Motion Coordination for Large Multi-Robot Teams in Obstacle-Rich Environments

Ph.D. Dissertation
submitted by

Wolfgang Hönig

A dissertation presented to the faculty of the
USC Graduate School in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the
University of Southern California

May 2019

Committee:

Nora Ayanian (Chair)
Gaurav S. Sukhatme
Sven Koenig
Vijay Kumar (Outside Member)

USC Viterbi
School of Engineering

Für Fünfchen.

Acknowledgment

This thesis would not have been possible without the guidance and support of many others.

I would like to thank my advisor, Prof. Nora Ayanian, who guided my development as a researcher. She was always supportive, both academically and personally, yet also left enough room for my own ideas. She initiated and encouraged collaborations with several other groups, which helped make my research more diverse and interesting. Finally, she prepared me well for life after my PhD, by allowing me to participate in mentorship, teaching, and research. The ACT Lab felt like an academic start-up, and I am grateful that Prof. Nora Ayanian let me influence it in its founding years.

I would also like to thank Professors Sven Koenig and Satish Kumar, who taught me many fascinating and crucial approaches to research in artificial intelligence. Their feedback and discussions were always very insightful, strongly influenced my research, and showed me the importance of paying attention to details.

I am also very grateful to my other committee members, Professors Gaurav S. Sukhatme and Vijay Kumar, for their helpful comments and insights.

Much of my work would have been impossible without my collaborators in RESL, IDM Lab, MxR, Bitcraze, and Amazon Robotics. In particular, I would like to thank James A. Preiss for an unforgettable and productive summer; Hang Ma and Liron Cohen for many great discussions on search-based planning; Thai Phan for his collaboration on Mixed Reality; Arnaud Taffanel for all his help and insight with the Crazyflie quadrotor; and Scott Kiesel for his insightful mentorship during and after my time at Amazon Robotics.

I was fortunate to work with and mentor two very talented Masters students, Mark Debord and Baskın Şenbaşlar, whose work is part of this thesis.

Graduate school can be stressful at times, and I am grateful for my lab members, with whom I shared both the good and the bad. Thanks to Nitin Kamra, Arash Tavakoli, Ameer Hamza, Jinyao Ren, and Elizabeth Boroson for all the fruitful discussions, support, and fun — you made the ACT Lab an enjoyable place to be.

My family has always been very supportive. My deepest gratitude to my parents, my sister, and her family for their encouragement, ideas, and love.

Finally, I am particularly thankful to Aliya, without whom I would not have started a PhD, who unconditionally supported me during my time as grad student, and who made my life wonderful.

Abstract

Using multiple robots is important for search-and-rescue, mining, entertainment, and warehouse automation, where robots must operate in constrained, perhaps even maze-like environments frequently. Motion coordination, which plans and executes the movement of robots, is a fundamental building block in such scenarios. Ideally, motion coordination should be capable of coordinating hundreds of robots efficiently even in obstacle-rich environments, of being executed on real physical robots, and of handling unforeseen dynamic changes.

Two components of motion coordination are considered: motion planning and motion execution, both of which are coupled. Motion planning assumes that the environment is known a priori, and produces motion plans with theoretical guarantees such as completeness or optimality. On the other hand, motion execution provides safety guarantees even in the case of unforeseen dynamic changes. In this thesis, we extend the state-of-art in motion planning by providing a planning solution that can plan for hundreds of heterogeneous robots within minutes. We also introduce motion execution frameworks that can be used for robust (with respect to dynamically appearing obstacles, imperfect motion execution, etc.) and persistent execution.

For motion planning, ideas from the artificial intelligence (AI) and robotics communities are combined. AI solvers are capable of computing plans for hundreds of robots in minutes with suboptimality guarantees. However, these solvers' simplified and unrealistic agent model assumptions make it challenging to execute the computed plans safely on real robots. Robotics solutions typically include richer kinodynamic models during planning, but are very slow when many robots and obstacles are taken into account. In this work, we combine the advantages of the two methods by using a two-step approach. First, we use and extend solvers from the AI community to solve a simplified coordination problem. The output is a discrete plan that, in its original form, cannot be executed on a real robot. Second, we apply a computationally efficient post-processing step that creates a smooth continuous plan, taking relevant kinematic constraints into account.

For motion execution, we couple motion planning and traditional motion execution methods by adding a feedback term. In warehouse applications, this feedback term overlaps planning and execution, enabling uninterrupted robot motions. In distributed settings, such a feedback term can be used to avoid livelocks of robots in many cases.

The approaches are demonstrated on different teams of robots, including ground robot teams, UAV teams, and heterogeneous teams.

Contents

CHAPTER 1

Introduction

Coordinating many collaborative robots is very useful for search-and-rescue, mining, entertainment and warehouse automation, as well as many other tasks. In many of these cases, robots must maneuver in tight spaces filled with obstacles to reach their objectives. Obstacles might be arranged in maze-like structures and thus require robots to actually plan their motions in advance rather than greedily advancing towards the goal. Moreover, tight spaces require coordination between robots: for example, deciding which robot should pass through a narrow passageway first. A purely reactive coordination approach can lead to livelock and is therefore undesirable. Instead, robots should coordinate to decide which task each robot should fulfill and the motions each robot must execute to finish their tasks. Some real-world systems already use many robots, such as an Amazon Fulfillment Center that might use hundreds of robots on a single floor [43]. Thus, there is a pressing need for planning solutions that scale to such problem sizes and that can deal with real physical systems and unforeseen effects, such as a broken robot or a priori unknown obstacles.

In this work, we propose a solution for motion coordination that combines the advantages of centralized planning and robust execution. First, we propose novel centralized *motion planning* techniques, that can produce kinodynamically feasible trajectories for hundreds of heterogeneous robots in obstacle-rich environments within minutes. Second, we introduce *robust execution*, an approach that can execute pre-planned trajectories safely under a variety of dynamic changes (such as dynamically appearing obstacles or external disturbances), while considering the pre-planned trajectories directly. All algorithms are demonstrated on real robots, including differential-drive ground robots and UAVs.

Motion Planning

In artificial intelligence, coordination tasks have been studied in the field of multi-agent systems. One example is the *Multi-Agent Path Finding* (MAPF) problem, where agents are tasked to move

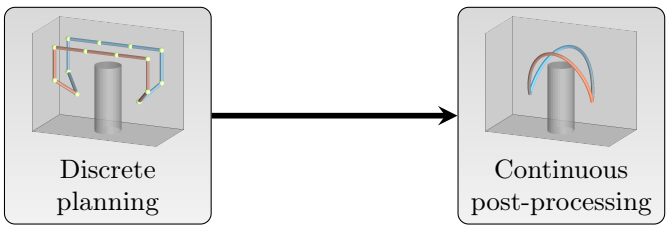


Figure 1.1: Overview of the motion planning approach.

from known start to goal locations. Recent advances allow the computation of paths for hundreds of agents in obstacle-rich environments within minutes while also providing bounded suboptimality guarantees. Such planners make simplifying assumptions about the environment and robots: both time and space are discretized, the agents are assumed to be point robots that can move in perfect synchronization on a graph, and the agents are assumed to be able to perfectly execute the plan. Thus, applying the planners' solution to real robots is challenging, because the assumptions are far from reality.

The robotics community has developed different approaches to deal with real robots. Many solutions use detailed models of the robots, including kinodynamic constraints. Combined with robust controllers, this ensures that the computed plans are likely able to be executed on real robots in practice. However, such planning is computationally expensive and often only works with a team of a few robots. Other approaches in robotics use reactive planning and work well on many robots, but cannot deal with maze-like environments and do not provide formal guarantees like completeness or suboptimality.

In this thesis, the advantages of the two approaches are combined. In particular, we develop a scalable algorithm that comes with theoretical guarantees and apply it to real robot systems. Our approach is based on planners from the AI community (and novel extensions thereof) and uses post-processing steps to allow safe execution on real robots, see Fig. 1.1. We leverage the fact that AI solvers essentially solve the coordination problem on an abstract level, creating a partial order between the robots. Such an abstract solution can be refined to include robot-specific properties. The post-processing step depends on the type of robots; this thesis includes examples for ground robots (in a warehouse-like domain), aerial vehicles, and heterogeneous teams.

Robust Execution

To compensate for changes in the environment or imperfect execution, one might apply cooperative collision avoidance strategies, such as ORCA [16], at runtime. However, such algorithms often operate locally and do not take the pre-planned trajectories into account. Robust execution, on the other hand, avoids future collision more effectively because it directly considers pre-planned trajectories. In case unforeseen changes occur (such as dynamically appearing obstacles), one might need to trigger (incremental) re-planning for the motions, adding a feedback term between motion planning and motion execution, see Fig. 1.2.

This work will show how robust execution can be used in different scenarios. First, we demonstrate in warehouse settings that robust execution allows us to apply planning algorithms from the AI community directly without the need of frequent replanning. Here, re-planning is triggered by newly appearing obstacles or new tasks. We then develop a novel data structure that allows us to overlap planning and execution, avoiding unnecessary idling during the robots' operation. Second, we generalize one of our motion planning algorithms to regenerate trajectories in a distributed way in real-time. Here, the feedback term is used to run a single-robot motion planner that avoids livelocks of robots in many cases.

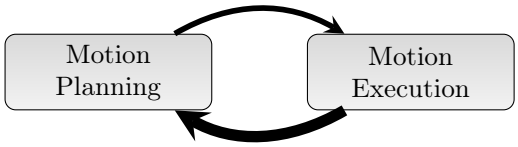


Figure 1.2: Overview of the motion execution approach.

1.1 Contributions

This dissertation presents scientific contributions in AI on multi-agent path finding, in robotics on motion planning, and in engineering on multi-robot system verification as follows[1]:

Artificial Intelligence

- A generalization of MAPF (MAPF/C) that allows the computation of collision-free paths for arbitrary roadmaps for robots of a known physical size, as well as the development of efficient solvers for MAPF/C problem instances, including labeled and unlabeled variants, with bounded suboptimality guarantees. Details are given in Section 6.3 and published as part of an IEEE Transactions on Robotics paper [172].
- A generalization of Conflict-Based Search that also finds the optimal task assignment (CBS-TA). Details are given in Section 6.2 and published as part of an AAMAS paper [166].

Robotics

- MAPF-POST, a post-processing method that can be used to safely execute paths on ground robots produced by any MAPF solver. MAPF-POST can take simple kinematic constraints into account and provides a user-defined safety distance between robots. Details are presented in Chapter 7 and the work has been conditionally accepted for publication in the Journal of Artificial Intelligence Research [169].
- A post-processing approach for quadrotors and ground robots in obstacle-rich environments that considers the asymmetric interaction between multiple UAVs, known as downwash. Details are given in Chapters 8 and 9 and published as part of an IEEE Transactions on Robotics paper [172] and an IROS paper [38].
- A framework for robust motion execution for robots in warehouses and differentially-flat robots. Details are given in Chapter 10 and are accepted at IEEE RA-L [167] and published at DARS [129].

Engineering

- A testbed for experimental validation of algorithms targeting aerial vehicles called the Crazyswarm. This testbed has been used successfully with up to 49 vehicles. Details are given in Chapter 4 and published at ICRA [119].
- A methodology that integrates Mixed Reality with robotics research. Details are given in Chapter 5 and published at IROS [171].

1.2 Outline

An overview on existing literature for both the AI and robotics communities is provided in Chapter 2, including theoretical hardness results and typical algorithms.

In Part I techniques and testbeds for experimental validation of multi-robot systems are presented. In particular, Chapters 3 and 4 show practical testbeds for ground robots and UAVs, respectively. Chapter 5 demonstrates how Mixed Reality can be used to simplify implementation and validation in robotics development and research.

[1] A full list of publications, including work that is not described in this thesis, is given in Appendix B.

Part II presents centralized motion planning techniques. A formal introduction to the Multi-Agent Path Finding problem and important variants is presented in Chapter 6. In particular, we discuss labeled, unlabeled, and multi-color cases and typical algorithms that can solve such instances. We use this background information to introduce multi-agent path finding with optimal task assignment and efficient solvers. Moreover, we introduce a generalization of MAPF, called MAPF/C and solvers for MAPF/C problem instances, which lays the foundation for some later chapters in Part II. Chapters 7 and 8 discuss post-processing steps for ground robots and quadrotors, respectively. In case of ground robots we introduce MAPF-POST, which uses the planned paths and assigns a realistic velocity profile using a simple temporal network. For quadrotors, we use trajectory optimization within safe corridors to compute smooth and safe trajectories. We generalize our approach for quadrotors to heterogeneous robot teams (including UAVs of different sizes and differential-drive ground robots) in Chapter 9.

Part III presents algorithms that can be used to execute motions safely, even if there are dynamic changes, such as new tasks, newly appearing obstacles, or external disturbances. In Chapter 10 we show how a variation of MAPF-POST can be used for robust and persistent operation in a warehouse domain. Chapter 11 introduces a distributed robust execution method, which combines ideas from collaborative collision avoidance and our motion planning work.

Conclusions and future work are discussed in Chapter 12.

CHAPTER 2

Background

Motion coordination is well represented in the literature, including both the AI and robotics communities. In the AI community, the focus is largely on planning and multi-agent path finding (MAPF) is a commonly used term. However, MAPF is also known under different names: for example Cooperative Path Finding (CPF) [145], Pebble Motion on a Graph (PMG) [86], and Multi-Robot Path Planning (MPP) [179]. In the robotics community, there are solutions for motion planning and motion execution, where the latter are often under the umbrella of collision avoidance.

We consider three general variants of the MAPF problem: labeled, unlabeled, and multi-color. In the *labeled* or *non-anonymous* case, the goal location for each agent is given a priori. The *unlabeled* or *anonymous* case allows the algorithm to assign goals from a given set of possible goal locations. The *multi-color* version partitions agents into groups and allows agents to swap goals with their respective group members.

This chapter surveys relevant work in the AI and robotics communities, respectively, and provides an overview of the possible relevant applications of motion coordination to demonstrate its practical importance.

2.1 Artificial Intelligence

MAPF solutions in the AI literature generally ignore robotic-specific properties, such as a finite size or kinematic constraints. Most approaches are based on a search graph, where agents can either stay at their current vertex or traverse to a neighboring vertex during a timestep. A formal problem description of MAPF as used in this thesis and some of the related work is given in Chapter 6.

2.1.1 Theoretical Results

The labeled MAPF variant was studied as *pebble motions on graphs*, as a generalization of the "15-puzzle". Here, the assumption is typically that there is at least one unoccupied vertex. A solution can be found in polynomial time [123, 86] and it is possible to test for feasibility in linear time [59, 100]. These results also apply in the special case when there are as many agents as vertices [181]. However, finding optimal solutions cannot be done in polynomial time in the general case. One might minimize the total time, the time until the last agent reaches its goal (known as the *makespan*), or the total distance. The total distance differs from the total time optimization as it does not require moving agents in parallel. Unfortunately, it has been shown that finding optimal solutions for any of these three objectives is NP-hard in the labeled case. In particular, the study of puzzles has revealed that minimizing the total distance is NP-hard [58, 120]. Later, Surynek [144] proved that minimizing the makespan is NP-complete and Ma et al. [98] showed that it is even NP-hard to

approximate with a factor less than 4/3. Finally, Yu and Lavalle [180] showed that minimizing the total time is NP-hard and that the three objectives cannot be simultaneously optimized. They also showed that this is true in the special case when there are as many agents as vertices (a case that had been neglected in other proofs.)

In the unlabeled case, it is possible to construct a time-expanded flow-graph representing the MAPF problem. Maximum-flow algorithms can then be used to find optimal solutions with respect to makespan and total distance in polynomial time. Furthermore, it is possible to provide upper bounds on the number of required steps [178].

The feasibility of the multi-color version can be checked in linear time as well [59], but finding an optimal solution with respect to the makespan is known to be NP-hard even to approximate within a factor less than 4/3 [98].

Some works consider special instances of the MAPF problem. If the underlying graph is acyclic (i.e. a tree), the feasibility can be checked in linear time [59, 100, 9]. Some algorithms are constructive, allowing the computation of a solution [9].

2.1.2 Algorithms

Several algorithms to compute solutions for various MAPF variants have been proposed. The different algorithms have different properties in terms of completeness (complete in the general case, complete in restricted environments, incomplete), optimality (optimal, bounded suboptimal, suboptimal) with respect to different objectives, and execution (centralized, decentralized). The approaches can be roughly categorized into reduction-based solvers, search-based solvers, and rule-based solvers.

2.1.2.1 Reduction-based Approaches

Labeled MAPF can be solved by reductions to other well-studied problems. Using answer set programming enables us to describe problems generically with applications in the vehicle routing domain and optimization of user-defined cost functions [45]. Integer linear programming has been used to handle cases with as many agents as vertices with varying objectives including minimizing makespan, total distance, and total time [179]. A reduction to the Boolean satisfiability problem (SAT) can find solutions that minimize the makespan [146], attempt to minimize the makespan sub-optimally [145], or minimize the total time [147].

2.1.2.2 Search-based Approaches

Search-based approaches can be used to solve labeled MAPF instances as well. In principle, it is possible to plan in the joint space of all agents. However, this leads to exponentially many states to explore (with respect to the number of agents), making this approach impractical [49]. Thus, search-based methods attempt to reduce the search-space. One approach is to decouple the planning, which leads to an incomplete but fast algorithm. Hierarchical Cooperative A* (HCA*) computes paths for agents sequentially, treating the paths for previously planned agents as a moving obstacles [134]. Such an approach is very fast in practice but not complete. Extensions such as Windowed Hierarchical Cooperative A* (WHCA*) [134] and Cooperative Partial-Refinement A* (CPRA*) [141] improve the runtime of the approach by including temporal and spatial hierarchies, respectively.

Other search-based approaches attempt to postpone the coupling of the search-space and usually find optimal solutions with respect to the total time. Independence Detection with Operator Decomposition [139] attempts to group the agents into independent groups, reducing the effective search space. Enhanced Partial Expansion A* [57] operates in the joint search-space, but avoids generating successor nodes in the search tree that are not required for the optimal solution.

Subdimensional Expansion and the M* algorithm [157, 156] dynamically increase the search-space locally in time and space. Increasing Cost Tree Search (ICTS) [132] and Conflict-Based Search (CBS) [131] use two-level search. In ICTS, the nodes in the high-level search define the solution cost for each individual agent. The low-level search stores all solutions of a requested cost for a given agent (avoiding conflicts with the other agents). In CBS, on the other hand, the high-level search defines the solution for each agent and spatio-temporal constraints. The low-level search finds solutions for single agents that are consistent with a list of such constraints. More details about CBS (and novel extensions thereof) are presented in Chapter 6.

2.1.2.3 Rule-based Approaches

Rule-based approaches only compute suboptimal solutions but can do so very fast. In some cases it is possible to give completeness guarantees. A well known but incomplete rule-based approach is Push and Swap [93, 126]. Push and Rotate is an extension of Push and Swap that is complete if there are at least two unoccupied vertices [165]. A decentralized version of Push and Swap is Push-Swap-Wait [164], which has completeness guarantees for some environments. The tree-based agent swapping strategy (TASS) [83] is an approach that is complete for acyclic graphs. Another work uses a multiphase planning algorithm [113] based on a spanning tree representation of the environment. Finally, BIBOX [143] is complete on bi-connected graph with 2 unoccupied vertices.

Some algorithms use both rules and search. For example, Flow Annotation Replanning [159] and MAPP [160] can solve labeled MAPF instances suboptimally. The latter provides completeness guarantees on graphs with a specific *slidable* property. More details and a limited performance evaluation for different labeled MAPF solving techniques are given in [49].

2.1.2.4 Other Approaches

The unlabeled MAPF problem can be solved optimally with respect to makespan and total distance in polynomial time [178]. The approach uses a time-expanded flow-graph and a min-cost max-flow algorithm to compute the resulting path for each agent.

Conflict-Based Min-Cost-Flow (CBM) can compute makespan-optimal solutions to the multi-color MAPF problem [95]. It is a search-based approach and combines ideas from CBS in the high-level and the flow-based approach in the low-level.

2.2 Robotics

Robots have, unlike agents, a physical embodiment and are subject to kinodynamic constraints. We first show that even relatively simple problems in continuous environments are PSPACE-hard. Afterwards, we discuss existing approaches that can solve multi-robot planning problems in practice.

2.2.1 Theoretical Results

It is known that even when ignoring kinodynamic constraints and looking at a pure motion planning problem, the resulting decision problem is PSPACE-hard. In all cases we assume that shapes of the robots and the environment are known and that we can arbitrarily move the robots (translation/rotation) in space. The motion planning decision problem determines whether it is possible to move the robots from their known start locations to their respective goal locations. If the robots and environments are arbitrary rectangles, the labeled problem, also called *Warehouseman's Problem*, is PSPACE-hard [70]. This is even true for domino-like sliding puzzles, where robot sizes have a fixed side length ratio of one-to-two [69]. PSPACE-hardness also holds in the unlabeled case considering unit-square robots and polygonal obstacles [136].

If restrictions on the environment and start/goal locations can be imposed, the problem is easier to solve. The *Well-Known Infrastructure property* requires robots' start and goal location to not obstruct each other [25]. In this case, planning for each robot sequentially leads to a complete (but not optimal) solution. If no obstacles are present, the unlabeled problem can be solved in polynomial time while minimizing the sum-squared distance traveled [152].

2.2.2 Algorithms

Many solutions for multi-robot motion planning have been proposed in the past and there is some similarity to AI-based solvers. Motion planning problems can be framed as optimization problems for all robots in joint space. This approach is similar to the reduction-based solvers from AI and provides completeness and optimality but poor scalability. Several methods try to decouple the robots (either statically or dynamically), similar to some of the search-based approaches in AI. Rule-based methods in robotics are local planners or collision avoidance methods, showing great scalability but no completeness or optimality guarantees. Sampling-based approaches attempt to reduce the large state-space using random sampling. Some of the variants work on discrete graphs while others work in continuous environments.

2.2.2.1 Meta-Robot Approaches

A simple approach to multi-robot motion planning is to repurpose a single-robot planner and represent the Cartesian product of the robots' configuration spaces as a single large joint configuration space [90]. Robot-robot collisions are represented as configuration-space obstacles. However, the high-dimensional search space is computationally infeasible for large teams.

A similar approach uses optimization techniques to optimize for a team of robots. Mixed Integer Linear Programming (MILP) [128], Mixed Integer Quadratic Programming (MIQP) [104], Sequential Convex Programming (SCP) [8], Particle Swarm Optimization (PSO) [101], and co-evolutionary genetic programming [77] have been used in the past. They provide optimal results (with respect to a cost function), but do not scale to large teams or obstacle-rich environments.

2.2.2.2 Decoupling Approaches

Some approaches attempt to decouple the robots. In the priority-based schema the planning is done sequentially, treating previously created robot motions as dynamic obstacles [46]. This approach is incomplete in the general case and highly depends on the order of the robots. Thus, many different approaches have been proposed that find priorities automatically to minimize the risk of deadlocks [21, 15, 13]. Decentralized variants of prioritized planning exist as well [155]. In case the start and goal locations of the robots never obstruct other vehicles, the priority-based approach can always find solutions [25]; this *Well-Known Infrastructure* property is in particular relevant for warehouse scenarios. It is possible to use a token-based decentralized method for motion planning in well-known infrastructures [27].

Another decoupling approach separates space and time. One can first plan paths for the robots individually and then compute velocity profiles that avoid any collisions during execution. This approach has been first applied to single robots navigating in environments with dynamic obstacles [80], but can also be used in the context of multi-robot systems [115]. A similar approach uses shortest paths on pre-defined roadmaps and then formulates a linear program (LP) to solve the task and path planning algorithm optimally [125].

Moreover, decoupling can also be used in combination with optimization techniques such as Sequential Convex Programming [31].

2.2.2.3 Collision Avoidance Approaches

Reactive collision avoidance is often distributed and only requires local information. Thus, it scales well to hundreds of robots, but it is susceptible to local minima.

One approach is to use artificial potential fields [162]. Here, artificial potentials are used to repel robots from obstacles and attract them towards their goal.

Optimal reciprocal collision avoidance (ORCA) [16] uses velocity obstacles and computes a locally optimal velocity for each robot at each timestep independently. It is distributed, but requires an estimate of the positions and velocities of neighboring robots. Velocity obstacles can also be used for human crowd simulations [63]. Another approach only requires position information and is based on Buffered Voronoi Cells [182].

Some collision avoidance strategies can also track a reference trajectory, which is similar to what we refer to as robust execution. For example, nonlinear model predictive control [78] can be used for collision avoidance of UAVs. Another method only changes the temporal execution of reference trajectories by compensating for delaying disturbances [60].

2.2.2.4 Sampling-based Approaches

Sampling-based planning approaches have been widely successful in robotics. They work well in high-dimensional state-spaces and some algorithms such as RRT* and PRM* are asymptotically optimal [81]. Several variants have been created that target multi-robot systems specifically. The Multi-agent RRT* (MA-RRT*) applies RRT* to the discretized joint space of all robots (using a graph) [26]. A similar idea is used for discrete RRT (dRRT), where RRT is executed over implicit representation of the composite roadmap [137]. Decentralized Multi-Agent RRT (DMA-RRT) uses a merit-based token passing strategy where individual robots plan sequentially (but not necessarily in a fixed order). A cooperative version allows robots to change the plan of their group members [40].

2.2.2.5 Other Approaches

One approach discretizes the possible robot motions using state lattices and decouples spatial and temporal planning [33]. This method is neither complete nor optimal, but can deal with highly constrained robots such as forklifts.

Spatial envelopes can be used to define the required space for robot motions. It is then possible to identify "critical sections", where robots might collide, and define a precedence relation between different robots for coordination [114].

A combination of sampling and graph search was used to move disc robots in a plane with polygonal obstacles for multi-color MAPF [135].

If no obstacles are present, unlabeled MAPF can be solved in polynomial time while minimizing the sum-squared distance traveled [152].

2.3 Applications

MAPF-style problems are basic blocks in many applications where multiple robots need to cooperate. In the following we discuss applications where MAPF is operating in mostly a priori known environments that are potentially filled with obstacles.

Warehouse Automation In some warehouse domains, robots must deliver shelves to humans, who then take items from the shelves to fulfill orders [175]. In order to maximize the warehouse's throughput and the utilization of its space, the robots must operate in tight and occluded environments, avoiding close proximity with obstacles, other robots, and human workers.

Traffic Autonomous aircraft-towing vehicles that tow aircraft all the way from the runways to their gates (and vice versa) can reduce pollution, energy consumption, congestion, and human workload [108]. Air traffic management conflict resolution can help to avoid mid-air collisions [150]. For cars, an automated intersection management can be used to reduce congestion [42].

Entertainment Real-time strategy games are in need of fast path planning algorithms and formation control. While optimality is not required, completeness and fast computation are a necessity [134, 159]. Physical robots can also be used to replace traditional methods of visualization. For example, differential drive robots with LEDs can be used to replace displays [4]. Similarly, quadrotors with high-powered LEDs might be used for light shows [74].

There are many more possible applications of multi-robot motion planning, including search and rescue [84], human crowd simulation [63], mining [99], and formation control [170]. Additional references are listed in other resources [179, 90].

Part I

Experimental Validation of Multi-Robot Systems

CHAPTER 3

Ground Robots

The ground robot platform, which is used for empirical evaluation in this thesis, is based on a commercially available platform with small changes that are well documented on our project's webpage[1].

For the hardware platform, we selected the iRobot Create 2, a refurbished vacuum cleaning robot specifically designed for *science, technology, engineering and mathematics* (STEM) education[2].

In the following section, the hardware, custom modifications, and software infrastructure are discussed in more detail.

3.1 Vehicle

The iRobot Create 2 robot is a differential-drive robot with a cylindrical shape with diameter 0.35 m, height 0.1 m, and weight 3.5 kg. Its robust mechanical build can withstand crashes and accidental drops. The robot can reach a translational velocity of up to 0.5 m/s. The robot's firmware is closed-source; however, a serial port is available, together with an API [75] describing how to control the different actuators and how to read sensors. The following actuators are supported:

- 2 wheels (pulse-width-modulation or velocity control)
- Motors (main brush, vacuum, and side brush)
- LEDs (four binary status indicators and one ring with variable color and intensity)
- Display (four-digit seven-segment)
- Speaker (playing pre-programmed tones)

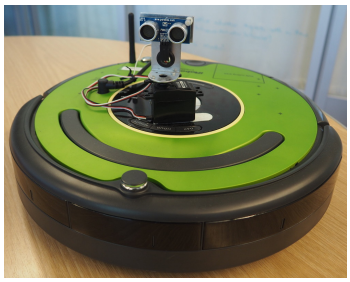
The following sensors are available:

- Wheel encoders
- 6 IR proximity sensors
- 4 cliff sensors (essentially downward facing IR-sensors)

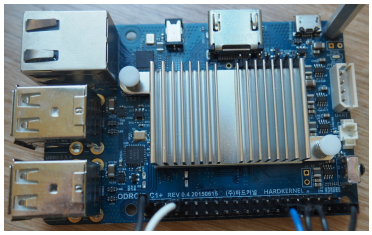
This chapter is based on **Wolfgang Hönig**, Arash Tavakoli, and Nora Ayanian. "Seamless Robot Simulation Integration for Education: A Case Study". In: *Workshop on the Role of Simulation in Robot Programming at SIMPAR 2016, San Francisco, CA, December 2016*. 2016. URL: http://act.usc.edu/publications/Hoenig_SimRP2016.pdf.

[1] https://pycreate2.readthedocs.io

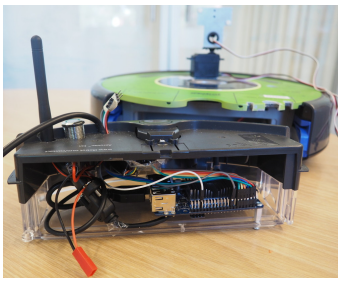
[2] More details are available at https://www.irobot.com/about-irobot/stem/create-2.aspx



(a) iRobot Create 2 with extensions.



(b) ODROID C1+ embedded computer.



(c) Removable extension containing added hardware components.

Figure 3.1: iRobot Create 2 robot (a), which we equipped with a small embedded computer (b). Everything is mechanically integrated (c) as a module for easy maintenance.

- 3 IR receivers (left, right, omni-directional)
- 2 wheel-drop sensors
- 2 bump sensors
- 8 buttons
- Temperature sensor
- Power-related sensors (voltage, current, and battery capacity)

Furthermore, there is a docking station which sends out IR beams such that the Create 2 can find the dock autonomously for self-charging.

To be able to run software without the need for a cable connection to a laptop, we use the small embedded computers ODROID C1+ or ODROID XU4[3], running Ubuntu 14.04 LTS. The ODROID C1+ features an ARM Cortex A5 1.5 GHz quad-core CPU, has 1 GB RAM, boots from microSD, and has four USB ports. The ODROID XU4 uses an ARM Cortex A15 2 GHz octa-core CPU and has 2 GB RAM. We attach a WiFi USB dongle to the ODROID to connect and control the robot wirelessly.

Additional sensors and actuators can be added using the *general purpose input/output* (GPIO) pins.

The ODROID is directly connected to the main battery, because the extension connector on the Create 2 has a low current limit. We integrate all components into the vacuum container of the robot, making it easier to transport and less affected by any crashes. Furthermore, connectors allow us to quickly swap our hardware additions between robots in order to simplify and speed up maintenance. Our final design, the extension container, and the embedded computer are shown in Fig. 3.1. The total cost per robot is approximately 350 USD. The list of parts and a wiring diagram are available online as part of the project's documentation[4].

3.2 Architecture

Each robot is running control and state estimation on-board its ODROID, using ROS as middleware. For state estimation, we support odometry (using the wheel encoders) as well as a motion-capture system. All robots are connected over WiFi to a central computer, which also runs the `rosmaster`.

[3] `https://www.hardkernel.com`
[4] `http://pycreate2.readthedocs.io`

We also implement simulation models of the robot in V-REP and GAZEBO. Our GAZEBO simulation uses ROS, while our V-REP simulation can be used either with ROS, or a custom high-level Python API that we developed specifically for educational purposes [173].

3.3 Control

There are two kinds of controllers for our ground robots. First, a controller that attempts to reach a waypoint at a desired time, controlling both heading and speed of the robot. Such a controller is useful for executing motions on a real robot that were planned on a grid. Second, a controller that follows a given trajectory (which encodes both a spatial path and a velocity profile). Such a controller is useful for executing motions that were planned using trajectory optimization techniques.

3.3.1 Go To Goal With Deadline

For the go-to-goal functionality, we use a simple PD control law. The control inputs are the current state in position $\mathbf{p}_{des}$, velocity $\mathbf{v}_{des}$, current yaw angle ψ, the goal location $\mathbf{p}_{des}$, and time until the robot should reach the goal t_{des}. The control output $u = [v, \omega]^T$ are the linear velocity and rotational velocity, which can be easily converted to individual motor speeds on a differential drive robot [127]. The current state position and heading can be directly measured in a motion-capture system and the velocity $\mathbf{v}$ can be numerically estimated.

The desired heading angle ψ_{des} can be computes as

$$\psi_{des} = \text{atan}_2(\mathbf{p}_{des}[y] - \mathbf{p}[y], \mathbf{p}_{des}[x] - \mathbf{p}[x]). \tag{3.1}$$

We can compute the angle error e_ψ as the shortest signed angle between ψ and ψ_{des}

$$e_\psi = \text{atan}_2(\sin(\psi_{des} - \psi), \cos(\psi_{des} - \psi)). \tag{3.2}$$

The desired speed v_{des} is computed using the remaining distance and time as follows

$$v_{des} = \frac{\|\mathbf{p}_{des} - \mathbf{p}\|}{t_{des}}. \tag{3.3}$$

The control law uses v_{des} as feed-forward term and a proportional feedback term as follows

$$u = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} v_{des} + K_p^v(v_{des} - \|\mathbf{v}\|) \\ K_p^\omega e_\psi \end{bmatrix}, \tag{3.4}$$

where K_p^v and K_p^ω are positive scalar control gains.

3.3.2 Trajectory Following

For trajectory following we use a provable stable tracking controller for non-holonomic vehicles [79]. The control inputs are the current state position $\mathbf{p} \in \mathbb{R}^2$, current yaw angle ψ, the setpoint in position $\mathbf{p}_{des}$, setpoint in velocity $\mathbf{v}_{des}$, and setpoint in acceleration $\ddot{\mathbf{p}}_{des}$. The control output $u = [v, \omega]^T$ are the linear velocity and rotational velocity, which can be easily converted to individual motor speeds on a differential drive robot [127]. The current state $(\mathbf{p}, \psi)$ can be directly measured in a motion-capture system; the setpoint can be given in different forms, including polynomials or Beziér curves.

We can compute the desired heading

$$\psi_{des} = \arctan \frac{\mathbf{v}_{des}[y]}{\mathbf{v}_{des}[x]}, \tag{3.5}$$

where $\mathbf{v}_{des}[y]$ and $\mathbf{v}_{des}[x]$ are the y and x components of the velocity setpoint, respectively. The desired angular velocity is:

$$\omega_{des} = \frac{\psi_{des}}{dt} = \frac{\mathbf{v}_{des}[x]\ddot{\mathbf{p}}_{des}[y] - \mathbf{v}_{des}[y]\ddot{\mathbf{p}}_{des}[x]}{\mathbf{v}_{des}[x]^2 + \mathbf{v}_{des}[y]^2} \tag{3.6}$$

The error in position can be computed as

$$\mathbf{e}_p = \begin{bmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{bmatrix} (\mathbf{p}_{des} - \mathbf{p}). \tag{3.7}$$

The heading error is simply $e_\psi = \psi_{des} - \psi$.

The control law has a feed-forward term and a proportional feedback term as follows:

$$u = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \|\mathbf{v}_{des}\| \cos e_\psi + K_x \mathbf{e}_p[x] \\ \omega_{des} + \|\mathbf{v}_{des}\|(K_y \mathbf{e}_p[y] + K_\psi \sin e_\psi) \end{bmatrix}, \tag{3.8}$$

where K_x, K_y, and K_ψ are positive scalar control gains.

3.4 Remarks

The proposed architecture is application agnostic and not only useful for multi-robot research validation, but also for single-robot use cases. For example, the robot hardware design has also been used for an undergraduate class "Introduction to Robotics", taught at the University of Southern California. Instead of ROS, students write their code in form of Python scripts, which can be executed without changes in a V-REP simulation environment or on the physical robot [173]. Additionally, the overall software and control architecture also works for ROBOTIS TurtleBot 3[5] robots, which have the advantage that the low-level control firmware is open source and can be modified.

We use this architecture in this thesis in Chapters 7, 9 and 10.

[5] http://www.robotis.us/turtlebot-3/

CHAPTER 4

Aerial Vehicles: Crazyswarm

Quadcopters are a popular robotic platform due to their agility, simplicity, and wide range of applications. Most research quadcopters are large enough to carry cameras and smartphone-grade computers, but they are also expensive and require a large space to operate safely. For very large swarms, smaller quadcopters are more attractive. The reduced size of these vehicles motivates different system design choices compared to a typical setup for larger vehicles in smaller numbers.

Here we describe the system architecture for a swarm of 49 very small quadcopters operating indoors. The vehicles use a motion-capture system for localization and communicate over three shared radios. Our system uses off-the-shelf hardware and performs most computation onboard. The software is available as open-source and already widely used in the research community. At the time of the initial publication in 2017, the system described here was, to our knowledge, the largest indoor quadcopter swarm, and had the largest number of quadcopters controlled per radio.

4.1 Related Work

Unmanned Aerial Vehicle (UAV) swarms have been used indoors for formation flight and collaborative behaviors, outdoors to demonstrate swarming algorithms, and in the media for artistic shows.

