free} while avoiding C_{obs}. Paths are shown by the green dashed line. Figure 2.3(b) shows different configurations for the robot, with the green circles representing the ones in C_{free} and the red one where it overlaps with C_{obs}, making it invalid. Image adapted from Jorge (2017).

2.2.2 Modeling the configuration space

Modeling the configuration space can make a considerable difference when running the planning algorithm. There are two physical resource limitations when trying to find a solution to a problem, **time** and **space**, and modeling affects both. During planning, being able to quickly check if a state transition is valid or not due to the interpolation of a robot and an obstacle impacts the algorithm's running time. Doing this poorly, for example, can lead to unwanted or impractical delays. At the same time, there are cases in which it is necessary to keep some of the details of your world during execution. As the algorithm expands more states, having a too-informed or verbose model can make it rapidly reach the limit of its memory space. Problems with a high degree of complexity, where the agent has to perform more intricate movement to succeed, are more likely to use 3D representations. Having a three-dimensional world during planning usually means having a larger space state, since it adds another dimension to the space. In counterpart, it results in more possibilities of movements, sometimes required to find an acceptable solution. (LAVALLE, 2006) shows many different geometric modeling techniques of configuration spaces, such as Polygonal Models or Semi-Algebraic Models, but explaining those models in detail is beyond the scope of this dissertation.

In our work, even though the algorithm is built with a focus on real scenarios, in practice, we do not deal with or consider three-dimensional movement. Therefore, it makes sense to choose a two-dimensional model, saving resources. Still, we cannot choose a model that is efficient but too far from reality, rendering the plan unusable. Thus, the chosen model – Gridmaps – satisfies this while still being compatible with our algorithm [1].

Gridmaps are widely used in computer graphics, since it is one way to represent images. It is also commonly used to represent and build maps when a robot explores an environment using its sensors. A gridmap is defined as a grid with a determined width and height divided into regular rectangles. Each smaller rectangle, called cell (or pixels in images) is a discrete unit and has a unique position determined by its coordinates on the grid. The amount of cells in each dimension of the grid is called resolution. Having a higher resolution results in finer detail, but also increases memory space.

Obstacles can be represented in different ways. Occupancy grids, frequently used when mapping unknown environments, use a scalar value in each grid cell according to the probability of said cell to contain an obstacle. In our algorithm, since we assume that the map is already known, we use a binary representation, meaning that a grid cell is either free or completely obstructed.

Regarding the spatial information, there's also flexibility in choosing how each pair of coordinates will be represented in the corresponding space. Intuitively, it might make sense to choose the starting coordinate $(0, 0)$ in the middle of the gridmap, resulting in certain quarters of the gridmap having negative coordinates, e.g., the bottom left quarter would have negative x and y values. To keep it simpler and deal with positive values only,

[1]Chapter 5 explains the workings of our proposed algorithm, and chapter 3 section 3.2 explains more thoroughly the problem we are dealing with. After reading those two parts, the reasoning for choosing this geometric model should be clearer.

we chose the starting point of our gridmap to be on the top left, meaning that coordinates only have increasing positive values.

One of the advantages of using this type of representation is that detecting collisions can be done very fast. When using a binary representation, checking for collisions is as simple as verifying whether a pair of coordinates is assigned as an obstacle. Another advantage is that it is intuitive and easy to implement while also being memory efficient. The flexibility of being able to choose different resolutions allows us to find a good balance between accuracy and memory space. We show an example of a gridmap using a binary representation in table 2.1.

0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0
0	0	0	0	1	1	1	0	0	0
0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Table 2.1 – An example of a gridmap with a binary representation. In this instance, we use 0s to represent the free areas and 1s to represent the obstacles.

3 MULTI-AGENT PATH FINDING

3.1 Problem Definition

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 $(t, t + y)$.

Figure 3.1 displays a graph that will be used to illustrate conflicts in the MAPF problem. An example of a successful set of paths being executed is displayed in Figure 3.2, where two agents traverse their paths without any interfering with each other, performing only *move* actions. Two agents cannot occupy the same location in the graph at the same time step. This occurrence is defined as a **vertex conflict** (STERN et al., 2019), 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. Figure 3.3 illustrates a vertex conflict between two agents, where both agents are trying to occupy vertex $g3$. Two agents also cannot traverse the same edge at the same time step. This is defined as an **edge conflict** (STERN et al., 2019), following the same inequality above. Figure 3.4 shows an edge conflict between two planned paths. In this scenario, while the red agent is trying to move to vertex $g4$, the green agent is trying to move to vertex $g3$. To do so, both have to traverse through the same edge, which classifies as a conflict.

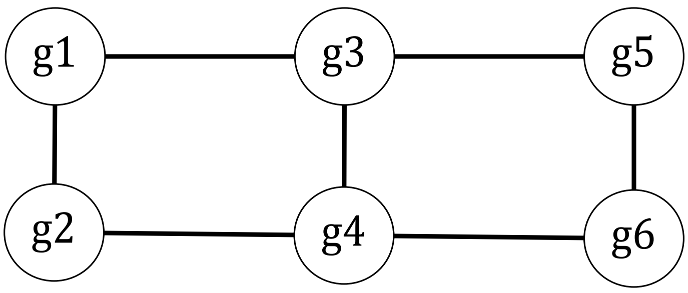
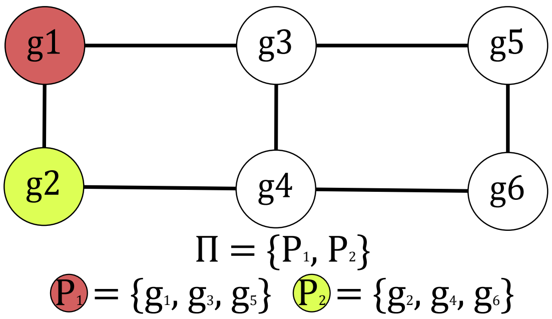
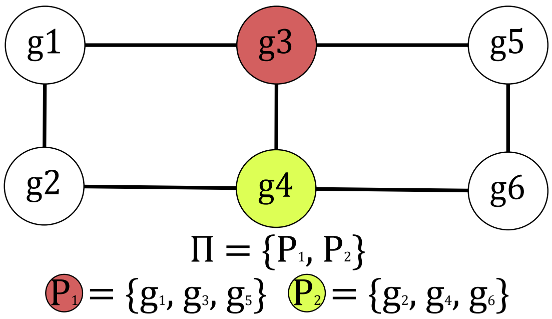


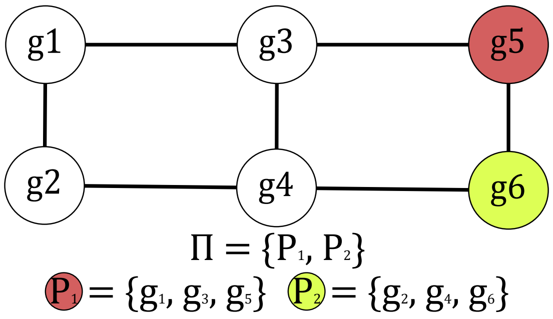
Figure 3.1 – Graph G used as a base for the examples of conflicts in MAPF.



(a) Path execution at t_0

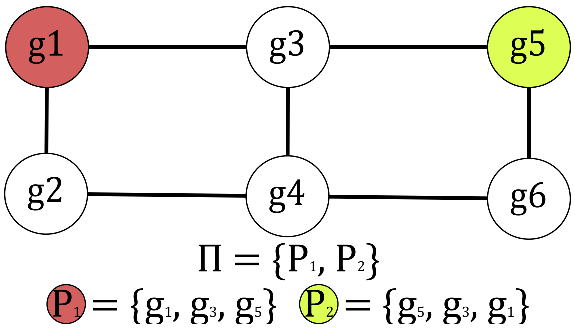


(b) Path execution at t_1

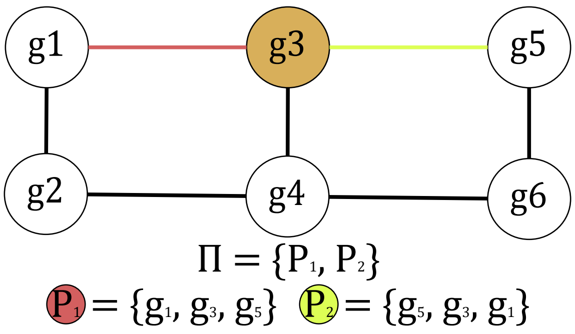


(c) Path execution at t_2

Figure 3.2 – Example of a step-by-step execution of a successful MAPF plan with two agents and no conflicts. Agent 1 is shown in red, and Agent 2 is shown in green.

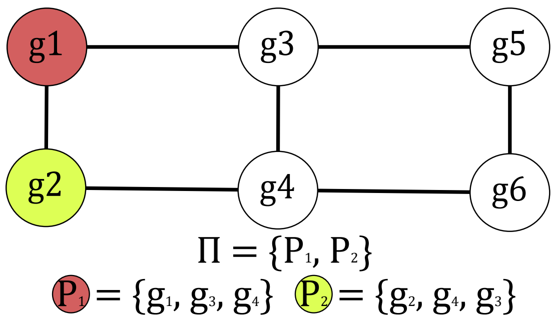


(a) Path execution at t_0

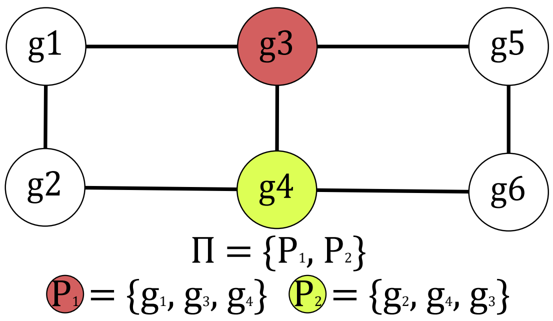


(b) Path execution at t_1

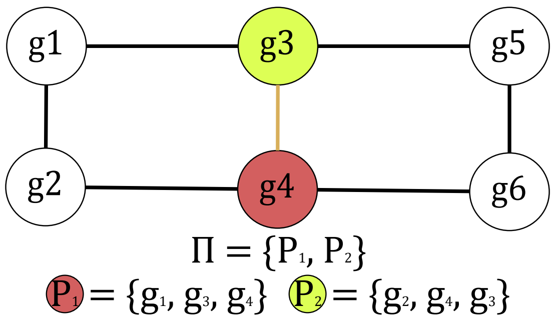
Figure 3.3 – Example of the occurrence of a vertex conflict between the two agents trying to occupy the same vertex $g3$ at time t_1. Agent 1 is shown in red, and Agent 2 is shown in green. The conflict is shown in orange.



(a) Path execution at t_0



(b) Path execution at t_1



(c) Path execution at t_2

Figure 3.4 – Example of the occurrence of an edge conflict between the two agents at time t_2, where both are trying to traverse the same edge between $g3$ and $g4$ at the same time step. Agent 1 is shown in red, and Agent 2 is shown in green. The edge conflict is shown in orange.

3.2 MAPF Generalizations

There are several generalizations of MAPF, each of them tailored to fit a different scenario (MA, 2022). In an automated warehouse environment, for example, robots receive multiple orders during a time period, meaning they don't have a single start and goal location. This is called Lifelong MAPF, solved by algorithms such as Rolling-Horizon Collision Resolution (RHCR) (LI et al., 2021). It uses a Bounded-Horizon Planning concept, in which the user defines two different parameters: a time horizon and a replanning period. The algorithm generates collision-free plans during the time horizon; after that, it assumes that every agent follows the shortest path to their goal locations. After reaching the replanning period threshold, the algorithm revisits the paths to solve the occurred collisions. Even though RHCR is not guaranteed complete or optimal, it achieved very good empirical results with high scalability. TP-SIPPwRT [1] (MA et al., 2019) also solves lifelong instances on a slightly different scenario called Multi Agent Pickup and Delivery (MAPD) (MA; KOENIG, 2017). The MAPD problem is adapted to scenarios like automated warehouses or delivery fleets. In this generalization, instead of setting goals for

[1]Abbreviation of "Token Passing Safe-Interval Path Planning with Reservation Table" algorithm.

each agent, there are tasks that can be inserted into the system at any time. Each task has a start and a finish position, which are the pickup and the delivery cells respectively. Agents that are not currently assigned or executing a task are deemed as free, meaning that they are available for unassigned tasks in the set. Instances are considered solved if and only if all the tasks in the set have been executed in a bounded amount of time after they've been inserted into the system. TP-SIPPwRT uses the Token Passing algorithm (TP), which has a token storing the paths for all current agents and the task set. The token is passed to the next free agent, who makes a decision based on different factors and passes it to the next available agent. Alongside that, it uses the Safe-Interval Path Planning (SIPP) algorithm, which will be explained further in chapter 4, with Reservation Tables (also part of our proposed algorithm, and explained in more detail in section 5.4). It was proved to be an efficient and effective algorithm for the MAPD problem.