Kushleyev *et al.* describe the design, planning, and control of a custom micro quadcopter with experiments involving up to 20 vehicles [88]. The group's infrastructure is described in [105]. While key aspects are similar, our work differs in the following ways: (a) our architecture is designed to reduce communication bandwidth and scales better to larger numbers of quadcopters; (b) we demonstrate interactivity with the swarm; and (c) we use a commercially available platform and provide our source code online.

The Flying Machine Arena at ETH Zurich supports both single- and multi-robot experiments. An overview of the system architecture and applications is given in [94]. Unlike our work, the position controller runs offboard, making the system less robust to packet drops. The additional computational power is used for a latency compensation algorithm to improve accuracy for high-speed flights.

This chapter is based on James A. Preiss*, **Wolfgang Hönig***, Gaurav S. Sukhatme, and Nora Ayanian. "Crazyswarm: A large nano-quadcopter swarm". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Star (*) refers to equal contribution. 2017, pp. 3299–3304. DOI: 10.1109/ICRA.2017.7989376.
James A. Preiss was a PhD student and mostly worked on the EKF and on-board trajectory planning and execution. I mostly worked on radio communication and the ROS host computer software layer. We both worked on frame-to-frame object tracking, controller, and physical experiments.

Figure 4.1: The Crazyflie 2.0 miniature quadcopter with four motion-capture markers, LED expansion board (not visible), and battery.

UAVs have been used to demonstrate swarming algorithms outdoors on fixed wing [66, 32] and quadcopter [154] platforms with up to 50 vehicles. Our platform uses a smaller vehicle, allowing us to execute such experiments indoors in a laboratory environment.

Quadcopter swarms have been featured in the media as well. A world record for the most UAVs airborne simultaneously was set in 2015 with 100 quadcopters flying outdoors [74]. In 2016, a TED talk featured about 35 nano-quadcopters flying without a net above a crowd indoors using an ultra-wideband localization system [37]. In 2018, Intel showed 110 drones indoors at the CES keynote [73]. However, not many technical details about those solutions are publicly available. Our solution is open-source, requires a motion-capture system, allows dense formations, and shows good scalability.

Other multi-robot testbeds include the Robotarium [118], a platform that is accessible remotely, but currently only supports ground robots.

4.2 Vehicle

The Crazyflie 2.0 quadcopter (Fig. 4.1) measures 92 millimeters between diagonally opposed motor shafts and weighs 27 grams with a battery. It contains a 32-bit, 168 MHz ARM microcontroller with floating-point unit that is capable of significant onboard computation. Software and hardware are both open-source. The Crazyflie communicates with a PC over the Crazyradio PA, a 2.4 GHz USB radio that transmits up to two megabits per second in 32-byte packets.

Our payload of motion-capture markers and an LED light expansion board brings the Crazyflie's mass to 33 grams. This reduces battery life from the listed seven minutes to six minutes, and results in a peak thrust-to-weight ratio of ∼1.8. The system's dynamics and empirically determined parameters have been discussed in previous Master's theses [89, 53].

The Crazyflie's small size makes it suitable for indoor flight in dense formations. It can survive high-speed crashes due to its low inertia and poses little risk to humans.

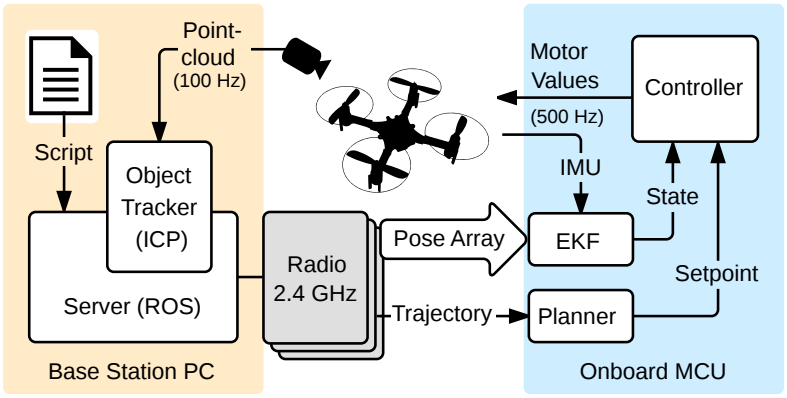


Figure 4.2: Diagram of major system components. A point cloud of markers detected by a motion-capture system is used to track the quadcopters. All estimated poses are broadcasted using three radios. Planning, state estimation, and control run onboard each vehicle at 500 Hz.

4.3 Architecture Overview

Our system architecture is outlined in Fig. 4.2. We track the vehicles with a Vicon motion-capture system using passive spherical markers. While the motion-capture system creates a single point of failure, we chose it over alternatives due to its high performance: typical position errors are less than one millimeter [94]. In comparison, a state-of-the-art decentralized localization system using ultra-wideband radio triangulation [91] showed position errors of over 10 centimeters, too large for dense formations. While vision-based methods are both accurate and decentralized [47], the required cameras and computers necessitate a much larger vehicle.

In contrast to related systems [94, 105], we implement the majority of in-flight computation onboard. The base station sends complete trajectory descriptions to the vehicle in the form of polynomials, ellipses, etc. For external feedback, the base station broadcasts vehicle poses using three radios operating on separate channels.

The main onboard loop runs at 500 Hz. In each loop cycle, the vehicle reads its Inertial Measurement Unit (IMU) and runs the state estimator, trajectory evaluator, and position controller. Messages with external pose estimates arrive asynchronously and are fused into the state estimate on the next cycle.

Since the full trajectory plan is stored onboard, the system is robust to significant radio packet loss. If a packet is dropped, the vehicle relies on its IMU to update the state estimate. We quantify our system's stability against simulated packet loss in Section 4.10.3.

4.4 Object Tracking

Standard rigid-body motion-capture software such as `Vicon Tracker` requires a unique marker arrangement for each tracked object. The Crazyflie's small size limits the number of locations to place a marker, making it impossible to form 49 unique arrangements that can be reliably distinguished. Therefore, we obtain only raw point clouds from the motion-capture system, and implement our own object tracker based on the *Iterative Closest Point* (ICP) algorithm [17] that handles identical marker arrangements. Our method is initialized with known positions, and subsequently updates the positions with frame-by-frame tracking.

4.4.1 Initialization

With identical marker arrangements for each vehicle, the object tracker needs some additional source of information to establish the mapping between vehicle identities (radio addresses) and spatial locations. We currently supply this information with a configuration file containing a fixed initial location for each vehicle. However, it is not feasible to place each vehicle at its exact configured position before every flight. To allow for small deviations, we perform a nearest-neighbor search within a layout-dependent radius about the initial position, and try ICP with many different initial guesses for vehicle yaw. We accept the best guess only if its resulting alignment error is low (less than 1 mm mean squared Euclidean distance between aligned points).

In future work we plan to eliminate the configuration file, using either infrared LEDs or programmed small motions for vehicle self-identification via the motion-capture system.

4.4.2 Frame-to-Frame Tracking

Each frame, we acquire a raw point cloud from the motion-capture system. For each object, we use ICP to register the marker positions corresponding to the object's last known pose against the entire scene point cloud. This process is independent for each object, so it can be executed in parallel. This approach assumes that there are no prolonged occlusions during the duration of the flight. We limit ICP to five iterations, which allows operation at 75 Hz with 49 robots using our computer hardware.

Motion-capture systems sometimes deliver a point cloud with spurious or missing points; if undetected, this may cause tracking errors. To mitigate these errors, we compute linear and angular velocities from the ICP alignments and reject physically implausible values as incorrect alignments. In combination with our on-board state estimation, a few missing frames do not cause significant instabilities, making tracking reliable in practice.

4.5 State Estimation

To reduce communication bandwidth requirements and maintain robustness against temporary communication loss, we fuse motion-capture and IMU measurements onboard in an Extended Kalman Filter (EKF). The filter is driven by IMU measurements at 500 Hz and estimates the states $(\mathbf{p}, \mathbf{v}, \mathbf{q})$ where $\mathbf{p} \in \mathbb{R}^3$ is the vehicle's position, $\mathbf{v} \in \mathbb{R}^3$ is its velocity, and $\mathbf{q} \in S^3$ is the unit quaternion transforming the vehicle's local coordinate frame into world coordinates. The system inputs are the accelerometer and gyroscope measurements, $\mathbf{a}_m$ and ω_m, respectively. The dynamics are:

$$\dot{\mathbf{p}} = \mathbf{v}, \quad \dot{\mathbf{v}} = \mathbf{q} \odot \mathbf{a}_m - \mathbf{g}, \quad \dot{\mathbf{q}} = \tfrac{1}{2}\Omega(\omega_m)\mathbf{q}, \tag{4.1}$$

where $\mathbf{g}$ is the gravity vector in world coordinates, $\Omega(\omega_m)$ is the quaternion multiplication matrix of ω_m as defined in [151], and $\odot$ denotes quaternion-vector rotation. The motion-capture system directly measures $\mathbf{p}$ and $\mathbf{q}$, resulting in a trivial measurement model.

Our EKF implementation follows the indirect error-state approach [151, 163], where IMU measurements drive a dynamics integration and the filter estimates the error and covariance of this integration. This method avoids the problematic extra dimension when representing a 3-DOF unit quaternion by four numbers. Whereas [151, 163] estimate accelerometer and gyroscope bias in the EKF, we have found that these biases do not drift significantly during the Crazyflie's battery life. During several six-minute test flights we found the maximum drift of any axis of the accelerometer to be $0.07\,\mathrm{m\,s^{-2}}$ and the maximum drift of any axis of the gyroscope to be $0.0018\,\mathrm{rad\,s^{-1}}$. Therefore, we measure biases on a level floor at startup and subtract these measurements for the duration of the flight. We also omit higher-order dynamics approximation terms, as we have found their effect unmeasurable in experiments.

4.6 Planning

We select a set of onboard trajectory planning methods that require a small amount of information exchange between the base station PC and the vehicles. Compared to architectures that evaluate trajectories on a PC and transmit attitude and thrust controls at a high rate [94, 105], we shift some planning effort onboard to reduce the needed radio bandwidth.

Quadcopter dynamics are differentially flat in the outputs $\mathbf{y} = (\mathbf{p}, \psi)$, where ψ is the yaw angle in world coordinates [103]. This means that the controls required to execute a trajectory in state space are functions of $\mathbf{y}$ and a finite number of its time derivatives. We take advantage of this fact and plan trajectories in $\mathbf{y}$ using functions that are at least four times differentiable.

4.6.1 Piecewise Polynomials

Piecewise polynomials are widely used to represent quadcopter trajectories. It is straightforward to construct a piecewise polynomial that satisfies given waypoint and continuity constraints. For a k-dimensional, q-piece, d-degree polynomial, the $k \cdot q \cdot (d+1)$ polynomial coefficients are concatenated into a single row vector $\mathbf{c}$. Waypoint and continuity constraints are then expressed as linear systems:

$$\mathbf{c}\mathbf{T}_w = \mathbf{w}, \quad \mathbf{c}\mathbf{T}_c = \mathbf{0}, \tag{4.2}$$

where $\mathbf{w}$ is a concatenated vector of waypoint values in both $\mathbf{y}$ and its derivatives, and $\mathbf{T}_w$ and $\mathbf{T}_c$ are both simple banded matrices obtained from the time values of the piece breaks. Constructions of $\mathbf{T}_w$, $\mathbf{T}_c$, and $\mathbf{w}$ are detailed in [68].

With a suitably high polynomial degree, the combined system Eq. (4.2) is underdetermined. The left null space of the system matrix thus provides a space for optimizing the trajectory for objectives such as min-risk [109] or observability [68] while implicitly satisfying the waypoint and continuity constraints. Additional constraints, such as thrust or angular velocity limits, are expressed as nonlinear inequalities in a general-purpose nonlinear optimization solver.

Our framework supports piecewise polynomials uploaded from the base station or built into the firmware image.

4.6.2 Online Single-Piece Polynomials

For short trajectories, a single polynomial can be sufficient. We implement an online single-piece polynomial planner for tasks such as vertical takeoff/landing and linear movement between hover points. This planner computes a closed-form solution for a degree-7 polynomial starting at the current state in $(\mathbf{p}, \mathbf{v}, \ddot{\mathbf{p}}, \psi, \dot{\psi})$ and ending at a desired state in the same variables. For linear movements, we restrict $\dddot{\mathbf{p}} = \mathbf{0}$ and $\ddot{\psi} = \dddot{\psi} = 0$, which results in a closed-form solution for the polynomial coefficients.

4.6.3 Ellipses

Elliptical motion is useful for demonstrations because it can generate a range of visually appealing behaviors from few parameters. Ellipses are parameterized by their center, axes, period, and phase, and are infinitely differentiable, so the differentially flat transformation applies.

Commanding a quadcopter to switch from hovering to elliptical motion produces a large step change in the controller setpoint, potentially causing instability. Our system overcomes this issue by using the single-piece polynomial planner to plan a trajectory that smoothly accelerates into the ellipse, iteratively replanning with longer time horizons until it achieves a trajectory that respects the vehicle's dynamic limits. The procedure is general and can be used to plan a smooth start from hover for any periodic trajectory.

Table 4.1: Consecutive packets dropped (see text for details.)

Delay	0	1	2	3	4	5+
3 ms	54458	2	280	1174	0	0
10 ms	51755	4111	4	1	0	0

4.6.4 Interactive Avoid-Obstacle Mode

The planners discussed so far can follow predefined paths, but are not suitable for dynamically changing environments. For the case of a single, moving obstacle at a known location, such as a human, we use a specialized avoid-obstacle mode. The planner is fully distributed and only needs to know the quadcopter's position $\mathbf{p}$, its assigned home position $\mathbf{p}_{home}$, and the obstacle's position $\mathbf{p}_{obst}$. Let $\mathbf{d}$ be the vector between the current position and the obstacle to avoid:

$$\mathbf{d} = \mathbf{p} - \mathbf{p}_{obst},\ \delta = \|\mathbf{d}\|_2. \tag{4.3}$$

The new desired position $\mathbf{p}_{des}$ can be computed as a weighted displacement from the home position, where we move further away if the obstacle is close:

$$\mathbf{p}_{des} = \mathbf{p}_{home} + \max\left(\frac{f}{\delta + \delta^2}, \delta_{max}\right)\frac{\mathbf{d}}{\delta}, \tag{4.4}$$

where f is a scalar gain and δ_{max} limits displacement, preventing collisions when appropriately small relative to the robot spacing. This is similar to a potential field approach. The attraction to the home position is encoded in Eq. (4.4) by displacing from $\mathbf{p}_{home}$ rather than $\mathbf{p}$. Instead of using the computed $\mathbf{p}_{des}$ directly as control input, we compute a smooth trajectory with bounded velocity and acceleration.

4.7 Communication

We use two different kinds of communication: request-response and broadcasting. Request-response is primarily used for configuration while the Crazyflie is still on the ground. This includes uploading a trajectory, changing flight parameters such as controller gains, and assigning each Crazyflie to a group. Broadcasting is used during the flight to minimize latency for position feedback, and to achieve synchronized behavior for taking off, landing, starting a trajectory, etc. Broadcast commands can be restricted to subsets of the swarm by including a group ID number.

Our communication does not use a standard transport layer and hence, we need to handle sporadic packet drops as part of our protocol. In order to achieve low latency, we do not aim for guaranteed packet delivery, but rather for a high probability of reliable communication. In our protocol, all commands are idempotent, so they can be received multiple times without any side effects. This allows us to repeat request-response commands until acknowledged or until a timeout occurs. Swarm-coordination commands, such as taking off, do not wait for an acknowledgment but are repeated several times for a high probability that all Crazyflies receive the command. Since external pose estimates are send at a high fixed rate every 10 ms, there is no need to explicitly repeat such messages.

Empirically, for repeated commands, the likelihood of packet loss depends on the rate at which the command is repeated. We show this effect in Table 4.1. We count the number of consecutive packets dropped from 0 (no packet dropped) to 5+ (5 or more consecutive packets dropped) for 10000 packets sent to six Crazyflies. The results show that repeating messages 5 times every 3 ms

for swarm coordination commands is likely to reach all Crazyflies. Furthermore, it is very unlikely to drop more than five consecutive position updates, which is sufficient for stable control.

For 49 vehicles we use three radios, with each vehicle permanently assigned to one radio. We broadcast positions $\mathbf{p}$ as 24-bit fixed-point numbers and compress quaternions $\mathbf{q}$ into 32 bits by transmitting the smallest three values in reduced precision and reconstructing the largest value from the unit quaternion property $\|\mathbf{q}\|_2 = 1$. Compression allows us to fit two position updates in one 32-byte radio packet without degrading the measurements beyond their inherent noise level. Furthermore, we transmit two such radio packets per USB request to the Crazyradio PA, allowing us to broadcast the pose of up to four Crazyflies per USB request.

4.8 Control

Our controller is based on the nonlinear position controller of [103], augmented with integral terms for position and yaw error. The control input is the current state $(\mathbf{p}, \mathbf{v}, \mathbf{q}, \omega_m)$ and the setpoint in position $\mathbf{p}_{des}$, velocity $\mathbf{v}_{des}$, acceleration $\ddot{\mathbf{p}}_{des}$, yaw ψ_{des}, and angular velocity ω_{des}.

The controller is a cascaded design with an outer loop for position and an inner loop for attitude. Both loops implement PID feedback terms; the outer position loop adds feedforward terms from the trajectory plan. We define position and velocity errors as

$$\mathbf{e}_p = \mathbf{p}_{des} - \mathbf{p}, \quad \mathbf{e}_v = \mathbf{v}_{des} - \mathbf{v}$$

respectively, and compute the desired force vector as follows:

$$\mathbf{F}_{des} = K_p\mathbf{e}_p + K_i \int_0^t \mathbf{e}_p dt + K_v\mathbf{e}_v + m\mathbf{g} + m\ddot{\mathbf{p}}_{des}, \tag{4.5}$$

where K_p, K_i, and K_v are positive diagonal gain matrices.

The desired body rotation matrix R_{des} is a function of $\mathbf{F}_{des}$ and ψ_{des}. The orientation error is computed as

$$\mathbf{e}_o = \frac{1}{2}\left(R^T R_{des} - R_{des}^T R\right)^\vee,$$

where R is the matrix form of the attitude quaternion $\mathbf{q}$ and $(\cdot)^\vee$ is the vee operator mapping $SO(3) \to \mathbb{R}^3$. The desired moments are then computed with another PID controller:

$$\mathbf{M}_{des} = K_o\mathbf{e}_o + K_m \int_0^t \mathbf{e}_o dt + K_\omega(\omega_{des} - \omega) \tag{4.6}$$

with positive diagonal gain matrices K_o, K_m, and K_ω. A simple linear transformation can compute the desired squared rotor speeds from $\mathbf{M}_{des}$ and the projection of $\mathbf{F}_{des}$ to the body z axis, as shown in [103].

This controller is identical to the one presented in [103] except for the added integral terms. The position error integral in Eq. (4.5) compensates for battery voltage drop over time (z-part) and unbalanced center of mass due to asymmetries (x, y-parts). From the attitude error integral in Eq. (4.6), we only use the z-component, which compensates for inaccuracies in yaw due to uneven wear on propellers and motors. Manual trimming and part replacement can compensate for such issues on a single vehicle, but is not feasible on a large fleet.

4.9 Software Tools

Many routine tasks become non-trivial when working with 49 robots. We have developed command-line tools for mass rebooting, firmware updates, firmware version query, and battery voltage checks

over the radio. A power-save mode turns off the main microcontroller and LED ring while leaving the radio active, permitting powering on the Crazyflie remotely. We can execute such commands and disable or enable subsets of the swarm using a graphical user interface. A Python scripting layer supports development of complex multi-stage formation flight plans.

Our decision to perform significant computation onboard complicates in-flight debugging due to constrained radio telemetry bandwidth and limited permanent storage for logs. To ease this problem, we structure major onboard procedures as platform-independent modules. We use the `SWIG` package to generate Python bindings to these modules, allowing firmware-in-the-loop testing in simulation via Python scripts.

4.10 Experiments

Our experiments quantify system performance and demonstrate the overall capabilities of our architecture. All experiments are conducted in a $6\,\mathrm{m} \times 6\,\mathrm{m} \times 3\,\mathrm{m}$ motion-capture space using a Vicon Vantage system with 24 cameras. `Vicon Tracker` obtains the marker point clouds on a PC with Windows 7, Xeon E5-2630 2.2 GHz, and 16 GB RAM. Our software is written in C++ using ROS Kinetic for visualization and scripting and is running on a PC with Ubuntu 16.04, Xeon E5-2630 2.2 GHz, and 32 GB RAM. To reduce latency, we do not use any ROS messages in the critical path between receiving data from the motion-capture system and broadcasting estimated poses to the Crazyflies. Sample flight videos are available at `http://youtu.be/D0CrjoYDt9w`.

4.10.1 Latency

The total time delay between physical movement and arrival of the corresponding position feedback message to the firmware significantly affects overall system performance [94]. Latency increases with swarm size due to both computational and communication overhead. We estimate the latency in two ways: first, we use the `Vicon DataStream SDK` to query an estimate of the motion-capture system latency, and instrument our code to estimate the runtime of our own software components. Second, we verify those numbers in the physical setup by inducing a sudden change in yaw to the Crazyflie. The delay between the immediate onboard IMU response and the corresponding change in external position measurement captures the full system latency. We read these values in real time via a JTAG debug adapter.

The estimated latencies from the base station software are shown in Fig. 4.3. The latency grows roughly linearly with the number of objects to track, with the largest portion caused by the motion-capture system (up to 14 ms for the 49-robot case). Object tracking and communication using the Crazyradio is done in separate threads, reducing the overall runtime. The communication latency increases in increments of four added Crazyflies per radio because of our compression algorithm (see Section 4.7.) Estimated total latency ranges from 8 ms to 23 ms.

The physical measurements show that the actual latency is a fixed 3 ms longer than the estimated numbers, resulting in an actual latency of 26 ms for the full 49-vehicle swarm.

4.10.2 Tracking performance

We estimate the tracking performance by integrating the overall Euclidean position error for a flight, excluding takeoff and landing. The flight path for each Crazyflie is a figure-8 repeated three times, with $0.76\,\mathrm{m\,s^{-1}}$ average speed and $1.5\,\mathrm{m\,s^{-1}}$ maximum speed. This trajectory is moderately fast for a small quadcopter, reaching a maximum roll angle of 20 deg. We try different square swarm sizes, ranging from 1 to 7×7, in a grid layout with 0.5 m spacing, flying at the same height. This tight formation causes additional aerodynamic disturbances, which affect the controller performance.

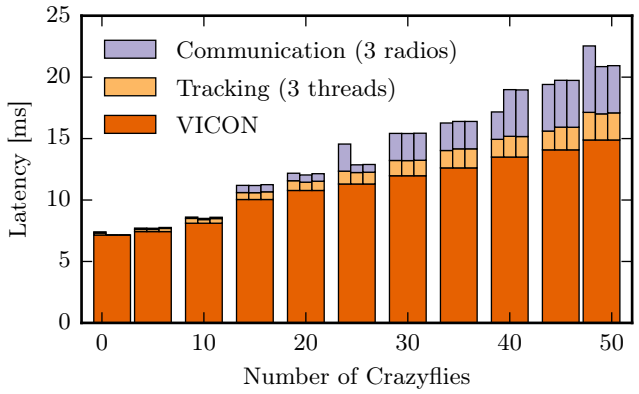


Figure 4.3: Software-estimated latency for different swarm sizes, averaged over 2000 measurements. The actual latency is 3 ms higher in all cases. The latency grows linearly with the swarm size and the largest portion is due to the motion-capture system.

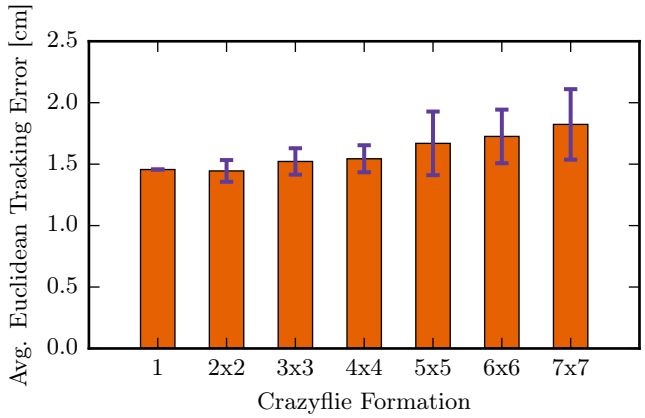


Figure 4.4: Measured tracking performance for different swarm sizes, averaged over the swarm and duration of the flight. The error bars show the standard deviation. The performance decreases in bigger swarms because of the additional aerodynamic effects between the quadcopters.

Table 4.2: Hover position error as a function of position update rate.

Position Rate [Hz]	100	10	5	$3.\bar{3}$	2.5	2
Avg. Tracking Error [cm]	0.58	0.68	0.91	1.48	1.95	2.18

Our average error is shown in Fig. 4.4. The error is below 2 cm per quadcopter even when the whole swarm is flying. However, the attitude controller starts to oscillate more to compensate for the additional airflow.

4.10.3 Effect of Position Update Rate on Hover Stability

We quantify the effect of position update rate on stability by varying the update rate from the maximal 100 Hz down to 2 Hz and measuring the mean Euclidean position error at hover state. Results are shown in Table 4.2. The range between 100 Hz and 10 Hz shows virtually no change. At

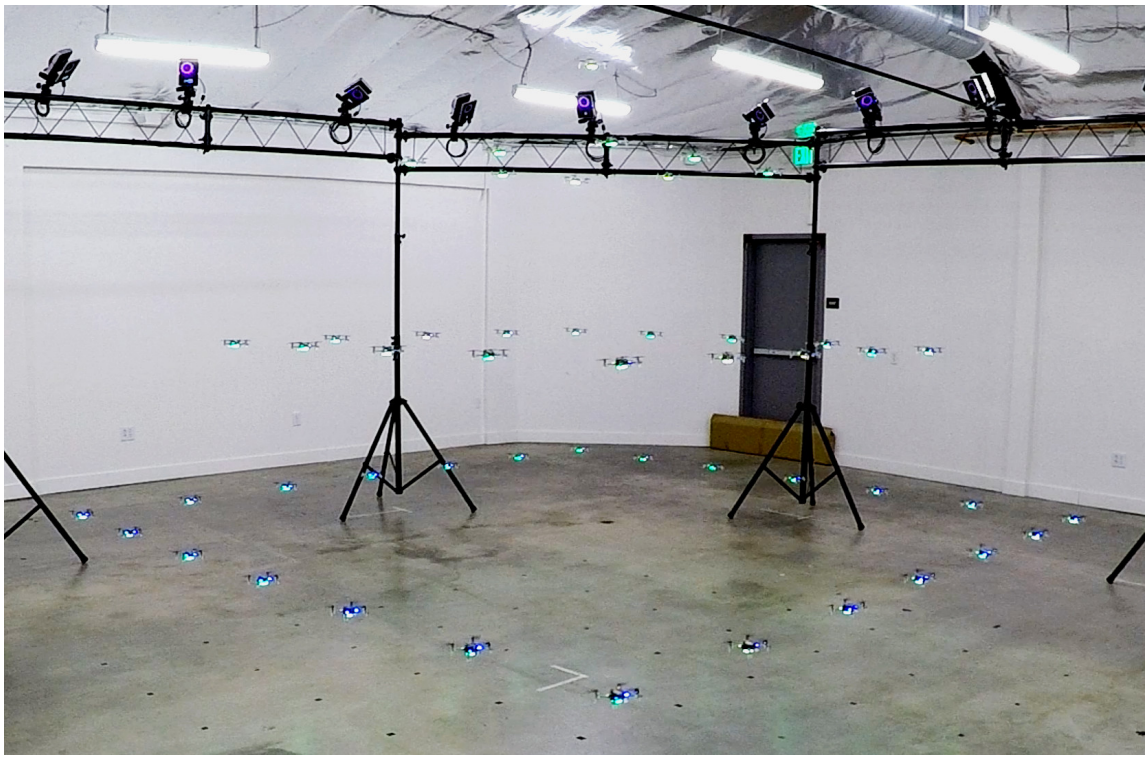


Figure 4.5: Forty-nine Crazyflies flying in a 4-layer rotating pyramid formation. The bottom layer is 3 m × 3 m with 0.5 m spacing between vehicles.

2 Hz, the system is still stable but shows significant oscillations. This demonstrates our system's robustness against communication loss during operation.

4.10.4 Coordinated Formation Flight

To demonstrate our scripting layer, we use a formation of 49 Crazyflies flying in a rotating pyramid as shown in Fig. 4.5. The Crazyflies are initially placed in a 7×7 grid with 0.5 m spacing. The takeoff is done in four different layers, with the user triggering the next action using a joystick or keyboard. The script initially assigns each Crazyflie a group number, according to its layer, and uploads the parameters for its ellipse to execute later. Because of the grouping, the takeoff of a whole layer can be achieved at the same time with broadcast messages. After all Crazyflies are in the air, the online planner plans a trajectory to smoothly enter the ellipse. The user can decide when to end the rotation, which causes another replanning onboard to reach a hover state above the initial takeoff location. Finally, landing is done using the same groups, minimizing aerodynamic disturbances during landing.

4.10.5 Swarm Interaction

A human operator is holding (or wearing) an object with motion-capture markers. We track the position of that object and broadcast its position alongside the positions of all quadcopters over the radio. Using the onboard planner described in Section 4.6.4, each Crazyflie computes its desired position, which avoids collisions with the obstacle or other quadcopters.

4.11 Remarks

We describe a system architecture for robust, synchronized, dynamic control of a large indoor quadcopter swarm. Our system fully utilizes the vehicles' onboard computation, allowing for robustness against unreliable communication and a rich set of trajectory planning methods requiring little radio bandwidth. Tests show good scalability for both latency and tracking performance with respect to the swarm size. For a swarm of 49 vehicles, only 3 radios are required and a latency below 30 ms is obtained, allowing moderately aggressive flight maneuvers using onboard state estimation with mean tracking errors below 2 cm. Our full source code, including tuned parameters such as EKF variances and controller gains, is available online[1]. We also maintain documentation, including setup instructions[2].

The Crazyswarm has been used in this work to verify motion planning algorithms (see Chapter 8) and by several other universities and research groups. Most of our custom firmware changes have been merged into the official firmware that is maintained by Bitcraze AB. Other extensions include support for custom UAVs (based on the Crazyflie control board, developed as part of the work presented in Chapter 9), and support for other motion-capture systems including PhaseSpace, OptiTrack, and Qualisys (mostly developed by users).

[1] https://github.com/USC-ACTLab/crazyswarm
[2] https://crazyswarm.readthedocs.io

CHAPTER 5

Mixed Reality

When robots operate in shared environments with humans, they are expected to behave predictably, operate safely, and complete the task even with the uncertainty inherent with human interaction. Preparing such a system for deployment often requires testing the robots in an environment shared with humans in order to resolve any unanticipated robot behaviors or reactions, which could be potentially dangerous to the human. In the case of a multi-robot system, uncertainty compounds and opportunities for error multiply, increasing the need for exhaustive testing in the shared environment but at the same time increasing the possibility of harm to both the robots and the human. Finally, as the number of components of the system (humans, robots, etc.) increases, controlling and debugging the system becomes more difficult.

Allowing system components to operate in a combination of physical and virtual environments can provide a safer and simpler way to test these interactions, not only by separating the system components, but also by allowing a gradual transition of the system components into shared physical environments. Such a *Mixed Reality* platform is a powerful testing tool that can address these issues and has been used to varying degrees in robotics and other fields. In this chapter, we introduce Mixed Reality as a tool for multi-robot research and discuss the necessary components for effective use. We will demonstrate four practical applications using different simulators to showcase the benefits of the Mixed Reality approach to simulation and development.

In this chapter, we establish Mixed Reality as a tool for research in robotics. To that end, we redefine Mixed Reality, and identify and describe the benefits of using Mixed Reality in robotics and multi-robot systems. We present novel use-cases which show the capabilities and benefits of Mixed Reality.

This chapter is based on:

- **Wolfgang Hönig**, Christina Milanes, Lisa Scaria, Thai Phan, Mark T. Bolas, and Nora Ayanian. "Mixed reality for robotics". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 5382–5387. DOI: 10.1109/IROS.2015.7354138
- Thai Phan, **Wolfgang Hönig**, and Nora Ayanian. "Mixed Reality Collaboration Between Human-Agent Teams". In: *IEEE Conference on Virtual Reality and 3D User Interfaces, (VR)*. 2018, pp. 659–660. DOI: 10.1109/VR.2018.8446542
  Thai Phan was an Industrial Design Engineer. He implemented the path planning, communication, and virtual environment described in Section 5.4.4. I implemented support for the Crazyflie quadrotors. Experiments were executed jointly.

5.1 Related Work

While we will refine this definition in the following section, the first published definition of *Mixed Reality* (MR) was given by Milgram and Kishino as the merging of physical and virtual worlds [106]. In their definition, *Augmented Reality* (AR) and *Augmented Virtuality* (AV) are seen as special instances of MR. In Augmented Reality, virtual objects are projected onto the physical environment, while in Augmented Virtuality, physical objects are incorporated into a virtual environment.

AR and AV have been used to help overcome challenges faced in implementing robotic systems. AR systems have been implemented in the form of a video overlay feed for a multi-robot system to display state information and internal data [56, 92]. With the addition of an overhead camera to do tracking, the pose of the physical robots can be incorporated into a virtual environment. Such a system has been suggested for educational purposes or to simplify debugging between simulation and practical experiments [55].

An implementation closer to a true MR approach merges virtual and physical sensor readings, allowing a robot to sense physical and virtual objects at the same time [30]. This approach allows testing a physical robot in unknown environments and simplifies the addition of obstacles.

Another unique usage of MR is in robot teleoperation. Freund and Rossman report a system that allows a human operator to manipulate objects in a virtual environment, which are then translated and executed by a robot in a physical environment [54]. Another teleoperation system enables head-coupled virtual reality viewing [19]. Similarly, it is possible to use a MR approach for so called tele-immersive environments. Such a setup allows humans to collaborate even if they are in different physical spaces [176]. While this previous work discusses human-human interaction only, we will show that it is beneficial to extend this use-case specifically to include robots.

In Milgram and Kishino's broad definition of Mixed Reality, anything containing pure or virtual elements may be considered as MR. Prior work in MR focuses on solving specific subproblems by incorporating physical or virtual components to enhance a singular physical or virtual workspace. We propose instead that a Mixed Reality system should allow bidirectional feedback between multiple virtual and/or physical environments. This would allow combining the advantages of both virtual and physical environments in a real-time setting. By presenting several use-cases of an MR workflow, we will show the usefulness of such complete framework in robotics research.

5.2 Mixed Reality Components

Mixed Reality creates a space in which both physical and virtual elements co-exist, allowing for easy interaction between the two. Rather than one space being secondary to another (like in Augmented Reality and Augmented Virtuality), our take on MR blurs the boundaries between environments, creating one larger space where components from both worlds can communicate in real-time. This enables elements in one world to react directly to what is happening in another via direct data communication as opposed to a reconfiguration or modification of existing components.

MR is often known by its applications in virtual reality. However, MR can be particularly useful in robotics applications, as it creates a platform that allows for flexible development and testing of control algorithms. Since users can select which elements are physical and which are virtual, a number of different experiments can be performed based on various constraints and environments.

We tighten Milgram and Kishino's broad definition of MR by specifying its properties and components. Following the definition of Augmented Reality by Azuma *et al.* [10], we define a *Mixed Reality system* as one that:

- combines physical objects in at least one physical environment and virtual objects in at least one virtual environment;
- runs interactively (often called real-time); and

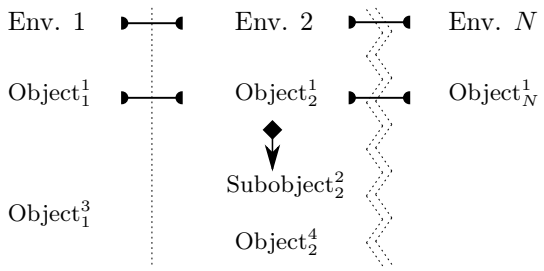


Figure 5.1: Mixed Reality merges different environments (physical and virtual). The environments and their objects are aligned with each other. In this case, Object^1 is at least partially available in three environments, e.g. the physical world and two virtual worlds. Environment 2 enhances Object^1 by adding a subobject (Subobject_2^2, e.g. a sensor). Object^3 and Object^4 are visible in their respective environments only.

- spatially maps physical and virtual objects to each other.

The relationship between objects can be shown schematically with the general form given in Fig. 5.1. We will use this schema to describe our applications. Lines symbolize links between objects in the different environments.

Note that the combination of objects can be in any of the two types of environments (physical or virtual) or both. Additionally, there is no restriction on the number of physical or virtual environments; for example, it is possible to bring together physical objects in distinct locations or to use several different virtual environments.

5.2.1 Physical Environment

The physical world in a MR space includes people, robots, sensors, obstacles, and other objects. Those objects are able to interact with each other and with elements from the virtual environment. The environments themselves can be protected, closed environments to accommodate physical elements that pose a safety hazard, or allow for the installation of special equipment (such as a motion-capture system) necessary to complete the physical-to-virtual communication link. Because of the direct communication of data between the physical and virtual environments, no changes or adjustments to physical-world components (e.g. the robot's camera and sensors) need to be made.

5.2.2 Virtual Environment

The virtual environment of a MR system can help overcome limitations of the physical environment and can be created using a variety of existing software such as robotic simulators or 3D game engines. Like the physical environment, the virtual environment can include robots and sensors. In addition, simulations of more complex objects are possible. Because MR makes direct interaction between the virtual and physical worlds possible, there is much flexibility on which elements exist in the physical world and which can exist in the virtual world. Components can be chosen to be physical or virtual based on user needs and convenience.

5.2.3 Physical-Virtual Interaction

The defining feature of MR is its ability to allow for direct interaction between physical and virtual environments as well as multiple physical environments in different locations. Achieving full interaction requires communication and synchronization between environments.

5.2.3.1 Physical To Virtual

An isotropic mapping function can be used to map physical environments to the virtual ones. This might be any homeomorphism, such as scaling. This requires knowledge of the pose of all physical objects, thus external localization or self-localization is needed.

5.2.3.2 Virtual To Physical

Synchronizing the virtual environments to the physical ones is useful to show the state of the virtual world in the physical environment. Additionally, it can be used to map invisible virtual entities, such as a virtual goal location, to a physical environment. In practice, this is much easier to realize than physical-to-virtual mapping, because the full pose information is already known.

5.3 Benefits of Mixed Reality

The interactive nature between different environments gives MR several advantages over AR or AV, including but not limited to the following.

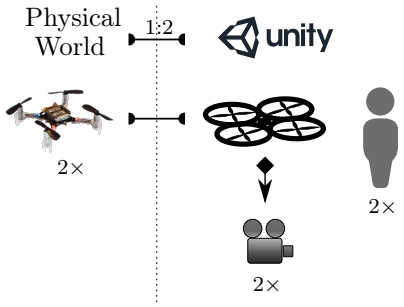
Spatial flexibility. Interaction between physical and virtual environments in MR allows experiments with robots to be performed remotely. This can expand collaboration between groups, as they are no longer limited by geographic constraints and can meet in a centralized virtual environment. Spatial flexibility also enables implementing scaled environments. A small physical environment can represent a large virtual one by using scaled coordinate transformation. Similarly, smaller robots might be used instead of larger ones. In combination with a motion-capture system, this approach can reduce the overall cost of the experiment.

Elimination of safety risks. With MR, safety issues typically associated with human-robot interaction can be resolved by separating them into distinct physical or virtual environments. In MR, physical robots can interact with virtual humans or physical humans in a different physical environment, eliminating concerns for human safety when errors occur. This can also be applied to experiments involving potentially dangerous interactions between robots. Because it allows interaction between multiple physical and virtual environments, MR creates a safer, lower-risk experiment space.

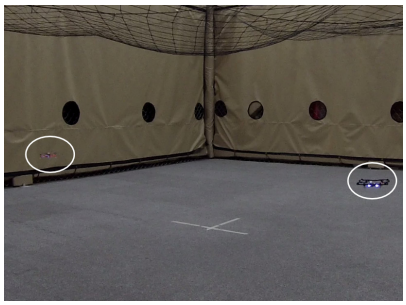
Simplification of debugging. The shared physical and virtual MR space reduces the gap between simulation and implementation. MR's virtual aspect allows for visualization of various states of the robots (e.g. field of view and planned path) so errors can be caught earlier. MR creates an enriched environment where all physical and virtual data interact directly in real-time; no further computation or calculation is necessary. This expedites and simplifies debugging.

Unconstrained additions to robots. A MR environment allows adding or changing virtual features of robots that may be too costly, time-consuming, or impossible in reality. For instance, one can add a virtual camera to a robot that is too small to carry one.

Scaling up swarms. Finally, MR simplifies experiments on robot swarms. Full interaction between the physical and virtual environments means most of the swarm can be simulated, as it may be sufficient for experiments to be performed on only a handful of real robots. This allows practical testing of swarm algorithms even if financial or spatial restrictions constrain the number of physical robots.



(a) MR Components



(b) Physical World

(c) Virtual World (Unity 3D)

Figure 5.2: Two virtual humans simulated by SmartBody are followed by two quadcopters. The quadcopters are flying in a motion-capture area and are synchronized with their virtual counterparts. This allows us to safely test algorithms while taking quadcopter dynamics into account.

5.4 Demonstrations

In the following we present several representative examples of using Mixed Reality in robotics research. In our demonstrations we use the Crazyflie 2.0, see Section 4.2 for a description of the platform.

Small size and low price (180 USD) make the Crazyflie an attractive option for research on multi-robot coordination. However, it has several limitations. Payload is limited to 15 g, which restricts sensors that can be added. The size also causes the Crazyflie to be more sensitive to external forces such as wind. Onboard processing abilities are also limited compared to larger research grade quadcopters such as the AscTec Hummingbird. These limitations make the Crazyflie a good candidate for algorithm prototyping in Mixed Reality.

Each of the following subsections will describe a practical use-case of Mixed Reality along with technical details on how to reproduce similar results. A video of the experiments is available at http://youtu.be/px9iHkA0nOI. To demonstrate the versatility of the approach, we use three different virtual environments: Unity 3D, v-rep, and Gazebo.

5.4.1 Human-Following UAVs

In this demonstration, we aim to develop a team of UAVs capable of following humans. Since human motion is complex and unpredictable, and UAVs are highly dynamic systems, the typical coarse-detail simulation is not expected to be very accurate. While pure simulation is a valuable tool as a proof of concept, new issues are often encountered when moving from simulation to reality. These issues may pose safety risks for humans sharing the environment with the robots.

Mixed Reality can add an intermediate step between simulation and practice. Note that while this demonstration uses virtual humans, it can also be applied to humans acting in a separate motion-capture space.

5.4.1.1 Technical Details

Typical robot simulators either do not support simulating humans (e.g. GAZEBO 2), or use a very simplified model (e.g. V-REP). We use the game engine UNITY 3D (4.6.2 Free Edition) [1], combined with the Virtual Human Toolkit [65] to simulate humans. This toolkit includes SMARTBODY [50], a character animation platform able to accurately simulate humans. The Behavioral Markup Language (BML) can be used to describe complicated behaviors, such as locomotion (including steering), lip-syncing, gazing, and object manipulation. SMARTBODY transforms those high-level descriptions into animations in real-time.

The scene in UNITY 3D contains two virtual humans and two quadcopters with forward-facing virtual cameras in an outdoor setting. Humans are controlled by SMARTBODY to follow a predefined path while obeying natural movement rules. To include more accurate quadcopter dynamics, we use two Crazyflies which fly in a space equipped with a 12-camera VICON MX motion-capture system. The robots' positions are tracked at 100 Hz, and a UNITY 3D script updates the position of the virtual quadcopters using the VRPN protocol [72]. To demonstrate the versatility of the approach, the virtual space is two times larger than the physical space ($5\,\text{m} \times 6\,\text{m}$), allowing the virtual humans to walk farther. Scaled coordinate transformations ($1 : 2$) between physical and virtual spaces align the objects. A simple controller takes the known positions of the quadcopter and the human to compute a goal position which keeps the human in the field of view. The Mixed Reality schema, virtual environment, and setup in the physical world are shown in Fig. 5.2.

5.4.1.2 Discussion

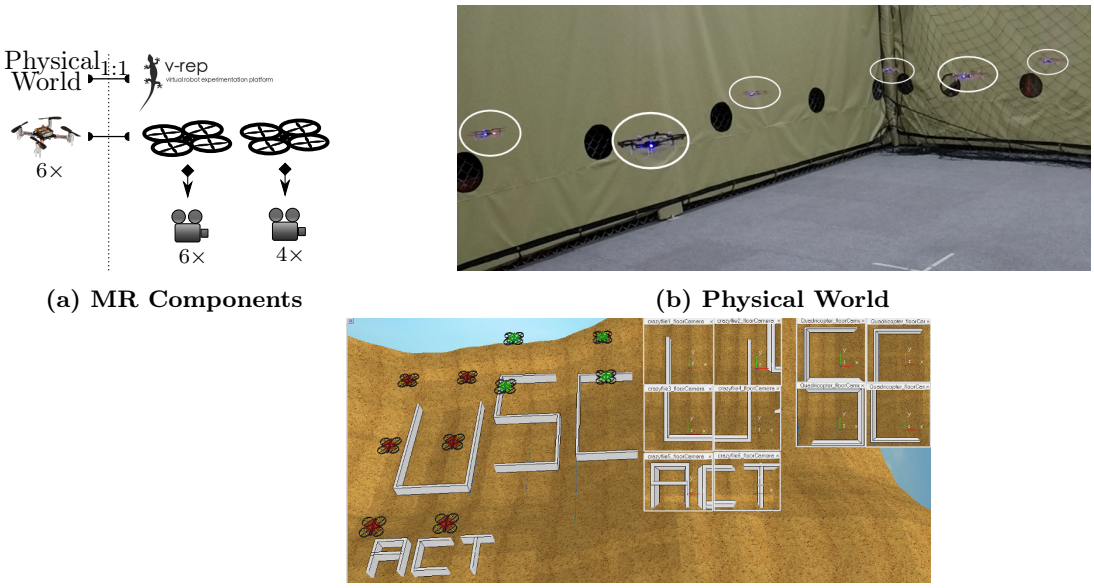
When humans and robots share environments, reducing the gap between simulation and practice can be crucial to ensure safety. For instance, the AscTec Hummingbird quadcopter can reach speeds up to 15 m/s. A crash at such high speeds can cause severe accidents, especially if the UAV operates close to humans as in the discussed surveillance application. Even if an algorithm works well in simulation, many added uncertainties in the physical world could put a human participant at risk. The MR approach reduces those risks by adding intermediate steps between simulation and realization. Additionally, MR allows the limited space of indoor motion-capture systems to be overcome by using a smaller robotics platform with added virtual sensors and by introducing scaling between the virtual and physical environments. For example, when outdoor UAV flight is restricted, outdoor components such as rocks, trees, or buildings can be fused with scaled indoor motion-capture data in a virtual environment. Thus, MR enables refining an algorithm in a safe and less restricted environment.

Similarly, it is possible to use actual humans in a separate physical environment rather than using virtual humans, combining the physical environments in a virtual one using a simulator. Such a multi-physical world approach provides safety and better approximation to reality compared to virtual simulations. This method can be used for different robots as well, improving collaboration between researchers. Furthermore, the possibility of scaling between different physical environments allows interaction between entities of different sizes.

5.4.2 Area Coverage using UAVs

For this demonstration, we are interested in testing an algorithm for a large swarm of robots when enough robots are not available or space does not allow so many to be used. We use area

[1] See http://unity3d.com



(a) MR Components

(b) Physical World

(c) Virtual World (V-REP)

Figure 5.3: A swarm of robots attempts to cover an area (outlined by the USC ACT letters). The six robots on the left are synchronized to a physical Crazyflie 2.0 swarm using a motion-capture system. The four robots on the right are simulated in V-REP. All downward facing cameras are virtual.

coverage with UAVs as the task with a simple centralized approach, where the goal position of each quadrotor is directly computed from the known position of the target and a fixed translation. This demonstration also conveys that the behavior of a physical robot swarm differs substantially from that of a simulated swarm.

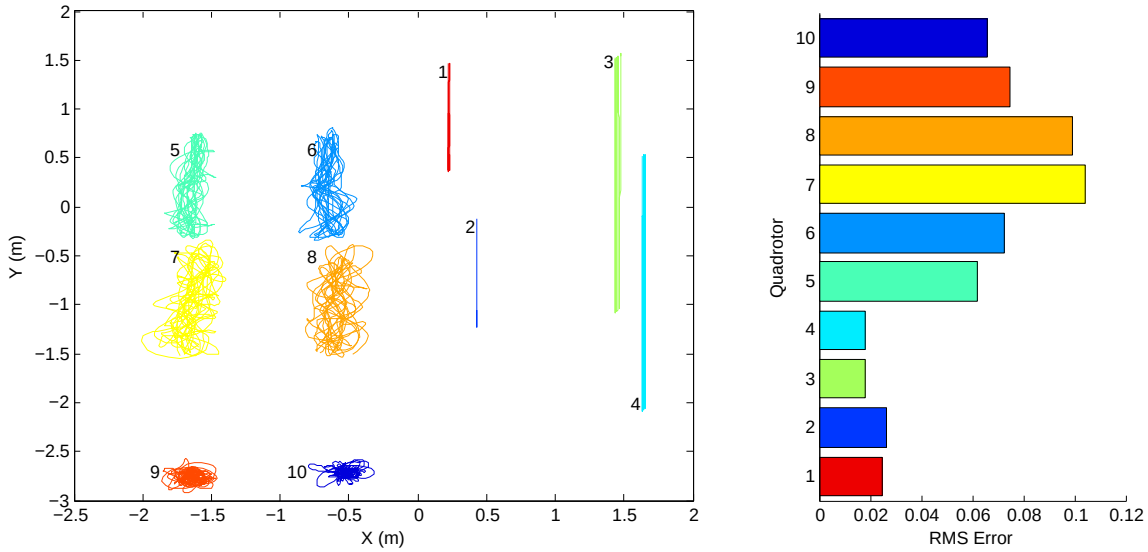
5.4.2.1 Technical Details

We fly six Crazyflies in our motion-capture arena; their pose is sent to V-REP (3.2.0) [124] over ROS. V-REP simulates four additional quadcopters using the included `Quadricopter` model and also simulates the desired area to cover. The area changes over time, thus the updated target pose for each quadcopter is computed by a child-script in V-REP based on the current area configuration. The target pose is visualized by red (physical robots) and green (simulated robots) spheres for each quadcopter in the virtual environment. The motion-capture space is mapped directly to the virtual environment (no scaling); the overall virtual space, however, is larger to accommodate the additional quadcopters. Screenshots from the demonstration and the MR schema are shown in Fig. 5.3.

5.4.2.2 Discussion

This approach showcases splitting a swarm of robots into virtual and physical components, which allows side-by-side comparison of the accuracy of simulation with real implementation.

For the given example, the physical quadcopters do not follow the target trajectory as smoothly as the simulated ones shown in Fig. 5.4, due to the added uncertainties of a real-world implementation and external forces such as air currents. Furthermore, connectivity issues and as a consequence increased latency in the control algorithm cause additional instabilities which are not part of a simulation. As a result, the physical quadcopters are unable to monitor their areas with the same accuracy as the virtual quadcopters.



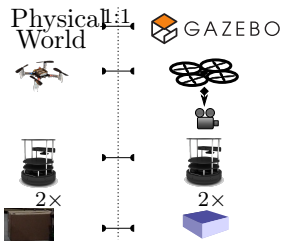
(a) Trajectory plot **(b) RMS Error**

Figure 5.4: Tracking performance of physical vs. virtual quadcopters. The six trajectories on the left in (a) were created by Crazyflies in the physical world, while the four on the right are simulation results. The root mean square (RMS) error between planned and actual position is much smaller for simulated quadcopters (b).

Since quadcopters are highly dynamic systems which are difficult to accurately simulate, especially as a team, this approach allows for motion comparisons between the quadcopters in the physical and virtual environments. Data collected from such an experiment can be used to design a model for a simulated quadcopter that is specific to the unique test environment the quadcopter will face. For instance, in order to simulate a swarm of UAVs under strong wind conditions, a wind model would be required. Tuning such parameters in an MR setting is simpler as the difference between model and reality is clearly visible. Once the simulation matches the physical robots, a larger swarm can be simulated by using both the physical robots and virtual robots using the newly tuned model, which can save money and time. This also allows for swarm algorithms to be tested in a simulation that more closely depicts the actual physical motions of a robot, prior to implementing the algorithms on their real counterparts. A similar approach could be used to identify limits of the simulation, for example by analyzing the behavior of UAVs flying in close proximity to each other.

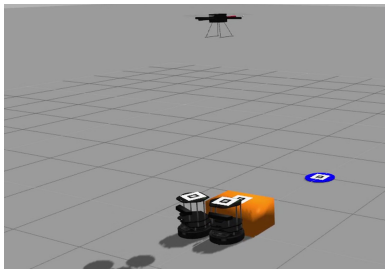
5.4.3 Object Moving with Limited Localization

In this demonstration, two robots without onboard vision must push a large box with the help of the overhead quadcopter. In environments that are GPS-denied and not equipped with motion-capture infrastructure, a camera-equipped UAV can be used to identify and track robots using special markers (so-called AR tags) and vision processing (see [67]). Based on the locations of the robots and the box, which are computed with vision processing from the overhead camera view, the robots drive toward the box to move it. While this is a simple scenario, it demonstrates the capabilities of the MR approach for testing vision-based approaches without having the required set of hardware. It also shows an example of a step-by-step transition from pure simulation to MR.

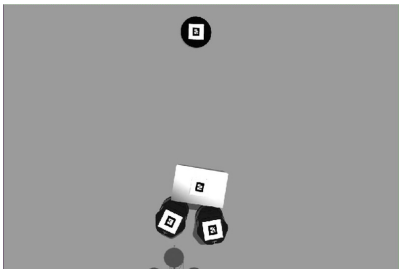


(a) MR Components

(b) Physical World



(c) Virtual World (Gazebo)



(d) Virtual Camera

Figure 5.5: Two TurtleBots collaboratively move a box using the guidance of a physical UAV with a virtual downward facing camera simulated with Gazebo (a). The pose of the robots and the box are estimated using AR tag detection.

5.4.3.1 Technical Details

We implemented the scenario in simulation using Gazebo 2.2.3 [85] and ROS Indigo by incorporating `turtlebot`, `hector_quadrotor`, and `ar_track_alvar` packages. After successful simulation, Mixed Reality testing was performed in stages.

In the first stage, we replace the simulated quadcopter by a Crazyflie in the physical environment, while keeping the camera simulated by Gazebo. Note that the 6 degree-of-freedom pose of the quadcopter captured by a motion-capture system is used in positioning the virtual camera and thus the camera angles are accurate. This allows analyzing the behavior of the algorithm while including more realistic dynamics and external influences such as wind.

In the second stage, we replace the virtual TurtleBots with physical ones (TurtleBot 2) while leaving the box in the virtual environment. The TurtleBots are localized using the external motion-capture, however the control algorithm is based solely on the virtual camera image. This introduces new uncertainties caused by the differences in the PID controller used, uneven floor, and other effects which are not considered in simulation. Similar to the previous demonstration, it is possible to have one physical and one simulated TurtleBot to compare the different behaviors side-by-side.

Finally, we introduce a physical box to include the additional forces created by the box movement. Screenshots and MR schema of this particular experiment are shown in Fig. 5.5.

5.4.3.2 Discussion

This experiment shows how Mixed Reality can help reduce the gap between simulation and implementation. The approach can be used to break the implementation into more controllable stages, where it may be easier to analyze and isolate failures. For example, a staged approach can be used to debug a particular issue by moving objects into a reproducible virtual environment and thus isolate a component that causes a failure in the physical environment.

In software engineering, such virtualized objects, known as mock objects, play a crucial role in development and testing [149]. In particular, moving physical objects to the virtual environment solves the same problems as mock objects do in software engineering, such as introducing deterministic behavior, improving execution time, or simplifying testing to isolate bugs.

5.4.4 Mixed Reality Collaboration between Human-Agent Teams

There is a need for a simulation testbed that allows multiple labs to conduct research cooperatively without the transportation of specialized equipment and resources between locations. We combine two ideas to simplify the collaboration of humans and physical entities, even when geographically dispersed: synthetic prototyping and Mixed Reality. Early synthetic prototyping [138] demonstrated that it is possible to use a game engine to play and interact with vehicle dynamics in a driving simulator, without the support of additional physical entities. Here, we use an MR approach that uses UNITY 3D to construct a virtual environment that combines multiple physical locations containing physically embodied entities such as humans and robots. Humans wear VR headsets to perceive other physical and virtual entities. We demonstrate our approach in a collaborative scenario, where humans, collocated in two physical locations, have to move through a narrow doorway. They are followed by autonomous drones and all entities must avoid physical and virtual obstacles while moving.

5.4.4.1 Technical Details

We consider two physical spaces, one with 2 users and 4 drones and the other with 1 user and 2 drones. One physical space has a physical door, while the other does not (see Fig. 5.6). Users in both physical spaces wear a Samsung Gear VR headset and are in the same virtual environment with the same door represented virtually. Each user has a pair of drones following in a simple line formation. The virtual environment also contains virtual obstacles such as walls, human avatars, and representations of the drones. As each user passes through the open door, the drones must break formation in order to pass through also.

For motion tracking, one space uses PhaseSpace and the other uses VICON. The major system components are outlined in Fig. 5.7. We use UNITY 3D for the server and clients, for rendering, path finding, and state synchronization. We use the Crazyswarm stack (see Chapter 4) to control the flight of Bitcraze Crazyflie 2.0 quadcopters. Each of the two locations uses a computer (base station) running ROS (Robot Operating System). ROS BaseStation A commands 4 Crazyflies while BaseStation B commands 2 Crazyflies. All VR headsets act as Wireless UNITY 3D Clients which connect to the the UNITY 3D Server.

Network Design An intracampus WAN allows our two tracking volumes to connect to each other with minimal number of hops. UNITY 3D provides High Level (HLAPI) and Low Level APIs (LLAPI) for networking, both of which are built on top of a transport layer, which uses UDP packets[2]. We implement a server-client architecture using the UNITY 3D game engine's High Level API (HLAPI), because it provides the core network components necessary to allow UNITY 3D clients to relay data to each other through the UNITY 3D server. One computer runs the UNITY 3D Server and UNITY 3D Client A in a single process. UNITY 3D Client A receives pose information of the PhaseSpace motion-tracked users and drones on its local LAN. It also receives pose information of all other users and drones via the UNITY 3D Server. UNITY 3D Client B receives pose information of the VICON motion-tracked users and drones on its LAN. It in turn receives pose information of the PhaseSpace-tracked users and drones via the UNITY 3D Server.

[2]See https://docs.unity3d.com/Manual/UNetUsingTransport.html

(a) Location A with physical door using PhaseSpace tracking.

(b) Location B using VICON tracking.

Figure 5.6: Photos of physical spaces.

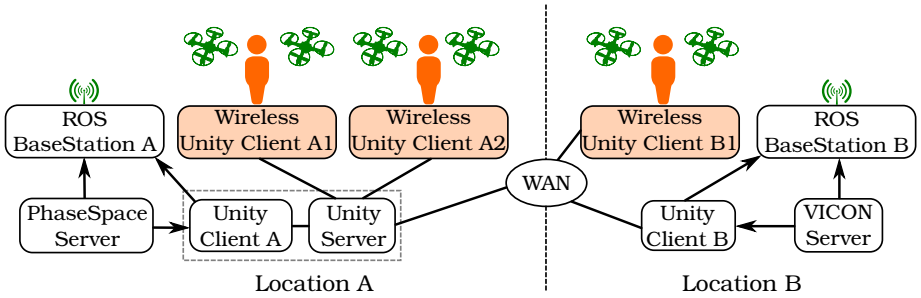


Figure 5.7: System diagram. Location A has 2 humans and 4 drones, while location B has 1 human and 2 drones. Humans and drones are tracked using PhaseSpace and VICON, respectively, and their poses are shared with the UNITY 3D Server.

The Wireless UNITY 3D Clients only receive pose information for all users and drones in order to render their correct placement inside the virtual environment. They do not transmit user data or input to the UNITY 3D Server. Both UNITY 3D Client A and B process the pose information locally. The UNITY 3D Server itself does not process the pose information as it simply updates all clients with the most current information.

For closed-loop control, UNITY 3D Client A sends a new goal position for each drone (at a rate of 10 Hz) via a Python TCP socket connection to ROS BaseStation A. Likewise, UNITY 3D Client B sends goal data to ROS BaseStation B. Both base stations communicate and command drones via a custom RF radio on the 2.4 GHz band and not through 802.11 WiFi. Wireless UNITY 3D Clients communicate on IEEE 802.11ac to reduce congestion on the 2.4 GHz band.

Motion Tracking All of the Gear VR headsets are outfitted with unique marker arrangements for 6DoF tracking; using PhaseSpace active LED markers or VICON passive retro-reflective markers.

The 4 drones tracked by PhaseSpace are modified to carry a PhaseSpace microdriver with 2 tracking LEDs, drawing 3.7 V from the drone's battery. UNITY 3D Client A only receives partial pose data from these drones – their yaw and position.

The 2 drones tracked by VICON are outfitted with small spherical markers adhered directly to the frame. The marker arrangements are unique, so UNITY 3D Client B receives full 6DoF pose

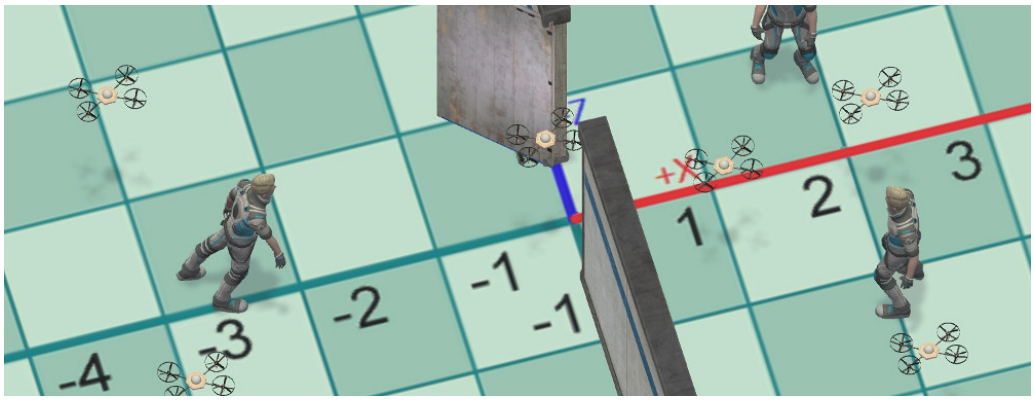


Figure 5.8: Overhead view of 3 users and 6 agents in total, sharing the same virtual environment.

data. UNITY 3D Client A uses the PhaseSpace SDK, while UNITY 3D Client B connects to VICON's integrated VRPN server. Data is unbuffered and only the most recent data is used.

Agent Navigation UNITY 3D Clients A & B calculate the paths for the drones by representing each one as an agent inside the virtual environment. UNITY 3D's navigation system uses the A* search algorithm[3]. This provides suitable reactive planning to avoid collisions with users, obstacles, and other agents. It also simplifies the maneuvers of the drones for the purpose of flying in close proximity with humans, each having an invisible 0.5 m radius boundary against collisions.

UNITY 3D generates a navigation mesh (NavMesh) for the environment in which paths for agents will be calculated on a 2D plane. Paths change as dynamic obstacles such as users, the door, and other agents cause obstructions. When an agent has no available path to reach a goal position, the agent stops moving until an obstruction is moved or a new path becomes available.

UNITY 3D Clients A & B will have identical local copies of the NavMesh, but will calculate paths independent of each other. If one of the client's network connection is interrupted, the other client will calculate paths using the most recent pose information reported by the UNITY 3D server. No client-side prediction is performed. Finally, invisible boundaries surrounding the virtual environment prevent drones in both tracking volumes from flying into untrackable areas.

Mixed Reality Design The physical door (2040 mm × 920 mm) is also tracked with PhaseSpace markers (see Fig. 5.6). As the door opens and closes, its virtual representation moves accordingly. One user wears a hand strap with PhaseSpace markers. When this user reaches for the doorknob, others see his/her avatar reach accordingly inside the virtual environment. We are also using an off-the-shelf solution for inverse kinematics (by Root-Motion[4]) to animate avatars as they locomote. A screenshot of the virtual environment (as rendered in the UNITY 3D Clients) is shown in Fig. 5.8.

The Wireless UNITY 3D Clients run on Samsung Galaxy S7 phones. Each client renders a stereoscopic view without distortion correction using a simplified rendering pipeline to prevent the phones from overheating. In order to give users improved spatial awareness, a HUD was designed to present them with an overhead view of their surroundings, indicating the location of other agents and users.

[3]See https://docs.unity3d.com/Manual/nav-InnerWorkings.html
[4]See http://root-motion.com/

5.4.4.2 Discussion

We have shown that Unity 3D's' multiplayer networking and AI architecture can facilitate mixed reality collaboration between human-agent teams. Using our approach, a wide variety of physical spaces can be interconnected for collaboration. Humans can work safely within their own physical confines, while robots might operate in more hazardous environments. This also enables the testing of heterogeneous human-agent teams. For example, UAVs might operate in a facility equipped to simulate air turbulence, while anthropomorphic robots might operate in a mock disaster zone.

5.5 Remarks

Through discussion of its novel advantages and three compelling use-cases for the approach, we have shown that Mixed Reality can provide many advantages for research and development in robotics. Furthermore, as one specific example of MR, we discuss how to use small UAVs such as the Bitcraze Crazyflie 2.0 instead of bigger and more expensive quadcopters, with practical tips for how to use different robotics simulation tools with Mixed Reality.

Our demonstrations show that it is possible to use Mixed Reality with typical, well-known robotics simulators as well as less familiar ones. More traditional robotics simulators such as Gazebo or v-rep can be used to first simulate algorithms and then, using the same platform, test intermediate steps using Mixed Reality. Unity 3D, combined with SmartBody, can be used to accurately simulate humans, creating a safer intermediate step for testing algorithms for robots in environments shared with humans without the typical risks. Furthermore, the Mixed Reality approach allows for early discovery, isolation, and correction of potential implementation issues, such as wireless communication dropouts. In addition, Mixed Reality enables experimenting with more robots than available or physically capable, not only as pertaining to virtual swarms as we have demonstrated, but also with physical robots in different locations. This expands the possibility of collaboration between different labs with different robotics hardware. Finally, Mixed Reality enables relying on cheaper, or in the case of robot swarms, fewer robots for initial experiments by simulating additional hardware, or sensors that cannot be easily installed on existing robot hardware.

Mixed Reality also permits a large amount of flexibility for testing environments. The choice of which objects are physical and which are virtual can be made by the user based on the experiment's needs. Additionally, it is easy to move physical objects to virtual ones and, given available hardware (e.g., the correct robots), vice versa. The different simulation platforms for virtual environments have different capabilities, which can be combined using multiple virtual environments.

While the Mixed Reality approach clearly has many advantages, it does have some limitations. First, the linking of virtual and physical objects requires localization. This can be a self-localization method such as SLAM, which requires in many cases a lot of sensors and onboard computation, or external localization, such as a motion-capture system. Second, a partially simulated environment does not have the same properties of an entirely physical environment; for example, virtual cameras may not show distortion or be affected by lighting conditions, and physical robots that interact with a virtual objects do not receive force feedback from the object. Therefore, as in pure simulation, a successful Mixed Reality experiment does not guarantee a working system in the physical world. The goal, however, is to add intermediate steps between pure simulation and full implementation.

In future work, issues related to scaling between virtual and physical environments should be addressed. In our demonstrations we used nano quadcopters to simulate bigger ones. Although we showed that this is more realistic compared to pure simulation, the behavior of the bigger drone would be different. For example, reaction to wind is highly dependent on the size of the UAV, which is amplified if scaling up between virtual and physical environments: while the scale allows flight in a smaller space, it also amplifies the noise. Furthermore, certain control algorithms such as trajectory following might require non-trivial changes (e.g., velocity adjustments) to accommodate the change in scaling.

In summary, Mixed Reality testing is one step closer to accurate implementation of a robotic system in the physical world, and can be an extremely valuable tool for research and development in robotics, particularly for multi-robot systems or robots which share environments with humans. We use Mixed Reality to verify our approach in Chapter 10.

Part II

Motion Planning for Static Environments

CHAPTER 6

Task and Path Planning for Agents

We now formally define standard MAPF problems and briefly describe commonly used AI solvers for each of them. We then generalize those problems in two ways. First, we present MAPF with Optimal Task Assignment (MAPF-TA) and efficient solvers. Second, we present MAPF with generalized conflicts (MAPF/C), which allow us to take the physical extent of robots into account and avoid collisions on any roadmap. We also present theoretical hardness results and solvers for MAPF/C that are based on solvers for the standard version.

6.1 Background

The standard problems rely on a undirected search graph representing the environment $\mathcal{G}_E = (\mathcal{V}_E, \mathcal{E}_E)$ with unit-length edges[1]. The set of N agents is $\{1, \ldots, N\}$. Let u_t^j be the vertex that agent j is planned to occupy at timestep t.

Definition 6.1.1. *A path $p^j = [u_0^j, \ldots, u_{T^j}^j, u_{T^j+1}^j, \ldots]$ for agent j is **valid** if and only if the following conditions hold.*

1. *Every action is either a move action along an edge or a wait action, that is, for all $t \in \{0, 1, \ldots, T^j - 1\}$, $(u_t^j, u_{t+1}^j) \in \mathcal{E}_E$ or $u_t^j = u_{t+1}^j$; and*
2. *Agent j remains at a vertex g^j, that is, there exists a minimum finite T^j such that, for each $t \geq T^j$, $u_t^j = g^j$.*

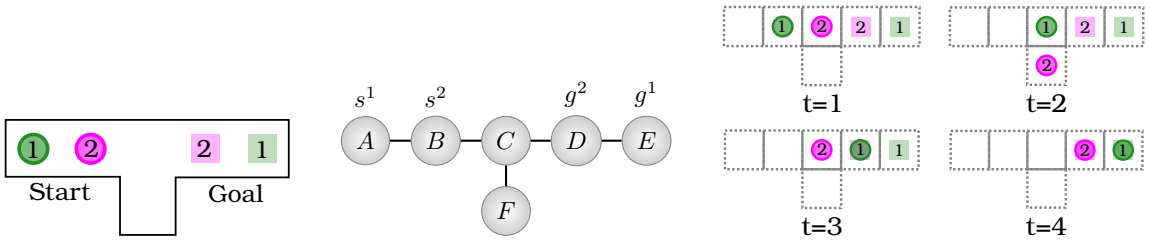
Definition 6.1.2. *The output of a solver is a **discrete plan**, which is a set of valid paths (one path for each robot) such that the following conditions hold.*

1. *There is no vertex collision between any two agents, that is, for all timesteps t and all agent pairs $j \neq k$ we have $u_t^j \neq u_t^k$; and*

Parts of this chapter appeared in:

- **Wolfgang Hönig**, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Conflict-Based Search with Optimal Task Assignment". In: *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. Software available at https://github.com/whoenig/libMultiRobotPlanning. 2018, pp. 757–765. URL: http://dl.acm.org/citation.cfm?id=3237495
- **Wolfgang Hönig**, James A. Preiss, T. K. Satish Kumar, Gaurav S. Sukhatme, and Nora Ayanian. "Trajectory Planning for Quadrotor Swarms". In: *IEEE Transactions on Robotics, Special Issue on Aerial Swarm Robotics* 34.4 (2018), pp. 856–869. DOI: 10.1109/TRO.2018.2853613

[1] Some MAPF solvers can work on directed graphs as well.



(a) Example environment. **(b) Undirected search graph $\mathcal{G}_E$.** **(c) Discrete output states.**

Figure 6.1: Example of a labeled MAPF instance. A makespan and flow time optimal solution is $\{p^1 = [A, B, C, D, E], p^2 = [B, C, F, C, D]\}$ with makespan $T = 4$ and flow time 8.

2. *There is no edge collision between any two agents, that is, for all timesteps t and all agent pairs $j \neq k$, it is not the case that both $u_t^j = u_{t+1}^k$ and $u_{t+1}^j = u_t^k$.*

Definition 6.1.3. *The **makespan** $T = \max_j T^j$ of a discrete plan is the earliest timestep when all agents have reached their goal vertices and remain there. In contrast, the **flow time** $\sum_j T^j$ of a discrete plan is the sum of the individual travel times.*

In the following, we will shorten the notation of a path using $p^j = [u_0^j, \ldots, u_T^j]$ instead of $p^j = [u_0^j, \ldots, u_T^j, u_{T+1}^j, \ldots]$.

6.1.1 Labeled Multi-Agent Path Finding

The labeled MAPF problem can be described informally as follows. Given a graph with vertices, each corresponding to locations, and edges, each connecting two different vertices (that correspond to passages between locations in which robots cannot pass each other), and a set of agents with assigned start and goal vertices, find collision-free paths for the agents from their start to their goal vertices (where the agents remain) that minimize the makespan or flow time. At each timestep, an agent can either wait at its current vertex or traverse a single edge. Two agents collide when they are at the same vertex at the same timestep or traverse the same edge at the same timestep in opposite directions.

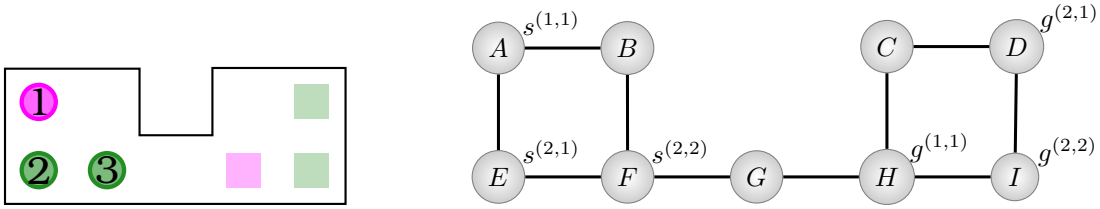
We use the following definitions to formalize the labeled MAPF problem. The search graph is $\mathcal{G}_E = (\mathcal{V}_E, \mathcal{E}_E)$, the start vertex of agent j is $s^j \in \mathcal{V}_E$, and its goal vertex is $g^j \in \mathcal{V}_E$.

Definition 6.1.4. *A discrete plan is **labeled MAPF valid** if and only if the following conditions hold for all j:*

1. *Agent j starts at its start vertex, that is, $u_0^j = s^j$; and*
2. *Agent j ends at its goal vertex and remains there, that is, there exists a minimum finite T^j such that, for each $t \geq T^j$, $u_t^j = g^j$.*

A valid labeled MAPF plan might be optimized with respect to makespan or flow time. An example is shown in Fig. 6.1. Suboptimal solutions to the labeled MAPF problem can be found in polynomial time [123, 86]. However, finding an optimal solution is NP-hard [144, 180, 98], for all common optimization objectives.