Some generalizations reduce the simplifications assumed in the original MAPF to produce more realistic plans. The Any-Angle SIPP (AA-SIPP) (YAKOVLEV; ANDREYCHUK, 2017) algorithm considers agents with different sizes that can occupy multiple vertices simultaneously in the same time step. Any-angle approaches allow agent movement into arbitrary directions instead of cardinal positions. Each move is then calculated as a line segment, whose ends are tied to either the center or corner of the origin and destination cells. AA-SIPP uses a decoupled and prioritized strategy, assigning a unique priority for each agent. Then, each agent plans in their priority order, each subsequent agent considering the paths of their predecessors as dynamic obstacles. Through experimenting, AA-SIPP has led to solutions with lower costs than optimal cardinal-only algorithms. It was later tested in differential-drive robots (YAKOVLEV; ANDREYCHUK; VOROBYEV, 2019), which required some tuning due to a lack of accuracy when executing the plans in a live scenario.

Research for algorithms that incorporate kinematic constraints, considering variations in acceleration and velocity, is also relevant. In Hoenig et al. (2017), it is pointed out that there are limitations when naively executing plans from standard discrete MAPF solvers. To deal with that, they present the MAPF-POST algorithm, which post-processes a MAPF plan and creates an execution with a guaranteed safety distance between robots. It takes into account their maximum rotational and translational speeds to generate an efficient execution. Successful post-processing is made possible by using a Temporal Plan Graph, a directed acyclic graph where edges between vertices indicate a time precedence. Plan adaptation is made by guaranteeing that each agent enters its location in the order given by the MAPF plan, and that the order in which each agent occupies the same vertex is also respected. Kinematic constraints are embedded by using a Simple Temporal Network (STN). STNs are directed acyclic graphs where vertices are connected through special edges. Each edge contains a lower bound (LB) and upper bound (UB), representing that an event must be scheduled between LB and UB time units before the event in the next vertex. The adaptation of the plan execution can be made in polynomial time. Actions are adapted from the MAPF solver, transforming up, down, left, and right actions into a forward plus a rotation amount, in increments of 90° relative to the robot's current orientation.

Discrete-time steps and simultaneous movement are also an impactful simplification in real-world situations. From this stems a generalization called MAPF$_R$, as previously defined in Walker, Sturtevant and Felner (2018), 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 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. A visual representation of a MAPF$_R$ environment is shown in figure 3.5. In this case, the usual edges are replaced by the distance between the center points of adjacent grid cells with distance being a $d \in \mathbb{R}$. The agent is represented by a disk with radius $r \in \mathbb{R}$. In this example, we assume that the agent moves in four directions (up, down, left, right) in its 2^2 neighborhood. Due to this, all edges would have the same value, but that is not always the case. There are instances, including the one in the algorithm presented in this dissertation, where the agent has a larger degree of freedom, resulting in edges with varying costs.

Scalable solutions were found for planning on roadmaps with continuous time (KASAURA; NISHIMURA; YONETANI, 2022). In real-world situations, robots' execution time of their plans might vary due to a multitude of reasons, such as malfunctions, drifting, or stuttering. Algorithms such as the one presented by Honig et al. (2019), focusing on warehouse scenarios, try to account for this by planning robust plans. They

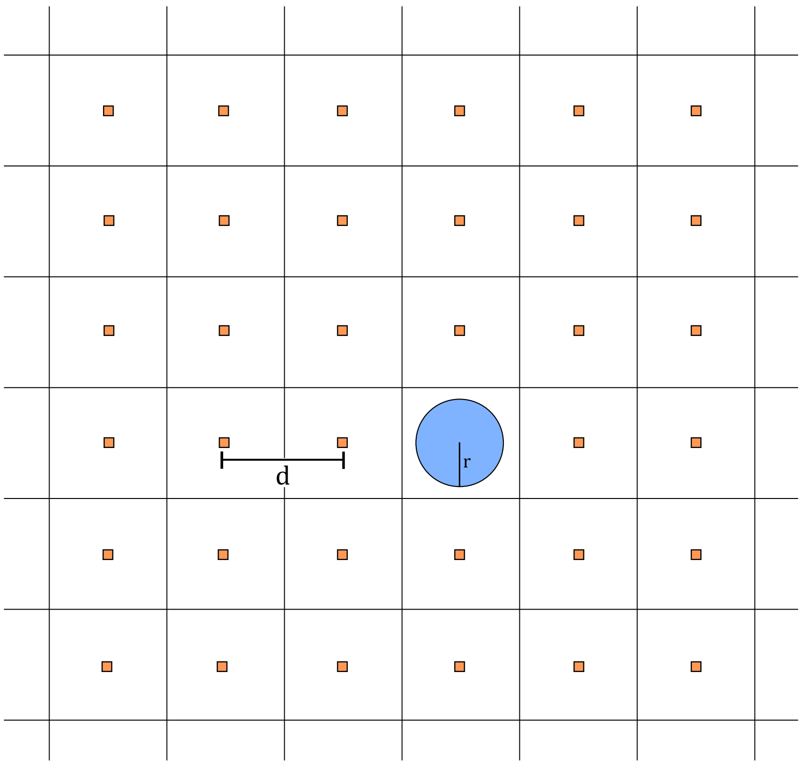


Figure 3.5 – Representation of a MAPF$_R$ grid. The agent is represented by a disk with radius r and moves between centers of adjacent cells with distance d. Obstacles were omitted, but they could occupy a whole cell, or, in some cases, only a part of it.

produce paths that remain collision-free despite unforeseen errors during the robots' execution. They start by addressing the fact that path planning algorithms are, in several cases, single-shot, meaning that they just get to their goal position from their start position and stay there, as in Ma et al. (2019). They manage to achieve efficient performance where (re)-planning and execution happen simultaneously through the use of an Action Dependency Graph (ADG) $\mathcal{G}_{ADG} = (\mathcal{V}_{ADG}, \mathcal{E}_{ADG})$ where $p^i_k \in \mathcal{E}_{ADG}$ and p^i_k refers to the kth tuple in plan p^i. Edges in the ADG represent inter-action dependencies. If $(p^i_k, p^{i'}_{k'}) \in \mathcal{E}_{ADG}$, then a robot is only allowed to start executing $a^{i'}_{k'}$ after a^i_k has been completed. This graph is created by post-processing the plan, transforming all actions into vertices based on all p^i_k and connecting them to subsequent actions through so-called Type 1 edges. These edges establish a progressive sequence for a single robot. Then, dependencies are found between different robots, indicating temporal precedence between actions (so-called Type 2 edges).[2] By using this method, it looks to overcome the fact that perfect synchronous execution is a property that is very hard to obtain in practical robots. The use of ADG guarantees during execution that a robot will only move into a location after the previous one has completely moved out of it, allowing the system to

2

track only finished actions instead of robots' positions. Their system also implemented path re-planning in case of unknown obstacle detection during execution. Experiments were made using complete simulations and mixed reality with different types of dynamic obstacles, achieving good results.

4 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 highest 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 (SURYNEK, 2010; YU; LAVALLE, 2013). However, extensive research has produced algorithms that are able to solve instances with large amounts of agents in minutes of runtime (MA, 2022).

Standard solvers are separated into different types: decentralized solvers like Walk, Stop, Count, and Swap (WSCaS) are tailored to work with large amounts of robots (WANG; RUBENSTEIN, 2020). The main issue of working with large-scale swarms of robots is that keeping track of every path while trying to avoid collisions can be computationally prohibitive. Using a decentralized system can help reduce this computational bottleneck since it distributes the load between agents. The algorithm proposes that each robot, while being aware of its own goal position and path, maintains communication with other robots in their range. They use the same protocol to signal their intention of movement and, through that, identify possible conflicts and act accordingly to avoid them. Each robot broadcasts its messages at a set frequency and can only perform orthogonal movement. None of them has knowledge of the swarm and relies on communication to resolve deadlocks. WSCaS first uses A* to plan the individual paths for each agent. Then, agents start walking towards their goals while using local communication. After each step, agents evaluate their traffic conditions, dividing them into five different types, some of them requiring different resolutions, such as a stop-and-wait action from one agent or a coordinated swap between two or more agents. For the algorithm to work, the environments where the agents are acting must be swappable, meaning that they must be able to perform the actions shown in Figure 4.1. While less efficient than the baseline algorithm used for comparison, it guarantees completeness and collision avoidance in swappable environments. The algorithm also succeeded with a swarm of 100 real robots in a high-density environment.

On the other hand, centralized solutions can be divided between Reduction-Based, Rule-Based, and Search-Based (MA, 2022). Reduction-based algorithms 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. For example, in Han and Yu (2019), they present an Integer Programming (IP)

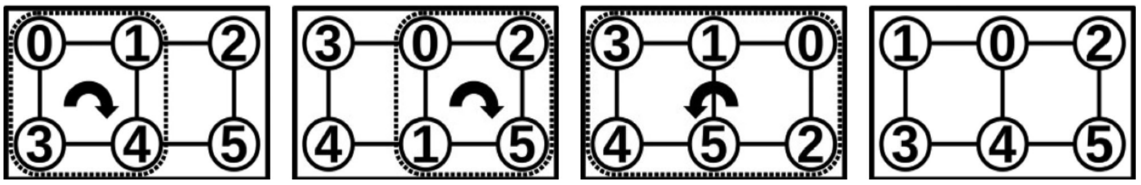


Figure 4.1 – Representation of a figure-8 swap, taken directly from Wang and Rubenstein (2020). Agent 1 and 0 are swapping their positions by taking the three steps depicted in the image.

methodology approach for challenging path-based problems. While polynomial-time algorithms can compute approximately optimal solutions, it does not always translate into optimality guarantees in real applications. Using greedy searches with heuristics is also an option, but sometimes, they might have issues when scaling to larger instances. Their methodology follows two steps: the first one is constructing an IP model that creates exclusively feasible paths. The second step is enforcing optimization criteria and additional constraints, e.g., collision avoidance in a multi-robot scenario. For the IP model, even though base graph encoding is mentioned and used, the time-expanded graph encoding was the method that could generate true collision-free graphs. Two problems were introduced for evaluation: the multi-robot minimum constraint removal, which is a generalization of the minimum constraint removal problem, and the multi-robot path planning with partial solutions, which is a generalization of the multi-robot path planning problem. When evaluated in multiple constraint removal problems, it was outperformed by exact and greedy search-based solvers, but heavily outperformed the same methods in the multi-robot minimum constraint removal. The reason for this is that single-robot environments are not “complex” enough to draw the complete advantages of the method, while the multi-robot scenario can draw its full potential. Overall, the method showed performance that was often competitive or better than other methods at the time, remaining simple to use.

Rule-based algorithms use a set of primitive operations to manage the robots. They usually can’t guarantee completeness for all problem classes, but can be very efficient. One notable example is the Push and Rotate algorithm (WILDE; MORS; WITTEVEEN, 2013), which can guarantee completeness in instances where there are at least two unoccupied locations in a connected graph. It is an adaptation of the Push and Swap algorithm (LUNA; BEKRIS, 2011), initially claimed to be complete for this class of instances, but later proved to fail to find a solution in certain types. The Push and Swap iteratively selects agents with an unspecified priority to move them to their respective goal locations. If an agent encounters another in its path, it can take two different actions.

When the encountered agent has lower priority, it gets pushed to another node towards their shortest path. In case the blocking agent has higher priority or is already in their goal location, the two agents perform a swap action, in which both of them move to the closest part of the graph where it is possible for them to switch positions. Any vertex with at least degree 3 is eligible for this operation. However, it falls short on graphs with only degree 2 vertices, not encountering possible valid solutions. Push and Rotate resolves this issue by accounting for graph sections that cause this, called *isthmuses* (WILDE; MORS; WITTEVEEN, 2013). The algorithm uses a three-stage approach to decompose this problem into subproblems, and by prioritizing the solution of those subproblems while making sure that higher priority agents move first, it manages to solve instances that Push and Swap was unable to. Push and Rotate achieved good results with empirical validation against similar solvers.

Search-based algorithms use heuristic search techniques to solve MAPF. They can be modified to achieve several different objectives. Notable examples of search-based algorithms are MAPF-LNS2 (LI et al., 2022) and PBS (MA et al., 2018).

MAPF-LNS2, i.e., MAPF with Large Neighborhood Search, is an algorithm that proposes using previous search efforts to improve future searches for the same MAPF instance to avoid memory-outs and time-outs, common occurrences in search and rule-based algorithms (LI et al., 2022). In most cases, failed solutions produced by algorithms are still making progress towards a feasible plan. The algorithm uses the Large Neighborhood Search (LNS) local search technique to improve the quality of the solutions. LNS destroys part of a solution and keeps the remaining paths fixed, iteratively replanning subsets of agents and keeping the best results. For efficiency purposes, MAPF-LNS2 needs a good single-agent path-finding algorithm. Even though Space-Time A* is mentioned and widely used, SIPPS, an improved version of the SIPP algorithm (PHILLIPS; LIKHACHEV, 2011), is presented, which is capable of handling soft obstacles. Another important factor for the algorithm's efficiency is the neighborhood selection. Three different selection methods are presented: Collision-based neighborhoods select a subset of agents that collide with each other. Failure-based neighborhoods focus on fixing reasons for failures by choosing a main agent and adding agents that collide with their path to the neighborhood subset. Random-based neighborhoods are a standard method that is very fast but doesn't always generate improvements. Another method, Adaptive LNS, records the relative success of different methods and uses the one with the highest probability for success in a given situation. Through various experiments, MAPF-LNS2 managed to solve 80% of the most challenging MAPF benchmark instances within the time limit of 5 minutes, significantly outperforming several state-of-the-art algorithms.