Two common approaches for solving labeled MAPF instances are variants of M* [157] and Conflict-Based Search (CBS) [131]. The core idea of M* is to plan for the robots individually as much as possible. Whenever a conflict occurs, the conflicting robots are joined together as a meta-agent. In contrast, Conflict-Based Search methods try to resolve conflicts one-by-one,



(a) Example environment. **(b) Search graph $\mathcal{G}_E$.**

Figure 6.2: Example of a multi-color MAPF instance with two groups. A makespan-optimal solution for this instance is $\{p^1 = [A, B, F, G, H], p^2 = [E, F, G, H, I], p^3 = [F, G, H, C, D]\}$ and has makespan $T = 4$.

starting with a conflict occurring at the earliest timestep. In both methods, the search might grow exponentially in the worst case.

CBS has several extensions that allow solving instances with hundreds of robots quickly. Improved CBS combines ideas from M* and CBS by merging multiple agents to meta-agents during the conflict resolution [20]. Enhanced CBS (ECBS) is a bounded w-suboptimal version that can approximate optimal labeled MAPF plans with a factor of w [11]. Edges on the underlying roadmap can be annotated with so-called *highways* that guide the robots and can help to find solutions much quicker [34]. The highways can be provided by a user or learned automatically [35].

6.1.2 Unlabeled Multi-Agent Path Finding

If agents are interchangeable, it might be beneficial to solve the task assignment and path planning simultaneously.

Let $s^j \in \mathcal{V}_E$ the start vertex for agent j and $g^1, \dots, g^N \subseteq V$ the set of goal vertices.

Definition 6.1.5. *A discrete plan is **unlabeled MAPF valid** if and only if the following conditions hold for all j:*

1. *Agent j starts at its start vertex, that is, $u_0^j = s^j$; and*
2. *Agent j ends at a unique goal vertex and remains there, that is, there exists a minimum finite T^j such that, for each $t \geq T^j$, $u_t^j = g^{\phi(j)}$ where ϕ is a permutation of $1, \dots, N$.*

It is possible to find a makespan-optimal solution to a unlabeled MAPF problem by reducing it to a max-flow problem [178].

6.1.3 Multi-Color Multi-Agent Path Finding

Labeled and unlabeled MAPF can be generalized to the multi-color MAPF problem[2]. We are given a team of agents partitioned into groups, where agents in the same group are interchangeable with each other. A formation specifies the locations occupied by each group without regard for which agent in a group occupies which location meant for the group. In marching bands, for example, all flute players are interchangeable with each other in the sense that it does not matter which location of the goal formation one of the flute players occupies as long as it is meant for a flute player. The objective of multi-color MAPF is to find collision-free paths that move all agents from a given start formation to a given goal formation, often minimizing makespan, see Fig. 6.2 for an example.

Labeled MAPF is a specialization of multi-color MAPF in which all groups have cardinality one (that is, no agents are interchangeable with each other and thus every agent is assigned a specific

[2]This is also sometimes called the target-assignment and path-finding problem (TAPF) [95].

location in the goal formation). Similarly, unlabeled MAPF is a special case of multi-color MAPF in which there exists only a single group. Solving a formation-change problem thus consists of two parts, namely assigning each agent a unique location in the goal formation meant for its group (called target assignment) and multi-agent path-finding for the resulting target assignments.

The N agents are partitioned into R agent groups $\{\text{group}^1, \text{group}^2 \dots \text{group}^R\}$, where group^i consists of R^i agents $\{a^{(i,1)}, a^{(i,2)}, \dots, a^{(i,R^i)}\}$ that are interchangeable with each other – for a total of $N = \sum_{i=1}^{R} R^i$ agents. Each agent $a^{(i,j)}$ has a unique start location $s^{(i,j)} \in \mathcal{V}_E$, and there are N unique goal locations. The goal locations are split into disjoint sets; each group group^i has a set of R^i unique goal locations $g^i = \{g^{(i,1)}, g^{(i,2)}, \dots, g^{(i,R^i)}\}$. A solution to a multi-color MAPF instance consists of a target assignment that assigns each agent a unique goal location meant for the same group (given by N one-to-one mappings from agents in a group to targets in the same group, one for each group) and a valid path from its start location to its goal location that avoids collisions with other agents.

Definition 6.1.6. *A discrete plan is* ***multi-color MAPF valid*** *if and only if the following conditions hold for all* (i, j)*:*

1. *Agent* $a^{(i,j)}$ *starts at its start vertex, that is,* $u_0^{(i,j)} = s^{(i,j)}$*; and*
2. *Agent* $a^{(i,j)}$ *ends at a goal vertex for its group and remains there, that is, there exists a minimum finite* $T^{(i,j)}$ *such that, for each* $t \geq T^{(i,j)}$, $u_t^{(i,j)} \in g^i$.

Multi-color MAPF can be solved optimally with respect to makespan using the *conflict-based min-cost-flow algorithm* (CBM) [95].

6.2 Multi-Agent Path Finding with Optimal Task Assignment

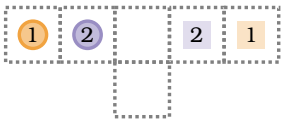
We now generalize multi-color MAPF to allow arbitrary assignment matrices. We are motivated by warehouse automation, where robots might be used to deliver shelves to pack stations [175]. In this domain, robots can initially choose which shelf to pick, but then the shelf must be moved to a specified station. Thus, a robot's task is partially unlabeled (any shelf can be picked) and partially labeled (the shelf needs to be delivered to a specified goal) at the same time. The objective in this domain is to minimize the idle time of human workers.

In the following, we extend Conflict-Based Search (CBS) [131] (and some of its variants) to simultaneously assign tasks and find paths for agents even for hybrid MAPF problem instances. Our approach is conceptually similar to a method which combines task reassignment and path planning in the M* framework [158]. However, our method works with an arbitrary assignment matrix not requiring that the number of tasks and robots is identical, and shows better scalability to large teams in particular when using our bounded suboptimal solver ECBS-TA.

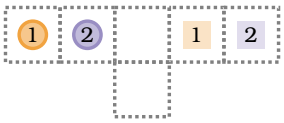
Another hybrid solver is the Conflict-Based Min-Cost-Flow (CBM) algorithm [95] that minimizes the makespan, which does not map well to minimizing idle time, where the sum of all costs is a better metric. We provide representative experimental results that demonstrate that our method, called CBS-TA, outperforms the naive (yet frequently used) approach of solving task assignment and path planning independently in dense environments. Finally, the CBS framework allows us to extend our approach to ECBS, a bounded suboptimal MAPF solver. We introduce our bounded suboptimal algorithm, ECBS-TA, and compare its solution quality and runtime to existing solutions.

6.2.1 Problem Definition

There are N agents at different start locations $s^i \in \mathcal{V}_E, i \in \{1, 2, \dots, N\}$. The set of M potential goal locations is $\{g^1, \dots, g^M\}$. The binary $N \times M$ matrix A indicates whether an agent can be



(a) Robot 2 must move into the alley temporarily, resulting in a total cost of 8.



(b) Both robots can move along their direct path without collision, resulting in a total cost of 6.

Figure 6.3: Example where task assignment and path planning cannot be decoupled for the optimal solution. Start positions are circles and goals are squares.

assigned to a specified goal: the entry a_{ij} is 1 if agent i is allowed to reach goal g^j and 0 otherwise. A solution to a MAPF with Optimal Task Assignment (MAPF-TA) instance consists of a target assignment that assigns each agent a unique goal location (given by N one-to-one mappings from agents to targets) and a valid path from its start location to its goal location that avoids collisions with other agents.

Definition 6.2.1. *A discrete plan is **MAPF-TA valid** if and only if the following conditions hold for all i:*

1. *Agent a^i starts at its start vertex, that is, $u_0^i = s^i$;*
2. *Agent a^i ends at one of its potential goal vertices and remains there, that is, there exists a minimum finite T^i such that, for each $t \geq T^i$, $u_t^i \in g^j$ and $a_{ij} = 1$*[3]*.*

Here, we consider a solution optimal if the sum of individual path costs (SIC) is minimized. The CBS framework requires that agents move in unit time steps. We want to optimize the total time and therefore we minimize $\sum_{i=1}^{N} T^i$.

This problem definition is identical to the traditional MAPF problem with the exception that we introduce the potential assignment matrix A. In case of $N = M$ and A being a permutation matrix (i.e., A contains exactly one 1 in each row and column, and all other elements are 0), our problem is identical to non-anonymous or labeled MAPF. In case of $N = M$ and $A = \mathbf{1}$, the problem is identical to the anonymous or unlabeled MAPF case. Our formulation also allows cases where there are more goals than agents, more agents than goals, or not all agents can reach all goals.

6.2.2 CBS-TA

A typical approach for task assignment and path planning is to separate them into two stages. However, both problems are tightly coupled, and certain task assignments may result in fewer collisions during path planning (see Fig. 6.3 for an example). To find an optimal solution, a naive approach would be to generate all possible assignments and solve the path planning for each of those assignments. However, there are $\binom{M}{N}$ assignments for N robots and M goals (assuming $M \geq N$), making this approach infeasible in practice. Instead, we generate an additional assignment on demand once we know that this assignment needs to be considered for the optimal solution, similar to an approach discussed for M* [158].

6.2.2.1 Algorithm

We start by briefly describing Conflict-Based Search (CBS), which we extend to incorporate task assignment. CBS is a two-level search. The low-level constructs paths for each individual agent given constraints provided by the high-level. The high-level finds conflicts (in our case, collisions) and

[3] If there is no assigned goal, we only require agent i to remain at some location eventually.

resolves them at their earliest start time. Conflict resolution works by adding two successor nodes in the high-level search tree and introducing an additional constraint for each agent participating in the conflict at the lower level. CBS is complete and optimal with respect to the sum of the cost of all agents [131].

Algorithm 6.1: high-level of CBS-TA

```
   Input: Graph, start and goal locations, assignment matrix
   Result: optimal path for each agent
 1 R.constraints ← ∅
 2 R.assignment ← firstAssignment()
 3 R.root ← True
 4 R.solution ← find individual paths using low-level()
 5 R.cost ← SIC(R.solution)
 6 insert R to OPEN
 7 while OPEN not empty do
 8     P ← best node from OPEN // lowest solution cost
 9     Validate the paths in P until a conflict occurs.
10     if P has no conflict then
11         return P.solution // P is goal
12     if P.root is True then
13         R ← new node
14         R.constraints ← ∅
15         R.assignment ← nextAssignment()
16         R.root ← True
17         R.solution ← find individual paths using low-level()
18         R.cost ← SIC(R.solution)
19         insert R to OPEN
20     Conflict ← (a_i, a_j, v, t) first conflict in P
21     for agent a_i in Conflict do
22         Q ← new node
23         Q.constraints ← P.constraints + (a_i, s, t)
24         Q.assignment ← P.assignment
25         Q.root ← False
26         Q.solution ← P.solution
27         Update Q.solution by invoking low-level(a_i)
28         Q.cost ← SIC(Q.solution)
29         Insert Q to OPEN
```

For CBS-TA we only need to change the high-level search; see Algorithm 6.1. Lines that were changed compared to CBS (Algorithm 1 in [130]) are highlighted. In CBS-TA, each high-level node has two additional fields: *root* describes if the current node is a root node and *assignment* describes the current task assignment which is used during the low-level search. CBS builds a search tree with a single root node. In comparison, CBS-TA creates a search forest, but expands new root nodes only on demand. CBS-TA starts with a single root node which uses the best task assignment, while ignoring possible conflicts between agents. Whenever a root node is expanded during the search, we create another root node with the next best assignment.

By design, CBS-TA requires an efficient way of computing the next-best assignment. It is possible to enumerate the K best solutions in various domains, including task assignment [44]. We base our method on existing algorithms [110, 29] but compute new solutions on demand, rather

than a set of K solutions. Our notation is closely based on prior literature [29]. We compute a lower bound of the cost for agent i to reach goal g^j (if $a_{ij} = 1$) by computing the shortest path, ignoring all other agents. A helper function $assignment(C)$ computes an optimal assignment for a given cost matrix C; this can be achieved, for example, by the Hungarian method [87] or flow-based approaches [2]. We introduce a new function $constrainedAssignment(I, O, C)$, where I is the set of assignments that must be part of the solution, O is the set of assignments that cannot be part of the solution, and C is the cost matrix. This function can be implemented as follows. First, we compute another cost matrix C' such that C' is identical to C, except that we change the cost to 0 for each entry in I and to infinity for each entry in O. Second, we execute any optimal assignment algorithm (e.g., the Hungarian Method) using C'. The pseudo code of our next-best assignment functions are shown in Algorithm 6.2 and Algorithm 6.3.

Algorithm 6.2: firstAssignment

Input: cost matrix C
Result: best assignment, initial ASG_OPEN
$R \leftarrow$ new node
$R.O \leftarrow \emptyset$
$R.I \leftarrow \emptyset$
$R.solution = constrainedAssignment(R.I, R.O, C)$
Insert R to ASG_OPEN
return $R.solution$

Algorithm 6.3: nextAssignment

Input: cost matrix C, ASG_OPEN
Result: next best assignment, updated ASG_OPEN
$P \leftarrow$ best node from ASG_OPEN // *lowest solution cost*
if P *does not exist* **then**
    return No next assignment
for $i \leftarrow 1$ **to** N **do**
    if i *not part of* $P.I$ **then**
        $Q \leftarrow$ new node
        $Q.O = P.O \cup \{P.solution[i]\}$
        $Q.I = P.I \cup \{P.solution[j] : j < i\}$
        $Q.solution = constrainedAssignment(Q.I, Q.O, C)$
        if $Q.solution$ *not empty* **then**
            Insert Q to ASG_OPEN
return solution of best node from ASG_OPEN

The central idea of the algorithm is to partition the solution space such that we forbid some assignments and forcefully include others. It has been shown that such a partitioning covers the complete solution space [110]. If the Hungarian Method is used and $N = M$, the complexity for finding the next solution is $O(N^4)$.

6.2.2.2 Properties of CBS-TA

In the following we show that CBS-TA, like CBS, is complete and optimal with respect to sum-of-cost.

Theorem 6.2.1. *CBS-TA is complete.*

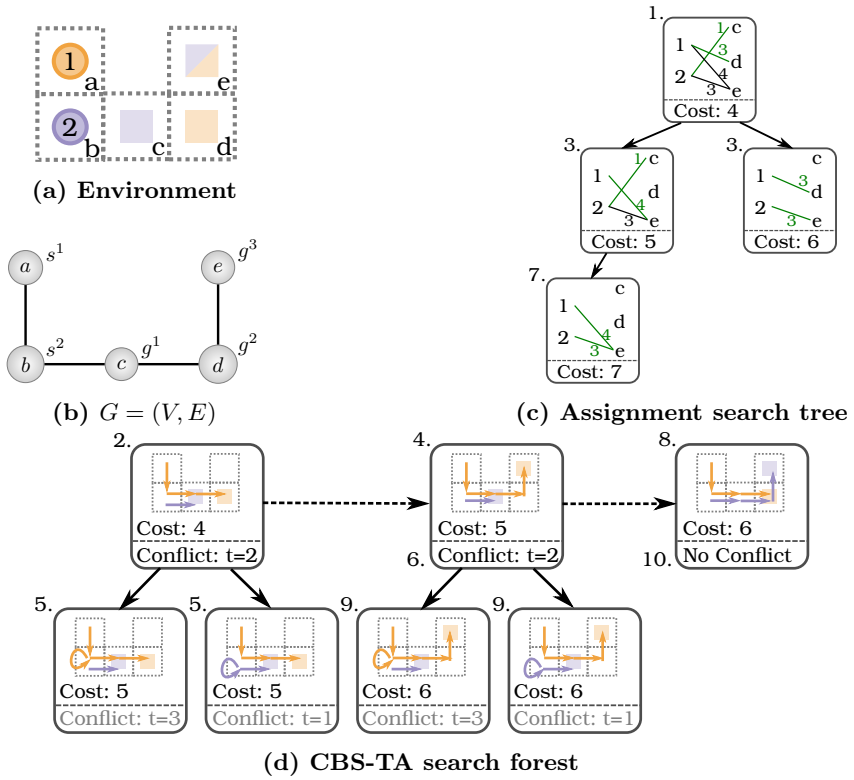


Figure 6.4: Example execution of CBS-TA. The environment (a) can be represented as graph (b) with some vertices being start and/or goal vertices. During the search, ASG_OPEN and OPEN are incrementally updated. The former can be visualized as a tree (c), while the latter forms a forest (d). See Section 6.2.2.3 for additional details.

Proof. It has been shown that CBS will return an optimal solution if one exists [131]. CBS-TA performs a CBS search on each root node. Whenever a root node is expanded the next best possible assignment is computed, until all possible assignments have been enumerated. Thus, the search is exhaustive in both task assignment and path planning. □

Theorem 6.2.2. *CBS-TA computes a solution that minimizes the sum of individual costs of all agents if one exists.*

Proof. If the assignment is fixed, the cost of each root node in the high-level search is a lower bound on the real cost (proof: Lemma 1, [131]). CBS-TA expands assignments in increasing cost order, therefore all expanded high-level nodes are a lower bound on the optimal cost. During each high-level search node expansion, the minimum cost either stays the same or increases because of the best-first expansion order in the high-level search. A different assignment can only be part of the optimal solution if its lower cost bound is identical or smaller than the current minimum cost in the high-level search. However, in this case this assignment was already added as new root node, because a previous root node (as a lower bound for its fixed assignment) must have been expanded. □

6.2.2.3 Example

The algorithm proceeds as follows. Consider an environment with $N = 2$ agents and $M = 3$ goals, see Fig. 6.4a. The problem can be formulated on a graph (see Fig. 6.4b), with an assignment matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} c & d & e \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \end{array}.$$

Based on G and A, we can compute the cost matrix C by considering the shortest path for each agent to the respective goal (ignoring all other agents):

$$C = \begin{array}{c} \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} c & d & e \end{array} \\ \begin{pmatrix} \infty & 3 & 4 \\ 1 & \infty & 3 \end{pmatrix} \end{array}.$$

We labeled the different steps in their respective orders in Fig. 6.4c and Fig. 6.4d. Using the cost matrix C, the first assignment commits agent 1 to goal d and agent 2 to goal c ($[1 \mapsto d, 2 \mapsto c]$) with cost 4. This creates an entry in the Assignment Open List (ASG_OPEN) (step 1) and a node in OPEN (step 2).

The path validation finds a conflict between the two agents at time step 2 (Line 9). When expanding a root node, we must also compute the next best assignment and add a new root node (Lines 12 to 19). For the next best assignment, we compute two possible successors in the assignment tree: the first one disallows the assignments $O = \{1 \mapsto d\}$ while the second one disallows $O = \{2 \mapsto c\}$ and enforces $I = \{1 \mapsto d\}$ (Lines 4 to 11 in Algorithm 6.3; step 3). In general, there might be up to N successors. The function `nextAssignment` returns the lowest cost option ($[1 \mapsto e, 2 \mapsto c]$). We compute the shortest path for each agent individually based on this assignment and add it to the OPEN list (step 4).

We now try to resolve the first conflict $(1, 2, c, 2)$, by adding additional nodes to the OPEN list (Lines 21 to 29). Namely, we consider the case where agent 1 is constrained to not be at node c at time step 2 and the case where agent 2 cannot be at node c at time step 2 (step 5).

In the next iteration we choose the second root node from the OPEN list (step 6)[4]. We need to compute the next best assignment ($[1 \mapsto d, 2 \mapsto e]$) and add an additional root node because the node being expanded is a root node (steps 7 and 8). The currently selected node from OPEN has a conflict (node c at time step 2) and we need to attempt to resolve it by adding two additional child nodes (step 9). Finally, we select the third root node from OPEN and return its solution because it is conflict free (step 10).

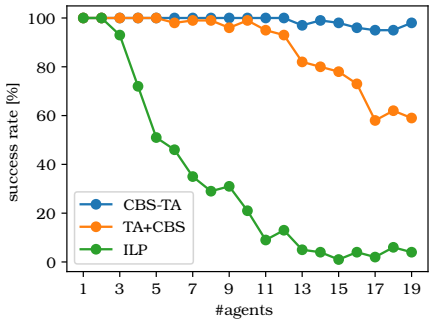
6.2.3 Extensions

We now show how CBS-TA can be extended to solve problem instances within a suboptimality bound and how it can be applied to additional interesting MAPF variants.

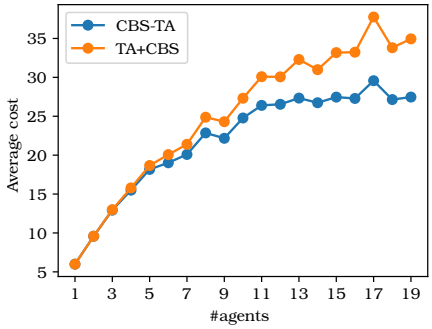
6.2.3.1 ECBS-TA

Enhanced CBS (ECBS) is a bounded suboptimal solver for MAPF [11]. ECBS uses focal search in both low- and high-level search algorithms. In focal search, a FOCAL list is maintained alongside the OPEN list. The FOCAL list contains a subset of the entries in the OPEN list, such that the

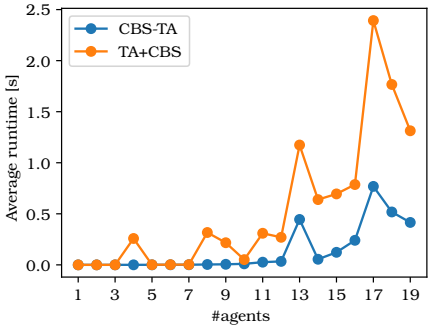
[4] We assume the FIFO principle as tie breaker in the OPEN list. An implementation could pick any of the nodes with cost 5.



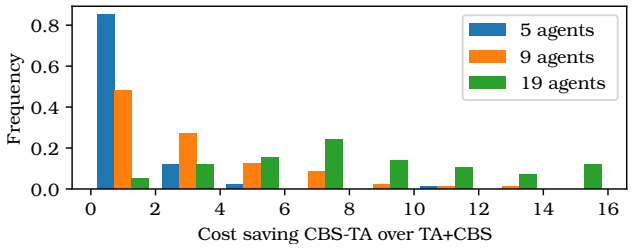
(a) Success rate.



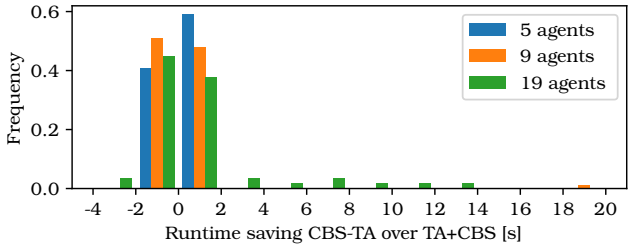
(b) Average cost.



(c) Average runtime.



(d) Histogram for cost savings.



(e) Histogram for runtime savings.

Figure 6.5: Benchmark results comparing CBS-TA with task assignment followed by CBS (TA+CBS) and ILP. (a)–(d) Each data point summarizes 100 randomly generated 8×8 4-connected grids with random start and goal locations. CBS-TA has a higher success rate, while achieving a lower average cost and runtime compared to TA+CBS. (e) and (f): Histograms comparing cost and runtime savings (i.e., TA+CBS minus CBS-TA) for 5, 9, and 19 agents. For few agents, there is frequently no cost or runtime difference between TA+CBS and CBS-TA, but a few outliers result in better average performance for CBS-TA. For many agents, CBS-TA computes better results in most cases, with large runtime benefits in a few cases.

cost of the entries in FOCAL are within a constant factor w of the best cost in OPEN. The low-level search of ECBS is changed in the following way compared to CBS: First, focal search (rather than A*) is used with a second inadmissible heuristic that estimates the number of conflicts. This is used to minimize the number of expected conflicts. Second, the lowest f-value of the low-level OPEN list is returned, in addition to the solution path. On the high-level, a lower bound $LB(n)$ is computed for each node n as the sum of the minimum f-values (from the low-level search). The high-level FOCAL list then only contains entries of the high-level OPEN list whose cost is less than or equal to $w \min LB(n)$. As in the lower-level search, an inadmissible heuristic that counts the number of expected conflicts is used for expansion. Keeping track of the lower bound through both search levels allows ECBS to use a single suboptimality factor w.

In order to jointly optimize for task assignment and path planning in the ECBS framework, we can use the same idea as for CBS-TA and generate a search forest with the root nodes referring to different assignments. However, the suboptimality bound w creates slack in the search, allowing us to be more flexible on when to generate additional root nodes. We consider three variants:

MaxRoot Add as many root nodes as possibly useful for the given w. In particular, we keep track of the highest cost of the already expanded root nodes. If that cost is smaller than $w \min LB(n)$, we add all additional root nodes whose cost is no larger than $w \min LB(n)$.

CBS-TA-style Following the same logic as CBS-TA, we add one additional root node each time a root node is expanded.

MinRoot Add as few root nodes as possible, without violating the suboptimality guarantee. In particular, we initially set $r = w \min LB(n)$. We only add an additional root node if the lowest-cost entry in the high-level OPEN list has a cost larger than r. In this case, we compute an additional assignment and update r.

The first variant (MaxRoot) can potentially compute low-cost solutions, even if high suboptimality bounds are used. However, the approach is impractical for large M and N, because there are too many potential assignments. Therefore this method is not implemented. We empirically evaluate CBS-TA-style and MinRoot on various instances, as described in Section 6.2.5.2.

6.2.4 Suboptimal Multi-Color MAPF

Our MAPF formulation permits agents to have a set of possible goals. One example of such a problem is the multi-color MAPF problem, in which each agent is part of a group and a set of goals is assigned to each group. Multi-color MAPF can be solved optimally with respect to the makespan using the Conflict-Based Min-Cost-Flow (CBM) algorithm [95], which uses a two-level search like CBS. Compared to CBS, CBM uses a maximum flow algorithm per group to find paths on the low-level rather than using A* per agent. We can model multi-color MAPF problem instances by setting $N = M$ and matrix A according to the group assignment. CBS-TA can compute optimal solutions with respect to the sum of costs, which can be more relevant in some scenarios (e.g. minimizing the total energy usage of the team). It has been shown that makespan and sum of cost cannot be simultaneously optimized [180]. Therefore, we need to consider our optimization objective directly. ECBS-TA can be used to find bounded suboptimal solutions to such problem instances. We present empirical results comparing CBM and ECBS-TA in Section 6.2.5.3.

6.2.5 Experiments

We implement CBS-TA and ECBS-TA in C++ using the boost library for fast heap data structures. We use a minimum-cost maximum-flow formulation that is part of the boost graph library to solve unconstrained assignment problems efficiently. All experiments were executed on a laptop (i7-4600U 2.1 GHz and 12 GB RAM).

Table 6.1: Benchmark results comparing task assignment followed by ECBS with ECBS-TA for different suboptimality bounds w. Each data point averages 100 randomly generated 4-connected grids with random start and goal locations.

			ECBS			ECBS-TA (CBS-TA style)			ECBS-TA (MinRoot)		
Grid Size	w	Agents	Success	Cost	Runtime	Success	Cost	Runtime	Success	Cost	Runtime
8×8	1.0	5	**1.00**	18.7	**0.00**	**1.00**	**18.2**	**0.00**	**1.00**	**18.2**	**0.00**
		9	0.96	24.3	**0.01**	**1.00**	**22.2**	**0.01**	**1.00**	**22.2**	**0.01**
		19	0.56	34.3	1.30	**0.98**	**27.2**	**0.31**	**0.98**	**27.2**	0.32
	1.1	5	**1.00**	18.7	**0.00**	**1.00**	**18.3**	**0.00**	**1.00**	18.4	**0.00**
		9	0.97	24.5	0.17	**1.00**	**22.7**	**0.00**	**1.00**	22.9	0.01
		19	0.56	34.6	0.93	**0.99**	**28.9**	**0.10**	**0.99**	29.3	0.11
	1.3	5	**1.00**	18.8	**0.00**	**1.00**	**18.7**	**0.00**	**1.00**	18.9	**0.00**
		9	**1.00**	25.6	0.01	**1.00**	**24.5**	**0.00**	**1.00**	25.4	0.01
		19	0.68	37.9	1.16	**1.00**	**32.8**	**0.05**	0.89	34.5	0.44
32×32	1.00	40	**0.92**	248	**0.5**	0.57	**244**	4.3	0.58	**244**	4.2
		70	**0.32**	283	**1.4**	0.01	**271**	3.0	0.01	**271**	2.8
		100	**0.01**	-	-	0.00	-	-	0.00	-	-
	1.05	40	0.95	264	**0.4**	**0.99**	**262**	0.5	0.95	263	0.7
		70	0.45	352	1.7	**0.52**	**350**	5.5	0.40	351	**1.6**
		100	**0.04**	**359**	2.5	0.03	**359**	16.8	**0.04**	**359**	**1.6**
	1.10	40	0.99	268	0.2	**1.00**	**267**	0.3	0.99	269	**0.1**
		70	**0.84**	371	0.6	0.82	**369**	2.6	0.80	371	**0.5**
		100	**0.33**	**417**	2.6	0.27	**417**	15.3	0.29	418	**2.3**

6.2.5.1 CBS-TA

We use a set of benchmark instances to compare CBS-TA to other existing methods. We randomly generated 8×8 4-connected grids with 20% obstacles and with random start and goal locations, such that it is guaranteed that there is at least one assignment where all agents can reach their respective goals. We limit the computation time to 30 s and mark a trial as a failure if no solution is found within the time limit. We vary the number of agents and report the success rate, average cost, and average runtime over 100 randomly created examples per number of agents. For CBS-TA, we use the shortest distance as heuristic in the low-level search.

TA+CBS versus CBS-TA We compare CBS-TA to task assignment followed by CBS (TA+CBS). To ensure fair runtime comparison, we implement TA+CBS by executing the same CBS-TA implementation with an artificial limit of a single root-node expansion. Our results (see Fig. 6.5) show that the success rate of CBS-TA is higher compared to TA+CBS. For examples that were successful with both algorithms, we compute the average cost and average runtime. CBS-TA achieves a lower average cost in a shorter average time compared to TA+CBS. We analyze the relative frequency of this effect by looking at individual histograms of the cost savings (that is cost(TA+CBS) - cost(CBS-TA)) and runtime savings, comparing only examples that were successful with both algorithms. With only 5 agents, over 80% of the test cases show no cost difference and the runtime is identical in nearly all cases. With 19 agents, however, the cost improvement peaks at an improvement of 7 (over 20% of the examples), while the runtime is identical in over 75% of the cases. This shows, that CBS-TA is in particular beneficial for dense cases, where the task assignment and path planning are more tightly coupled. Additionally, CBS-TA does not seem to require additional runtime, even for sparser examples.

CBS-TA versus ILP Integer Linear Program (ILP) formulations have been used for the non-anonymous MAPF problem with different objectives, including minimizing makespan and sum-of-cost [179]. The idea behind such formulations is to construct a time-expanded flow graph and formulate a multi-commodity flow problem. We implement an ILP based on this idea assuming $M = N$ and a fully anonymous assignment. This is a challenge for CBS-TA (since $N!$ possible assignments have to be considered), but easier for the ILP formulation because it can be framed as a single commodity flow. Such instances can be solved in polynomial time when optimizing for makespan [98]. In order to be able to minimize for the sum-of-cost instead, we use the following steps. First, we generate the time-expanded flow graph and formulate an ILP, similar to [179], but using a single commodity for all agents, rather than one commodity per agent. Second, we add one additional auxiliary integer variable for each goal capturing the time until an agent reaches and stays at that goal. Third, we set our optimization objective to minimize the sum of all such auxiliary variables [98].

We use Gurobi 7.5 as ILP solver [62]. In order to solve an instance, we need an upper bound of the makespan of the optimal solution. We find an upper bound dynamically, by doubling the makespan on each attempt. Only if the cost between two successive attempts did not change do we report a solution. This avoids solutions where the makespan but not sum-of-cost is minimal.

The ILP solver computes results with the same minimum cost in all solved cases, as expected. However, the runtime is significantly higher compared to CBS-TA. For example, the average runtime for 10 agents is 21 s. This also affects the success rate (see Fig. 6.5), which is significantly lower compared to CBS-TA for larger numbers of agents.

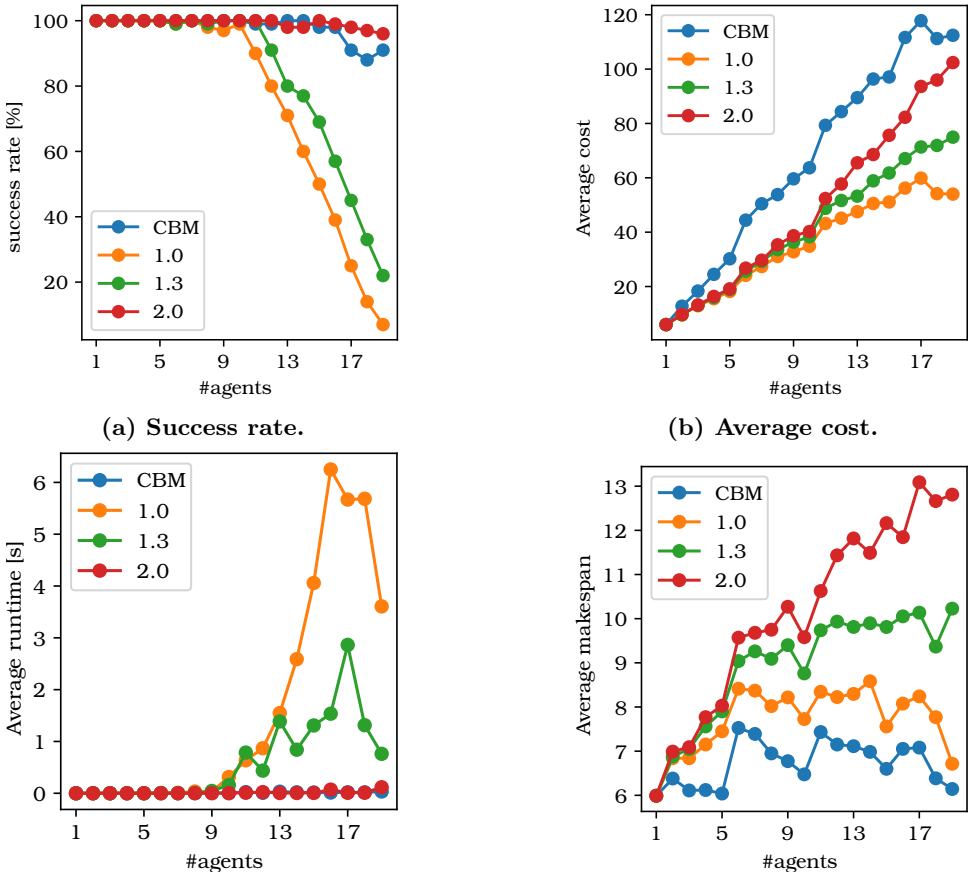
6.2.5.2 ECBS-TA

We use benchmarks to compare the two ECBS-TA variants to optimal task assignment followed by ECBS using different environment sizes and suboptimality bounds. Note that even though we use the same suboptimality bound for ECBS-TA and ECBS, they have different semantics. In the ECBS-case we guarantee that the returned result is within a factor of w given that the task assignment is fixed. ECBS-TA, on the other hand, guarantees to return a solution that is within a factor of w for the optimal valid task assignment. Thus, the guarantee given by ECBS-TA is stronger. In all cases we numerically verified that the suboptimality bounds are fulfilled.

Small Environments In the first set of tests, we use the same 8×8 4-connected grids with 20% obstacles as for the CBS-TA analysis in the previous section. We vary the number of agents and report the success rate, average cost, and average runtime over 100 examples per numbers of agents. A subset of our results is shown in Table 6.1. Both ECBS-TA variants achieve higher success rates, lower or comparable costs, and lower runtime compared to ECBS when used with the same suboptimality bound. When comparing the two different ECBS-TA versions, we notice that the CBS-TA style root-node expansion results in lower costs at higher suboptimality bounds. In case ($w = 1.3$), this version also provides a higher success rate at a lower runtime, compared to the MinRoot expansion policy.

Large Environments In another set of tests (see Table 6.1) we used 32×32 4-connected grids, again with 20% obstacles. Instead of up to 20 robots we test with up to 100 robots. This results in longer required paths for each robot, but has a lower robot-to-free-space density of 12% compared to the 38% of the smaller maps.

When computing the optimal solution ($w = 1$), the success rate of ECBS-TA (both variants) is now significantly lower than ECBS. For instances that could be solved by all variants, the solution found by ECBS-TA has a lower cost, but not significantly (less than 5% on average). Higher suboptimality bounds ($w = 1.05$ and $w = 1.1$) improve the success rate of ECBS-TA to a comparable level and the solution quality is nearly identical for ECBS and ECBS-TA. However, the runtime



(a) Success rate. **(b) Average cost.**

(c) Average runtime. **(d) Average makespan.**

Figure 6.6: Benchmark results comparing CBM with ECBS-TA for 8×8 4-connected grid environments. ECBS-TA achieves a higher success rate at comparable runtime with $w = 2$. For all used suboptimality bounds the achieved sum of costs is lower when using ECBS-TA when compared to CBM. However, because CBM optimizes makespan, it outperforms ECBS-TA with respect to makespan.

of ECBS-TA using the CBS-TA style expansion is significantly higher and grows quickly with the number of agents. Our MinRoot expansion on the other hand has the same (and sometimes better) runtime than ECBS.

This effect can be explained as follows. The number of possible task assignments grows factorially in the number of robots. Thus, instances with many robots have many possible assignments with identical cost (we noticed several hundred possible assignments with optimal cost for some of our examples). Adding another root node in the ECBS forest is time-consuming, because another assignment needs to be computed and low-level search for each robot for this assignment needs to be executed. Therefore, our CBS-TA style expansion will create many additional root nodes, but those root nodes do not help significantly to find lower cost solutions. The MinRoot expansion delays creating additional root-nodes as long as possible for the given w. High suboptimality bounds might not trigger the creation of any additional root node.