The Priority-Based Search (PBS) algorithm (MA et al., 2018) was presented considering that prioritized MAPF algorithms are usually very efficient when solving MAPF. Prioritized planners follow the simple principle that agents involved in the problem have determined unique degrees of priority between them. Agents plan their paths in order of priority, and lower-level agents must respect the higher-level agents' plans, treating their occupied cells as obstacles. This method can translate into fast execution times but has shortcomings, as some predetermined orders can produce bad results or even completely fail to find solutions in solvable instances. To avoid these limitations, PBS uses a systematic depth-first search to lazily explore the total of all priority orderings. Selecting priorities can be done in several different ways, for example, according to each agent's distance to the goal using heuristics, or by preferring certain types of paths. Doing this is the first level out of the two levels of the PBS algorithm, dynamically constructing priority orderings that result in a Priority Tree. The Priority Tree is used for backtracking in case no solution is found on the current branch. On the low level, the algorithm uses a mix between space-time A* and standard A* (HART; NILSSON; RAPHAEL, 1968). Due to the nature of the algorithm, there might be no path that reaches a target vertex, so space-time A* is useful to avoid deadlocking in situations that could have infinite constraints. For every other case, standard A* is used. As a contribution, the algorithm was presented with a theoretical framework that improved the discussion on prioritized planning.

Finally, one of the most popular MAPF algorithms and influential for developing our algorithm, the Conflict-Based Search (CBS) (SHARON et al., 2012), is explained in further detail in the next section.

4.1 Conflict Based Search

Conflict Based Search (CBS) is a two-level complete algorithm that provides optimal solutions for the classical MAPF problem (SHARON et al., 2012). When using A* to expand states in a regular MAPF environment, the size becomes exponential in relation to the number of agents. CBS overcomes this issue of having one problem with many agents into smaller single-agent pathfinding problems. These instances can be solved in a time proportional to the length of the map and solution; however, there can still be potentially an exponential size of those single-agent problems.

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. A conflict occurs when two agents are attempting to occupy the same edge or vertex at the same time step. 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.

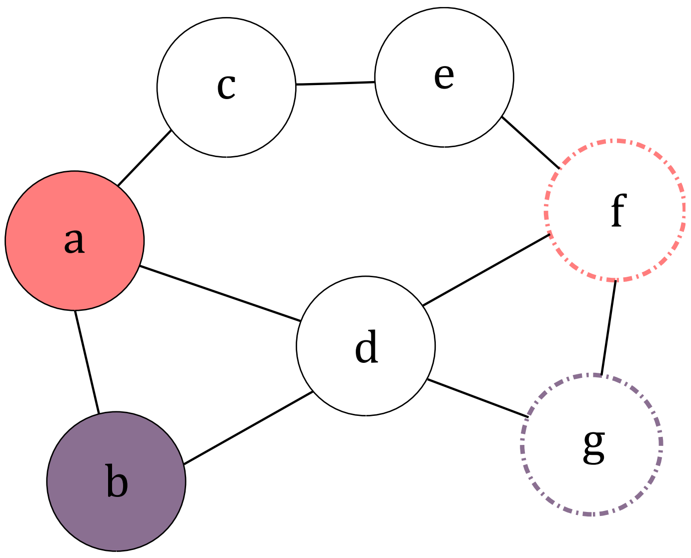


Figure 4.2 – A graph representing a MAPF instance, with both agents starting positions represented by the color-filled nodes and the goal nodes represented by the dash-stroked nodes in their respective colors.

We use the following example to illustrate the construction of the CT. In Figure 4.2, we show an example of a MAPF instance, where $agent_1$ starting at node a, represented in red, and $agent_2$ starting at node b, represented in purple, have to reach their goals. The goals are f and g, respectively, represented by the dash-stroked nodes in corresponding colors. With both agents individually planning for their goals and trying to find the shortest path, $agent_1$ finds the path $P_1 = (a, t_0), (d, t_1), (f, t_2)$ and $agent_2$ finds the path $P_2 = (b, t_0), (d, t_1), (g, t_2)$. This means that both agents want to occupy node d at time t_1, which is a conflict. After creating the starting node without any constraints, due to the detected conflict, new constraints need to be created to find collision-free paths. For the sake of completeness, the algorithm creates two nodes, one with the constraint $< agent_1, d, t_1 >$ and another node with the constraint $< agent_2, d, t_1 >$. Figure 4.3

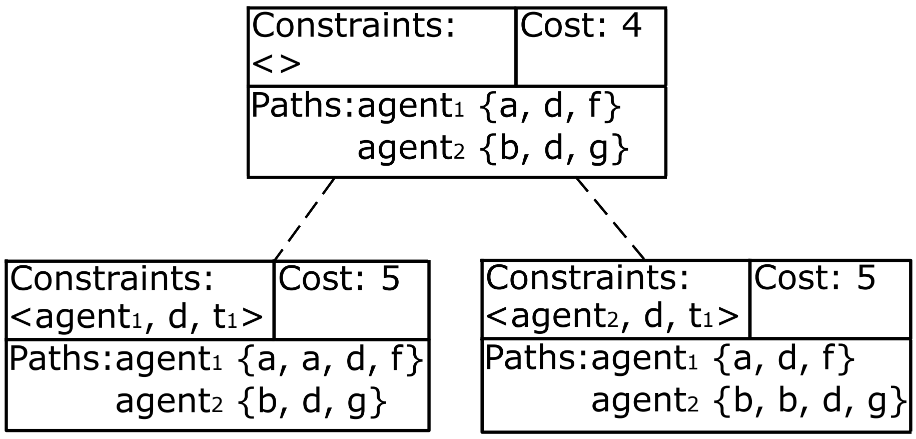


Figure 4.3 – An illustration of a Constraint Tree created through the process of solving the MAPF instance in Figure 4.2.

shows the CT created by the algorithm in this instance. Besides storing the constraints, the CT nodes also store information about the cost of the solution, which is used to select nodes for exploration and the paths for all agents in the instance. Since both nodes have the same cost, either one of them can be chosen for exploration. The agent involved in the constraint is replanned, having a new path that complies with the constraint in the node. It is important to note that constraints are always additive, meaning that if there were more conflicts generated through replanning of agents, created nodes would add the new constraint on top of the ones in the parent 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. In our example in Figure 4.3, either one of the nodes could be a goal node, since either $agent_1$ or $agent_2$ can wait in their starting nodes before moving to the d node, solving all conflicts. $Agent_1$ still has the same-cost alternative of going through nodes c and e, in case wait actions aren't allowed, as well as other different possible paths with equal cost.

CBS is optimal and complete; thus, it has become a very popular solver for MAPF instances. It was noted that CBS has a better performance in certain types of scenarios, outperforming A*. One of these scenarios is like the one where the shortest path for the agents goes through a bottleneck. CBS expands much fewer nodes in this case, and is also more efficient when expanding them, since CBS nodes have single-agent states and every A* node has multi-agent states. In scenarios that have a small open area where multiple agents have to go through, A* ends up outperforming CBS, since it ends up ruling out conflicted solutions fast. The empirical evaluation in Sharon et al. (2012) produced good results, outperforming different algorithms in various scenarios. 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 (CCBS) algorithm (ANDREYCHUK et al., 2019) changes the CBS algorithm to reduce this shortcoming. We further explain the algorithm in the following section.

4.2 Continuous time Conflict-Based Search

As an extension of the CBS algorithm to work with Continuous time, CCBS (ANDREYCHUK et al., 2019) is a complete and optimal algorithm designed to solve MAPF$_R$ problems. The system implements a collision-detection ability that manages to compute safe intervals for each agent more accurately by not discretizing time. However, considering agents' geometry and dealing with continuous time can make it slower than grid-based solutions.

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, collisions can't be trivially detected on a high level by checking the vertex disputed between two agents. Therefore, the conflicts in CCBS are defined between actions; formally, the conflict is defined as a tuple $\langle a_i, t_i, a_j, t_j \rangle$, meaning that if agent i executes action a_i at time t_i and agent j executes action a_j at time t_j, there will be a collision. Collision detection is a non-trivial problem, but there are algorithms that are efficient enough to be used in these situations. CCBS uses a fast closed-loop collision detection mechanism (GUY; KARAMOUZAS, 2015), but it is agnostic regarding the type of collision detection used. Conflict resolution is similar to the one done in CBS. A node expansion in CCBS will select the best node in the CT, and if it is not a goal node, it will 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 maximal time interval where an agent performs an action a_i at time t_i and will cause a collision with agent a_j at time 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) (PHILLIPS; LIKHACHEV, 2011), 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 Andreychuk et al. (2019) modified it to handle CCBS constraints. Let $\langle i, a_i, [t_i, t_i^u) \rangle$ be a constraint in CCBS over agent i. The adaptation was made by distinguishing between two different cases:

- a_i is a move action, and the pair v, v' are the agent's source and target destination: If the agent arrives to v in step $t \in [t_i, t_i^u)$ the action that moves it from v to v' at time t is removed, adding in its place an action that represents waiting at v until time t_i^u and then moving to v'.
- a_i is a wait action and v is the vertex where agent i is waiting: The agent is forbidden to wait at v in the range $[t_i, t_i^u)$ by splitting the safe intervals of v accordingly. For example, if v is associated with a single safe interval $[0, \infty)$, then split it into two intervals $[0, t_i]$ and $[t_i^u, \infty)$.

The algorithm was also proved to be sound and complete, with full proof available at Andreychuk et al. (2019).

4.3 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 Atzmon et al. (2021). Unlike the traditional MAPF, k-Robust MAPF (or kR-MAPF) plans are only considered valid if conflict-free and do not contain k-delay conflicts. A k-delay conflict between two agents occurs if and only if a $\Delta \in [0, k]$ exists 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 can still follow their plan if they suffer a delay that is lower or equal to k. It is noted that the standard MAPF is a case of kR-MAPF with $k = 0$.

The k-Robust CBS (kR-CBS) (ATZMON et al., 2021) was proposed as a solution for kR-MAPF problems. One of the main differences from the standard CBS algorithm is that kR-CBS must identify k-Robust conflicts. After having a consistent plan through the low-level solver, the algorithm must scan the CT node for conflicts, as it happens in standard CBS. Finding k-delay conflicts means identifying conflicts between two agents $\langle a_i, a_j, t\rangle$ in a CT node N, with a set of paths $N.\pi$ where $N.\pi_i(t) = N.\pi_j(t + \Delta)$ for $\Delta \in [0, k]$. Implementing this additional check isn't hard, but has a runtime larger by a factor of k when compared to the original CBS. Regarding the solution of those conflicts, considering the case above, there must be a vertex v and a value $\Delta \in [0, k]$ such that $v = N.\pi_i(t) = N.\pi_j(t + \Delta)$. From this, it follows that two constraints that solve this conflict are a possible fit, $\langle a_i, v, t\rangle$ and $\langle a_j, v, t + \Delta\rangle$. Thus, CT spawns two child nodes, each with one of those constraints.

kR-CBS was proved to be sound, since it doesn't halt until it finds a CT node without k-delay conflicts. It was also proved to be complete, because the splitting of the CT does not lose valid plans. Complete proof is available at Atzmon et al. (2021).

A more efficient version (in the sense that can find a goal node sooner) was also proposed in the same paper, Improved k-Robust CBS (IkR-CBS). The improvement comes from the understanding that imposing stricter constraints can reduce the size of the CT. To do this, IkR-CBS generates range constraints for its successors, defined by the tuple $\langle a_i, v, [t_1, t_2]\rangle$, meaning that agent i must avoid vertex v from timestep t_1 to t_2.

Optimality coming from CCBS, albeit desirable, leaves no room for deviation or error, things that are very likely to occur with real agents. With this in mind, we propose in chapter 5 a new algorithm that has the useful features of planning with continuous time, and also deals with a certain degree of uncertainties by having robust features.

5 K-ROBUST CCBS

Our algorithm modifies the CCBS algorithm to generate plans that are robust to k delays. Although our goal is to generate plans that remain safely executable in the occurrence of unforeseen events, we still want to maintain effectiveness. Satisfying the k-Robust condition of planned paths requires considering new types of conflicts that do 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 time. This prohibits the agents from performing certain patterns, such as train-like motions in close-quarters. Agent's aren't completely 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. Coming back to the afterimage visualization, the distance between agents would be as large as the afterimage of the robot in front would last in the path. These distances are also subject to change during execution, this being exactly one of the reasons for planning with the k-Robust feature, since in a standard scenario, where a stutter or unforseen delay from the robot in front would very likely cause a collision, in the case of our algorithm, the robot behind would close the gap between them (as long as the delay affects the robot for a shorter amount of time than the k value set).

Figure 5.1 shows an example of the new type of conflict. Agent a_1 is in position $(0, 0)$ at time 1, and wants to move to position $(0, 1)$, where agent a_2 is located. Agent a_2 moves down to an empty location at $(1, 1)$. In this case, with a k-value equal to 2, agent a_1 occupying position $(0, 1)$ at time 2 is considered a k-conflict. Due to the presence of a_2 in $(0, 1)$ at $t = 1$, a_1 would only be able to safely occupy $(0, 1)$ at time $t = a_{2time} + k_{value} = 3$.

5.1 Collision detection

In our strategy, we use the closed-loop collision detection proposed by Guy and Karamouzas (2015) but extend it by adding another layer of collision checking for k-Robust conflicts. Avoiding collision between agents is related to predicting whether they will collide according to current movement. The agents are assumed to be disc-shaped, and a collision happens when two discs intersect. A time to collision (denoted τ) is used to reason over interactions. Specifically, a collision is said to occur in a time $\tau \geq 0$, if

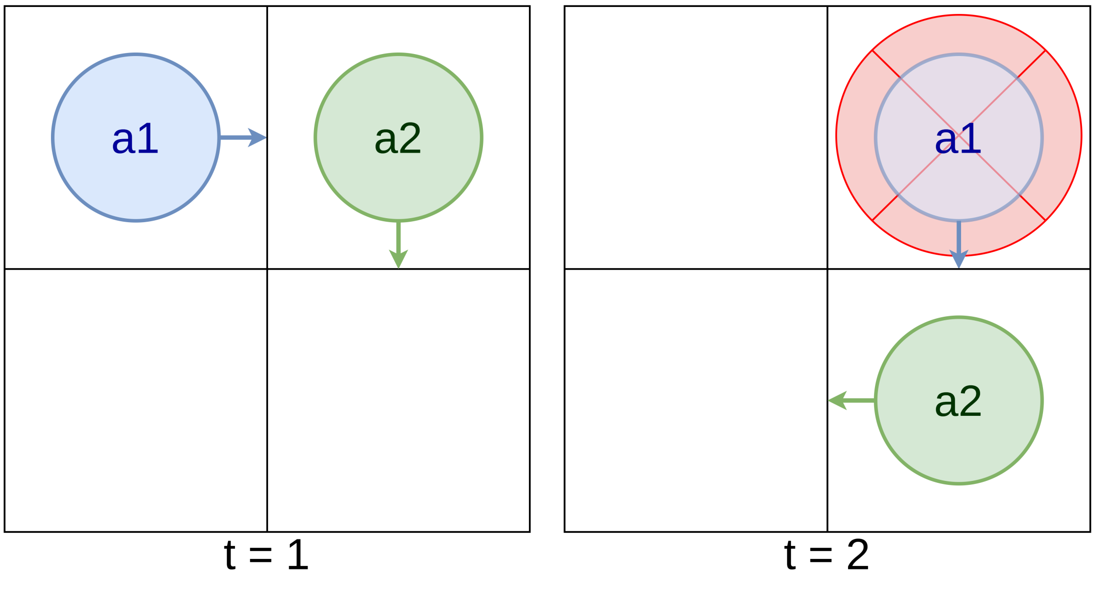


Figure 5.1 – Example of a k-conflict occurrence, using discrete values for simplicity, in a configuration with a $k = 2$. (a) At time 1, a_1 is at position $(0, 0)$ and a_2 is at position $(0, 1)$. (b) At time 2, a_1 is at position $(0, 1)$, which is a k-conflict, and a_2 is at position $(1, 1)$.

the two discs of the agents intersect. To estimate τ, we extrapolate the agents' trajectories based on their current velocities. Then, the problem is simplified into computing the distance between the two extrapolated trajectories and comparing them against the sum of the radii of the agents.

Formally, as presented in Guy and Karamouzas (2015), a collision between agents A and B exists if:

$$\| (x_B + v_B\tau) - (x_A + v_A\tau) \| = r_A + r_B \ , \tag{5.1}$$

where x_A and x_B are position vectors and v_A and v_B are velocity vectors. To estimate the extrapolated positions of the agents it is assumed constant velocities. In practice, it works very well for avoiding upcoming collisions, especially in the short run, although the assumption doesn't hold in all cases.

Squaring and expanding equation 5.1 leads to the following quadratic equation for τ:

$$(v \cdot v)\tau^2 + 2(w \cdot v)\tau + w \cdot w - (r_A + r_B)^2 = 0 \tag{5.2}$$

where $w = x_B - x_A$ and $v = v_B - v_A$. For ease of notation, let $a = v \cdot v, b = 2(w \cdot v)$ and $c = w \cdot w - (r_A + r_B)^2$. Then, the equation can be solved following the quadratic formula, $\tau^{\pm} = (-b \pm \sqrt{b^2 - 4ac})/(2a)$, estimating the possible time for collision between the two agents. Three different situations can happen regarding the solutions of the equation:

1. If there is no solution ($b^2 < ac$), or only one (double) solution ($b^2 = ac$), or if both solutions are negative, then no collision takes place, and τ is undefined.
2. If one solution is negative and the other is nonnegative, then the agents are currently colliding and $\tau = 0$.
3. If both solutions are nonnnegative, then a collision occurs at $\tau = min(\tau^+, \tau^-)$.

In practice, it is not necessary to account for all cases. Assuming that the agents are not currently colliding, it is sufficient to test if τ^- is nonnegative. Otherwise, τ is undefined. Alongside this collision detection algorithm, we also perform k-Robust collision checks, which are slightly different. It is possible to take advantage of the trajectory detection algorithm to save some extra processing because if their trajectories overlap, it also means that the two agents will occupy the same coordinate in the grid. If $k \geq 0$, we can see that this would be considered a k-conflict as well. We detail in 5.2 how this is adapted and turned into constraints.

Besides that, assuming that the agents are not colliding, we still need to check if they're occupying the same position in the grid, and if they are, we check if the time difference between them is safe according to the k parameter. As previously shown in the example illustrated by Figure 5.1, when a situation like this is detected, a conflict is assigned where either agent a_1 cannot perform their action arriving at $(0, 1)$ at time 2, or agent a_2 cannot be occupying position $(0, 1)$ at time 1. There are exceptions regarding how conflicts are handled, since sometimes it can be impossible for one of the agents to comply with the restriction. One example is a k-Robust conflict in one of the agents' starting positions in a time that is lower than the time it takes for the agent to move from their starting position to a different one. Since it is impossible for the agent to move any faster, the algorithm doesn't consider this type of conflict.

5.2 Conflict resolution

When a collision is detected, it is identified as a conflict and added to the list of conflicts in the set of paths. A conflict is defined as a tuple $\langle a_i, t_i, a_j, t_j \rangle$, meaning that if agent i performs action a_i at time t_i and agent j performs action a_j at time t_j, they are going to collide. Constraints are created based on the conflict, to be split into the two nodes from the Constraint Tree. They are determined by calculating the unsafe intervals for each action concerning the other action. The unsafe interval of action a_i regarding action a_j is defined as the maximum time interval starting from t_i during which executing a_i would result in a collision with the execution of a_j at time t_j. Then, one constraint is added to node N_i that prohibits agent i from performing action a_i in the unsafe interval related to a_j, and another constraint is added to node N_j that is similar, but with agent j with action a_j regarding the unsafe interval related to a_i. We also take advantage of the collision detection here by adding the k parameter value to the unsafe intervals. Since the interval is calculated as the time estimate for a collision to occur, we consider that as a time estimate for both agents occupying similar positions. Then, instead of the usual action interval $\langle t_{begin}, t_{end} \rangle$, we turn this interval into $\langle t_{begin} - k, t_{end} + k \rangle$. Detected k-Robust conflicts, unrelated to collisions, are dealt with and processed in the same way.

Even though 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' actions; thus, we limited the collision checks to be during the intervals when the agents are located in the middle of the cell when dealing with k-Robust conflicts. This has to be taken into 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.

5.3 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 until a solution is found.

We explain this more clearly by showing an example of a full algorithm run and how the constraint tree is formed considering the scenario shown in Figure 5.2. The step-by-step construction of the Constraint Tree is shown in Figure 5.3. Consider that we are using a gridmap, with coordinate $(0, 0)$ being on the top left of the grid, and coordinate

Algorithm 1: High-Level Algorithm

```
Initialization;
Create root node;
while there are nodes in tree do
    if there are no conflicts then
        return solution
    end if
    get first conflict;
    for each agent in conflict do
        create_constraint(agent);
        LowLevelAlgorithm(agent);
        Find new conflicts (regular and k-robust);
        creates node with paths;
        add node to Constraint Tree;
    end for
end while
return "No solution";
```

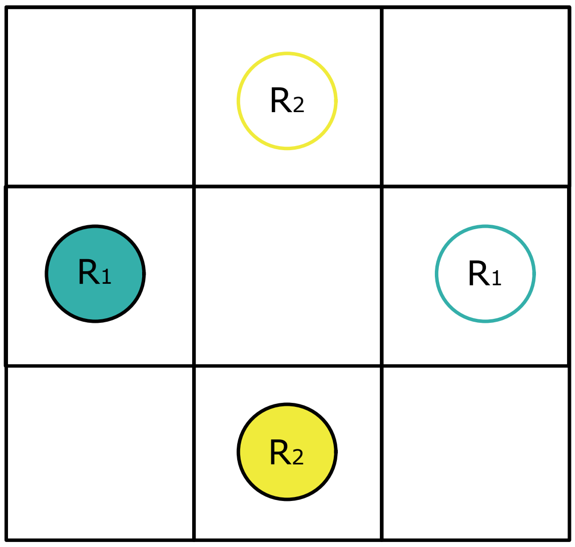
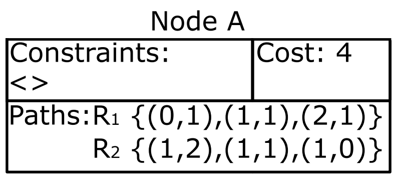


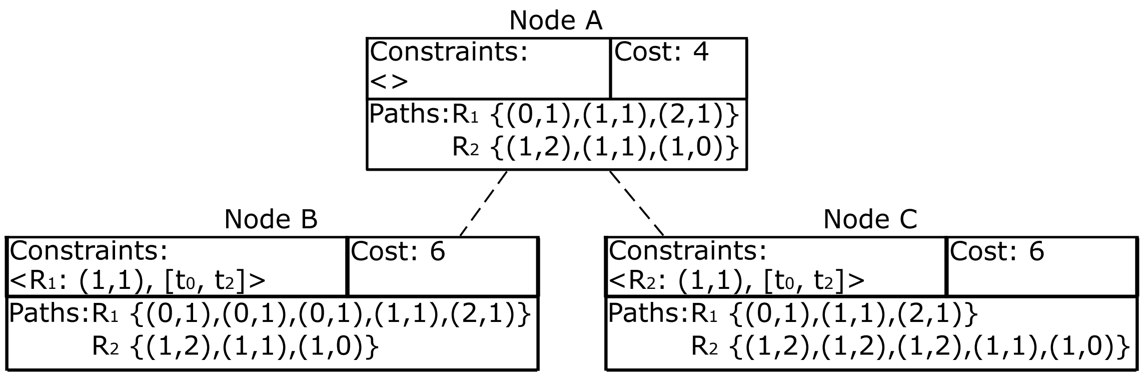
Figure 5.2 – Map of the scenario used to exemplify the workings of the k-Robust CCBS algorithm. The filled circles represent the starting positions for both robots, and the border-colored circles represent their goal positions, in their respective colors.

$(2, 2)$ being on the bottom right of the grid. Robot R_1 is in starting position $(0, 1)$ and its goal position is $(2, 1)$. Robot R_2 is in starting position $(1, 2)$ and its goal position is $(1, 0)$. In this scenario, the robots can move in a 2^2-neighborhood (or 4-neighborhood), meaning that they can only move up, down, left, or right. The robust parameter is defined as $k = 1$.

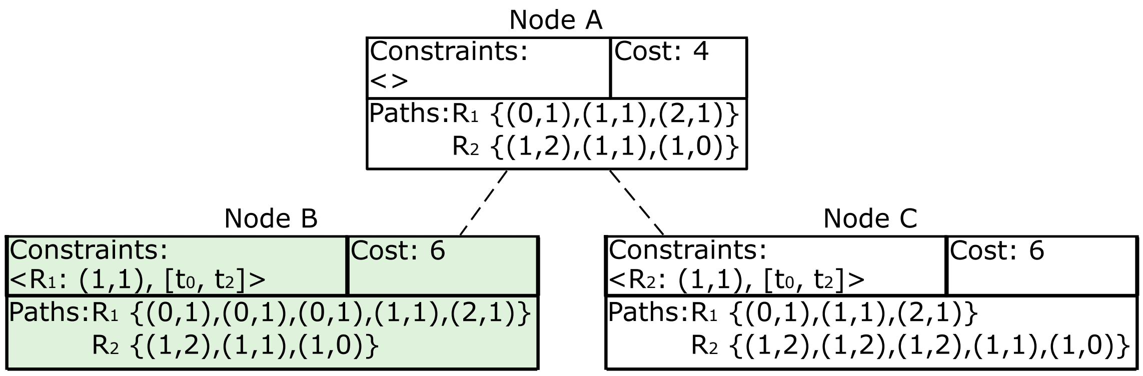
The algorithm receives all the data regarding the map, starting and goal positions, and initializes all variables. Then, it builds the root node A of the Constraint Tree. Since it is the first time that paths are being planned and there aren't any known constraints,



(a) Root node of the constraint tree.