Consequently, instances with many robots should use ECBS-TA with the MinRoot expansion. This provides stronger suboptimality guarantees compared to ECBS and better results with low suboptimality bounds. At the same time ECBS-TA achieves the same results in terms of runtime

and cost as ECBS for high suboptimality bounds.

6.2.5.3 Multi-color MAPF

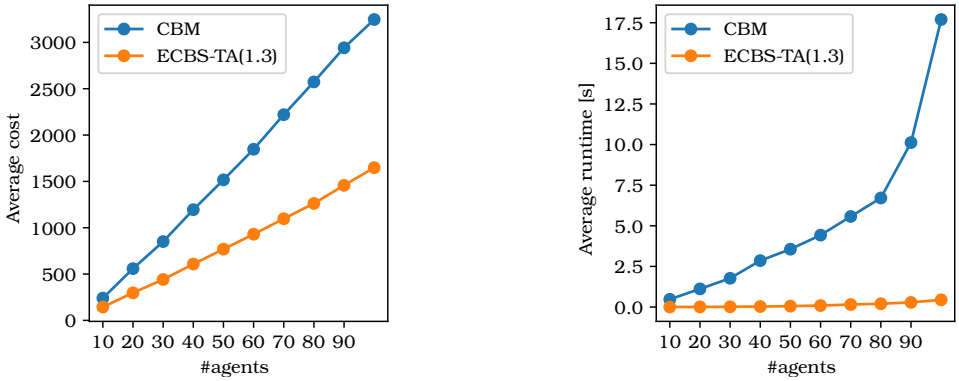
Thus far our experiments have only considered the unlabeled case. We now evaluate cases where the target assignment is more constrained. To be able to compare to a baseline we set $N = M$ and arrange agents into groups, such that agents within the same group are interchangeable. We solve the same problem instance using the Conflict-Based Min-Cost-Flow (CBM) algorithm [95], and compare it with ECBS-TA (MinRoot expansion) with varying suboptimality bounds. Both algorithms use different objective functions: CBM finds solutions with minimal makespan, while ECBS-TA minimizes the sum of cost up to a given suboptimality factor. The CBM implementation is written in C++ and uses boost graph as well.

Small Environments In the first set of tests we run both algorithms on the same 8×8 maps with varying number of agents, but fixed group size of 5 agents per group[5]. We report the success rate, average cost, runtime, and makespan for CBM and ECBS-TA with different suboptimality bounds ($w = 1.0, 1.3, 2.0$), see Fig. 6.6. We notice that the achieved average cost (that is, the sum of the individual agent costs) is smaller for ECBS-TA in all cases, while the makespan (as the metric being optimized in CBM) is lowest for CBM. When we choose $w = 2.0$, ECBS-TA achieves a higher success rate at comparable runtimes compared to CBM. Using lower suboptimality bounds results in lower cost solutions, but the success rate is significantly lower and the runtime is longer compared to CBM.

Large Environments We use the same fixed group size of 5 agents per group on the larger 32×32 maps that were also used in the ECBS-TA experiments. Using ECBS-TA to compute cost-optimal solutions ($w = 1.0$) leads to a low success rate (no instance with 100 agents can be solved within the timeout), while bounds with $w \geq 1.3$ can solve all instances. In comparison, CBM solved 995 out of the 1000 test instances in the given time limit. For brevity, we report the results for $w = 1.3$ in Fig. 6.7 on instances that were solved by both algorithms. The achieved cost of CBM is more than twice as high compared to ECBS-TA in the 100 agent case. It is surprising that the difference is so large considering the relatively high suboptimality bound used for ECBS-TA. Not surprisingly, CBM achieves a lower makespan (the metric it is minimizing): in the 100 agent case CBM has an average makespan of 33 while ECBS-TA achieves an average makespan of 47. The runtime of ECBS-TA is significantly better compared to CBM, especially when more agents are considered.

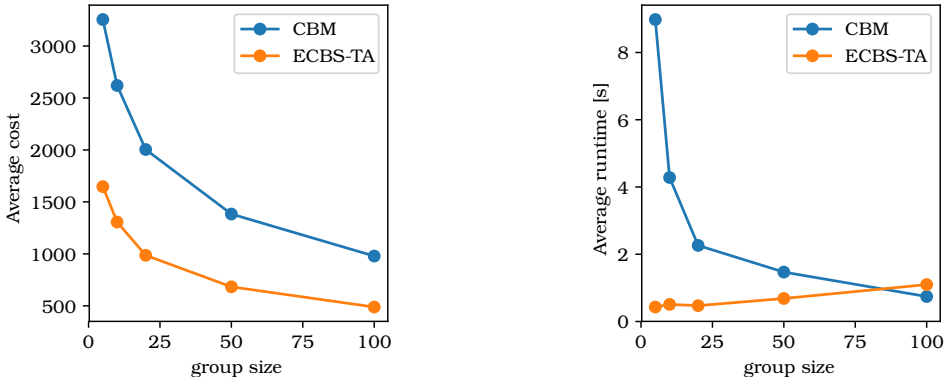
Varying Group Size To evaluate the influence of the group size, we use the 32×32 maps with 100 agents while varying the group size. The results (using $w = 1.3$ for ECBS-TA) are shown in Fig. 6.8. As before, we limit the computation time to 30 s. ECBS-TA was able to compute solutions for all cases within the given time limit. If using 100 groups (group size of 1), CBM was not able to compute a single solution, but it was successful for all other group sizes and instances. If the group size is 1, we get the labeled case where each agent has an assigned goal. The other extreme is the unlabeled case, which we achieve with a group size of 100. CBM uses a maximum flow algorithm in the low-level search that is executed per robot group. Thus, it performs best if there is just a single group (of size 100) and worst if there are 100 groups (of size 1). ECBS-TA, on the other hand, considers a single possible assignment if there are 100 groups, and 100! assignments if there is a single group. The runtime results reflect this behavior: ECBS-TA can find solutions much faster for small group sizes compared to CBM. The runtime requirements slowly increase for larger group sizes. The runtime of CBM decreases with the group size. As in the previous results, the cost of the

[5]The last group may have less than 5 agents.



(a) Average cost. (b) Average runtime.

Figure 6.7: Benchmark results comparing CBM with ECBS-TA for 32×32 4-connected grid environments. ECBS-TA achieves lower cost and runtime compared to CBM.



(a) Average cost. (b) Average runtime.

Figure 6.8: Benchmark results comparing CBM with ECBS-TA for varying group sizes. The runtime of ECBS-TA slowly grows, while the runtime of CBM rapidly decays with the group size. The average cost of the solution is always smaller when using ECBS-TA, independent of the group size.

solution is lower for ECBS-TA in all cases. Similarly, as expected, the makespan of CBM solutions is always better than for ECBS-TA solutions.

6.3 MAPF with Generalized Conflicts (MAPF/C)

We are given a roadmap of the environment ($\mathcal{G}_E$) with additional conflict sets for generalized vertex-vertex ($conVV$), edge-edge ($conEE$), and edge-vertex ($conEV$) conflicts. Additionally, we are given a unique start vertex for each robot $v_s^i \in \mathcal{V}_E$. In the labeled case we are given a unique goal for each robot $v_g^i \in \mathcal{V}_E$; in the unlabeled case the goals can be freely assigned to robots.

As before, a robot can either wait at its current vertex or traverse an edge. However, at each timestep we need to enforce that the generalized vertex-vertex constraints are enforced. Additionally, during each movement we need to ensure that the edge-edge and edge-vertex conflicts are not violated.

Definition 6.3.1. *A path $p^j = [u_0^j, \ldots, u_{T^j}^j, u_{T^j+1}^j, \ldots]$ for agent j is **MAPF/C valid** if and only if the following conditions hold.*

1. *Agents obey inter-robot constraints when stationary (*generalized vertex-vertex collision*), that is, for all timesteps t and all agents pairs $j \neq k$ we have $u_t^j \notin conVV(u_t^k)$;*
2. *Agents obey inter-robot constraints while traversing an edge (*generalized edge-edge collision*), that is, for all timesteps t and all agent pairs $j \neq k$ we have $(u_t^j, u_{t+1}^j) \notin conEE((u_t^k, u_{t+1}^k))$;*
3. *Agents obey inter-robot constraints between stationary and traversing robots (*generalized edge-vertex collision*), that is, for all timesteps t, all agents pairs $j \neq k$ and $u_t^j = u_{t+1}^j$ we have $u_t^j \notin conEV((u_t^k, u_{t+1}^k))$; and*
4. *p^j is valid.*

Based on the MAPF/C validity of a path, we can define labeled, unlabeled, and multi-color MAPF/C variants by strengthening the requirements for the solution paths. Similar to traditional MAPF instances, it is possible to optimize for different criteria, including makespan and sum-of-cost.

Solving MAPF/C instances optimally is NP-Complete even in the unlabeled case, as the following two theorems show. In contrast, unlabeled MAPF can be solved in polynomial time using a maximum-flow formulation [178].

Theorem 6.3.1. *Solving labeled MAPF/C optimally with respect to cost or makespan is NP-Complete.*

Proof. This follows directly from the NP-Completeness proof for labeled MAPF [180]. □

Theorem 6.3.2. *Solving unlabeled MAPF/C optimally with respect to makespan is NP-Complete.*

Proof. We reduce maximum independent set (MIS) to unlabeled MAPF/C. Consider the decision MIS instance $(\mathcal{G}_{MIS}, k_{MIS})$ that decides if there is a maximal independent set of cardinality k_{MIS}. An independent set is the set of vertices in $\mathcal{G}_{MIS}$ such that there are no two vertices in the set that are adjacent to each other. We create a new graph $\mathcal{G}_{MAPF/C}$ by copying $\mathcal{G}_{MIS}$ and adding $2k_{MIS}$ vertices $v_s^1, \ldots, v_s^k, v_g^1, \ldots, v_g^k$, each of them fully connected to the vertices which were copied from $\mathcal{G}_{MIS}$. We add generalized vertex-vertex constraints to the vertices we copied from $\mathcal{G}_{MIS}$ to enforce the MIS property that no selected vertices should be adjacent to each other. We can then solve the decision version of the constructed MAPF/C instance to check if there is a solution with makespan $T = 2$, which disallows any wait actions. An example is shown in Fig. 6.9.

If we are given a solution to the MAPF/C problem of makespan $T = 2$, each path must contain exactly three nodes and the middle nodes form the set S since they fulfill all the constraints. On the other hand, if we are given a maximum independent set $S = \{v^1, \ldots, v^k\}$, we can create a solution for the MAPF/C instance by adding k_{MIS} paths connecting v_s^i, v^i, and v_g^i for each $i \in \{1, \ldots, k_{MIS}\}$. □

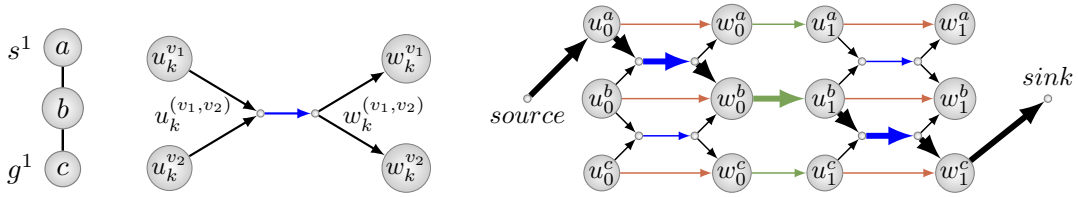
6.3.1 Unlabeled Planner

Unlabeled MAPF can be solved in polynomial time if K is given by finding the maximum flow of a time-expanded flow-graph [178]. This maximum-flow problem can also be expressed as an Integer Linear Program (ILP) where each edge is modeled as a binary variable indicating its flow and the objective is to maximize the flow subject to flow conservation constraints [179]. An ILP formulation allows us to add generalized constraints.

In order to find an optimal solution with respect to makespan, we use a two-step approach. First, we find a lower bound for K, by ignoring the generalized constraints. We search the sequence

(a) $\mathcal{G}_{MIS}$ (b) $\mathcal{G}_{MAPF/C}$

Figure 6.9: Example for maximum independent set to unlabeled MAPF/C reduction with $k_{MIS} = 2$. The red sets define $conVV(\cdot)$ of nodes a, b, c. A solution for the MAPF/C instance is shown with the bold edges, indicating that $\{a, c\}$ is a maximum independent set.



(a) $\mathcal{G}_E$ (b) "Gadget" (c) $\mathcal{G}_F$ with $K = 2$

Figure 6.10: Example flow-graph (c) for environment shown in (a) with a single robot. The construction uses a graph "gadget" (b) for each edge in $\mathcal{G}_E$. The blue edges are annotated with edge-edge and edge-vertex conflicts and the green edges are annotated with vertex-vertex conflicts. The bold arrows in (c) show the maximum flow through the network, which can be used to compute the robots' paths.

$K = 1, 2, 4, 8, \ldots$ for a feasible K, and then perform a binary search to find the minimal feasible K, which we denote as $LB(K)$. Because we ignore the generalized constraints, we can check the feasibility in polynomial time using the Edmonds-Karp algorithm on the time-expanded flow-graph. Second, we execute a linear search from the known lower bound, solving the fully constrained ILP. In practice, we have found that the lower bound $LB(K)$ is sufficiently close to the final K such that a linear search is faster compared to another modified binary search using the ILP.

We build the time-expanded flow-graph $\mathcal{G}_F = (\mathcal{V}_F, \mathcal{E}_F)$ as an intermediate step to formulate the ILP. Compared to the existing detailed discussions [178, 179, 95], we add additional annotations to some of the edges indicating the generalized constraints. For each timestep k and vertex $v \in \mathcal{V}_E$ we add two vertices u_k^v and w_k^v to $\mathcal{V}_F$ and create an edge connecting them (red edges in Fig. 6.10c). For each timestep k and edge $(v_1, v_2) \in \mathcal{E}_E$ we create a "gadget" connecting $u_k^{v_1}, u_k^{v_2}, w_k^{v_1}$, and $w_k^{v_2}$, see Fig. 6.10b. Furthermore, we connect the different timesteps by adding additional edges (w_k^v, u_{k+1}^v) (green edges in Fig. 6.10c). Additionally, we add vertices *source* and *sink*, which are connected to vertices $u_0^{v_s^i}$ and $w_K^{v_g^i}$, respectively. If a maximum flow is computed on this graph, the flow describes a path for each robot, fulfilling the regular path validity requirements. We annotate the edges $e = (w_k^v, u_{k+1}^v)$, which connect different timesteps, with generalized vertex collisions (green edges in Fig. 6.10c). Specifically, we set $con(e)$ to be the set of other edges $e' = (w_k^{v'}, u_{k+1}^{v'})$ where $v' \in conVV(v)$. Furthermore, we add annotations for generalized edge-edge and edge-vertex collisions to all helper edges e of the gadget (blue edges in Fig. 6.10c) that were created for $(v_1, v_2) \in \mathcal{E}_E$. For edge-edge conflicts we set $con(e)$ to include the set of other edges e' that were created for $(v_1', v_2') \in conEE((v_1, v_2))$. For edge-vertex conflicts we set $con(e)$ to include the set of other edges $e' = (u_k^{v'}, w_k^{v'})$ that were created to allow wait actions (red edges in Fig. 6.10c) where $v' \in conEV((v_1, v_2))$. Finally, we disallow that the blue edges of the

"gadget" are used for wait actions by adding constraints $con((u_k^{v_1}, u_k^{(v_1,v_2)})) = \{(w_k^{(v_1,v_2)}, (w_k^{v_1})\}$ and $con((u_k^{v_2}, u_k^{(v_1,v_2)})) = \{(w_k^{(v_1,v_2)}, (w_k^{v_2})\}$.

For each edge $(u, v) \in \mathcal{E}_F$, we introduce a binary variable $z_{(u,v)}$. The ILP can be formulated as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{(source,v)\in\mathcal{E}_F} z_{(source,v)} \\ \text{subject to} \quad & \sum_{(u,v)\in\mathcal{E}_F} z_{(u,v)} = \sum_{(v,w)\in\mathcal{E}_F} z_{(v,w)} \\ & \forall v \in \mathcal{V}_F \setminus \{source, sink\} \\ & z_e + \sum_{(u',v')\in con(e)} z_{(u',v')} \leq 1 \\ & \forall e \in \mathcal{E}_F, con(e) \neq \emptyset \end{aligned} \tag{6.1}$$

Here, the first constraint enforces the flow conservation and the second constraint enforces the generalized conflicts. A solution to the ILP assigns a value to all variables, indicating the flow on each edge. We can then easily create the path p^i for each robot by setting x_k^i based on the flow in $\mathcal{G}_F$. The permutation $\phi(i)$ is implicitly given by x_K^i.

6.3.2 Labeled Planner

Our labeled planner is based on a bounded suboptimal variant of CBS, called Enhanced CBS (ECBS), which has been shown to solve practical instances with hundreds of robots in maze-like environments [11].

ECBS uses two different search levels: high- and low-level. The high-level search creates a binary constraint tree. Each node in that tree has a set of constraints. A constraint either disallows a robot to occupy a specific vertex or to traverse a specific edge at a fixed timestep. A node also contains a solution path p^i for each robot that is consistent with the set of constraints, and an associated cost for the solution paths. The root node does not have any constraints and thus the solution paths p^i can be created by finding the path with the lowest cost in the given roadmap for each robot individually. If a node contains only MAPF/C valid paths, a solution is found. Otherwise, the first MAPF/C violation between two robots r^i and r^j is found, and two new nodes are created. Both children inherit the constraints from the parent node. The first child imposes an additional constraint on r^i and the second node adds one additional constraint for r^j. In CBS the nodes are traversed using the best-first search strategy with respect to the cost. The low-level search is used to compute a solution path for a single agent that satisfies the constraints for that agent. Each time a new high-level node is created, the low-level search needs to be executed just once for the agent with the newly added constraint. The bounded suboptimality in ECBS is achieved by using focal search with heuristics on both levels [11].

ECBS can solve MAPF/C instances by using the generalized conflict definition for both high-level search and search heuristics. The algorithm is identical to the one outlined in the original paper [11], with the following adjustments. For the high-level search, we now need to consider conflicts caused by the generalized constraints. In case the first conflict is a violation of the generalized vertex-vertex constraints between agents r^i and r^j at timestep k, we create two child nodes. The first child adds a constraint for r^i to avoid visiting x_k^i at timestep k. The second child adds a constraint for r^j to not visit x_k^j at timestep k. Similarly, if the first conflict is a violation of the generalized edge-edge constraints between r^i and r^j, we add two child nodes with the additional conflicts of r^i not traversing (u_k^i, u_{k+1}^i) and r^j not traversing (u_k^j, u_{k+1}^j), respectively. In case the first conflict is a violation of the generalized edge-vertex constraints where, without loss of generality, r^i is waiting at a vertex and r^j is traversing an edge, we add two child nodes with the additional conflicts of

r^i not waiting at u^i_k at timestep k and r^j not traversing (u^j_k, u^j_{k+1}), respectively. Furthermore, we need to adjust the heuristics to count violations caused by the generalized constraints as well. For the high-level search we use the number of pairs of robots that have at least one conflict. For the low-level search we use the number of conflicts in the high-level search node.

ECBS is bounded suboptimal with respect to to the cost, i.e. for a user-provided suboptimality bound w, the cost of the returned solution will satisfy

$$\sum_i pathcost(p^i) \leq w \sum_i pathcost(p^i_*),$$

where p^i_* is a optimal solution as computed by CBS.

6.3.3 Goal Assignment

ECBS can be used to solve an unlabeled problem approximately, by finding a goal assignment first. Frequently the Hungarian method is used in robotics, which finds the the optimal assignment with respect to the sum of the costs [87]. This might, for example, correspond to minimizing the total energy usage of the robots. The unlabeled case is typically a formation change problem, where it is more desirable to minimize the time until the last robot reaches its goal. This problem is also known as the linear bottleneck assignment problem and can be solved efficiently using, for example, the Threshold algorithm [22].

6.4 Remarks

We describe labeled, unlabeled, and multi-color MAPF and existing solvers. We extend those problem formulations in two ways: First, we present MAPF with optimal task assignment (MAPF-TA). Second, we present MAPF with generalized conflicts (MAPF/C).

For MAPF-TA, we extend Conflict-Based Search to simultaneously assign tasks and plan paths for multiple agents. The key insight is the extension of the high-level search to operate on a search forest rather than a search tree, where each root node represents a fixed assignment. The forest can be efficiently constructed on demand, avoiding the need to consider all irrelevant possible task assignments. The use of the CBS framework provides two significant advantages. First, other extensions of CBS, such as the bounded suboptimal ECBS, are directly applicable. Second, we can optimize for the sum of individual costs, which is more appropriate in some domains than makespan.

We evaluated our algorithms, CBS-TA and ECBS-TA, extensively, with the following important results:

1. We can compute solutions significantly faster than an ILP-based solver while providing the same optimality guarantees.

2. The traditional method of independently assigning tasks followed by path planning can be improved in terms of solution quality and runtime by using (E)CBS-TA in dense environments with few agents. However, larger environments with many agents do not benefit significantly from a joint optimization. Nevertheless, ECBS-TA (MinRoot expansion) provides stronger suboptimality guarantees than before with negligible additional runtime overhead.

3. ECBS-TA produces lower cost solutions than CBM (which optimizes for a different objective) even when high suboptimality bounds are used. For small group sizes, ECBS-TA can produce such a solution in significantly shorter time compared to CBM.

We believe that (E)CBS-TA can be used in all cases where task assignment and path planning might be optimized jointly with respect to the sum of costs of the individual agents' plans. We use MAPF-TA in Chapter 10.

For MAPF/C, we generalize MAPF by considering vertex/vertex, vertex/edge, and edge/edge conflicts. Our solvers are based on existing MAPF solvers, that directly consider and resolve the added conflicts. We use our MAPF/C solvers in Chapters 8 and 9.

CHAPTER 7

Task and Motion Planning for Ground Robots

We now consider ground robots such as differential drive robots to operate in warehouse environments. We demonstrate our ideas with a simple running example of two robots in a narrow corridor of a grid-world with 1 m × 1 m cells, see Fig. 7.1a. The maximum speed limit of Robot 1 is $1/4$ m/s, and the maximum speed limit of Robot 2 is $1/16$ m/s. Furthermore, edge (C, D) in Fig. 7.1b has a minimum speed limit of $1/32$ m/s. Robot 1 must pass Robot 2 to reach its goal location, which requires Robot 2 to move into an alcove temporarily, no matter what the maximum speeds and plan-execution capabilities of the robots are. We can use a discrete solver from AI to discover such critical intermediate configurations and then use a post-processing step to create a plan-execution schedule that takes the maximum and minimum speed limits and plan-execution capabilities of the robots into account. Our approach consists of four stages:

1. We formulate the task as a labeled MAPF instance in discrete time and space on a graph of the environment, see Fig. 7.1b.

2. We use a standard solver from AI to find a discrete plan consisting of collision-free paths for all robots, assuming point-robots, uniform edge lengths, and synchronized robot movement from vertex to vertex at discrete timesteps, see Fig. 7.1c. However, this discrete plan is nearly impossible to execute safely on actual robots due to their imperfect plan-execution capabilities. It is communication-intensive for the robots to remain perfectly synchronized as they follow their paths, and their individual progress will thus deviate from the discrete plan, for example, because the edges have non-uniform lengths or speed limits (due to dynamics constraints or safety concerns) or because the robots cannot move at a uniform speed (due to dynamics constraints, slip, and other robot and environmental limitations). For example, if Robot 2 is slightly slower than Robot 1 in our running example, the robots can collide while they execute their first action. In that case, it is not effective to slow down all robots. Instead, we control the speeds of the robots and enforce safety distances between them to avoid collisions.

3. We execute our post-processing step, MAPF-POST, to create a plan-execution schedule that takes information about the edge lengths and speed limits into account to provide user-specified guaranteed safety distances between the robots.

This chapter is based on **Wolfgang Hönig**, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. "Path Finding for Multi-Robot Systems with Kinematic Constraints". In: *Journal of Artificial Intelligence Research (JAIR)* (2019). Accepted. In Revision.

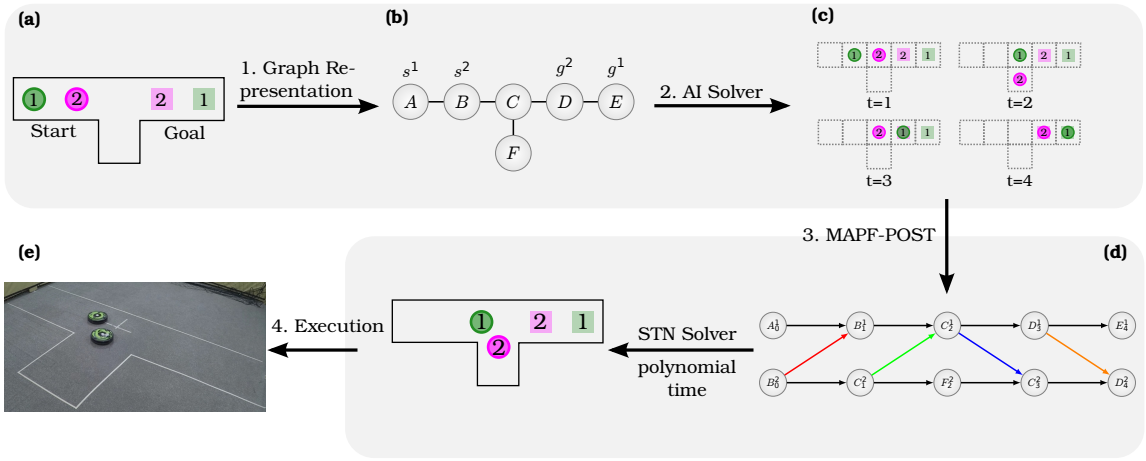


Figure 7.1: Architecture of our approach. Boldly colored circles mark robots' locations. (a) Example environment with two robots at their start location. Lightly colored squares mark the goal locations of the robots with corresponding colors/numbers. (b) Graph representation of the example environment. (c) Discrete plan with 4 timesteps. (d) Post-processing of the discrete plan using MAPF-POST. (e) Plan execution.

4. We execute the schedules in simulation and on physical robots.

We now describe MAPF-POST in detail and demonstrate experiments in simulation and on physical robots in Section 7.3.

7.1 MAPF-POST

We now convert a discrete plan to a continuous plan-execution schedule. This post-processing step, MAPF-POST, takes any MAPF valid discrete plan as input. The output is a sequence of timed waypoints for each robot, such that the resulting constant speed between any two such waypoints is within the specified speed limits of the robot and edge. We assume that the distances between the nearest points on any two robots do not depend on their orientations, e.g., robots are cylindrical in a 2-D environment or spherical in a 3-D environment. We also assume that they can change their speeds at waypoints instantaneously. If the robots move from waypoint to waypoint using constant speeds (and thus only change their speeds at waypoints), we can guarantee a user-specified safety distance between any two robots. While the given discrete plan might have wait actions, the output of MAPF-POST only allows the robots to stop once they have reached their final location. However, robots might be required to move with a speed close to zero. It is guaranteed that we can construct such continuous schedule for any valid MAPF plan if the minimum speed limits of all robots and edges are zero. If we have a non-zero minimum speed limit, it is possible that there is no continuous schedule for a given valid MAPF plan that is consistent with the robot and edge speed limits.

MAPF-POST relies on a data structure called the *Temporal Plan Graph* (TPG). We first introduce this data structure formally, and then describe the construction and usage of TPGs for multi-robot planning.

7.1.1 Temporal Plan Graph

A TPG is a directed acyclic graph $\mathcal{G}_{TPG} = (\mathcal{V}_{TPG}, \mathcal{E}_{TPG})$, see Fig. 7.2 for an example. Each vertex $v \in \mathcal{V}_{TPG}$ represents an event, which corresponds to a robot entering a location. Each edge $(u, v) \in \mathcal{E}_{TPG}$ is a temporal precedence between events u and v indicating that event u must

Algorithm 7.1: Algorithm for constructing the TPG.

Input: A discrete plan with a makespan of T for N robots consisting of path $p^j = [u_0^j, \ldots, u_T^j]$ for each robot j.
Result: The temporal plan graph $\mathcal{G}_{TPG}$ for the discrete plan.

```
1  /* Step 1: add vertices and Type 1 edges */
2  for j ← 1 to N do
3      v_0^j ← Add vertex to V_TPG
4      x ← v_0^j
5      for t ← 1 to T do
6          if u_t^j ≠ u_{t-1}^j then
7              v_t^j ← Add vertex to V_TPG
8              Add Type 1 edge (x, v_t^j) to E_TPG
9              x ← v_t^j
10 /* Step 2: add Type 2 edges */
11 for t ← 0 to T do
12     for j ← 1 to N do
13         if v_t^j in V_TPG then
14             for t' ← t + 1 to T do
15                 for k ← 1 to N do
16                     if j ≠ k and v_t'^k in V_TPG and u_t^j = u_t'^k then
17                         Add Type 2 edge (v_t^j, v_t'^k) to E_TPG
18                         Break out of two loops
```

be scheduled strictly before event v. For MAPF-POST, the TPG imposes two types of temporal precedences between events as dictated by the discrete plan.

Type 1 For each robot, precedences enforce that it enters locations in the order given by its path in the discrete plan.

Type 2 For each pair of robots and each location that they both enter, precedences enforce the order in which the two robots enter the location in the discrete plan.

A *plan-execution schedule* assigns a real-valued time to each event, corresponding to an entry time for each location, and can be computed using the TPG. Robots that execute a plan-execution schedule enter all locations at these entry times. We prove later that the robots do not collide if they execute a plan-execution schedule that is consistent with these precedences. The discrete plan uses timesteps and specifies a total order among the events. The TPG, however, does not discretize time and specifies only a partial order among the events, which provides it with flexibility to take into account both speed limits and imperfect plan-execution capabilities of actual robots.

7.1.2 Constructing the Temporal Plan Graph

Algorithm 7.1 formalizes the construction of the TPG, given any discrete plan consisting of the paths $p^j = [u_0^j, \ldots, u_T^j]$ for each robot j. For each robot, we create a *route* from its given path by removing the wait actions but keeping the move actions (Line 6), that is, for each robot j, we extract the route r^j from path p^j by keeping only the first of any consecutive identical locations on the path. For each (kept) location u_t^j on route r^j, we create a location vertex $v_t^j \in \mathcal{V}_{TPG}$ (with associated robot j and associated location u_t^j; Line 7). For each two successive (kept) locations u_t^j

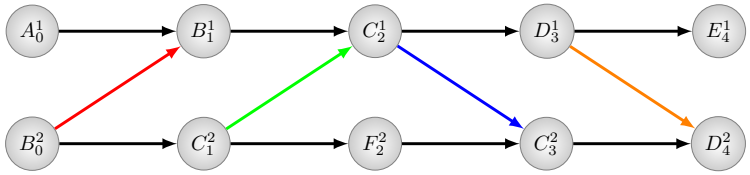


Figure 7.2: TPG for our running example. Each vertex labeled l_t^j in the TPG represents the event "robot j arrives at location l at timestep t." Each edge represents a temporal precedence between the events represented by its incident vertices. Black edges are Type 1 edges, and colored edges are Type 2 edges.

and $u_{t'}^j$ on the same route, we create a Type 1 edge $(v_t^j, v_{t'}^j) \in \mathcal{E}_{TPG}$ (Line 8), shown as horizontal black edges in Fig. 7.2. This edge corresponds to a precedence of Type 1, indicating that robot j enters location u_t^j directly before location $u_{t'}^j$ and thus enters locations in the order given by its path in the discrete plan. Type 1 edges thus correspond to move actions. For each two identical (kept) locations $u_t^j = u_{t'}^k = u$ on different routes (and thus $j \neq k$) with $t < t'$, we create a Type 2 edge $(v_t^j, v_{t'}^k) \in \mathcal{E}_{TPG}$ (Line 17), shown as non-horizontal colored edges in Fig. 7.2. This edge corresponds to a precedence of Type 2, indicating that robot j enters location u before a different robot k enters the same location.

Some Type 2 edges are implied by transitivity and are not created (Line 18). Consider the Type 2 edges $(v_t^j, v_{t'}^{j'})$ and $(v_t^j, v_{t''}^{j''})$ where $t < t' < t''$, $j \neq j'$, and $j \neq j''$. Because we iterate over timesteps, we first add $(v_t^j, v_{t'}^{j'})$. A subsequent iteration will consider $v_{t'}^{j'}$ and add the Type 2 dependency $(v_{t'}^{j'}, v_{t''}^{j''})$. Thus, the Type 2 edge $(v_t^j, v_{t''}^{j''})$ is implied by transitivity and does not need to be created.

The TPG for a discrete plan for N robots with a makespan of T has $O(NT)$ location vertices, because there are up to $T-1$ move actions in each of the N paths. There are $O(NT)$ edges because each location vertex has at most one outgoing Type 1 edge and one outgoing Type 2 edge. Algorithm 7.1 constructs the TPG in $O(N^2T^2)$ time.

7.1.3 Augmenting the Temporal Plan Graph

While the basic TPG models the precedences between robots, it does not provide any safety distance between them. We now add additional vertices, called safety markers, to the TPG to provide a guaranteed safety distance between robots. The safety markers correspond to new locations, called auxiliary locations, and allow us to change the meaning of the edges in the TPG. Each edge $(u, v) \in \mathcal{E}_{TPG}$ can now be a temporal precedence indicating that event u must be scheduled no later than (rather than strictly before) event v.

We assume that all edge lengths are larger than the user-specified parameter $\delta > 0$, which is the guaranteed safety distance between any two robots on the graph. We also assume that each robot traverses each edge with constant speed and stays at (main or auxiliary) locations only for an instant before it reaches its goal location. However, its speed can change instantaneously at each location. Our implementations use controllers that approximate this assumption.

7.1.3.1 Basic Approach

We first assume that δ divides all edge lengths or that there are only two robots. The pseudo code of this basic approach is given by the highlighted lines of Algorithm 7.2. This construction of the augmented TPG is a slight variation of Algorithm 7.1.

We make use of two relations $vertex(t, j)$ and $succloc(t, j)$ that map to a vertex $v_t^j \in \mathcal{V}_{TPG}$ and location $u_t^j \in \mathcal{V}$, respectively. Specifically, $vertex(t, j)$ maps to the location vertex v_t^j if one exists

as before, or $NULL$ if no such location vertex was created. $succloc(t, j)$ maps to the next main location that robot j will reach after timestep t or $NULL$ if there is no such location. Step 1 adds vertices referring to main locations as well Type 1 edges as before (see Lines 1 to 17). Compared to Algorithm 7.1, we also construct the two relations $vertex(t, j)$ and $succloc(t, j)$ as part of that first step.

The second step adds Type 2 edges, similar to before. Instead of connecting corresponding main locations directly, we add safety markers that are δ apart from the main locations. For the construction of Type 2 edges, we still iterate over every pair of vertices v_t^j and $v_{t'}^k$ in the TPG where two different robots j and k arrive at the same main location (Lines 19 to 24). We then use the helper function `addType2Edge` (pseudo code in Appendix A) that adds a Type 2 edge to $\mathcal{E}_{TPG}$ and inserts new vertices if required. For example, the call in Line 25 adds a Type 2 edge (v_t^j, x'), where x' is located δ before $v_{t'}^k$. If such a vertex does not exist, we add it to $\mathcal{V}_{TPG}$. Furthermore, we split the existing Type 1 edge between $v_{t'}^k$ and its predecessor x'_p in the TPG into two so-called microedges: one microedge from x'_p to x' and one microedge from x' to $v_{t'}^k$. The splitting is done by another helper function `getOrCreateVertex` that either finds a specified vertex (specified by a vertex and relative distance), or creates such a vertex and splits the existing Type 1 edge into microedges accordingly (pseudo code in Appendix A). Similarly, we add a second Type 2 edge from a vertex δ after v_t^j to $v_{t'}^k$ (Line 26). In case the two robots are following the same path, that is if $succloc(t, j)$ and $succloc(t', k)$ are identical, we add additional Type 2 edges with a spatial distance of δ each (Lines 30 to 32).

As before, we can avoid some Type 2 edges that are implied by transitivity (Lines 35 to 37). In case the two robots are following the same path, we can break out of both loops, following the same argument as in Section 7.1.2. If the two robots j and k follow different paths, there might be another robot k' that occupies the same location as j in the future such that j and k' follow the same path. Thus, we cannot break out of both loops in this case and there might be multiple outgoing Type 2 edges for each vertex. Because there might now be up to N outgoing Type 2 edges for each vertex, Algorithm 7.2 creates $O(N^2T)$ edges.

An example of the augmented TPG for our running example with $\delta = 0.75\,\text{m}$ is shown in Fig. 7.3a. Compared to the TPG from Fig. 7.2, the number of Type 2 edges has doubled, since each Type 2 edge of the original TPG has been replaced by at least two Type 2 edges. For an intuition what the additional Type 2 edges accomplish, consider the two red edges. The first one (originating at B_0^2) enforces that Robot 2 has to start moving out of B before Robot 1 can move closer than $0.75\,\text{m}$ towards B. The second red edge (aiming towards B_1^1) enforces that Robot 2 has to be at least $0.75\,\text{m}$ away from B before Robot 1 can move into B. The combination of the two edges and the fact that robots have to move with a constant speed along an edge enforce the safety distance.

If the safety distance is decreased, the constructed TPG might contain more Type 2 edges. An example for $\delta = 0.25\,\text{m}$ is shown in Fig. 7.3b. Here, additional safety markers and Type 2 edges are added for the portions of the paths where the two robots traverse the same edges, namely the traversal from B to C and the traversal from C to D. Consider the four red edges, for an intuition of what those additional safety markers accomplish. The first two red edges enforce that Robot 1 can only move into B after Robot 2 left B, as before. After Robot 1 reached B it is scheduled to move towards C. Without any coordination between Robots 1 and 2, Robot 1 might come arbitrary close to Robot 2 while both of them move towards C. The two additional red Type 2 edges ensure that Robot 1 has to stay at least $0.25\,\text{m}$ behind Robot 2 while they both move from B to C.