(b) Constraint tree expanding the root node after detecting a conflict, splitting it into two other nodes.



(c) After replanning, the algorithm checks that there are no new conflicts and considers Node B a goal node.

Figure 5.3 – Constraint Tree for our running example.

the high-level algorithm runs the low-level planner for each robot as if they were alone in the map, finding the shortest path. On Figure 5.3(a), we can see that the low-level algorithm found paths for both robots, $P_{R_1} = \{\langle(0,1), t_0\rangle, \langle(1,1), t_1\rangle, \langle(2,1), t_2\rangle\}$ and $P_{R_2} = \{\langle(1,2), t_0\rangle, \langle(1,1), t_1\rangle, \langle(1,0), t_2\rangle\}$, with a total cost of 4. The cost is calculated as the sum of the time taken for all robots to reach their destination. To simplify it, we use discrete time for this example, but the algorithm supports continuous-time units. We do this by determining a cell size and a traveling speed regarding the robot's radius. The cell size has to be slightly larger than the robot to allow robots to move or stand side by side without immediately touching. The cost for each move is calculated by the time taken from the robot to go from the center of the starting cell to the center of the destination cell. We assume constant speed and disregard acceleration.

The algorithm then performs a conflict check for the paths in Node A, finding one that occurs in position $(1,1)$. Then, this conflict is processed and transformed into a constraint involving the two robots. Now, Node A is split into two nodes, B and C. Figure 5.3(b) shows the information contained in the Constraint Tree at this point in the algorithm. Replanning occurs only for the agent involved in the current conflict, so Node B replans the path for robot R_1. Node B receives the constraint for robot R_1, which prevents him from occupying position $(1,1)$ from time interval $[t_0, t_2]$. Note that this time interval comes from the collision detection, but with the added k parameter to the safe interval. The unsafe interval would originally be $[t_0, t_1]$, since this is the time taken by the agent to perform the move that causes a collision, but we transform this interval into $[t_0 - k, t_1 + k] = [t_0, t_2]$, making this constraint not only collision-safe but also k-Robust. Doing this avoids the creation of two extra nodes, since if we did not change the unsafe interval, replanning for agent R_1 would result in a path arriving at $(1,1)$ at the next earliest possible time, which would be t_2. However, during the next round of collision checks, this would be determined as a k-Robust conflict, creating two more nodes splitting from node B. Therefore, in this scenario, the two types of conflicts occur, but in this case, they are solved in the same conflict-check round of the algorithm. Node C receives the constraint for robot R_2, with similar parameters described for the previous node. After replanning, the algorithm checks for new conflicts in the node, and adds the node to the Constraint Tree. Next, the algorithm selects the node with the lowest cost for expansion, as shown in Figure 5.3(c). Node B is selected, and since there are no conflicts in the node, it is deemed as a goal node and the paths are returned as the solution for this instance.

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 function returns the first conflict found, which can be of either type.

5.4 Low-level solver

On the low level of our proposed algorithm, the SIPP structure originally used in CCBS (ANDREYCHUK et al., 2019) suffered adaptations to work within the framework of the k-robust CCBS algorithm. An overview of the Low-level solver is displayed in Algorithm 2. The Space-time A* algorithm searches through a pair (location, time), while, in the standard implementation, only being aware of the constraints originated from the high-level CT node. Now, the agent selected in the Constraint Tree node is set for replanning using SIPP while avoiding the conflicts detected by the high-level algorithm of the k-robust CCBS.

Algorithm 2: Low-Level Algorithm

```
make constraints for agent;
add initial position to open list;
while open not empty do
    if node is goal then
        return path;
    end if
    find successors using reservation table;
end while
return "No path";
```

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 is 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. In our approach, having the knowledge of the other agents' paths during replanning helps avoid extra conflicts, that would generate more nodes in the Constraint Tree down the line. To work with that efficiently, we incorporated a *reservation table* (MA et al., 2019) to our SIPP algorithm. A reservation table is a feature in Space-Time A*, 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 and avoiding the replanned path going through another agent's path.

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 into 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. We show in Figure 5.4 the reservation tables built in the example shown in the previous section. As we can see, since node A planned independently for each robot, the Reservation Table A is empty. For nodes B and C, a replan is made after knowing the path planned by the other robots. Therefore, in this instance, node B builds Reservation Table B with the safe intervals related to the path planned by robot R_2. As we can see, the reserved interval in coordinates $(1, 1)$ in this table was $[0, 2]$, which is the same interval deemed unsafe by the constraint in the node. Though the constraint intervals are added to the reservation tables when they are built, the $[0, 2]$ interval was added regardless of the constraint, since this was the path planned by robot R_2. Albeit redundant sometimes, reserving paths like this helps avoid creating new conflicts during replanning.

Reservation Table A

(0, 0)		(1, 0)		(2, 0)	
(0, 1)		(1, 1)		(2, 1)	
(0, 2)		(1, 2)		(2, 2)	

Reservation Table B

(0, 0)		(1, 0)	[1,3]	(2, 0)	
(0, 1)		(1, 1)	[0,2]	(2, 1)	
(0, 2)		(1, 2)	[0]	(2, 2)	

Reservation Table C

(0, 0)		(1, 0)		(2, 0)	
(0, 1)	[0]	(1, 1)	[0,2]	(2, 1)	[1,3]
(0, 2)		(1, 2)		(2, 2)	

Figure 5.4 – Information stored in the reservation tables for each node in the Constraint Tree for the previous example. Tables are named after their corresponding nodes. The coordinates are written as (x, y), and to the right of each coordinate is the time interval that is reserved for them as $[t_{begin}, t_{end}]$

The resulting algorithm produces plans with a slight increase in cost (shown in the next section), but with increased safety. Still, the algorithm is precise due to the avoidance of some abstractions that are part of the standard MAPF problem. In the next section, we present the experiments and results.

6 EXPERIMENTS

6.1 Configurations

In this section, we present the experiments conducted to compare our algorithm with the standard implementation of CCBS, as proposed in Andreychuk et al. (2021). Other algorithms have made progress in bringing abstract plans to simulations that have a higher fidelity to real-world applications. For example, MAPF-POST (HOENIG et al., 2017) is an algorithm that receives as input an abstract MAPF plan and post-processes it to form a plan that accounts for the kinematic movement of the robots. Though they both share the same objective, we choose not to compare our algorithm with MAPF-POST, since we consider this a different variation of the problem. MAPF-POST is independent of the solver, since it receives a generated plan as input; on the other hand, we prepare the path for execution during the planning stage, so comparing the two algorithms is something we could consider doing in the future, but for now we feel that they are different enough for us to omit comparison. The ICTS algorithm (WALKER; STURTEVANT; FELNER, 2018) works with non-unit costs and it is compared to CCBS in (ANDREYCHUK et al., 2019), so the comparison was omitted, since CCBS outperforms ICTS. Therefore, in this dissertation, we chose to compare the standard CCBS algorithm with our proposed k-Robust CCBS. We also considered comparing k-Robust CCBS to its discrete time counterpart, k-Robust CBS. However, since an implementation of the algorithm wasn't readily provided by the authors, we weren't able to produce one for our experiments.

To evaluate the performance of both algorithms, we performed multiple experiments in simulated scenarios and a few practical experiments with real robots. In the simulated experiments, we generated ten random configurations of start and goal positions for each different set of agents, with numbers ranging from 6 to 13. The map we used was an open 10x10 grid, with the diagonal of each cell being two times the diameter of the robot.

In the experiments, agents were allowed to perform a single move action to the center of every cell in their 2^3 neighborhood. We tested the algorithm with two different k-Robustness parameters, $k = 2$ and $k = 3$. Although the k values can be non-integer, we chose these values based on observations of the impact of incrementing the k-value when generating and executing plans. We found that these two parameters were sufficient to showcase the variation of the measured parameters and plan execution. 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.

For each algorithm, we set a time limit of 60 seconds 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 given to the robots, which executed them with minimal post-processing.

We used standard Turtlebot3 Burger robots from ROBOTIS (ROBOTIS, 2023) and conducted the simulations on Gazebo robot simulator. 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 we limited it to 0.2 m/s in our experiments. The maximum rotational speed is 2.84 rad/s, limited to 2.5 rad/s in our experiments.

To control the movement of the robots, we implemented a PID controller using the goal as the center of the next cell in the path. To ensure that the planned and simulated plans maintained fidelity, we only allowed the robots to accelerate once their angles were sufficiently aligned with the center of the goal cell. 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.

6.2 Real experiments

We had two Turtlebot3 burger robots available in our laboratory for practical experiments, the exact same model used in our Gazebo experiments. Since the number of robots is not large enough to reproduce similar scenarios as the ones presented in the previous section, we opted to create specific conditions that could replicate a case that would be very common in large spaces with multiple robots. Additionally, since the area of tests is small, we used the odometry of both robots for localization. Besides changing the precision threshold to check for a reached cell to 0.9cm, (which was 0.3cm in our simulations) due to a slightly lower accuracy in the real odometry, the code for execution was exactly the same as in the simulated experiments.

An abstract map was defined to create the conflict condition, as shown in Figure 6.1. This map has only one row of cells, meaning that the two robots have no option but to cross each others paths. There is also a single cell at the top row, meaning that to avoid collisions, one agent has to use that cell to maneuver around the other. Figure 6.2 shows the real robots in the practical scenario with virtual grid cells associated to the grid described above.

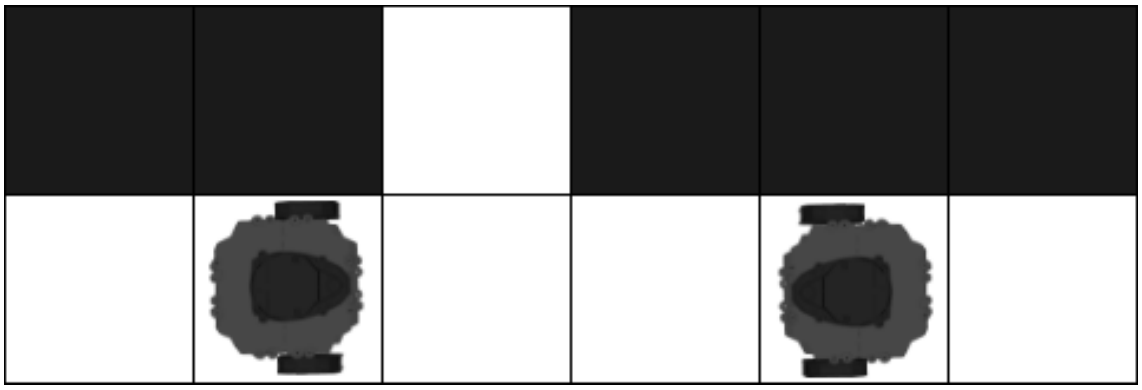


Figure 6.1 – Drawing of the grid describing the scenario used in the practical experiments.

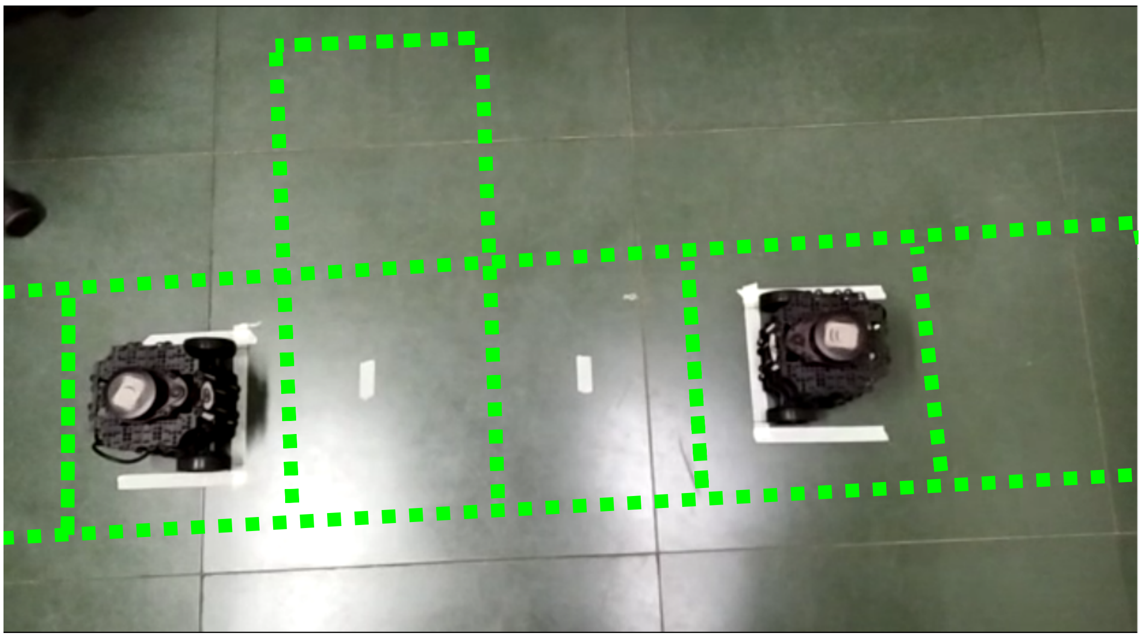


Figure 6.2 – Robots in the practical experiment scenario. Virtual grid cells highlighted in green.