7.1.3.2 Adding Additional Type 2 Edges

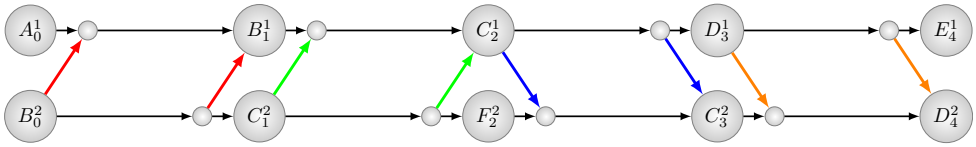
We now consider cases where δ does not need to divide all edge lengths. This creates additional challenges as outlined in the following example. Consider three robots in an environment with an intersection as shown in Fig. 7.4, with the following given paths: $\{p^1 = [A, A, B, C, D]$, $p^2 = [B, B, C, D, E]$, and $p^3 = [F, C, G]\}$. The highlighted part of Algorithm 7.2 creates an augmented

Algorithm 7.2: Constructing the augmented TPG.

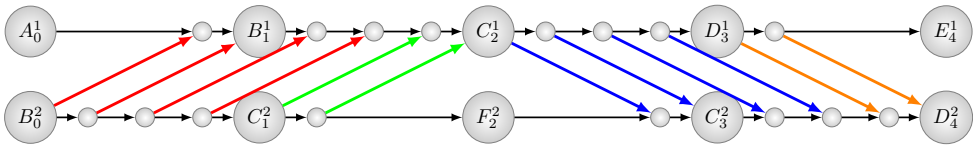
Input: A discrete plan with a makespan of T for N robots consisting of path $p^j = [u_0^j, \ldots, u_T^j]$ for each robot j.

Result: The augmented temporal plan graph $\mathcal{G}_{TPG}$ for the discrete plan.

```
/* Step 1: add vertices and Type 1 edges */
for j ← 1 to N do
    x ← Add vertex to V_TPG
    vertex(0, j) ← x
    t' ← 0
    for t ← 1 to T do
        if u_t^j ≠ u_{t-1}^j then
            y ← Add vertex to V_TPG
            Add Type 1 edge (x, y) to E_TPG
            succloc(t', j) ← u_t^j
            constraints((x, y)) ← ∅
            t' ← t
            vertex(t, j) ← y
            x ← y
        else
            vertex(t, j) ← NULL
    succloc(t', j) ← NULL
/* Step 2: add Type 2 edges */
for t ← 0 to T do
    for j ← 1 to N do
        if vertex(t, j) ≠ NULL then
            for t' ← t + 1 to T do
                for k ← 1 to N do
                    if j ≠ k and vertex(t', k) ≠ NULL and u_t^j = u_t'^k then
                        addType2Edge(G_TPG, vertex(t, j), 0, vertex(t', k), −δ)
                        addType2Edge(G_TPG, vertex(t, j), δ, vertex(t', k), 0)
                        C ← (vertex(t, j) → vertex(t', k))
                        addConstraint(G_TPG, vertex(t, j), 0, δ, C)
                        addConstraint(G_TPG, vertex(t', k), −δ, 0, C)
                        if succloc(t, j) = succloc(t', k) then
                            for d ← 2δ to dist(u_t^j, succloc(t, j)) step δ do
                                addType2Edge(G_TPG, vertex(t, j), d, vertex(t', k), d − δ)
                                addConstraint(G_TPG, vertex(t, j), d − δ, d, C)
                                addConstraint(G_TPG, vertex(t', k), d − 2δ, d − δ, C)
                            Break out of two loops
                        else
                            Break
/* Step 3: add additional Type 2 edges */
addAdditionalType2Edges(G_TPG)
```



(a) Augmented TPG for the running example with $\delta = 0.75\,\text{m}$.



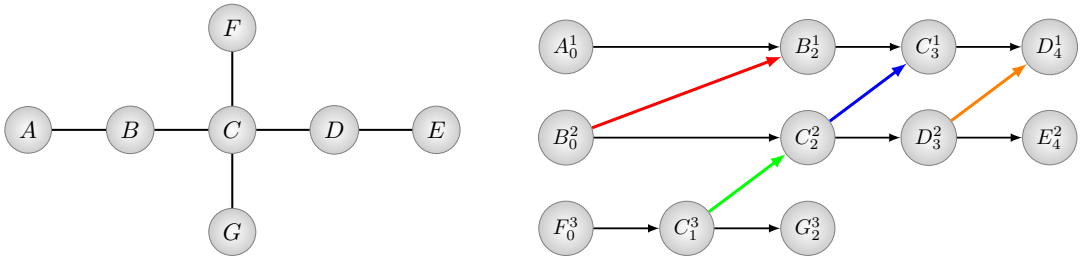
(b) Augmented TPG for the running example with $\delta = 0.25\,\text{m}$.

Figure 7.3: Augmented TPGs for the running example. The labeled vertices are associated with main locations as in Fig. 7.2. The unlabeled vertices are safety markers (associated with auxiliary locations).

TPG as shown in Fig. 7.5. Here, additional vertices were added in Step 2, causing some of the original Type 1 edges to be split into microedges. For example, the first safety marker after B_0^2 caused a split of the edge from B_0^2 to the safety marker before C_2^2. Because the robot model assumes constant velocities on edges only, it would now be possible for Robot 2 to change its speed between B_0^2 and the safety marker directly before C_2^2. Such a speed change can cause a safety distance violation, as described in the following example. Robot 2 might start moving out of B very quickly until it reaches the first safety marker and than change to a slower speed until it reaches the second safety marker. At the same time, Robot 1 moves with a constant speed from its first safety marker after A towards B. Because Robot 2 moves much slower away from B after it passed the first safety marker than Robot 1 moves towards B, Robot 1 will be closer than δ to Robot 2 while traversing. In order to prevent such a possible violation, we need to add an additional Type 2 edge, preventing Robot 2 from moving too slowly out of location B compared to Robot 1 moving towards location B. Similar to the case where two robots follow the same path, we add the new Type 2 edge such that is is parallel to the existing Type 2 edges.

We now describe how we can find such violations and add additional Type 2 edges, see Algorithm 7.2. The part that is not highlighted is discussed in this section. We use another relation $\mathit{constraints}(e)$ that maps an edge $e \in \mathcal{E}_{TPG}$ to a set of constraints of the form $v_t^j \to v_{t'}^k$. For Type 1 edges the constraint $v_t^j \to v_{t'}^k$ indicates that robot j cannot enter location u_t^j before robot k moved at least δ away. The *constraints* relation is initialized in Line 11, and updated in Lines 27 to 29 and Lines 33 to 34. We use a helper function `addConstraint` (pseudo code in Appendix A), that adds constraints for multiple microedges. For Type 2 edges the constraint $v_t^j \to v_{t'}^k$ indicates that this edge was created to enforce that robot j cannot enter location u_t^j before robot k moved at least δ away. There is exactly one constraint per Type 2 edge and it is initialized as part of the `addType2Edge` function (see Appendix A). A potential violations occurs if a Type 1 edge requires a constraint that is not satisfied by the associated Type 2 edges. We execute `addAdditionalType2Edges` in Step 3 (Line 39), which finds such discrepancies and adds additional Type 2 edges.

The pseudo code for `addAdditionalType2Edges` is shown in Algorithm 7.3. We iterate over all Type 1 edges (u, v) where u is a safety marker (Lines 3 to 5). The set of constraints that need to be satisfied is given by $\mathit{constraints}(e)$ (Line 6). The set of constraint that are already satisfied is given by the constraints that are satisfied by all incoming and outgoing Type 2 edges with respect to u (Lines 7 to 11). We can now iterate over all constraints $v_t^j \to v_{t'}^k$ that have not been satisfied yet (Lines 12 to 14). If the constraint originated from the current robot, that is j equals the robot associated with the Type 1 edge (u, v), we add an additional Type 2 edge originating from the



(a) Graph of environment. All edge lengths are 1 m. **(b) TPG for an example with three robots.**

Figure 7.4: Example environment where additional Type 2 edges need to be added to the TPG to enforce the safety distance.

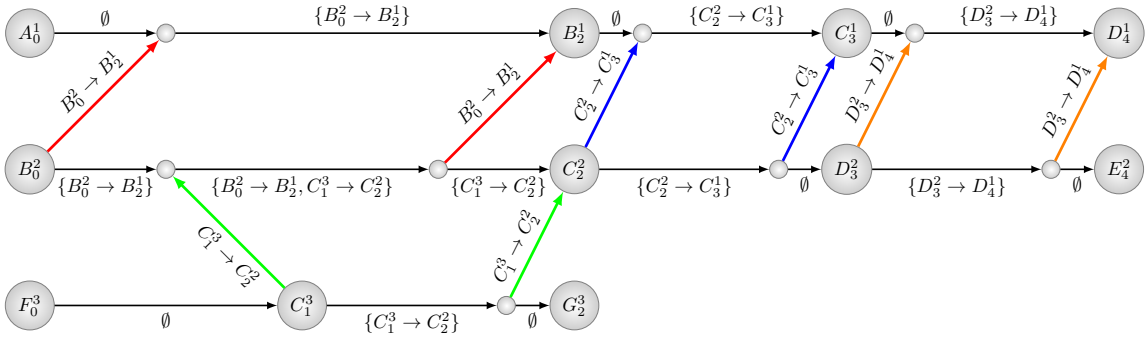


Figure 7.5: Augmented TPG for the example from Fig. 7.4 with $\delta = 0.75$. The annotations on the edges show the set of constraints which need to be satisfied along that edge for Type 1 edges and the constraint which needs to be satisfied for Type 2 edges.

current robot (Lines 15 to 16). Otherwise, we add an additional Type 2 edge targeting the current robot (Lines 17 to 19). This process is repeated until no more violations are found (Lines 1, 2 and 20).

For the example first introduced in Fig. 7.4, the *constraints* relation is shown as edge annotations in Fig. 7.5. Consider the microedge (u, v) for Robot 2 that starts at the first safety marker after B_0^2 and ends at the next safety marker. The constraints are $\{B_0^2 \to B_2^1, C_1^3 \to C_2^2\}$. There is only one incoming Type 2 edge, satisfying the second constraint $(C_1^3 \to C_2^2)$. The constraint $B_0^2 \to B_2^1$ is missing; because this constraint originated from the current robot, we add an additional Type 2 edge originating from Robot 2 (see red dashed line in Fig. 7.6). Now consider the microedge (u, v) for Robot 2 that start at the safety marker directly before C_2^2 and ends at C_2^2. Here, the constraints are $\{C_1^3 \to C_2^2\}$ and only another constraint $(B_0^2 \to B_2^1)$ is satisfied. The missing constraint originates from Robot 3; thus, we add an additional Type 2 edge targeting Robot 2 (see green dashed line in Fig. 7.6).

The number of possible iterations in Algorithm 7.3 is finite. Each time a violation is found, a Type 1 edge is split and another vertex added. This new vertex in turn might cause additional constraints not being satisfied. However, this can only happen up to N times, because the TPG is a directed acyclic graph (which we prove in Lemma 7.1.5). Each iteration has a time complexity of $O(N^2T)$, resulting in an overall time complexity of $O(N^3T)$ for `addAdditionalType2Edges`.

Algorithm 7.3: addAdditionalType2Edges

Input: TPG $\mathcal{G}_{TPG}$ as created by the first two steps of Algorithm 7.2.
Result: Updated $\mathcal{G}_{TPG}$, such that all constraints are satisfied.

```
repeat
    violationsFound ← False
    for i ← 1 to N do
        foreach Type 1 edge e = (u, v) of robot i do
            if u is safety marker then
                required ← contraints(e)
                satisfied ← ∅
                foreach Type 2 edge e' = (u, v') do
                    satisfied ← satisfied ∪ contraints(e')
                foreach Type 2 edge e' = (v', u) do
                    satisfied ← satisfied ∪ contraints(e')
                missing ← required \ satisfied
                foreach m = (v_t^j → v_t'^k) ∈ missing do
                    violationsFound ← True
                    if j = i then
                        addType2Edge(G_TPG, v_t^j, dist(v_t^j, u), v_t'^k, dist(v_t^j, u) − δ)
                    else
                        v' ← getOrCreateVertex(G_TPG, v_t'^k, −δ)
                        addType2Edge(G_TPG, v_t^j, dist(v', u), v_t'^k, dist(v', u) − δ)
until violationsFound = False
```

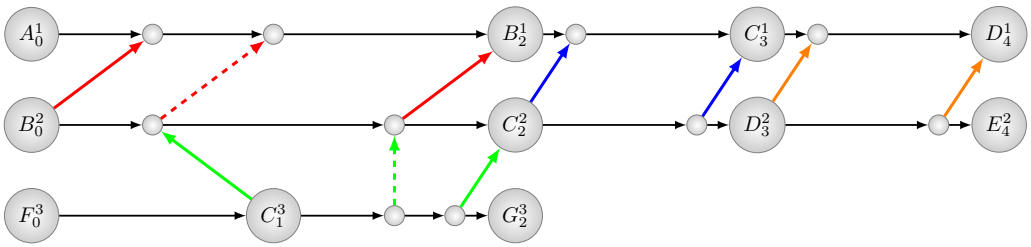


Figure 7.6: Augmented TPG for the example from Fig. 7.4 with $\delta = 0.75$. The dashed lines show additional Type 2 edges which need to be added after the first two steps were executed.

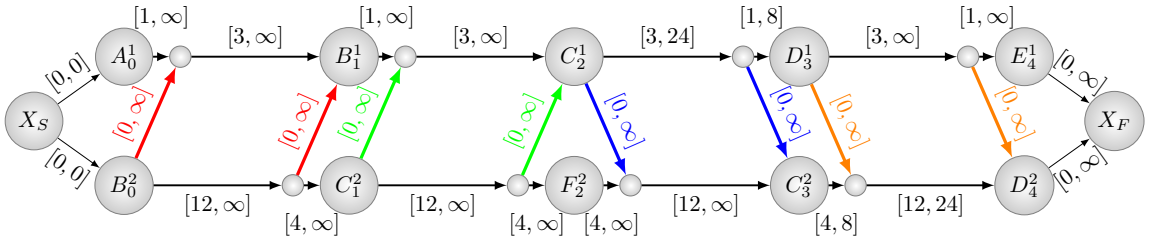


Figure 7.7: STN for our running example with $\delta = 0.75\,\text{m}$. Each edge annotation is of the form $[LB(e), UB(e)]$. A robot must take at least $LB(e)$ time units and at most $UB(e)$ time units to traverse a Type 1 edge e.

7.1.4 Encoding Dynamics Constraints

We now associate quantitative information with the edges of the augmented TPG, transforming it into a *Simple Temporal Network* (STN). An STN is a directed acyclic graph $\mathcal{G}_{STN} = (\mathcal{V}_{STN}, \mathcal{E}_{STN})$. Each vertex $v \in \mathcal{V}_{STN}$ represents an event. Each edge $e = (u, v) \in \mathcal{E}_{STN}$ annotated with the STN bounds $[LB(e), UB(e)]$ is a simple temporal constraint between events u and v indicating that event u must be scheduled between $LB(e)$ and $UB(e)$ time units before event v. We add two additional vertices. X_S represents the start event and therefore has edges annotated with the STN bounds $[0, 0]$ to all vertices without incoming edges. Similarly, X_F represents the finish event and therefore has edges annotated with the STN bounds $[0, \infty]$ to all vertices without outgoing edges.

The STN bounds allow us to express non-uniform edge lengths or speed limits (due to dynamics constraints or safety concerns). We now explain which STN bounds to associate with the edges of the augmented TPG to transform it into an STN. Each edge $(v, v') \in \mathcal{E}_{STN}$ is a precedence indicating that event v must be scheduled no later than event v'. Thus, we must associate the STN bounds $[0, \infty]$ with all edges. However, we can assign tighter STN bounds to Type 1 edges. Consider any Type 1 edge $e = (u, v)$ with associated robot j and associated length $l(e) = dist(u, v)$. The lower STN bound corresponds to the minimum time needed by robot j for moving from the location associated with vertex u to the location associated with vertex v, and the upper STN bound corresponds to the maximum time. From now on, we assume that robot j has a finite maximum speed limit $v^*_{\max}(e)$ and finite minimum speed limit $v^*_{\min}(e)$ for the move, for example, due to the dynamics constraints of robot j or safety concerns about traversing edge e with high or low speeds. Then, robot j needs at least $l(e)/v^*_{\max}(e)$ time units to complete the move, meaning that it enters the location associated with vertex v at least $l(e)/v^*_{\max}(e)$ time units after it enters the location associated with vertex u, resulting in a tighter lower STN bound than 0. Similarly, we can compute an upper bound $UB(e)$ as $l(e)/v^*_{\min}(e)$ if $v^*_{\min}(e)$ is greater 0 and ∞ otherwise. Thus, we associate the STN bounds $[l(e)/v^*_{\max}(e), UB(e)]$ with the edge.

Figure 7.7 shows the STN for our running example. Remember that the length of all edges in $\mathcal{E}_E$ is 1 m, the maximum speed limit of Robot 1 is $1/4$ m/s, and the maximum speed limit of Robot 2 is $1/16$ m/s. Furthermore, edge $(C, D) \in \mathcal{E}_E$ has a minimum speed limit of $1/32$ m/s. We use $\delta = 0.75$ m. The simple temporal constraint between the safety marker after location vertex B_0^2 and location vertex B_1^1 enforces that Robot 1 cannot move at maximum speed until it enters location B since it needs to let the slower Robot 2 exit location B before it enters the location. The upper bounds on the four Type 1 edges between locations C and D reflect the minimum speed constraint on that edge.

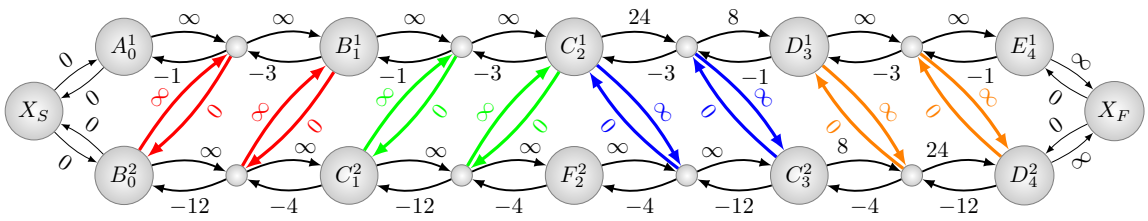


Figure 7.8: Distance graph for our running example.

7.1.5 Calculating a Plan-Execution Schedule

One can calculate a schedule (that assigns a time $t(v)$ to each event v with the convention that $t(X_S) = 0$) that satisfies all simple temporal constraints in polynomial time using minimum-cost path computations on the directed distance graph of the STN, typically done with the Bellman-Ford algorithm. Every vertex of the STN is translated into a vertex of the distance graph. Every edge of the STN $e = (u, v) \in \mathcal{E}_{STN}$ annotated with the STN bounds $[LB(e), UB(e)]$ is translated into two edges of the distance graph, namely one forward edge (u, v) of cost $UB(e)$ and one reverse edge (v, u) of cost $-LB(e)$. The absence of negative cost cycles in the distance graph is equivalent to the existence of a schedule that satisfies all simple temporal constraints [39]. The distance graph of our running example is shown in Fig. 7.8. The assigned time $t(v)$ for each event v is given by the negative of the shortest distance from v to X_S in the distance graph. For example $t(B_1^1) = 12$, because the shortest distance from B_1^1 to X_S is -12. We compute all such times by running the Bellman-Ford algorithm on the distance graph with reversed edges starting from X_S. Similarly, an upper bound for a valid arrival time is given by the shortest distance from X_S to v and all such times can be computed by running the Bellman-Ford algorithm on the distance graph starting from X_S.

7.1.6 Properties

In the following, we show that the user-specified parameter δ is indeed a guaranteed safety distance. In all cases, we assume that for any edge with length l in the underlying search graph, $\delta < l$. Furthermore, we discuss the runtime and space complexity of MAPF-POST and the cases in which it is guaranteed to find a plan-execution schedule given a valid discrete plan.

7.1.6.1 Safety Guarantee

We first prove that δ is the guaranteed safety distance in terms of the distance in the search graph $\mathcal{G}_E$. In practice, the Euclidean distance (that is, the straight-line distance in the continuous environment) between any two robots is more relevant. In Theorem 7.1.2, we relate the graph-based and Euclidean distances for the two cases of four-neighbor square grids and six-neighbor cubic grids.

Lemma 7.1.1. *If a Type 2 edge connects two vertices x and x' in the TPG then their corresponding locations are δ away from each other.*

Proof. This follows from Lines 25, 26 and 32 in Algorithm 7.2 and Lines 16 and 19 in Algorithm 7.3. □

Lemma 7.1.2. *Two vertices x and x' in the TPG that correspond to the same location and different robots a and a' are connected by a path of temporal precedences with at least one Type 1 edge.*

Proof. Suppose x and x' correspond to the same main location. Two robots cannot be at that main location at the same time in the discrete plan since the paths in the discrete plan are valid. Without

loss of generality, let robot a be the robot that arrives before robot a' at the main location in the discrete plan. Then there is a path of temporal precedences starting with a Type 1 edge from x to the safety marker δ after x and a Type 2 edge from this safety marker to a vertex x'' (Line 26 in Algorithm 7.2), where x'' is either identical with x' or a vertex in the TPG that corresponds to the same main location as x but that is for a different robot a'' that arrives at the main location after a' but before a''. If x'' corresponds to a different robot, then a future iteration of the loop at Line 19 in Algorithm 7.2 creates an edge from δ after x'' to x' or to some other robot a''' that arrives at the same main location after a'' but before a'. Since there is only a finite number of discrete steps between the arrival times of a and a', there will eventually be a path of temporal precedences between x and x' with at least one Type 1 edge.

Suppose x and x' correspond to the same auxiliary location. Let x be somewhere between vertices u and v, and x' be somewhere between u' and v', where u, u' and v, v' correspond to the same main locations, respectively.

Let's analyze the possibility where robot a moves from u to v and robot a' moves from u' to v'. Both robots cannot be at the same main location at the same time nor swap locations in one timestep in the discrete plan. Without loss of generality, let robot a be the robot that arrives at the location that corresponds to u before robot a' arrives at the location that corresponds to u'. We now consider two cases. First, the case where x is a multiple of δ away from u. Then, a temporal precedence including at least one Type 2 edge is created in Line 26 or Line 32 in Algorithm 7.2 connecting x with a vertex δ before x'. Thus, there is a path of temporal precedences including at least one Type 1 edge (from that vertex to x') in this case. Second, the case where x is not a multiple of δ away from u. Then, x is created dynamically by splitting a Type 1 (micro)edge into two Type 1 microedges using `getOrCreateVertex` (Line 19 in Algorithm 7.3). Both newly created microedges retain the constraints of the split edge (Lines 16 and 17 in Algorithm A.2). Then, a Type 2 edge connecting x with a vertex δ before x' will eventually be created in `addAdditionalType2Edges` (Lines 16 and 19 in Algorithm 7.3). Thus, there is a path of temporal precedences including at least one Type 1 edge (from that vertex to x') in this case.

Let's analyze the possibility where robot a moves from u to v and robot a' moves from v' to u'. Without loss of generality, let robot a be the robot that arrives at the location that corresponds to v before robot a' arrives at the location that corresponds to v'. Because v and v' correspond to the same main location, there is a path of temporal precedences from v to v' with at least one Type 1 edge. By construction, there are Type 1 edges connecting x to v and v' to x', concluding our analysis of all cases. □

Lemma 7.1.3. *Two robots cannot be at the same main or auxiliary location nor traverse the same microedge in opposite directions at the same time.*

Proof. Consider two vertices x and x' in the TPG that correspond to two robots arriving at the same main or auxiliary location. According to Lemma 7.1.2, there is a path of temporal precedences between x and x' with at least one Type 1 edge. Since Type 1 edges have non-zero length and robots have finite speed limits, x and x' cannot be simultaneous events.

Now, consider the case where two robots traverse a microedge in opposite directions. Robot a traverses the microedge (u, v) somewhere between vertices x and y and robot a' traverses the microedge (v', u') between vertices y' and x', where x, x' and y, y' correspond to the same main locations and u, u' and v, v' correspond to the same auxiliary locations, respectively. Both robots cannot be at the same main location at the same time nor swap locations in one timestep in the discrete plan. Without loss of generality, let robot a be the robot that arrives at y before robot a' arrives at y'. This means that there is a path of temporal precedence edges from y to y' that contains at least one Type 1 edge (Lemma 7.1.2). Since Type 1 edges have non-zero length, both robots cannot traverse the microedge in opposite directions at the same time. □

Lemma 7.1.4. *The distance between any two robots in the graph is at least δ if at least one of them is at a main or auxiliary location.*

Proof. Robots always stay at locations only for an instant before they reach their goal location. Thus, the times they are at (main or auxiliary) locations coincide with the times they arrive at them. Consider a vertex x in the TPG that corresponds to robot a arriving at location u. For any other robot a' the worst case for its proximity to a arises when it also goes through u (indicated by vertex x' in the TPG). We distinguish two cases:

First, robot a' moves away from u. In this case, robot a' arrived at u before robot a. According to Lemma 7.1.2, there is a path of temporal precedence edges from x' to x with at least one Type 1 edge. This path must also include a Type 2 edge and therefore, by Lemma 7.1.1, the robots' distance is at least δ when robot a arrives at u (and continues to be at least δ in case robot a has reached its goal location).

Second, robot a' moves towards u. In this case, robot a arrived at u before robot a'. According to Lemma 7.1.2, there is a path of temporal precedence edges from x to x' with at least one Type 1 edge. This path must also include a Type 2 edge and therefore, by Lemma 7.1.1, the robots' distance is at least δ when robot a arrives at u. Since robot a cannot have reached its goal location, because the discrete plan avoids robots a and a' at u at the same timestep, the robots' distance is at least δ when robot a is at u. □

Theorem 7.1.1. *The distance between any two robots in the graph is at least δ at all times.*

Proof. Consider any two robots. The theorem follows from Lemma 7.1.4 if at least one of the two robots is at a main or auxiliary location. Otherwise, we distinguish four cases where the distance between the two robots could potentially be less than δ in the graph and show that it is not:

First, both robots traverse the interior of the same microedge in the same direction or the interiors of two incident microedges toward each other at the same time. One of the robots will arrive at a location incident on its microedge first according to Lemma 7.1.3. Their distance in the graph will be less than δ at that time, which is a contradiction with Lemma 7.1.4. Thus, this case is impossible.

Second, both robots traverse the interior of the same microedge in opposite directions at the same time, which is a contradiction with Lemma 7.1.3. Thus, this case is impossible.

Third, both robots traverse the interiors of incident microedges e_1 and e_2 away from each other at the same time. Let e_1 be an edge from location u to v and e_2 be an edge from location u to w. One of the robots left u last and the robots' distance in the graph was less than δ at that time, which is a contradiction with Lemma 7.1.4. Thus, this case is impossible.

Fourth, both robots traverse incident microedges in the same direction at the same time. Let robot a traverse microedge e_1 (starting at location u and ending at location v) and robot a' traverse microedge e_2 (starting at location v and ending at location w). Robot a' left v no later than robot a left u since otherwise their distance in the graph would be less than δ at that time, which would be a contradiction with Lemma 7.1.4. Robot a' will arrive at w no later than robot a will arrive at v for the same reason. Thus, both robots traverse their microedges from the time when robot a leaves u to the time when robot a' arrives at w. Their distance in the graph is at least δ at those times according to Lemma 7.1.4. Since both robots move with constant speeds, their distance in the graph is at least δ between those times as well. □

Theorem 7.1.2. *The Euclidean distance between the center of any two circular robots is at least $\delta/\sqrt{2}$ for four-neighbor square grids $\mathcal{G}_E$ and six-neighbor cubic grids $\mathcal{G}_E$ at all times.*

Proof. Consider two robots a and a'. Their distance in the graph is at least δ according to Theorem 7.1.1. We distinguish three cases and show that their Euclidean distance is at least $\delta/\sqrt{2}$ in each case:

First, both robots are on the same microedge. Then, their Euclidean distance is at least δ according to Theorem 7.1.1.

Second, both robots are on incident microedges. Let robot a be on a microedge (starting at location u and ending at location v) at distance x from v and robot a' be on a microedge (starting at location v and ending at location w) at distance y from v. Then, their distance in the graph is $x + y \geq \delta$. If the two microedges are parallel, then the Euclidean distance between the two robots is equal to their distance in the graph and thus $x + y \geq \delta$. If the two microedges are orthogonal, then the Euclidean distance between the two robots is $\sqrt{x^2 + y^2}$ with $\sqrt{x^2 + y^2} \geq \delta/\sqrt{2}$ since $x^2 - 2xy + y^2 = (x-y)^2 \geq 0 \Rightarrow x^2 + y^2 \geq 2xy \Rightarrow 2(x^2+y^2) = 2x^2 + 2y^2 \geq x^2 + y^2 + 2xy \geq (x+y)^2 \geq \delta^2$ $\Rightarrow \sqrt{x^2 + y^2} \geq \delta/\sqrt{2}$.

Third, both robots are on non-incident microedges. If these two microedges are on the same macroedge then the robots' Euclidean distance is at least δ according to Theorem 7.1.1. If these two microedges are on parallel macroedges then the Euclidean distance is at least the length of one macroedge, which is greater than δ. If these two microedges are on orthogonal macroedges then a similar calculation as in the second case (incident microedges) applies. □

7.1.6.2 Complexity

We first show that Algorithm 7.2 terminates for realistic values of δ and then remark on its time and space complexities.

Theorem 7.1.3. *Algorithm 7.2 terminates if $\mathcal{G}_E$ has uniform edges of rational length and $\delta \in \mathbb{Q}^+$.*

Proof. Let $\delta = a/b$, where $a, b \in \mathbb{Z}^+$, and let $l = c/d$ be the uniform edge length, where $c, d \in \mathbb{Z}^+$. The set of possible distances of auxiliary locations to main locations is given by $D = B \cup \{l - e | e \in B\}$, where $B = \{(i \cdot \delta) \,(\mathrm{qmod}\, l) | i \in \mathbb{Z}^+\}$. Here, (qmod) is the modulus for rational numbers defined such that $(k \cdot l + \delta)(\mathrm{qmod}\, l) = \delta, k \in \mathbb{Z}^+$. We can rewrite $B = \{(i \cdot \frac{a \cdot d}{b \cdot d}) \,(\mathrm{qmod}\, \frac{b \cdot c}{b \cdot d}) | i \in \mathbb{Z}^+\} = \{((i \cdot ad) \,(\mathrm{qmod}\, cb))/(bd) | i \in \mathbb{Z}^+\}$, which is a finite set because $\mathbb{Z}^+/(cb)$ is a finite group. Thus, D is a finite set as well. Therefore, the number of auxiliary locations is finite.

Steps 1 and 2 in Algorithm 7.2 use loops with finite bounds and thus terminate. Step 3 in Algorithm 7.2 (see Algorithm 7.3) terminates because the number of auxiliary locations is finite. □

For any given potentially irrational δ, we can execute Algorithm 7.2 by conservatively transforming δ to a close-enough rational number. If $\mathcal{G}_E$ has non-uniform or irrational edge length, Algorithm 7.2 might not terminate; however, if it does terminate, the computed schedule is collision-free.

The TPG for a discrete plan for N robots with makespan T has $O(NT)$ vertices corresponding to main locations and $O(N^2T)$ edges. Algorithm 7.2 creates more vertices and edges corresponding to auxiliary locations. The total size of the TPG depends on δ, but remains finite by virtue of Theorem 7.1.3 and increases the complexity of the algorithms that work on it only by a constant factor.

Algorithm 7.2 constructs the TPG in $O(N^2T^2)$ time. The STN bounds can be computed in constant time for any given edge, resulting in $O(N^2T)$ time.

We can compute a schedule by executing the Bellman-Ford algorithm twice: once on the distance graph starting from X_F and once on the distance graph with all edges reversed starting from X_S. Since our STN has $O(NT)$ vertices and $O(N^2T)$ edges, we can compute a schedule in $O(N^3T^2)$ time. Thus, the overall runtime complexity of MAPF-POST is $O(N^3T^2)$ and its space complexity is $O(N^2T)$.

7.1.6.3 Existence of Solution

In the following we prove that MAPF-POST can compute a schedule for any given discrete plan as long as there are no specified minimum speed limits.

Lemma 7.1.5. *The TPG as constructed by Algorithm 7.2 is a directed acyclic graph.*

Proof. We use the notation $v^j_{(t,d)}$ for each vertex in the TPG that corresponds to an auxiliary or main location, where j indicates that the vertex corresponds to robot a_j, t indicates the discrete timestep of the current or previous main location in the discrete plan, and d indicates the distance to the current or previous main location of a_j. We define a relation $v^{j'}_{(t',d')} \succ v^j_{(t,d)}$ iff $t' > t$ or $t' = t$ and $d' > d$. This relation defines a valid partial order on the vertices of the TPG, because the tuple (t, d) is composed of ordinal values. It now suffices to prove that each edge (x, x') in the TPG respects this partial order, i.e., $x' \succ x$.

By construction, Type 1 edges connect consecutive vertices associated with a path for each robot (Line 9). Therefore, Type 1 edges respect the partial order.

There are five places where Algorithm 7.2 adds Type 2 edges of the form $(v^j_{(t,d)}, v^{j'}_{(t',d')})$. The three places in Lines 25, 26 and 32 respect this order because $t' \geq t + 1$. The remaining two places in Algorithm 7.3 (Lines 16 and 19) invoked by Algorithm 7.2 also respect this order. This is because these lines require the condition that $t' \geq t + 1$ or $t' = t$ and $d' - d = \delta$.

□

Theorem 7.1.4. *There always exists a plan-execution schedule that is consistent with the simple temporal constraints of the STN for a MAPF plan and assigns finite plan-execution times to all vertices in the augmented TPG, if no minimum speed limits are specified.*

Proof. By Lemma 7.1.5, the constructed TPG is a directed acyclic graph. The distance graph has one forward and one reverse edge for each edge in the TPG. All reverse edges in the distance graph are finite negative, since all lower bounds in the STN represent positive and finite speed limits. However, all forward edges in the distance graph are infinity, since all upper bounds in the STN represent minimum speed limits that are not specified. This guarantees that there are no negative-cost cycles in the distance graph. Thus, there exists a plan-execution schedule that satisfies all simple temporal constraints [39]. The earliest plan-execution time of any vertex v in the augmented TPG is the negative of the cost of a shortest path from v to X_S in the distance graph and is thus finite (since all lower STN bounds are finite). □

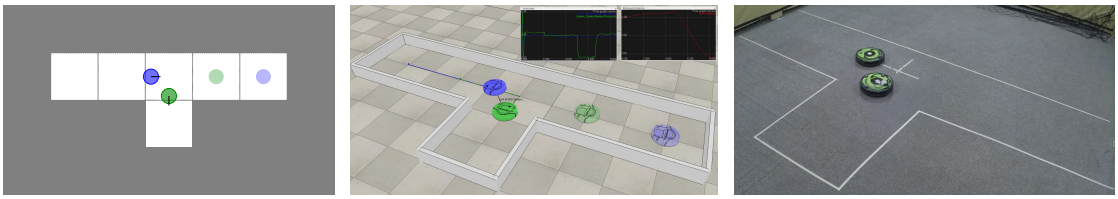
In the case of robots with minimum speed limits, although Theorem 7.1.4 is no longer true, MAPF-POST will detect and report such a case correctly by finding a negative cost cycle in the distance graph. Unfortunately, however, such a negative cost cycle only shows that there is no such schedule for the given discrete plan. Another discrete plan, perhaps with the same makespan, might very well lead to a valid continuous schedule.

7.2 Extensions

We now discuss how MAPF-POST can be used for differential drive robots by taking rotation actions into account. Furthermore, we discuss how to minimize the representation of the TPG to reduce the solving time.

7.2.1 Differential Drive Robots

So far, we have assumed that the robots are holonomic and thus able to move in all directions. However, many robots, such as differential-drive robots, are non-holonomic. A differential-drive robot cannot move sideways; instead, it has to first turn and then continue moving forward (it can also move forward while turning). Such a rotation takes additional time, which our standard MAPF formulation does not take into account. If we model the robot closer to reality, we can set the maximum speed of the robots closer to their nominal speeds without risking safety distance



(a) Agent simulation. **(b) V-REP simulation.** **(c) Implementation on Create2 robots.**

Figure 7.9: Screenshots of the three validation settings for our running example.

violations. We address the problem during discrete planning and MAPF-POST. We make the following changes to accommodate differential-drive robots that operate on grid-worlds. First, we change the definition of the MAPF problem and adapt the MAPF solver appropriately. Vertices now are pairs of locations (cells) and orientations (discretized into the four compass directions) and actions either wait, move forward to the next location or rotate in place 90 degrees clockwise or counter-clockwise. Edges and collisions change accordingly. Second, when we determine the routes from a MAPF plan we remove the wait actions but keep the move and rotate actions. Then, we merge several consecutive rotate actions into one rotate action whose rotation angle is the sum of the merged rotations (and delete the rotate action if its rotation angle is zero). Third, we adapt the placement of Type 2 edges in the TPG since two consecutive vertices on a route can now correspond to the same location (with different orientations). Type 2 edges connect the last such vertex on a route to the first such vertex on another route. Fourth, we split those Type 1 edges (and introduce safety markers) that correspond to move actions but not those Type 1 edges that correspond to rotate actions. Fifth, we associate the STN bounds $[L(e)/V^*_{\max}(e), L(e)/V^*_{\min}]$ if $V^*_{\min} > 0$ and $[L(e)/V^*_{\max}(e), \infty]$ if $V^*_{\min} = 0$ with a Type 1 edge $e \in \mathcal{E}_{STN}$ that corresponds to a rotate action, where $L(e)$ is the absolute value of the rotation angle and $V^*_{\max}(e)$ and $V^*_{\min}(e)$ are the maximum and minimum rotational speed limits, respectively. Sixth, Theorem 7.1.1—as stated in this chapter—requires a non-zero minimum translational speed $v_{\min}$ to guarantee a positive safety distance but we can simply assume for the purpose of the theorem that non-holonomic robots do not rotate in place in the locations associated with location vertices but move slowly toward the locations of the next safety markers on their paths while rotating.

7.2.2 Parsimonious Representation

Algorithm 7.2 avoids creating some transitive edges, however it is still possible that some Type 2 edges are implied by transitivity. In particular, the algorithm only considers pairs of robots and not robot groups with more than two members. It is possible to remove transitive edges in polynomial time for directed graphs [1] and even linear time for specific kinds of acyclic directed graphs [64]. It requires less time to find a schedule for an STN with fewer edges. There are two advantages in computing the transitive reduction. First, it might lead to an overall faster running time, if the time for the graph reduction and finding a schedule for the reduced graph is smaller than the time required to find a schedule for the original graph. Second, the number of removed edges gives an insight into the number of unnecessarily created edges in Algorithm 7.2. We present empirical results of that approach in Section 7.3.

7.3 Experimental Validation

We implemented MAPF-POST in C++ using the `boost graph` library [133] for the graph operations. All experiments were run on a laptop computer with an i7-4600U 2.1 GHz processor and 12 GB

Table 7.1: Number of vertices, edges, and runtimes for different problem instances and safety distances.

		$\delta = 0.5$				$\delta = 0.8$			
N	T	$\lvert\mathcal{V}_{TPG}\rvert$	$\lvert\mathcal{E}_{TPG}\rvert$	t_{TPG} [s]	t_{solve} [s]	$\lvert\mathcal{V}_{TPG}\rvert$	$\lvert\mathcal{E}_{TPG}\rvert$	t_{TPG} [s]	t_{solve} [s]
50	29	1320	3074	0.1	0.1	3125	7540	0.1	0.1
100	44	3563	9293	0.2	0.1	8470	22823	0.2	0.6
150	63	6797	19109	0.6	0.2	16389	47221	0.7	2.4
200	61	9642	27416	1.0	0.3	23353	67853	1.3	3.7
250	65	13952	40934	2.0	0.6	34031	101631	2.3	8.3
300	79	17839	54521	3.6	1.0	43692	135639	4.4	13.4
350	90	22355	68773	6.0	1.3	54891	171220	6.6	20.8
400	91	27448	86850	8.6	2.1	67469	216307	9.6	26.5

RAM. We validated our approach experimentally in three different settings, namely using an agent simulation (which implements the uniform speed model perfectly), the robot simulation tool V-REP, and an implementation on actual robots. Pictures of the different approaches are shown in Fig. 7.9. A supplemental video showcasing some of our experiments is available at `https://youtu.be/Da7PzIrLnfM`.

7.3.1 Agent Simulation

We discuss the runtime behavior of MAPF-POST for practical problems in four different perspectives. In all cases, we used an ECBS solver to compute the discrete plans for MAPF instances. First, we look at the different runtime for varying number of robots (N) and discrete makespan T. Second, we discuss the influence of the user-provided safety distance δ for the runtime of MAPF-POST. Third, we run various experiments to compare original and parsimonious representations of the TPG. Finally, we demonstrate the effect of re-solving the STP to avoid replanning in dynamic environments.

7.3.1.1 Varying Number of Robots and Makespan

To determine the runtime for practical problems, we randomly generate a grid environment with $1\,\text{m} \times 1\,\text{m}$ cells and 10% occupied grid cells. The robots' start and goal locations and occupied grid cells are randomly assigned. We use ECBS [11] as our discrete solver. Table 7.1 shows some of our results for varying numbers of robots and two different safety distance settings. The time to find a schedule varies depending on the number of robots (N), the makespan of the discrete plan (T), and the safety distance (δ). We report the number of created vertices ($\lvert\mathcal{V}_{TPG}\rvert$), the number of created edges ($\lvert\mathcal{E}_{TPG}\rvert$), the time to execute Algorithm 7.2 (t_{TPG}), and the time to solve the STN (t_{solve}). The larger safety distance of $\delta = 0.8\,\text{m}$ creates more vertices and edges compared to $\delta = 0.5\,\text{m}$, because of the `addAdditionalType2Edges` function. The increased number of vertices and edges leads to a larger runtime for solving the STN compared to $\delta = 0.5\,\text{m}$.

7.3.1.2 Influence of Safety Distance

We execute MAPF-POST on the environment shown in Fig. 7.10. Here, 100 robots try to cross to their respective opposite sides. We record the achieved minimum distance as reported by numerically executing the path, the number of vertices and edges of the TPG, the runtime to compute a continuous plan-execution schedule, and the achieved makespan for different user-provided safety distances. The results are shown in Table 7.2. It is noticeable that the STN solving runtime t_{solve} varies a lot, between 0.5 s and 13.5 s. Furthermore, the achieved safety distances are always very

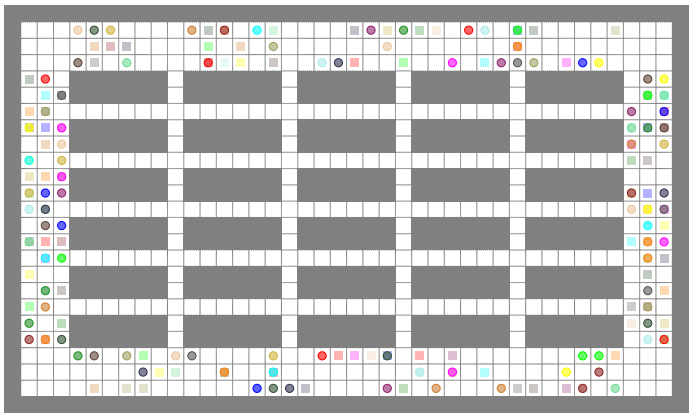


Figure 7.10: Example environment with 100 robots for the analysis of the influence of the requested safety distance δ. Each robots start location is shown as a circle and its goal location is on the opposite side from the start location, marked as a colored square.

Table 7.2: Runtime of MAPF-POST for different safety distance δ for the example shown in Fig. 7.10.

δ [m]	$\frac{\delta}{\sqrt{2}}$ [m]	$minDist$ [m]	$\lvert\mathcal{V}_{TPG}\rvert$	$\lvert\mathcal{E}_{TPG}\rvert$	t_{TPG} [s]	t_{solve} [s]	$makespan$ [s]
0.1	0.07	0.10	42596	83706	0.8	1.8	33.0
0.2	0.14	0.20	21816	46416	0.5	1.1	33.1
0.25	0.18	0.18	17660	38958	0.6	0.9	33.1
0.3	0.21	0.22	40374	87335	0.8	8.8	33.1
0.4	0.28	0.28	21978	50437	0.6	2.7	33.2
0.5	0.35	0.36	9157	23851	0.4	0.5	33.2
0.6	0.42	0.45	21073	50177	0.6	3.2	33.3
0.7	0.49	0.50	29404	72827	0.7	4.9	33.3
0.8	0.57	0.57	22603	59415	0.6	4.7	33.4
0.9	0.64	0.64	44637	118006	0.9	13.5	33.4

closely above the requested safety distances. For $\delta = 0.5\,\text{m}$ a schedule was computed quickest. The search graph for our example is a grid with all edges being 1 m long. Therefore, a safety distance of 0.5 m leads to a TPG, where the only safety markers might be added in the middle of two vertices indicating main locations. Thus, the TPG contains significantly fewer edges and vertices compared to TPGs created by using a different δ. Requested safety distances that divide the edge length (in this example $\delta \in \{0.1, 0.2, 0.25, 0.5\}$) tend to result in faster solving times even if the number of edges and vertices are similar. For example, the result for $\delta = 0.2$ and $\delta = 0.4$ results in similar number of edges and vertices, but the time to find a schedule is longer in the latter. The makespan increases with higher requested safety distances, but not significantly.

7.3.1.3 Parsimonious Representation

We generated 39 random environments with varying number of robots, start- and goal locations with a cell size of 1 m × 1 m. For each case, we execute MAPF-POST for safety distances $\delta \in \{0.2, 0.5, 0.8\}$ and record the runtime with and without the transitive reduction as well as statistics about the number of edges in the TPG. To compute the transitive reduction, we implemented an existing algorithm [64]. The number of edges was reduced by 33 % on average (standard deviation: 0.07 %). The smaller graphs resulted in the solving time being reduced by 23 % on average. However,

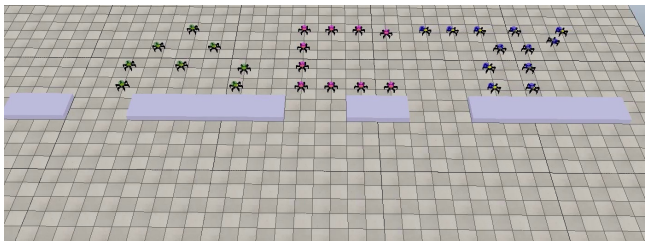


Figure 7.11: Formation-change example with V-REP, where three groups of spider-like robots form different letters.

computing the transitive reduction took longer than the time saving during solving on average, resulting in an average increase of the runtime of 11 % when the transitive reduction was enabled.

Those results indicate that it is not advisable to use the parsimonious representation for one-time schedule computations.

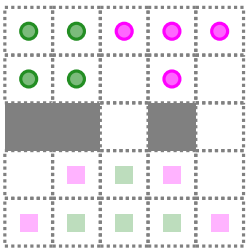
7.3.2 Simulations

We run robot simulations using the robot simulator V-REP for three different robots: iRobot Create2 differential drive robots, spider-like six-legged robots, and quadcopters. For the Create2, we ensure the similarity of the robot simulation and implementation on actual robots by using the robot operation system ROS as middleware and implementing a robot controller directly in ROS that drives either virtual robots in V-REP or actual robots. The controller controls the state $[x, y, \theta]^T$ and tries to meet the deadline specified by the plan-execution schedule of MAPF-POST by setting the translational (or rotational) speed of a robot to the ratio of the remaining time to reach the next location (or orientation) and the remaining translational (or rotational) distance, which approximates the uniform velocity model well. A PID-controller corrects for heading error and drift by driving the two motors independently and using the known location of the robot from either the simulation or motion-capture system. Here, the controllers are sufficient to absorb deviations from the planned path and neither re-solving the STP nor replanning was used.

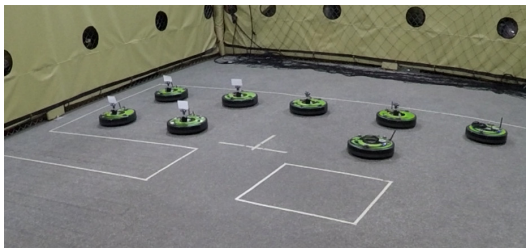
For the other robots we use the controllers provided by V-REP. A screenshot of the Create2 simulation is shown in Fig. 7.9b; examples of the other simulation experiments are shown in Fig. 7.11. Compared to the simulated example, the environments and number of robots are small, resulting in runtimes of less than a second for MAPF-POST.

7.3.3 Physical Robot Experiments

We use eight iRobot Create2 robots equipped with single-board computers (such as ODROID C1+), that interface to the robots via their serial ports. The computers run the PID controller on Ubuntu 14.04 with ROS Jade and communicate via WiFi with a single `roscore` on a host computer that runs MAPF-POST. Localization is provided by a 12-camera VICON MX optical motion-capture system in a space approximately 5 m × 4 m in size. The discrete environment uses a grid cell size of 0.75 m × 0.75 m. Our chosen value of $\delta = 0.74\,\text{m}$ guarantees a safety distance of $0.74\,\text{m}/\sqrt{2} \approx 0.52\,\text{m}$, assuming perfect execution on holonomic robots. To prevent collisions of the physical non-holonomic robots (whose diameter is 0.35 m), the improved MAPF-POST limits their maximum translational speed to 0.2 m/s, even though the robots can move with a translational speed of up to 0.5 m/s. This parameter adjustment was sufficient to absorb deviations from the planned path in the controller. Figures 7.12a and 7.12b show an example formation-change instance on the robots.



(a) Multi-color MAPF instance with two groups of four robots each, differentiated by color. The start formation is on top, and the goal formation is at the bottom.



(b) Formation-change instance on Create2 robots that corresponds to the multi-color instance in (a). The four robots with a white flag (on the left) belong to the same group.

Figure 7.12: Physical robot experiment.

7.4 Remarks

We described MAPF-POST, a post-processing step which takes the output of any discrete multi-robot planner and produces a continuous schedule in polynomial time. This method allows the use of existing planners, which work well for hundreds of robots and provide sub-optimality guarantees, with physical robots. Thus, we combine the scalability of AI solvers and the practicality of planners in robotics to efficiently solve problems which arise in real-world scenarios such as warehousing and towing.

MAPF-POST has the following features: (a) it incorporates kinodynamic constraints of the robots such as speed limits; (b) it maintains a user-specified safety-distance; (c) it attempts to minimize makespan; and (d) it runs efficiently for hundreds of robots. We demonstrated the efficiency and versatility in several simulations and on physical robots.

One of the key insights of MAPF-POST, the ability to extract dependencies from a given schedule in polynomial time, is also used for the execution framework presented in Chapter 10.

CHAPTER 8

Task and Motion Planning for Aerial Vehicles

We now consider the formation change problem for a team quadrotors in obstacle-rich environments. It is possible to use a graph representation of the 3-dimensional environment, execute MAPF and MAPF-POST to find and execute a schedule [170]. However, the quadrotors would need to operate at low speeds or stop at each vertex. Additionally, this model ignores the downwash effect which disallows quadrotors to fly in close vertical proximity.

In the following, we present an optimization-based postprocessing step that can compute smooth trajectories that can be safely executed on a team of quadrotors.

8.1 Approach

We now formalize the trajectory planning problem for any homogeneous robot team and outline our method to solve this class of problems. Furthermore, we introduce the robot model for quadrotors. Later sections discuss the steps of our approach in more detail, using a quadrotor swarm as example.

8.1.1 Problem Statement

Consider a team of N robots in a bounded environment containing convex obstacles $\mathcal{O}_1, \ldots, \mathcal{O}_{N_{obs}}$. Boundaries of the environment are defined by a convex polytope $\mathcal{W}$. The convex set of points representing a robot at position $q \in \mathbb{R}^3$ is $\mathcal{R}_\mathcal{E}(q)$. The free configuration space for a single robot is thus given by

$$\mathcal{F} = \left(\mathcal{W} \setminus \left(\bigcup_{h=1}^{N_{obs}} \mathcal{O}_h \right) \right) \ominus \mathcal{R}_\mathcal{E}(\mathbf{0}) \tag{8.1}$$

where $\ominus$ denotes the Minkowski difference. We allow a separate inter-robot collision model and define $\mathcal{R}_\mathcal{R}(q)$ to be the convex set of points a robot at position q requires to operate safely when close to another robot. For example, in the case of quadrotors $\mathcal{R}_\mathcal{R}(q)$ can model the downwash effect.

This chapter is based on **Wolfgang Hönig**, James A. Preiss, T. K. Satish Kumar, Gaurav S. Sukhatme, and Nora Ayanian. "Trajectory Planning for Quadrotor Swarms". In: *IEEE Transactions on Robotics, Special Issue on Aerial Swarm Robotics* 34.4 (2018), pp. 856–869. DOI: 10.1109/TRO.2018.2853613.

James A. Preiss was a PhD student and mostly worked on continuous portion: corridor computation, trajectory optimization, and iterative refinement. I mostly worked on the discrete portion: Octomap environment mapping, roadmap generation, and MAPF/C solver. Experiments were executed jointly.

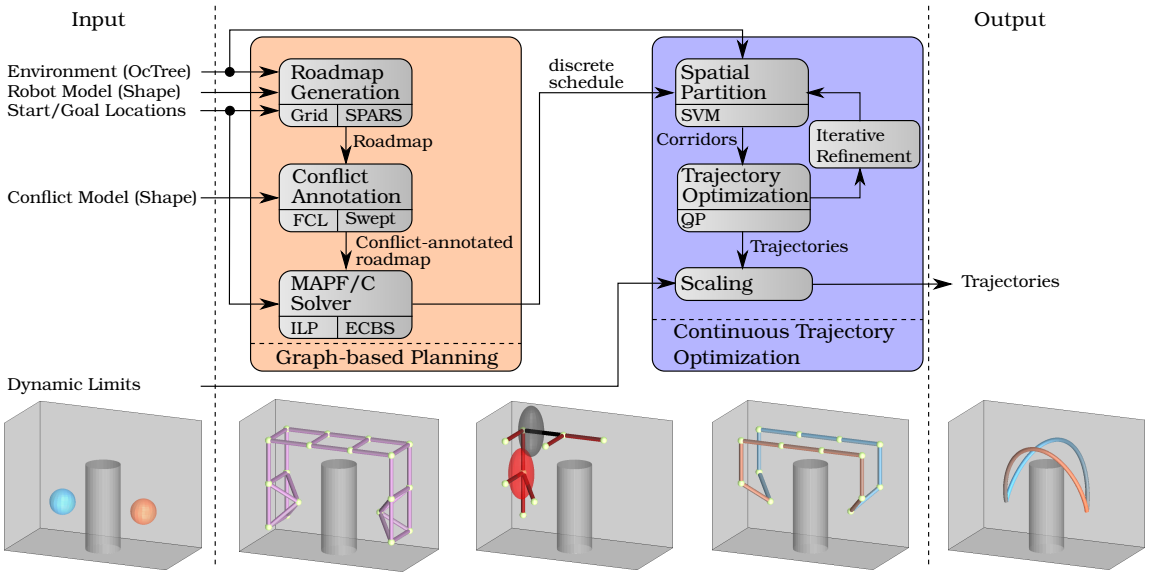


Figure 8.1: Components of our approach. The user specifies a model of the environment (e.g. using an octree), a model of the robot (e.g. sphere), start/goal locations, a robot-robot collision model (e.g. ellipsoid), and dynamic limits (e.g. maximum acceleration). We generate a sparse roadmap from the environment, including the start and goal locations as vertices. The collision model is used to annotate the roadmap with additional inter-robot conflicts. This annotated roadmap can be used to find a discrete schedule for each robot. This schedule defines a corridor for each robot in which trajectory optimization is used to generate smooth trajectories. Finally, the trajectories are scaled in order to fulfill the dynamic limits. The pictures on the bottom show one example of two quadrotors swapping their positions.

Let $f^i : [0, T] \to \mathbb{R}^3$ be a trajectory for each robot r^i, where $T \in \mathbb{R}_{>0}$ is the total time duration until the last robot reaches its goal. All trajectories are considered *collision-free* if there are no robot-environment collisions, i.e. $f^i(t) \in \mathcal{F}$ and no robot-robot collisions, i.e.

$$\mathcal{R}_{\mathcal{R}}(f^i(t)) \cap \mathcal{R}_{\mathcal{R}}(f^j(t)) = \emptyset \quad \forall\, i \neq j,\ 0 \leq t \leq T. \tag{8.2}$$

In the *labeled* trajectory planning problem we are given a start and goal position for each robot $s^i, g^i \in \mathcal{F}$, where start and goal inputs must satisfy the robot-robot collision model, i.e. $\mathcal{R}_{\mathcal{R}}(s^i) \cap \mathcal{R}_{\mathcal{R}}(s^j) = \emptyset$ and $\mathcal{R}_{\mathcal{R}}(g^i) \cap \mathcal{R}_{\mathcal{R}}(g^j) = \emptyset$ for all $i \neq j$.

We seek the total time duration T and collision-free trajectories f^i such that:

- $f^i(0) = s^i$
- $f^i(T) = g^i$
- f^i is continuous up to user-specified derivative order C
- f^i is kinodynamically feasible for robot i.

In the *unlabeled* case we allow the robots to exchange goal locations, i.e. we additionally seek an assignment of each robot to a goal position such that $f^i(T) = g^{\phi(i)}$, where ϕ is a permutation of $1, \ldots, N$.

8.1.2 Components

Our method combines the strengths of two conceptually different approaches to multi-robot trajectory planning problems. First, we use MAPF/C by discretizing time and space to find collision-free paths for all robots. This can be done efficiently for hundreds of robots even in maze-like environments. However, dynamic constraints of robots are ignored. Furthermore, creating an appropriate roadmap for an environment that fulfills all requirements of the search algorithm is challenging. The second approach, commonly used in robotics, is based on trajectory optimization. This approach can deal with kinodynamic constraints, but does not scale well to hundreds of robots.

Our approach is outlined in Fig. 8.1. We use models of the environment and the robots to generate a roadmap suitable for planning for a single robot. We annotate the roadmap with additional edge and vertex conflicts to model constraints caused by inter-robot dependencies. We can then use an extended multi-robot planner to find discrete schedules for each robot. These schedules can be executed on real quadrotors without collisions, but they require the quadrotors to stop at each waypoint. Finally, a trajectory optimization stage generates smooth and conflict-free trajectories based on the discrete schedule. This approach can be used for any multi-robot trajectory planning problem, however the exact details for each step vary. In this chapter we present the components needed to plan trajectories for a swarm of quadrotors, directly taking downwash into account.

8.1.3 Robot Model for Quadrotor Trajectory Planning

As aerial vehicles, quadrotors have a six-dimensional configuration space. However, as shown in [103], quadrotors are *differentially flat* in the *flat outputs* (x, y, z, ψ), where x, y, z is the robot's position in space and ψ its yaw angle (heading). Differential flatness implies that the control inputs needed to move the robot along a trajectory in the flat outputs are algebraic functions of the flat outputs and a finite number of their derivatives. Furthermore, in many applications, a quadrotor's yaw angle is unimportant and can be fixed at $\psi = 0$. We therefore focus our efforts on planning trajectories in three-dimensional Euclidean space.

While some multi-robot planning work has considered simplified dynamics models such as kinematic agents [170] or double-integrators [8], our method produces trajectories with arbitrary smoothness up to a user-defined derivative. This goal is motivated by [103], where it was shown that a continuous fourth derivative of position is necessary for physically plausible quadrotor trajectories, because it ensures that the quadrotor will not be asked to change its motor speeds instantaneously.

Rotorcraft generate a large, fast-moving volume of air underneath their rotors called *downwash*. The downwash force is large enough to cause a catastrophic loss of stability when one rotorcraft flies underneath another. We model downwash constraints as inter-robot collision constraints by treating each robot as an axis-aligned ellipsoid of radii $0 < r_x = r_y \ll r_z$, illustrated in Fig. 8.2. Empirical data collected in [177, 105] support this model. The set of points representing a robot at position $q \in \mathbb{R}^3$ is given by

$$\mathcal{R}_{\mathcal{R}}(q) = \{Ex + q : \|x\|_2 \leq 1\} \tag{8.3}$$

where $E = \mathbf{diag}(r_x, r_y, r_z)$.

8.2 Roadmap Generation

A roadmap is an undirected connected graph of the environment $\mathcal{G}_E = (\mathcal{V}_E, \mathcal{E}_E)$, where each vertex $v \in \mathcal{V}_E$ corresponds to a location in $\mathcal{F}$ and each edge $(u, v) \in \mathcal{E}_E$ denotes that there is a linear path in $\mathcal{F}$ connecting u and v. We also require that there exists a vertex $v_s^i \in \mathcal{V}_E$ corresponding to each start location s^i and that there exists a vertex $v_g^i \in \mathcal{V}_E$ for each goal location g^i. Let $loc : \mathcal{V}_E \to \mathbb{R}^3$ be a function that returns the location for each vertex.

The ideal roadmap should have the following properties:

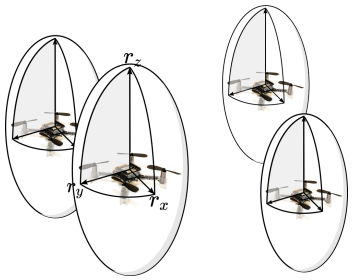
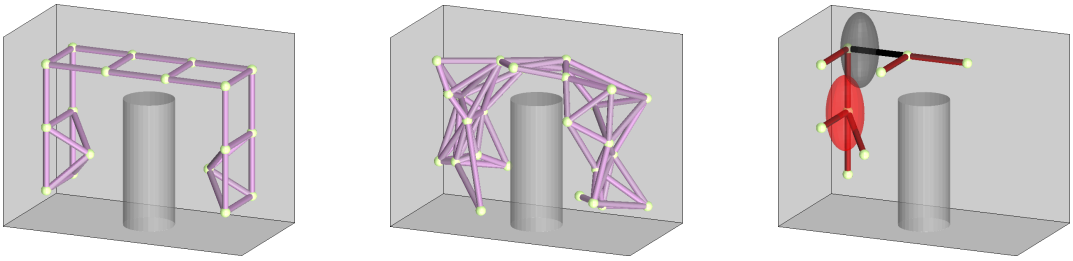


Figure 8.2: Axis-aligned ellipsoid model of robot volume. Tall height prevents downwash interference between quadrotors.



(a) Grid-based roadmap. **(b) SPARS-based roadmap.** **(c) Conflicting edges.**

Figure 8.3: Example of generated roadmaps using the grid approach (a) and SPARS (b) with the same desired dispersion. The roadmaps can be annotated with edge and vertex conflicts based on the inter-robot conflict model $\mathcal{R}_\mathcal{R}$. Conflicting edges for black edge are marked in red in (c). The edges are in conflict because it is possible to place the robots on the edge such that Eq. (8.2) is violated.

1. be connected, i.e., if a path between two points in $\mathcal{F}$ exists, there should be a path in the roadmap as well;
2. lead to optimal results, i.e., the shortest path between two points in $\mathcal{F}$ can be approximated well by a path in the roadmap; and
3. be sparse, i.e., have a small number of vertices and edges.

The last property is desired because dense roadmaps result in a higher number of inter-robot conflicts, which can create a significant computational burden for the discrete planning stage. The first two properties are in conflict with the last property. Thus, roadmap generation should balance those goals. A useful property of a roadmap is its *dispersion*, which is the radius of the largest sphere centered in $\mathcal{F}$ that does not contain any vertex location. Dispersion is a measure of the uniformity of the roadmap. In this work we tested two different approaches, namely 6-connected grid structures, and sparse roadmap spanners. Here, *spanner* refers to a concept from graph theory where a spanner of a graph is a subgraph with fewer edges such that the distance between any two vertices is within a user-provided bound compared to the original graph.

The grid roadmap generator places potential vertices equidistant from each other. A vertex is only added if it is in $\mathcal{F}$. Edges are added to the six closest neighbors if there would be no collision between the robot and the environment when traversing that edge. Roadmaps generated this way have a regular structure, reducing inter-robot conflicts and providing a low dispersion for a fixed number of vertices. However, this approach is resolution-complete and completeness and optimality are only achieved as the grid-size approaches zero, which is not desirable due to our sparseness

requirement. Thus, the generated roadmaps are often missing important edges and vertices and fail to achieve the first two desired properties.

The family of SPArse Roadmap Spanner algorithms (*SPARS*) [41] attempts to generate roadmaps that are bounded sub-optimally with respect to distance (up to a user-provided stretch factor). Unlike graph spanners, the algorithm attempts to reduce both vertices and edges, rather than edges only. The algorithms are probabilistically complete [90], i.e., the probability that a path is found if one exists converges to one as more states are sampled. The SPARS algorithm uses a dense roadmap similar to those generated by PRM* [81] and only keeps a subset of vertices and edges such that the distance sub-optimality is guaranteed. The SPARS2 algorithm removes the requirement of explicitly constructing a dense roadmap, reducing the memory overhead but at the cost of a larger roadmap. We do not have a requirement for a small memory usage and used the SPARS algorithm.

In both grid and SPARS cases we add vertices to the roadmap corresponding to the start and goal locations if such vertices are not already part of the roadmap. We connect those additional vertices to up to six neighbors within a search radius if the resulting edge could be traversed without any collision with the environment. Example roadmaps for the same environment are shown in Fig. 8.3.

8.3 Conflict Annotation

Roadmaps are typically generated to plan the motion for a single robot. If the same roadmap is used by multiple robots, there are additional constraints:

1. Vertex-Vertex Constraints: two robots may not occupy two vertices that are in close proximity to each other at the same time.
2. Edge-Edge Constraints: two robots may not traverse two edges if a collision could occur during the traversal, see Fig. 8.3c.
3. Edge-Vertex Constraints: one robot may not traverse an edge if a collision could occur with another stationary robot.

The goal of the conflict annotation step is to identify the set of conflicts for each edge and vertex. The output is a graph where each vertex and each edge is annotated with a conflict set.

We define the following functions:

$$\begin{aligned} conVV(v) &= \{u \in \mathcal{V}_E \,|\, \mathcal{R}_\mathcal{R}(loc(u)) \cap \mathcal{R}_\mathcal{R}(loc(v)) \neq \emptyset\} \\ conEE(e) &= \{d \in \mathcal{E}_E \,|\, \mathcal{R}^*_\mathcal{R}(d) \cap \mathcal{R}^*_\mathcal{R}(e) \neq \emptyset\} \\ conEV(e) &= \{u \in \mathcal{V}_E \,|\, \mathcal{R}_\mathcal{R}(loc(u)) \cap \mathcal{R}^*_\mathcal{R}(e) \neq \emptyset\} \end{aligned} \tag{8.4}$$

where $\mathcal{R}^*_\mathcal{R}(e)$ is the set of points swept when traversing edge e. We refer to this method as *swept* collision model.

The swept collision model is conservative, allowing edge traversals with an arbitrary velocity profile. Alternatively, we can assume that the motion on the edge uses a known velocity profile (such as constant velocity), where $mot(e,t)$ refers to the position of the robot traversing edge e for $t \in [0,1]$. We can then use a standard continuous collision definition as used in the Flexible Collision Library (FCL) [112]:

$$\begin{aligned} conEE'(e) = \{&d \in \mathcal{E}_E \,|\, \exists t \in [0,1] \\ &\mathcal{R}_\mathcal{R}(mot(d,t)) \cap \mathcal{R}_\mathcal{R}(mot(e,t)) \neq \emptyset\} \\ conEV'(e) = \{&u \in \mathcal{V}_E \,|\, \exists t \in [0,1] \\ &\mathcal{R}_\mathcal{R}(loc(u)) \cap \mathcal{R}_\mathcal{R}(mot(e,t)) \neq \emptyset\} \end{aligned} \tag{8.5}$$

Collision checking is required for all vertex and edge pairs, leading to quadratic time complexity.

8.4 Discrete Planning

We can formulate our discrete planning problem as an unlabeled MAPF/C instance (see Section 6.3). The output of a MAPF/C solver is a discrete schedule p^i for each robot that is composed of a sequence of $K+1$ locations:

$$\begin{aligned} p^i &= loc(u_0^i), loc(u_1^i), \ldots, loc(u_K^i) \\ &\triangleq x_0^i, x_1^i, \ldots, x_K^i. \end{aligned} \tag{8.6}$$

Robots are expected to be synchronized in time, such that robot i is at waypoint x_k^i at timestep k. In between waypoints x_k^i and x_{k+1}^i, we assume that robot i travels on the line segment between x_k^i and x_{k+1}^i. We denote this line segment by ℓ_k^i.

Because we are interested in minimizing the makespan, we use the Threshold algorithm for task assignment (see Section 6.3.3) before solving the MAPF/C instance with our ECBS-based solver.

8.5 Trajectory Optimization

In the continuous refinement stage, we convert the waypoint sequences p^i generated by the discrete planner into smooth trajectories f^i. We use the discrete plan to partition the free space $\mathcal{F}$ such that each robot solves an independent smooth trajectory optimization problem in a region that is guaranteed to be collision-free. We assign a time $t_k = k\Delta t$ to each discrete timestep, where Δt is a user-specified parameter. This is an initial guess for $T = K\Delta t$. The exact total time T is computed in a post-processing stage such that all trajectories meet given physical limits such as a maximum thrust constant.

8.5.1 Spatial Partition

The continuous refinement method begins by finding *safe corridors* within the free space $\mathcal{F}$ for each robot. An example from a real problem instance is shown in Fig. 8.4. The safe corridor for robot r^i is a sequence of convex polyhedra $\mathcal{P}_k^i, k \in \{1, \ldots, K\}$, such that, if each r^i travels within $\mathcal{P}_k^i$ during time interval $[t_{k-1}, t_k]$, both robot-robot and robot-obstacle collision avoidance are guaranteed. For robot r^i in timestep k, the safe polyhedron $\mathcal{P}_k^i$ is the intersection of:

- $N-1$ half-spaces separating r^i from r^j for $j \neq i$;
- N_{obs} half-spaces separating r^i from $\mathcal{O}_1, \ldots, \mathcal{O}_{N_{obs}}$.

We separate r^i from r^j by finding a separating hyperplane $(\alpha_k^{(i,j)} \in \mathbb{R}^3, \beta_k^{(i,j)} \in \mathbb{R})$ such that:

$$\begin{aligned} \ell_k^i &\subset \{x : {\alpha_k^{(i,j)}}^T x < \beta_k^{(i,j)}\} \\ \ell_k^j &\subset \{x : {\alpha_k^{(i,j)}}^T x > \beta_k^{(i,j)}\}. \end{aligned} \tag{8.7}$$

While this hyperplane separates the line segments ℓ_k^i and ℓ_k^j, it does not account for the robot ellipsoids. Without loss of generality, suppose the hyperplanes are given in the normalized form where $\|\alpha_k^{(i,j)}\|_2 = 1$. We accommodate the ellipsoids by shifting each hyperplane according to its normal vector, resulting in the final trajectory constraint halfspaces:

$$\begin{aligned} {\alpha_k^{(i,j)}}^T f^i(t) &< \beta_k^{(i,j)} - \|E\alpha_k^{(i,j)}\|_2 \quad \forall t \in [t_{k-1}, t_k] \\ {\alpha_k^{(i,j)}}^T f^j(t) &> \beta_k^{(i,j)} + \|E\alpha_k^{(i,j)}\|_2 \quad \forall t \in [t_{k-1}, t_k] \end{aligned} \tag{8.8}$$

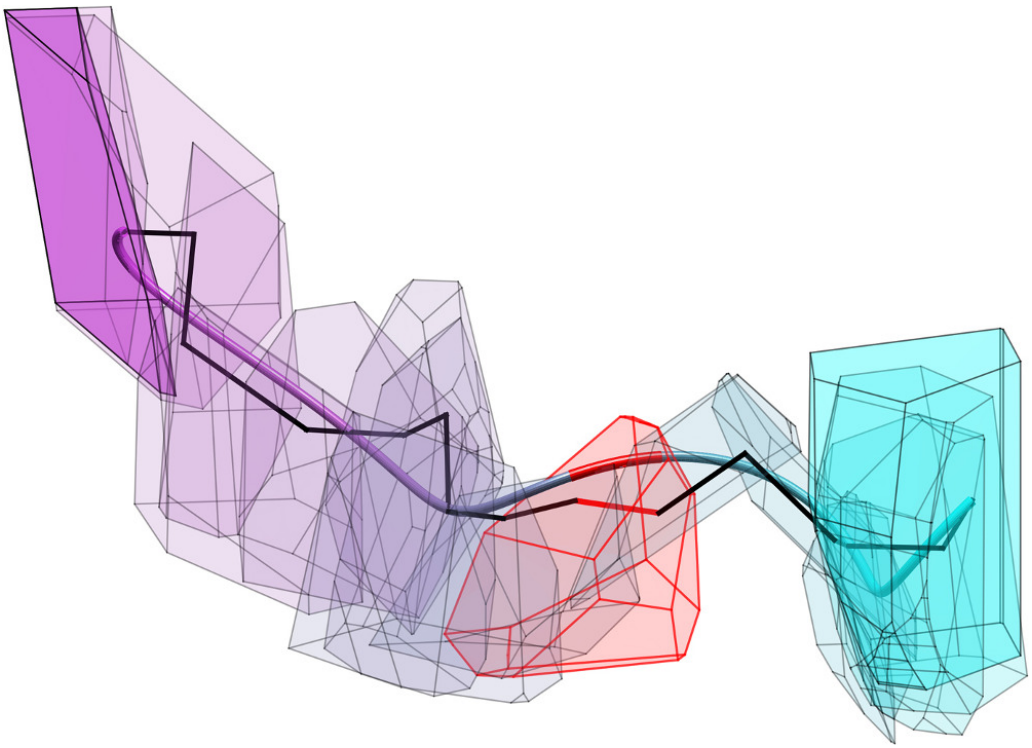


Figure 8.4: Safe corridor for one robot over entire flight. Corridor polytopes are colored by timestep. *Black line*: underlying discrete plan. *Shaded tube*: smooth trajectory after first iteration of refinement. *Highlighted in red*: polytope, discrete plan segment, and trajectory polynomial piece for a single timestep.

where $E = \mathbf{diag}(r_x, r_y, r_z)$ is the ellipsoid matrix. Robot-obstacle separating hyperplanes are computed similarly, except using a different ellipsoid E_{obs} for obstacles to model the fact that downwash is only important for robot-robot interactions.