Figure 6.3 shows screenshots taken from the videos of experiments in this scenario[1]. 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 Figure 6.4). The collision occurred at a point where 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

[1]Complete videos of experiments are available at
<https://www.youtube.com/playlist?list=PLAst3_ReOaDwqgDiNFzVL6f3Lh9IdbMEW>

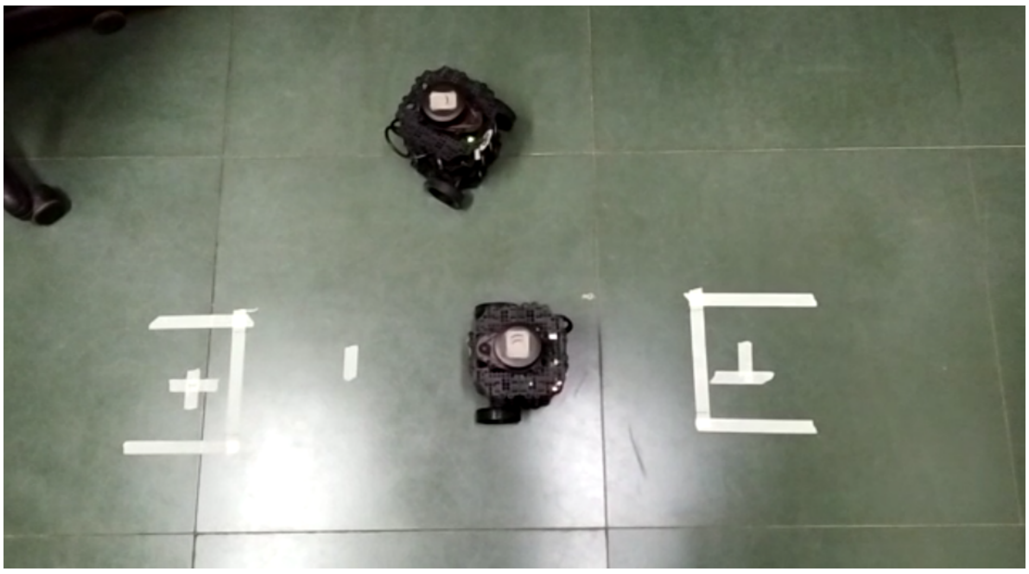
their goal position. This happened because, as previously stated, the generated plans of CCBS expect a perfect execution, which usually isn't the case in practice.

6.3 Simulated experiments

Aiming to further assess our method's scalability, we executed simulated experiments with a larger number of robots. We selected Gazebo as the simulation tool for our experiments because it uses the DART physics library, which accurately simulates rigid bodies (LEE et al., 2018). This means that the simulated robot's movements are subject to real-world factors such as friction, which may cause stuttering or different velocities on both wheels, resulting in unbalanced movement. Live robot executions are similarly affected, and by using this tool, we demonstrated that our algorithm can create safer plans that can be executed without collisions, even when executed with imperfect movement (to a certain degree). Figure 6.5 shows the start and goal configuration of a successful execution.

For our evaluation, we compared the success rate of executions and the planned flowtime and makespan values with the simulation values. Figure 6.6 shows the success rate 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 3. It it is also important to point out that CCBS achieved a 0% success rate with 12 robots, while still having a 25% success rate with 13 robots. While this might seem strange, since every problem instance was randomly generated, so it is entirely possible that in a sample of 10 different experiments, the instances with 12 robots were particularly unfavorable with the CCBS algorithm. We believe that if there were a larger sample of tests, the success rate would be larger than 0, but it wouldn't be much different than the success rate for 13 robots. 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.

Figure 6.7 compares the average flowtime values of the CCBS algorithm and the k-Robust CCBS algorithm. Figure 6.8 compares the average makespan values for the two algorithms. The *Planned* category in both Figures 6.7 and 6.8 represent the values measured in the abstract plan. The *Real* category in the same figures represent the values measured in the simulated executions. As expected, higher k values result in higher


(a) Middle point of the k-Robust CCBS algorithm.

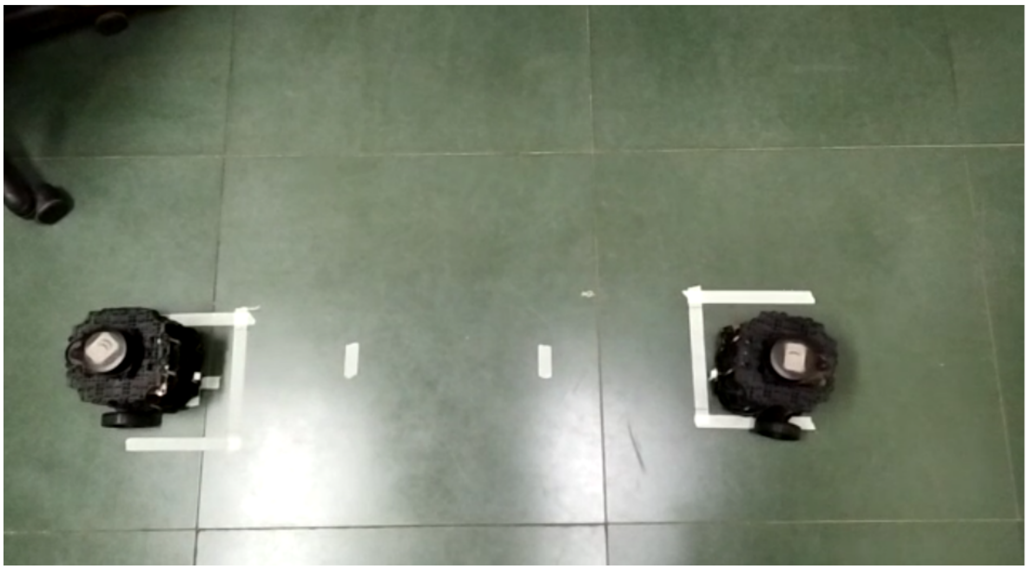

(b) After avoiding collision, robots safely reach their goals.

Figure 6.3 – Example of execution of the k-Robust CCBS algorithm, taken from videos of real-life executions with two Turtlebot3 robots. The top image highlights a point where one of the robots changes course to avoid a collision. Meanwhile, the other robot waits according to the determined k-robust parameter. The bottom image shows the final configuration.

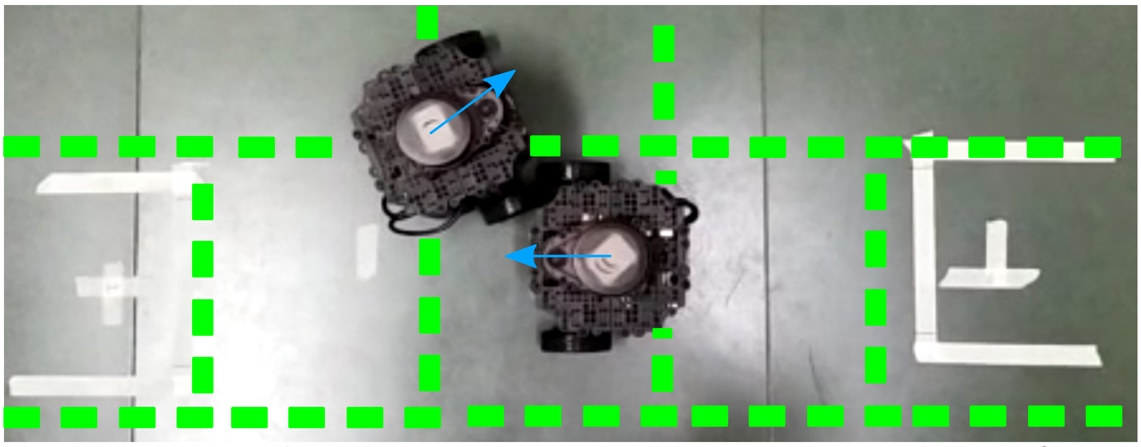


Figure 6.4 – Example of one of the experiments where we executed the plan without the k-robust parameter, causing a collision between the two robots. The grid is drawn to show an estimate of where in the grid the collision happened. The blue arrows above each robot shows the direction they were currently moving towards.

flowtime and makespan values, while the CCBS algorithm yields slightly lower values.

We further analyze the differences between the planned and executed times in the experiments by looking at Table 6.1. It shows the flowtime and makespan increase when executing the generated plans, as well as the average success rate for the two configurations of k-robust CCBS and for standard CCBS. When seeing the difference in values alongside the difference in success rate, we can see that there is compensation when using our algorithm.

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

Table 6.1 – 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 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 consistent, while the success rate for CCBS drops. This shows that the proposed algorithm is more robust than standard CCBS for an increased number

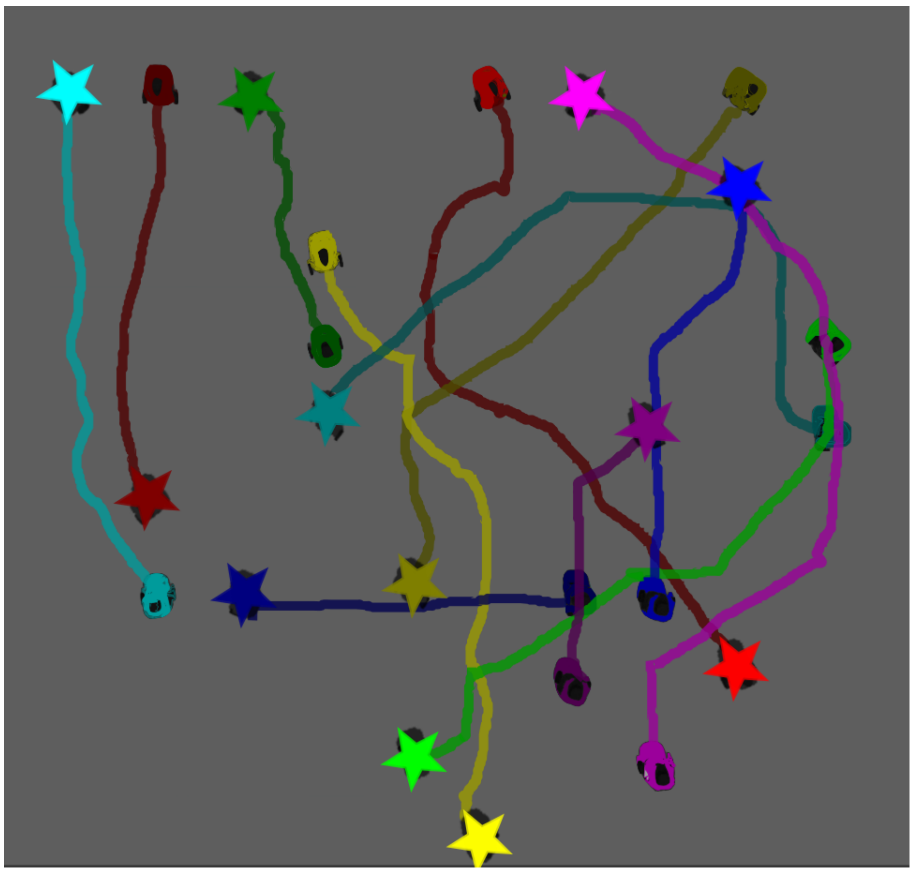


Figure 6.5 – Example of a successful execution, where robots reach their goal positions without collisions. Instance completed using k-robust CCBS with k-value = 2. Robots were colored for ease of visualization. The starting position is highlighted, with the path in the same color leading to the goal position. The goal position for each robot is represented by a star in its corresponding color.

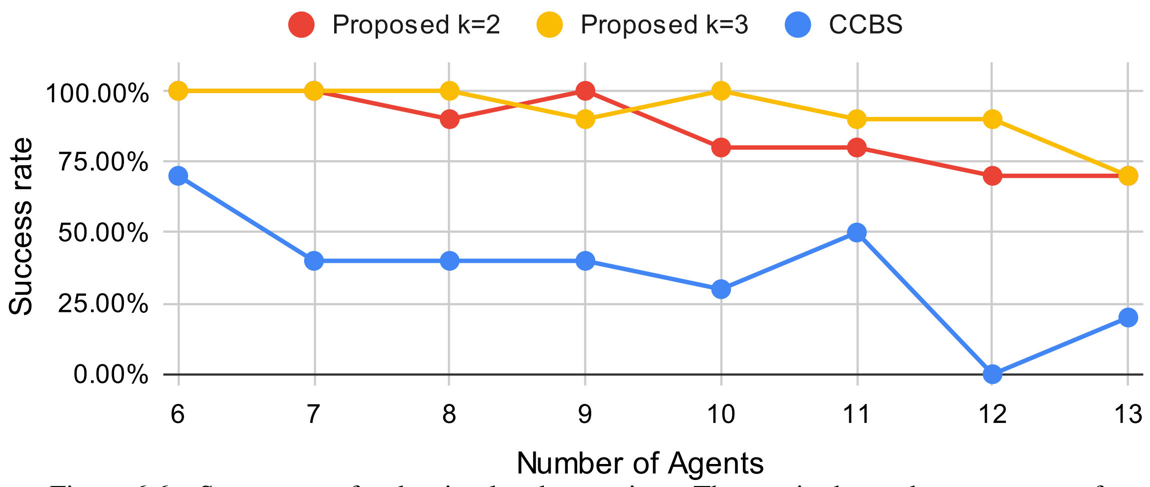


Figure 6.6 – Success rate for the simulated executions. The y axis shows the percentage of instances completed without collisions, while the x axis shows the number of agents on the executed instances.

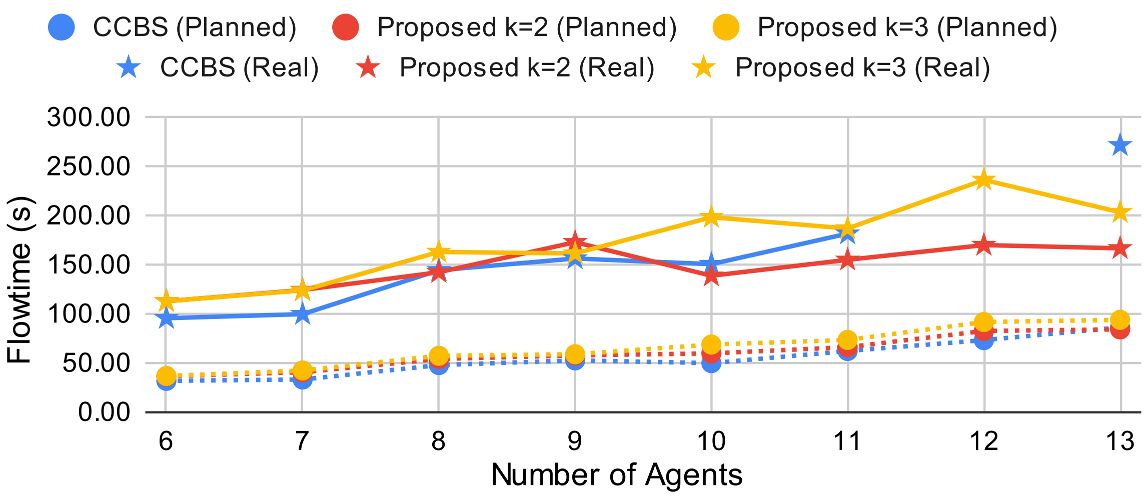


Figure 6.7 – Average flowtime values for the plans between all instances and for the flowtime measured after execution in Gazebo.

of agents.

Finally, for a more in-depth analysis of the data from the experiments, we present the complete times for robots in the simulation. To maintain brevity while being informative, we chose to present one completed instance for each different size of the set of robots. When possible, we presented an instance where execution happened without collisions for all three algorithms (CCBS, $k = 2$ k-RCCBS, $k = 3$ k-RCCBS). Each table shows the time each robot takes to reach its destination, in increasing order, with the last robot's time being highlighted as the makespan value for that run. Below that, we show the planned makespan and the planned flowtime, finishing with the flowtime value of the simulation. Tables 6.2 to 6.9 show the data for experiments done for the same instance. We note that in 6.8 and 6.9, we could not find an instance where all three algorithms were successful. With 12 agents, CCBS could not complete any instances, and with 13 agents, we chose an instance where CCBS could complete even though one of the k-RCCBS configurations could not, because we were more interested in highlighting the difference in times between algorithms.

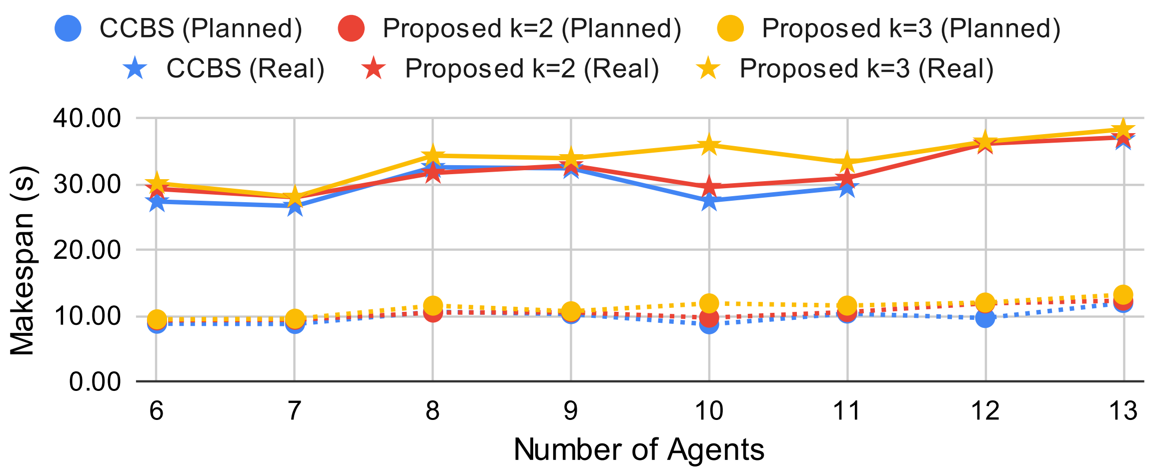


Figure 6.8 – Average makespan values for the plans between all instances and for the makespan measured after real execution in Gazebo.

	k = 2	k = 3	CCBS
Robot 1	11.064	11.064	10.322
Robot 2	11.865	11.865	11.105
Robot 3	16.006	16.006	17.438
Robot 4	18.796	18.796	19.032
Robot 5	20.586	20.586	20.623
Robot 6 (Makespan sim)	**27.561**	**27.561**	**27.189**
Makespan (plan)	9.899	9.899	9.899
Flowtime (plan)	35.799	35.799	35.799
Flowtime (sim)	105.878	105.878	105.709

Table 6.2 – Execution for an instance with 6 robots. All values are in seconds.

	$k = 2$	$k = 3$	CCBS
Robot 1	5.064	5.853	5.35
Robot 2	8.198	9.175	7.315
Robot 3	12.42	12.689	10.053
Robot 4	17.09	19.366	18.177
Robot 5	18.444	20.566	18.376
Robot 6	21.764	25.29	18.967
Robot 7 (Makespan sim)	**28.782**	**31.428**	**24.901**
Makespan (plan)	9.657	9.828	7.243
Flowtime (plan)	35.385	36.728	32.971
Flowtime (sim)	111.762	124.367	103.139

Table 6.3 – Execution for an instance with 7 robots. All values are in seconds.

	$k = 2$	$k = 3$	CCBS
Robot 1	9.75	10.752	11.142
Robot 2	10.868	11.328	12.684
Robot 3	15.627	21.151	12.877
Robot 4	18.62	23.702	14.438
Robot 5	31.115	28.781	26.055
Robot 6	31.26	35.552	27.797
Robot 7	31.927	39.902	28.953
Robot 8 (Makespan sim)	**35.667**	**43.083**	**33.225**
Makespan (plan)	11.485	13.657	11.485
Flowtime (plan)	59.698	73.870	56.941
Flowtime (sim)	184.834	214.251	167.171

Table 6.4 – Execution for an instance with 8 robots. All values are in seconds.

	$k = 2$	$k = 3$	CCBS
Robot 1	7.991	7.932	7.257
Robot 2	8.528	8.798	14.595
Robot 3	10.754	15.59	15.778
Robot 4	14.655	16.094	16.332
Robot 5	15.76	19.026	20.005
Robot 6	17.408	20.687	22.061
Robot 7	24.945	24.629	23.954
Robot 8	25.529	27.07	28.71
Robot 9 (Makespan sim)	**35.511**	**31.735**	**37.387**
Makespan (plan)	10.243	10.243	11.071
Flowtime (plan)	50.799	54.456	58.284
Flowtime (sim)	161.081	171.561	186.079

Table 6.5 – Execution for an instance with 9 robots. All values are in seconds.

	$k = 2$	$k = 3$	CCBS
Robot 1	2.927	2.801	2.187
Robot 2	5.805	5.131	5.341
Robot 3	7.172	7.295	8.081
Robot 4	8.689	8.493	8.65
Robot 5	11.116	10.93	10.144
Robot 6	15.578	15.101	14.236
Robot 7	19.117	19.435	15.365
Robot 8	19.809	20.67	18.735
Robot 9	26.436	32.599	19.892
Robot 10 (Makespan sim)	**28.178**	**35.047**	**26.405**
Makespan (plan)	9.485	10.071	9.485
Flowtime (plan)	47.799	49.385	45.456
Flowtime (sim)	144.827	157.502	129.036

Table 6.6 – Execution for an instance with 10 robots. All values are in seconds.

	$k = 2$	$k = 3$	CCBS
Robot 1	2.34	2.39	2.521
Robot 2	5.054	5.052	4.947
Robot 3	6.532	8.77	4.947
Robot 4	8.425	9.085	6.543
Robot 5	8.585	12.305	7.364
Robot 6	12.02	15.762	12.253
Robot 7	15.49	19.964	12.581
Robot 8	16.82	22.112	17.535
Robot 9	23.615	29.068	21.477
Robot 10	29.689	30.830	26.522
Robot 11 (Makespan sim)	**30.727**	**37.339**	**28.771**
Makespan (plan)	11.24	13.24	10.07
Flowtime (plan)	53.627	70.698	51.627
Flowtime (sim)	159.297	192.677	145.461

Table 6.7 – Execution for an instance with 11 robots. All values are in seconds.

	$k = 2$	$k = 3$	CCBS
Robot 1	2.389	2.416	-
Robot 2	5.749	5.326	-
Robot 3	19.39	15.113	-
Robot 4	21.677	16.745	-
Robot 5	23.078	18.529	-
Robot 6	23.366	24.885	-
Robot 7	24.868	28.247	-
Robot 8	25.475	30.207	-
Robot 9	27.265	34.79	-
Robot 10	27.430	35.301	-
Robot 11	28.536	38.641	-
Robot 12 (Makespan sim)	**37.467**	**40.370**	**-**
Makespan (plan)	13.657	14.657	-
Flowtime (plan)	90.113	97.698	-
Flowtime (sim)	266.690	290.570	-

Table 6.8 – Execution for an instance with 12 robots. All values are in seconds.

	$k = 2$	$k = 3$	CCBS
Robot 1	-	2.894	2.321
Robot 2	-	3.192	3.227
Robot 3	-	10.994	10.925
Robot 4	-	16.432	13.96
Robot 5	-	20.207	16.419
Robot 6	-	27.926	18.474
Robot 7	-	28.913	23.098
Robot 8	-	30.21	24.176
Robot 9	-	30.234	24.352
Robot 10	-	31.205	24.671
Robot 11	-	33.054	27.720
Robot 12	-	34.541	30.542
Robot 13 (Makespan sim)	**-**	**36.814**	**35.907**
Makespan (plan)	-	12.243	10.243
Flowtime (plan)	-	98.012	82.012
Flowtime (sim)	-	306.616	255.792

Table 6.9 – Execution for an instance with 13 robots. All values are in seconds.

7 CONCLUSION AND FUTURE WORK

In this dissertation, we presented k-Robust Continuous Conflict-Based Search, an algorithm for multi-agent path planning that plans with continuous time and implements a safety layer. We highlighted the benefits and reasoning behind using continuous time in a multi-agent scenario, especially when bringing the plans to high-fidelity simulations or real robots. Applying plans in real-life situations is not always seamless, sometimes resulting in collisions, so we justify using a safety layer during planning, the k-Robust concept.

Our approach, the k-robust CCBS, is an algorithm that implements a safety layer for multi-agent path planning with continuous time. We explained the general structure of the algorithm, highlighting how it deals with collision detection and conflict resolution, and described how the high and low-level solvers work together to solve MAPF problems. Through various experiments, we showed the effectiveness of the algorithm when executing the generated plans on physically accurate simulations and real robots on a smaller scale. Our results indicated that the cost increase in plans using k-robust CCBS, especially in a larger amount of agents, is a reasonable trade-off considering the higher rate of success of our executions.

For future work, we could improve the algorithm's scalability by adding landmarks and disjoint splitting, as in (ANDREYCHUK et al., 2021). There is more room for experiments with other algorithms to see how well it performs compared to them. Alternatively, there's also room for experimenting with hybrid solutions for the MAPF problem, combining pre-execution planning with online route correction. For example, adding an online rule for every robot during execution that stops when detecting another robot in its vicinity, and comparing this reactive solution with our algorithm, which is preventive regarding collisions. It is also worth experimenting how the algorithm would perform in situations where robots aren't allowed to perform wait actions, which sometimes can happen depending on the type of robot or environment. Checking how this could alter the algorithms performance and success rate might be a worthy analysis. Other experiments were run informally with different map configurations, with the addition of obstacles, but they weren't run enough to be able to gather relevant data. Therefore, it is also worth verifying in the future if the algorithm performs well in maps with different characteristics (i.e., choke points versus wide spaces). Adapting the algorithm to work on roadmaps could also generate more organic paths.

REFERENCES

ANDREYCHUK, A. et al. Multi-agent pathfinding with continuous time. In: **Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19**. [S.l.]: International Joint Conferences on Artificial Intelligence Organization, 2019. p. 39–45.

ANDREYCHUK, A. et al. **Improving Continuous-time Conflict Based Search**. [S.l.]: arXiv, 2021. ArXiv:2101.09723 [cs].

ATZMON, D. et al. Robust Multi-Agent Path Finding. **Proceedings of the International Symposium on Combinatorial Search**, v. 9, n. 1, p. 2–9, sep. 2021. ISSN 2832-9163, 2832-9171.

GUY, S.; KARAMOUZAS, I. Guide to anticipatory collision avoidance. In: ____. **Game AI Pro 2**. [S.l.]: CRC Press, 2015. p. 195–208. ISBN 9781482254792.

HAN, S. D.; YU, J. Integer programming as a general solution methodology for path-based optimization in robotics: Principles, best practices, and applications. In: **2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. [S.l.: s.n.], 2019. p. 1890–1897.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. **IEEE Transactions on Systems Science and Cybernetics**, v. 4, n. 2, p. 100–107, jul. 1968. ISSN 2168-2887. Conference Name: IEEE Transactions on Systems Science and Cybernetics. Available from Internet: <https://ieeexplore.ieee.org/document/4082128>.

HO, F. et al. Decentralized multi-agent path finding for uav traffic management. **Trans. Intell. Transport. Sys.**, IEEE Press, v. 23, n. 2, p. 997–1008, feb 2022. ISSN 1524-9050. Available from Internet: <https://doi.org/10.1109/TITS.2020.3019397>.

HOENIG, W. et al. Multi-Agent Path Finding with Kinematic Constraints. p. 9, 2017.

HONIG, W. et al. Persistent and Robust Execution of MAPF Schedules in Warehouses. **IEEE Robotics and Automation Letters**, v. 4, n. 2, p. 1125–1131, abr. 2019. ISSN 2377-3766, 2377-3774.

JORGE, V. A. M. **Enabling loop-closures and revisits in active SLAM techiniques by using dynamic boundary conditions an local potential distortions**. Thesis (PhD) — Universidade Federal do Rio Grande do Sul, 2017. Accepted: 2017-06-14T02:32:45Z. Available from Internet: <https://lume.ufrgs.br/handle/10183/159492>.

KASAURA, K.; NISHIMURA, M.; YONETANI, R. Prioritized Safe Interval Path Planning for Multi-Agent Pathfinding With Continuous Time on 2D Roadmaps. **IEEE Robotics and Automation Letters**, v. 7, n. 4, p. 10494–10501, oct. 2022. ISSN 2377-3766. Conference Name: IEEE Robotics and Automation Letters.

LAVALLE, S. M. **Planning Algorithms**. [S.l.]: Cambridge University Press, 2006. ISBN 0521862051.

LEE, J. et al. Dart: Dynamic animation and robotics toolkit. **J. Open Source Softw.**, v. 3, n. 22, p. 500, 2018. Available from Internet: <http://dblp.uni-trier.de/db/journals/jossw/jossw3.html#LeeGHKJYSSL18>.

LI, J. et al. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 36, n. 9, p. 10256–10265, jun. 2022. ISSN 2374-3468, 2159-5399.

LI, J. et al. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. **arXiv:2005.07371 [cs]**, mar. 2021. ArXiv: 2005.07371.

LUNA, R.; BEKRIS, K. E. Push and swap: Fast cooperative path-finding with completeness guarantees. In: **Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One**. [S.l.]: AAAI Press, 2011. (IJCAI'11), p. 294–300. ISBN 9781577355137.

MA, H. Graph-Based Multi-Robot Path Finding and Planning. **Current Robotics Reports**, v. 3, n. 3, p. 77–84, jun. 2022. ISSN 2662-4087. ArXiv:2206.11319 [cs].

MA, H. et al. Searching with Consistent Prioritization for Multi-Agent Path Finding. **arXiv:1812.06356 [cs]**, dec. 2018. ArXiv: 1812.06356.

MA, H. et al. Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 33, p. 7651–7658, jul. 2019. ISSN 2374-3468, 2159-5399.

MA, H.; KOENIG, S. AI Buzzwords Explained: Multi-Agent Path Finding (MAPF). **AI Matters**, v. 3, n. 3, p. 15–19, oct. 2017. ISSN 2372-3483. ArXiv:1710.03774 [cs].

MA, H. et al. Feasibility Study: Moving Non-Homogeneous Teams in Congested Video Game Environments. **arXiv:1710.01447 [cs]**, oct. 2017. ArXiv: 1710.01447.

PHILLIPS, M.; LIKHACHEV, M. SIPP: Safe interval path planning for dynamic environments. In: **2011 IEEE International Conference on Robotics and Automation**. Shanghai, China: IEEE, 2011. p. 5628–5635. ISBN 978-1-61284-386-5.

ROBOTIS. **ROBOTIS e-Manual, Turtlebot3 specifications**. 2023. Available from Internet: <https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications>.

SHARON, G. et al. Conflict-based search for optimal multi-agent path finding. In: FELNER, A. et al. (Ed.). **MAPF@AAAI**. [S.l.]: AAAI Press, 2012. (AAAI Workshops, WS-12-10). ISBN 978-1-57735-575-5.

STERN, R. et al. **Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks**. 2019.

SURYNEK, P. An Optimization Variant of Multi-Robot Path Planning Is Intractable. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 24, n. 1, p. 1261–1263, jul. 2010. ISSN 2374-3468, 2159-5399.

WALKER, T. T.; STURTEVANT, N. R.; FELNER, A. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. In: **Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence**. Stockholm, Sweden: International

Joint Conferences on Artificial Intelligence Organization, 2018. p. 534–540. ISBN 978-0-9992411-2-7.

WANG, H.; RUBENSTEIN, M. Walk, Stop, Count, and Swap: Decentralized Multi-Agent Path Finding With Theoretical Guarantees. **IEEE Robotics and Automation Letters**, v. 5, n. 2, p. 1119–1126, abr. 2020. ISSN 2377-3766, 2377-3774.

WEN, L.; LIU, Y.; LI, H. Cl-mapf: Multi-agent path finding for car-like robots with kinematic and spatiotemporal constraints. **Robotics and Autonomous Systems**, v. 150, p. 103997, 2022. ISSN 0921-8890. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0921889021002530>.

WILDE, B. de; MORS, A. W. ter; WITTEVEEN, C. Push and rotate: Cooperative multi-agent path planning. In: **Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems**. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2013. (AAMAS '13), p. 87–94. ISBN 9781450319935.

YAKOVLEV, K.; ANDREYCHUK, A. **Any-Angle Pathfinding for Multiple Agents Based on SIPP Algorithm**. [S.l.]: arXiv, 2017. ArXiv:1703.04159 [cs].

YAKOVLEV, K.; ANDREYCHUK, A.; VOROBYEV, V. Prioritized Multi-agent Path Finding for Differential Drive Robots. **2019 European Conference on Mobile Robots (ECMR)**, p. 1–6, sep. 2019. ArXiv: 1911.10578.

YU, J.; LAVALLE, S. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 27, n. 1, p. 1443–1449, jun. 2013. ISSN 2374-3468, 2159-5399.

APPENDIX A — RESUMO EM PORTUGUÊS DA DISSERTAÇÃO

No campo da ciência da computação e robótica, a coordenação de múltiplos robôs apresenta-se como um problema fundamental e complexo, importante no aumento de eficiência e funcionalidade em diversas aplicações na vida real. O núcleo deste problema consiste em descobrir caminhos livres de colisões para múltiplos agentes dentro de um ambiente compartilhado - um problema que tem sido extensivamente explorado através de algoritmos de Planejamento de Caminhos para Múltiplos Agentes (MAPF). Alguns algoritmos, apesar de bem-sucedidos, frequentemente simplificam certas complexidades do mundo real a fim de manter a tratabilidade, o que pode causar impraticalidades quando os planos são aplicados diretamente em situações reais. Neste contexto, nós introduzimos a Busca k-Robust Baseada em Conflitos com Tempo Contínuo (k-Robust CCBS), um algoritmo novo que mitiga estas limitações incorporando o elemento de tempo contínuo ao planejamento, e uma camada adicional de segurança atravez de k-robustez.

A motivação deste trabalho vem de dois pontos principais. Primeiro, existe a necessidade de alinhar soluções de MAPF de maneira mais próxima com situações reais, onde fatores como movimentação perfeita dos robôs e precisão de tempo não são garantidas. Algoritmos de MAPF tradicionais, operando sob condições de tempo e espaço discretos, frequentemente relevam estes elementos, tornando necessários ajustes pós-execução que não são somente custosos em termos de recursos, mas também sujeitos a erros. Segundo, a garantia de segurança em um sistema de coordenação de múltiplos agentes é essencial. Ao introduzir o conceito de k-robustez, o algoritmo não só planeja caminhos que são mais resilientes a incertezas, mas também significantemente reduz os riscos de colisão, melhorando a confiabilidade geral do sistema.

k-Robust CCBS encara estes dois desafios através de duas soluções principais. O uso de tempo contínuo para planejamento de caminhos permite um cálculo mais preciso e realista de rotas, separando-se das restrições resultantes de intervalos de tempo discretos. Esta abordagem faz com que os caminhos não sejam somente eficientes, mas que também possam ser mais próximos da movimentação real dos robôs. Além disso, a inclusão de k-robustez ao algoritmo introduz um buffer de segurança que leva em conta potenciais desvios no comportamento dos agentes, seja por erros mecânicos, obstáculos inesperados ou discrepâncias de tempo. Este buffer garante que, mesmo quando os agentes não aderem estritamente ao seus planos, o sistema ainda pode manter um grau alto de segurança e permanecer livre de colisões, desde que estes imprevistos não excedam o tamanho do buffer.

A estrutura do k-Robust CCBS é construída baseada no tradicional algoritmo de Busca Baseada em Conflitos (CBS), modificando-a para o funcionamento com tempo contínuo e planejamento robusto. Em seu núcleo, o algoritmo opera em dois níveis: o solucionador de alto-nível, responsável por gerenciar a estratégia geral e resolução de conflitos, e o solucionador de baixo-nível, que tem a tarefa de encontrar caminhos viáveis para agentes individuais, respeitando as restrições determinadas pelo solucionador de alto-nível. Através desta abordagem hierárquica, o k-Robust CCBS é capaz de produzir caminhos que maximizam a eficiência, paralelamente com o objetivo de manter a segurança.

Avaliações empíricas do algoritmo k-Robust CCBS demonstraram sua superioridade sobre o algoritmo usado como base de comparação, Busca Baseada em Conflitos com Tempo Contínuo (CCBS). Os testes foram realizados em um grid de 10x10 células, sem obstáculos, com o k-Robust CCBS tendo valores de $k = 2$ e $k = 3$. Foram realizados 10 testes com configurações iniciais e finais aleatórias, para cada grupo de 6 a 13 agentes simultâneos. O resultado de cada execução era dado como sucesso caso os agentes chegassem até os seus destinos sem nenhuma colisão, e falhas eram consideradas caso houvesse alguma colisão durante a execução ou se o algoritmo falhasse em encontrar um plano para a configuração em um tempo limite de 60 segundos. Após a realização dos testes, como mostra a Tabela A.1, foi observado que o algoritmo k-Robust CCBS obteve uma taxa de sucesso com $k = 2$ de 86.25% e com $k = 3$ de 92.50%, com uma taxa média de sucesso de 89.38%. Comparado com o algoritmo base CCBS, que obteve uma taxa de sucesso de 36.25%, nota-se uma melhora significativa. Enquanto foi demostrado uma taxa de sucesso alta, mesmo assim não foi notado um aumento significativo nos valores de tempo medidos pelo k-Robust CCBS e CCBS durante a execução real e no planejamento abstrato dos caminhos, mostrando que nosso algoritmo conseguiu um aumento de sucesso sem um grande aumento de custo de execução.

Concluindo, o k-Robust CCBS representa uma contribuição no campo de planejamento de caminhos para múltiplos agentes. Através do uso de tempo contínuo e a integração de k-robustez, o algoritmo oferece uma solução mais realística, que ainda é segura e eficiente no cenário de coordenação de múltiplos robôs. Os nossos resultados indicaram que o aumento de custo de planejamento utilizando k-Robust CCBS, especialmente em uma quantidade maior de agentes, é uma troca razoável considerando a alta taxa de sucesso em nossas execuções. Para nossos trabalhos futuros, é possível adicionar no-

	Aumento no "*Flowtime*"	**Aumento no "*Makespan*"**	**Taxa de sucesso**
k = 2	196.89%	204.74%	86.25%
k = 3	190.77%	202.54%	92.50%
CCBS	201.30%	207.50%	36.25%

Table A.1 – Aumento médio do *flowtime* (tempo total de fluxo dos caminhos) e do *makespan* (tempo máximo de um caminho) e taxa média de sucesso para todas as instâncias. O aumento é calculado comparando os valores de flowtime e de makespan do plano abstrato com os valores do plano executado na simulação e, em seguida, calculando a média da diferença entre todas as instâncias concluídas para cada algoritmo.

vas funcionalidades ao algoritmo que possam melhorar sua eficiência, como landmarks e disjoint splitting. Também há espaço para experimentos comparados com soluções híbridas, que reagem a obstáculos dinâmicos através do uso de sensores. Foram realizados experimentos com outras configurações de mapas com obstáculos estáticos, mas não o suficiente para obtermos dados relevantes. Logo, outro ponto a ser verificado no futuro seria a avaliação do algoritmo em diferentes configurações de mapa.