In our implementation, we require that obstacles $\mathcal{O}_i$ are bounded convex polytopes described by vertex lists. Line segments are also convex polytopes described by vertex lists. Computing a separating hyperplane between two disjoint convex polytopes $\Psi = \mathbf{conv}(\psi_1, \ldots, \psi_{m_\Psi})$ and $\Omega = \mathbf{conv}(\omega_1, \ldots, \omega_{m_\Omega})$, where **conv** denotes the convex hull, can be posed as an instance of the hard-margin support vector machine (SVM) problem [61]. However, the ellipsoid robot shape alters the problem: for a separating hyperplane with unit normal vector α, the minimal safe margin is $2\|E\alpha\|_2$. Incorporating this constraint in the standard hard-margin SVM formulation yields a slightly modified version of the typical SVM quadratic program:

$$\begin{aligned} \text{minimize} \quad & \alpha^T E^2 \alpha \\ \text{subject to} \quad & \alpha^T \psi_i - \beta \leq 1 \quad \text{for } i \in \{1, \ldots, m_\Psi\} \\ & \alpha^T \omega_i - \beta \geq 1 \quad \text{for } i \in \{1, \ldots, m_\Omega\} \end{aligned} \tag{8.9}$$

We solve a problem of this form for each robot-robot pair and each robot-obstacle pair to yield the safe polyhedron $\mathcal{P}_k^i$ in the form of a set of linear inequalities. Note that the safe polyhedra need not be bounded and that $\mathcal{P}_k^i \cap \mathcal{P}_{k+1}^i \neq \emptyset$ in general. In fact, the overlap between consecutive $\mathcal{P}_k^i$ allows the smooth trajectories to deviate significantly from the discrete plans, which is an advantage when the discrete plan is far from optimal.

8.5.2 Bézier Trajectory Basis

After computing safe corridors, we plan a smooth trajectory $f^i(t)$ for each robot, contained within the robot's safe corridor. We represent these trajectories as piecewise polynomials with one piece per time interval $[t_k, t_{k+1}]$. Piecewise polynomials are widely used for trajectory planning: with an appropriate choice of degree and number of pieces, they can represent arbitrarily complex trajectories with an arbitrary number of continuous derivatives.

We denote the k^{th} piece of robot i's piecewise polynomial trajectory as f^i_k. We wish to constrain f^i_k to lie within the safe polyhedron $\mathcal{P}^i_k$. However, when working in the standard monomial basis, i.e., when the decision variables are the a_i in the polynomial expression

$$p(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_D t^D,$$

bounding the polynomial inside a convex polyhedron is not a convex constraint. Instead, we formulate trajectories as Bézier curves. A degree-D Bézier curve is defined by a sequence of $D+1$ *control points* $y_i \in \mathbb{R}^3$ and a fixed set of Bernstein polynomials, such that

$$f(t) = b_{0,D}(t)y_0 + b_{1,D}(t)y_1 + \cdots + b_{D,D}(t)y_D \tag{8.10}$$

where each $b_{i,D}$ is a degree-D Bernstein polynomial with coefficients[1] given in [76]. The curve begins at y_0 and ends at y_D. In between, it does not pass *through* the intervening control points, but rather is guaranteed to lie in the convex hull of all control points. Thus, when using Bézier control points as decision variables instead of monomial coefficients, constraining the control points to lie inside a safe polyhedron guarantees that the resulting polynomial will also lie inside the polyhedron. We define f^i as a K-piece, degree-D Bézier curve and denote the d^{th} control point of f^i_k as $y^i_{k,d}$. The degree parameter D must be sufficiently high to ensure continuity at the user-defined continuity level C.

8.5.3 Distributed Optimization Problem

The set of Bézier curves that lie within a given safe corridor describes a family of feasible solutions to a single robot's planning problem. We select an optimal trajectory by minimizing a weighted combination of the integrated squared derivatives:

$$\text{cost}(f^i) = \sum_{c=1}^{C} \gamma_c \int_0^T \left\| \frac{d^c}{dt^c} f^i(t) \right\|_2^2 dt \tag{8.11}$$

where the $\gamma_c \geq 0$ are user-chosen weights on the derivatives.

Our decision vector $\mathbf{y}$ consists of all control points for f^i concatenated together:

$$\mathbf{y} = \left[{y^i_{1,0}}^T \cdots {y^i_{1,D}}^T, \quad \ldots, \quad {y^i_{K,0}}^T \cdots {y^i_{K,D}}^T \right]^T \tag{8.12}$$

The objective function Eq. (8.11) is a quadratic function of $\mathbf{y}$, which can be expressed in the form:

$$\text{cost}(f^i) = \mathbf{y}^T (B^T Q B) \mathbf{y} \tag{8.13}$$

where B is a block-diagonal matrix transforming control points into polynomial coefficients, and the formula for Q is given in [121]. The start and goal position constraints, as well as the continuity

[1] The canonical Bernstein polynomials are defined over the time interval $[0, 1]$, but they are easily modified to span our desired time interval.

constraints between successive polynomial pieces, can be expressed as linear equalities. Thus, we solve the quadratic program:

$$\begin{aligned} \text{minimize} \quad & \mathbf{y}^T (B^T Q B) \mathbf{y} \\ \text{subject to} \quad & y^i_{k,d} \in \mathcal{P}^i_k \quad \forall\, i,k,d \\ & f^i(0) = s^i,\ f^i(T) = g^{\phi(i)} \\ & f^i \text{ continuous up to derivative } C \\ & \frac{d^c}{dt^c} f^i(t) = 0 \quad \forall\, c > 0,\ t \in \{0, T\}. \end{aligned} \tag{8.14}$$

It is important to note that there are N quadratic programs which can be solved in parallel, allowing a distributed implementation where each robot receives the halfspace coefficients of its safe corridor Eq. (8.8) and solves the quadratic program Eq. (8.14) onboard. Computation of the spatial partition may also be distributed with a $2\times$ constant factor of redundant work.

The quadratic program Eq. (8.14) may not always have a solution due to our conservative assumptions regarding velocity profiles and the decoupling of robots using hyperplanes. In these cases, we fall back on a solution that follows the discrete plan exactly, coming to a complete stop at corners. Details of this solution are given in [148].

The corridor-constrained Bézier formulation presents one notable shortcoming: for a given safe polyhedron $\mathcal{P}^i_k$, even if no Bézier curve with control points that are contained within $\mathcal{P}^i_k$ can be found, there might exist degree-D polynomials that lie inside the polyhedron. Empirical exploration of Bézier curves suggests that this problem is most significant when the desired trajectory is near the faces of the polyhedron rather than the center. Further research is needed to characterize this issue more precisely.

8.5.4 Iterative Refinement

Solving Eq. (8.14) for each robot converts the discrete plan into a set of smooth trajectories that are locally optimal given the spatial decomposition. However, these trajectories are not globally optimal. In our experiments, we found that the smooth trajectories sometimes lie quite far away from the original discrete plan. Motivated by this observation, we implement an iterative refinement stage where we use the smooth trajectories to define a new spatial decomposition, and use the same optimization method to solve for a new set of smooth trajectories.

For time interval k, we sample f^i_k at S evenly-spaced points in time to generate a set of points $\mathcal{S}^i_k$. The number of sample points S is a user-specified parameter, set to $S = 32$ in our experiments. We then compute the separating hyperplanes as before, except we separate $\mathcal{S}^i_k$ from $\mathcal{S}^j_k$ instead of ℓ^i_k

Figure 8.5: Illustration of discrete plan postprocessing. (a) In timestep k, robot r^j arrives at a graph vertex v and robot r^i leaves v. The separating hyperplane between ℓ^i_k and ℓ^j_k (with ellipsoid offset shaded in grey) prevents both robots from planning a trajectory that passes through v. (b) Subdivision of discrete plan ensures that this situation cannot occur.

from ℓ_k^j. This problem is also a (slightly larger) ellipsoid-weighted SVM instance. While the sample points $\mathcal{S}_k^i$ are not a complete description of f_k^i, $\mathcal{S}_k^i$ is guaranteed to be linearly separable from $\mathcal{S}_k^j$ for $i \neq j$, because the polynomial pieces f_k^i, f_k^j lie inside their respective disjoint polyhedra $\mathcal{P}_k^i, \mathcal{P}_k^j$.

These new safe corridors are roughly "centered" on the smooth trajectories, rather than on the discrete plan. Intuitively, iterative refinement provides a chance for the smooth trajectories to further minimize the cost of the quadratic objective Eq. (8.14) beyond that allowed by the constraints of the previous spatial partition. Iterative refinement is similar in spirit to sequential quadratic programming (SQP) [18] and sequential convex programming (SCP) [24]. These methods find a local minimum for a nonlinear, nonconvex optimization problem by solving an iterative sequence of convex approximations. Our method is not derived directly from such an underlying problem but follows the same pattern of alternating between updates on the approximated constraints and on the solution. The max-margin spatial partition objective Eq. (8.9) is not directly related to the primary energy minimization objective Eq. (8.13), but updating the separating hyperplanes in a separate step, allows us to decompose each iteration into N independent subproblems for efficient and decentralized computation.

Iterative refinement can be classified as an anytime algorithm. If a solution is needed quickly, the original set of f^i can be obtained in a few seconds. If the budget of computational time is larger, iterative refinement can be repeated until the quadratic program cost Eq. (8.13) converges.

8.5.5 Dynamic Limits

Unlike related work (e.g. [121]), we do not take dynamic limits such as acceleration constraints into account during optimization to avoid tight coupling of the robots in the optimization. Instead, we leverage the fact that the discrete planning stage already finds a synchronized solution for the quadrotors and we scale all trajectories uniformly in a postprocessing step.

The user-supplied timestep duration Δt directly affects the magnitudes of dynamic quantities such as acceleration and snap that are constrained by the robot's actuation limits. We use a binary search to find a uniform temporal scaling factor such that no trajectory f^i violates a dynamic constraint. For quadrotors, as the temporal scaling goes to infinity, the actuator commands are guaranteed to approach a hover state [103], so kinodynamically feasible trajectories can always be found.

8.5.6 Discrete Postprocessing

If the FCL collision model is used, the discrete planner might produce waypoints p^i that require some postprocessing to ensure that they satisfy the collision constraints Eq. (8.2) under arbitrary velocity profiles. In particular, we must deal with the case when one robot r^j arrives at a vertex $v \in \mathcal{V}_E$ in the same timestep k that another robot r^i leaves v. This situation creates a conflict where neither robot's smooth trajectory can pass through v, as illustrated in Fig. 8.5. We ensure that this situation cannot happen by dividing each discrete line segment in half. In the subdivided discrete plan, at odd timesteps robots exit a graph-vertex waypoint and arrive at a segment-midpoint waypoint, while at even timesteps robots exit a segment-midpoint waypoint and arrive at a graph-vertex waypoint. Under this subdivision, the conflict cannot occur. However, the increased number of timesteps requires more time to solve the quadratic program Eq. (8.14).

In our experiments, we noticed that the continuous trajectories typically experience peak acceleration in the vicinity of $t = 0$ and $t = T$ due to the requirement of accelerating from and to a complete stop. We add an additional wait state at the beginning and end of the discrete plans to reduce the acceleration peak.

Table 8.1: Influence of different parameters for "Wall32" problem instance, see Section 8.6.2. Bold entries indicate parameters changed compared to row 1. Gray entries indicate results that are not expected to change compared to row 1.

Row	Mapping		Roadmap					Conflicts				
	d_{octo}	nodes	Method	d_{road}	$\lvert\mathcal{V}_E\rvert$	$\lvert\mathcal{E}_E\rvert$	t_{rm}	Method	$\overline{C_{\mathcal{VV}}}$	$\overline{C_{\mathcal{EE}}}$	$\overline{C_{\mathcal{EV}}}$	t_{conf}
1	0.1	17k	Grid	0.5	978	3331	0.2	FCL	1.5	3.2	2.6	9.0
2	**0.04**	677k	Grid	0.5	978	3331	0.2	FCL	1.5	3.2	2.6	9.0
3	0.1	17k	**SPARS**	0.5	888	3495	36	FCL	0.8	7.5	1.7	9.6
4	0.1	17k	Grid	**0.2**	13k	40k	1.5	FCL	15.9	11.9	24.0	1043
5	0.1	17k	Grid	0.5	978	3331	0.2	**Swept**	1.5	26	2.6	0.6
6	0.1	17k	Grid	0.5	978	3331	0.2	FCL	1.5	3.2	2.6	9.0
7	0.1	17k	Grid	0.5	978	3331	0.2	FCL	1.5	3.2	2.6	9.0

Row	Discrete			Continuous		
	Method	K	t_{dis}	iter	t_{con}	T
1	ECBS(1.5)	24	0.4	6	47	6.5
2	ECBS(1.5)	24	0.4	6	380	6.5
3	ECBS(2.0)	30	1.5	6	56	11.5
4	ECBS(1.5)	54	10	6	103	7.7
5	ECBS(2.5)	36	1.4	6	29	8.5
6	**ILP**	20	222	6	32	5.4
7	ECBS(1.5)	24	0.4	**2**	17	9.5

Table 8.2: Results for different problem instances using SPARS, ECBS and the swept collision model, see Section 8.6.2.

Example	labeled	N	Env. Size	occupied	Roadmap			Conflicts			
					$\lvert\mathcal{V}_E\rvert$	$\lvert\mathcal{E}_E\rvert$	t_{rm}	$\overline{C_{\mathcal{VV}}}$	$\overline{C_{\mathcal{EE}}}$	$\overline{C_{\mathcal{EV}}}$	t_{conf}
Flight Test	No	32	$9 \times 5.5 \times 2.2$	4 %	873	3430	50	0.8	28	1.8	0.8
Wall32	No	32	$7.5 \times 6.5 \times 2.5$	6 %	921	3536	36	0.8	27.7	1.7	0.8
Maze50	No	50	$10 \times 6.5 \times 2.5$	30 %	1045	3221	82	1.1	24.2	2.2	0.7
Sort200	No	200	$14.5 \times 14.5 \times 2.5$	31 %	3047	7804	85	1.1	18.8	2.0	3.5
Swap50	Yes	50	$7.5 \times 6.5 \times 2.5$	6 %	869	3371	34	0.8	27	1.6	0.7

Example	Discrete		Continuous			
	K	t_{dis}	$t_{1(hp)}$	$t_{1(qp)}$	t_{con}	T
Flight Test	28	0.5	2.0	3.9	28	6.5
Wall32	41	4.5	1.6	3.5	35	11.6
Maze50	48	4.4	7.3	12.3	133	16.3
Sort200	39	25	21	34	411	11.7
Swap50	48	15	3.4	9.4	78	14.4

8.6 Experiments

We implement the roadmap generation, conflict annotation, and discrete planning in C++. We use FCL [112] for collision checking, OMPL [142] for the SPARS roadmap generator, OctoMap [71] for the environment data structure, Boost Graph for maximum flow computation, and Gurobi 7.0 as ILP solver. For the swept collision model we use a quadratic program for collision checking. The continuous refinement stage is implemented in Matlab.

We use the total number of actions as a cost metric in ECBS, where move actions along an edge

and wait actions have uniform cost. Whenever we use ECBS for an unlabeled planning problem, we use the Threshold algorithm to compute the goal assignment first.

When computing the robot-obstacle separating hyperplanes for each robot during each timestep, we limit the search space to a box containing the path segment to reduce computation. We expand the bounding box of the segment by 1 m to allow the continuous plan to deviate from the discrete plan. We consider only the nodes of the octree that intersect this box.

To compute separating hyperplanes for the safe corridors, our method requires solving $O(KN^2 + KN_{obs}N)$ small ellipsoid-weighted SVM problems. For these problems, we use the CVXGEN package [102] to generate C code optimized for the exact quadratic program specification Eq. (8.9). The per-robot trajectory optimization quadratic programs Eq. (8.14) are solved using Matlab's `quadprog` solver. Since these problems are independent, this stage can take advantage of up to N additional processor cores. In our experiments, we enforce continuity up to the fourth derivative ($C = 4$) by using a polynomial degree of $D = 7$. We evaluate our method in simulation and on the Crazyswarm, see Chapter 4.

8.6.1 Downwash Characterization

In order to determine the ellipsoid radii E, we executed several flight experiments. For r_z, we fly two Crazyflie quadrotors directly on top of each other and record the average position error of both quadrotors at 100 Hz for varying distances between the quadrotors. We noticed that high controller gains lead to very low position errors even in this case, but can cause fatal crashes when the quadrotors are close. We determined $r_z = 0.3$ m to be a safe vertical distance. For the horizontal direction, we use $r_x = r_y = 0.12$ m. We set E_{obs} to a sphere of radius 0.15 m based on the size of the Crazyflie quadrotor.

8.6.2 Runtime Evaluation

We execute our implementation on a PC running Ubuntu 16.04, with a Xeon E5-2630 2.2 GHz CPU and 32 GB RAM. This CPU has 10 physical cores, which improves the execution runtime for the continuous portion significantly.

Each stage of our framework can be configured and the choices influence the runtime and quality of results in the later stages. We will first use one specific example and discuss the variations at each stage. In later experiments we report the results for one specific choice on different examples.

Our example "Wall32" uses 32 quadrotors, which begin in a grid in the $x - y$ plane, fly through a wall with three windows, and form the letters "USC" in the air. For the roadmap, we report the number of vertices and edges created, the desired dispersion (d_{road}) and the runtime to create the roadmap (t_{rm}). For the conflict annotation we report the average number of conflicting vertices per vertex ($\overline{C_{\mathcal{VV}}}$), average number of conflicting edges per edge ($\overline{C_{\mathcal{EE}}}$), average number of conflicting vertices per edge ($\overline{C_{\mathcal{EV}}}$), and runtime to annotate the roadmap (t_{conf}). For the MAPF/C solver we report the makespan of the solution (K) and the runtime to find a solution (t_{dis}), including solving the assignment problem if ECBS is used for an unlabeled instance. For the continuous trajectory planning we report the runtime of six iterations (t_{con}) and the duration of the trajectories (T).

Each stage might influence the results as follows, see Table 8.1 for example results:

Mapping We use an occupancy grid representation of the environment, stored in an octree. Larger leaf-nodes result in a more compact data structure, but might be overly pessimistic. On the other hand, smaller leaf-nodes require more computation during the roadmap generation and trajectory optimization stages. Row 2 shows an example where a significantly higher number of leaf nodes has no effect on the discrete side, but impacts computation time on the continuous side due to the higher number of hyperplanes that must be considered.

Roadmap Generation A 6-neighbor grid is fast to compute and results in fewer conflicts during the conflict annotation stage, which in turn improves the performance of solving the MAPF/C instance. However, such a grid approach might miss narrow corridors entirely. A SPARS generated roadmap takes longer to compute and results in more conflicts due to its irregular shape, as shown in row 3. Furthermore, the non-uniform edge length might cause the discrete solver to make worse scheduling decisions because the distance traveled by each robot during a single timestep varies widely. In row 3, this caused the continuous solver to find trajectories which had to be executed much more slowly to stay within the physical limits of the quadrotor.

Lower dispersion creates denser roadmaps, which might produce better results. However, the time for conflict annotation and number of identified conflicts increase significantly. Moreover, this results in MAPF/C instances that are harder to solve in practice, see row 4.

Conflict Annotation Using the FCL collision checking model Eq. (8.4) results in fewer edge conflicts on average compared to the swept model, decreasing the MAPF/C solving time, as shown in row 5. However, it requires the continuous side to postprocess the discrete plan as visualized in Fig. 8.5, resulting in larger solving times. Moreover, we have seen various instances where the FCL model results in infeasible quadratic programs for the continuous stage, thus necessitating reversion to the piecewise linear plan [148]. The FCL collision checking model is slower compared to the swept model because it is generically implemented, supporting a wide range of possible collision models.

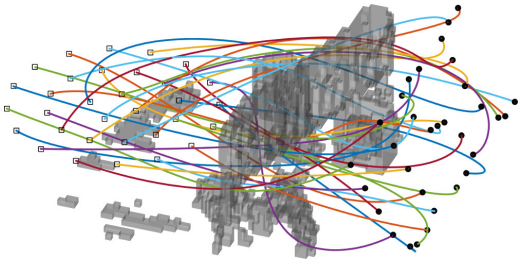
Discrete Solver The ILP-based solver can be used to solve the unlabeled problem optimally with respect to makespan. However, ILP takes significantly longer (see row 6) and in cases of large makespans or high number of conflicts might not find a solution in reasonable time at all. ECBS is more versatile and computes bounded suboptimal solutions very quickly. We noted that the suboptimality bound has a big impact on solution time with a smaller impact on the resulting makespan. Therefore, we used several different bounds, as noted in Table 8.1.

Trajectory Optimization The number of refinement stages affects the quality of the trajectories. In our experiments six refinement iterations were sufficient for convergence in all cases. If trajectories are used after fewer iterations, the resulting time T tends to be larger because the trajectories exhibit larger acceleration terms, see row 7.

In another set of experiments, we fix the parameters and apply our method to different problem instances, see Table 8.2. We use a set of parameters such that we are able to find a good solution quickly. However, tuning the parameters for a particular problem might result in shorter times T at a higher computational cost, as discussed before. We use a leaf-node size of 0.1 m, the SPARS roadmap planner with an average dispersion of 0.5 m, the swept collision model, ECBS as discrete solver and six iterations of continuous refinement. For ECBS we mostly used suboptimality bound 3.0 except for the labeled example ("Swap50") where we used a higher suboptimality bound of 4.0. We compute plans for five different examples for 32 to 200 robots navigating in obstacle-rich environments. The most significant time portion is the roadmap generation, taking up to 85 s. Conflict annotation and discrete solving can be done within 30 s. The continuous refinement finds the first smooth plan in less than a minute.

8.6.3 Flight Test

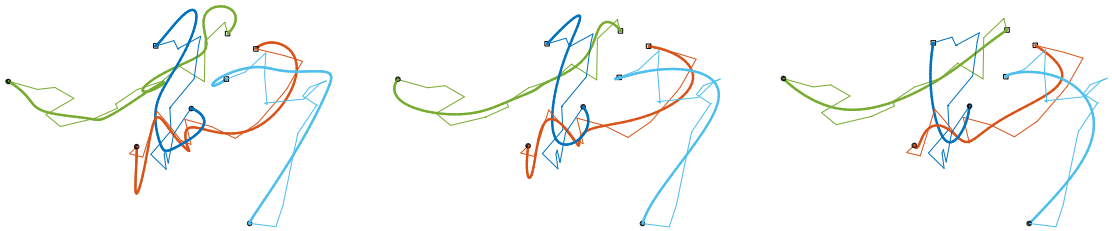
We discuss the different steps of our approach using a concrete task with 32 quadrotors. In this task, the quadrotors begin on a disc in the $x-y$ plane, fly through randomly placed obstacles (e.g. boxes, bicycle, chair), and form the letters "USC" in the air. We execute the experiment in a space which is 10 m × 16 m × 2.5 m in size and equipped with a Vicon motion-capture system with 24 cameras.



(a) Full 32-robot trajectory plan after six iterations of refinement. The start and end positions are marked by squares and filled circles, respectively.

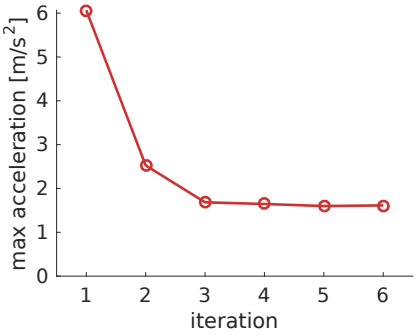
(b) Picture of the final configuration after the test flight. A video is available as supplemental material.

Figure 8.6: Formation change example where quadrotors fly from a circle formation to a goal configuration spelling "USC" while avoiding obstacles.

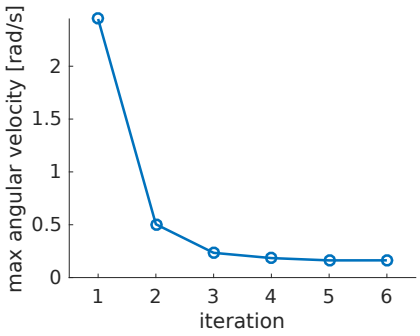


(a) 1 iteration (b) 2 iterations (c) 6 iterations

Figure 8.7: Subset of results for the example shown in Fig. 8.6 after different numbers of refinement iterations. Fine lines represent the discrete plans p^i; heavy curves represent the continuous trajectories f^i. The remaining 28 robots are hidden for clarity. Increased smoothness and directness can be observed with more iterations.



(a) Peak acceleration was reduced from 6.1 to 1.6 m/s^2.



(b) Peak angular velocity was reduced from 2.4 to 0.2 rad/s.

Figure 8.8: Illustration of worst-case acceleration and angular velocity over all robots during six iterative refinement cycles.

Figure 8.9: Long exposure of 32 Crazyflie nano-quadrotors flying through an obstacle-rich environment.

In an initial step we use a structured light depth camera tracked by our motion-capture system and the `octomap_mapping` ROS stack to map the environment using 0.1 m as octree resolution. We generate a roadmap within a bounding-box of 9 m × 5.5 m × 2.2 m using SPARS in 50 s, generating 873 vertices and 3430 edges. The conflict annotation and discrete planning take less than one second combined, finding discrete schedules p^i with $K = 28$. The continuous planner needs six seconds to find the first set of smooth trajectories and finishes six iterations of refinement after 28 seconds.

Figure 8.8 demonstrates the effect of iterative refinement on the quadrotor dynamics required by the trajectories f^i. For each iteration, we take the maximum acceleration and angular velocity over all robots for the duration of the trajectories. Iterative refinement results in trajectories with significantly smoother dynamics. This effect is also qualitatively visible when plotting a subset of the trajectories, as shown in Fig. 8.7. The final set of 32 trajectories is shown in Fig. 8.6a.

We upload the planned trajectories to a swarm of Crazyflie 2.0 nano-quadrotors before takeoff, and use the Crazyswarm infrastructure (see Chapter 4) to execute the trajectories. State estimation and control run onboard the quadrotor, and the motion-capture system information is broadcasted to the UAVs for localization. Figure 8.6b shows a snapshot of the execution when the quadrotors reached their final state. The executed trajectories can be visualized with long-exposure photography, as shown in Fig. 8.9. The supplemental video shows the full trajectory execution.

8.7 Remarks

We present a trajectory planning method for large robot teams, combining the advantages of graph-based AI planners and trajectory optimization. We validate our approach on a real-world example of downwash-aware planning for quadrotor swarms.

Our approach creates plans where robots can safely fly in close proximity to each other. We create the trajectories using three stages: roadmap generation with inter-robot conflict annotation, discrete planning, and continuous optimization. The roadmap generation stage creates a sparse roadmap for a single robot and can use any existing algorithm for that purpose. We annotate the roadmap with generalized conflicts, describing possible inter-robot dependencies which might be violated if two robots are in close proximity to each other. We can then formulate a MAPF/C

instance and either solve it optimally with respect to makespan using an ILP-based formulation, or solve it with bounded suboptimality using a search-based method. The continuous optimization finds smooth trajectories for each robot and is decoupled, allowing easy parallelization and improving performance for large teams. Our approach is resolution-complete [90], but sub-optimal due to the decoupling of discrete and continuous planning.

To our knowledge, our approach is the first solution which can compute safe and smooth trajectories for hundreds of quadrotors in dense continuous environments with obstacles in a few minutes. Our results have been tested and executed safely in numerous trials on a team of 32 quadrotors.

CHAPTER 9

Task and Motion Planning for Heterogeneous Robot Teams

In this chapter, we extend the method in Chapter 8 to heterogeneous teams of robots. Trajectory planning for heterogeneous teams of robots is a core problem for many potential applications of multi-robot systems. To accomplish complex tasks, it could be beneficial for a team to be composed of different types of robots with varied capabilities. This complicates trajectory planning due to differing dynamics and mixed requirements for allowable interactions between robots. For example, downwash from rotorcraft is an effect that other nearby rotorcraft must consider in order to maintain stable flight, but ground robots are not necessarily affected by downwash. Figure 9.1 shows an example of a physical experiment in which many quadrotors of different sizes must fly in close proximity and thus be aware of other quadrotors' downwash while also considering the motion of ground robots.

The proposed method is centralized and is designed for *a priori* calculation of trajectories in known environments. The high-level structure of the approach in Chapter 8 is retained. First, a graph-based planning method is used to compute a collision-free discretized schedule for all robots in the team. Then, a second parallelizable optimization stage refines these schedules into locally optimal smooth trajectories. This chapters extension introduces the capability of utilizing independent and asymmetric collision constraints between different pairs of robot types. Additionally, it generates trajectories that obey the dynamic limits of all robots types in the team while retaining temporal alignment. Like the original method, the one presented here scales well to teams with large numbers of robots, with additional processing time mostly due to offline roadmap generation.

9.1 Related Work

Motion planning for heterogeneous robots can employ optimization-based methods, graph-based methods, or reactive planners. Mixed-integer quadratic programs (MIQPs) can plan for a team of different quadrotors, taking their different sizes and aerodynamic effects between quadrotors into account [104]. This method has been applied to up to four physical quadrotors (three different sizes) and produces optimal trajectories, but does not scale well to a large number of robots and/or

This chapter is based on Mark Debord, **Wolfgang Hönig**, and Nora Ayanian. "Trajectory Planning for Heterogeneous Robot Teams". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 7924–7931. DOI: 10.1109/IROS.2018.8593876.
Mark Debord was a Master's student whom I supervised. He implemented the continuous optimization changes and I implemented the discrete planning changes. Experiments were executed jointly.

Figure 9.1: Heterogeneous robot team with ten small UAVs (blue, only eight visible), two medium UAVs (red), one large UAV (green), and two ground robots (yellow). The robots have to navigate through a cluttered environment, avoiding obstacles and taking asymmetric inter-robot constraints into account.

obstacles. Similar to our approach, inter-robot constraints are modeled asymmetrically, allowing large quadrotors to fly underneath smaller ones. Optimization-based methods can also be combined with sequential planning to achieve better scalability [122]; this method, unlike our work, does not consider asymmetric interactions between differently sized robots and has not been demonstrated in 3D.

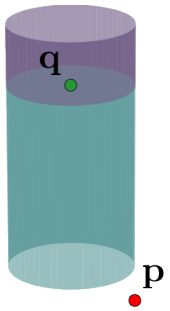
Graph-based planning can be combined with controller-based motion primitives to include complex dynamics during planning. Planning in the joint space of all robots can be used to model heterogeneous teams such as a collaboration between a UGV and a UAV [23], but it does not scale well to a large number of robots. Other approaches use graph-based planning for homogeneous robots that have different operating modes, such as flying cars [6]. Here, the motion planner can consider switching between driving and flying, but the resulting trajectories are not smooth and asymmetric inter-robot constraints are not considered.

Velocity obstacle approaches have also been applied to heterogeneous robot teams [5], but while such local planners can find solutions quickly, they may not find solutions in cluttered environments.

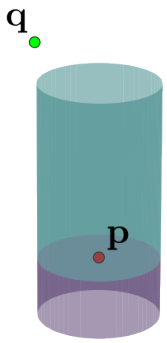
9.2 Approach

The first step is to specify the types of robots in the heterogeneous team and the geometric interaction models that are required to define collision-free trajectories. We then outline the major components of our approach and explain the major extensions to the previous work that enable heterogeneous teams.

A team of robots is considered heterogeneous if it consists of multiple robots with different physical capabilities or dynamic limits. We focus our efforts on two classes of robots, quadrotors



(a) Collision geometry for a large quadrotor of type k at position $\mathbf{q}$: $R_{\mathcal{R}}^{(k,l)}(\mathbf{q})$.



(b) Collision geometry for a small quadrotor of type l at position $\mathbf{p}$: $R_{\mathcal{R}}^{(l,k)}(\mathbf{p})$.

Figure 9.2: An illustration of the cylindrical collision geometries for a large quadrotor of type k at position $\mathbf{q}$ and a small quadrotor of type l at position $\mathbf{p}$. In this case the tuple $R_{\mathcal{R}}^{(k,l)}(\mathbf{q}) : \langle r, a, b \rangle$ has $b > a$ to model the asymmetry between the downwash zones as shown in (a). $R_{\mathcal{R}}^{(l,k)}(\mathbf{p})$ is symmetric to $R_{\mathcal{R}}^{(k,l)}(\mathbf{q})$ as shown in (b).

and differential drive wheeled robots, however, the presented method applies for any differentially flat system. Each of these classes can be further delineated into *types* which may be of different sizes or have varied dynamic limits, such as maximum accelerations or velocities.

Quadrotors generate a fast-moving volume of air called downwash that impacts the ability of other quadrotors to fly directly below them. In Chapter 8, we modeled the collision volume of a quadrotor as an axis-aligned ellipsoid centered at the quadrotor's position. This is not sufficient in the heterogeneous case, as the effect is asymmetric with respect to the sizes of the interacting quadrotors. For example, a large quadrotor is likely able to fly below a smaller one without difficulty, but the opposite is not true. Additionally, wheeled robots likely do not need to consider the downwash effect at all. Thus, we define independent collision volumes for every possible pair of interacting robot types.

9.2.1 Collision Model

Consider a team of N robots, where each robot is one of M types. The environment is defined as a set of convex obstacles $\mathcal{O}_1, \ldots, \mathcal{O}_{N_{obs}}$ within a boundary defined by a convex polytope $\mathcal{W}$. For each robot of type $k \in \{1, \ldots, M\}$ we define a convex volume $\mathcal{R}_{\mathcal{E}}^{k}(\mathbf{q})$ that represents the robot-environment collision volume for the k type robot at position $\mathbf{q} \in \mathbb{R}^3$. The set $\mathcal{F}^k$ describes the free configuration space for a robot of type k with respect to the environment. $\mathcal{F}^k$ is defined:

$$\mathcal{F}^k = (\mathcal{W} \backslash (\bigcup_{h=1}^{N_{obs}} \mathcal{O}_h)) \ominus \mathcal{R}_{\mathcal{E}}^{k}(\mathbf{0}), \tag{9.1}$$

where $\ominus$ denotes the Minkowski difference.

We define separate convex geometries to describe the unique collision constraints that exist between every robot type-pair. Consider a robot $r^{(i,k)}$ with index $i \in \{1, \ldots, N\}$ and type $k \in \{1, \ldots, M\}$ at position $\mathbf{q}$ and another robot $r^{(j,l)}$ at position $\mathbf{p}$. If $r^{(i,k)}$ is at position $\mathbf{q}$, then $r^{(j,l)}$ cannot occupy the convex volume $\mathcal{R}_{\mathcal{R}}^{(k,l)}(\mathbf{q})$. Likewise, if $r^{(j,l)}$ is at position $\mathbf{p}$, then $r^{(i,k)}$ cannot occupy the convex volume $\mathcal{R}_{\mathcal{R}}^{(l,k)}(\mathbf{p})$. Specifically, we say there is a collision between the two robots if $\mathbf{p} \in \mathcal{R}_{\mathcal{R}}^{(k,l)}(\mathbf{q})$, or equivalently, $\mathbf{q} \in \mathcal{R}_{\mathcal{R}}^{(l,k)}(\mathbf{p})$.

If k and l both specify types of quadrotors, then $R_{\mathcal{R}}^{(k,l)}(\mathbf{q})$ defines both the downwash and physical collision volume of the k type quadrotor with respect to l type. If either k or l specify a type of wheeled robot, then $R_{\mathcal{R}}^{(k,l)}(\mathbf{q})$ only specifies the physical collision volume.

For all cases of (k, l) type pairs, we parameterize the collision volumes with a tuple $R_{\mathcal{R}}^{(k,l)}(\mathbf{q}) : \langle r, a, b\rangle$ that specifies an axis-aligned cylinder of radius r where the top of the cylinder is located at $q_z + a$ and the bottom at $q_z - b$. The parameter r represents the minimum safe horizontal distance between the positions of the two robot types. The parameters a and b specify the minimum safe vertical distance for the l type robot above and below $\mathbf{q}$, respectively. These parameters are experimentally determined for real robots as described in Section 9.5.1.

Note that the definition of the cylinder for the (k, l) pair is symmetric with respect to the (l, k) pair. That is, if $R_{\mathcal{R}}^{(k,l)}(\mathbf{q}) : \langle r, a, b\rangle$, then $R_{\mathcal{R}}^{(l,k)}(\mathbf{p}) : \langle r, b, a\rangle$. Figure 9.2 shows an example of these cylinder definitions for the case of a large and small quadrotor. Also note that the cylinder model is an approximation of the downwash effect as it does not directly account for orientation or acceleration. To account for inaccuracies, we assume cylinder parameters are conservative.

Let $f^{(i,k)} : [0, T] \mapsto \mathbb{R}^3$ be the trajectory of robot $r^{(i,k)}$ where T represents the time that the last robot in the team reaches its goal. Trajectories are considered collision free if there are no robot-environment collisions and no inter-robot collisions:

$$\begin{aligned} f^{(i,k)}(t) &\in \mathcal{F}^k && \forall i,\ 0 \le t \le T \\ f^{(i,k)}(t) &\notin \mathcal{R}_{\mathcal{R}}^{(l,k)}(f^{(j,l)}(t)) && \forall i \neq j,\ 0 \le t \le T. \end{aligned} \tag{9.2}$$

9.2.2 Problem Statement

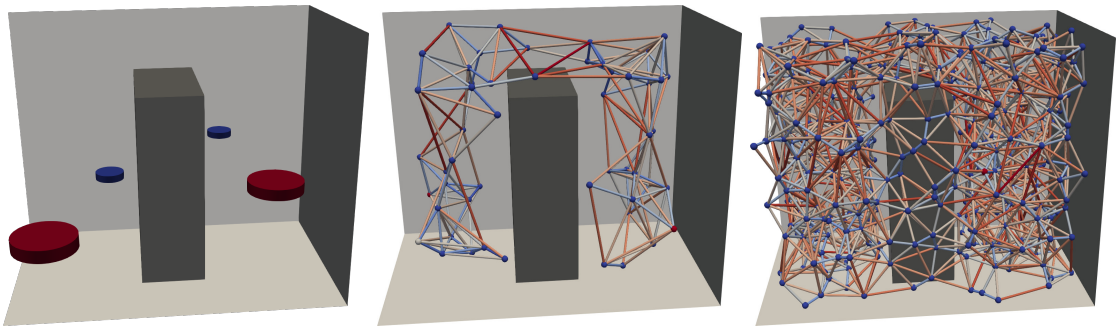
Given the following:

- an environment specified by $\mathcal{W}$ and $\mathcal{O}_1, \ldots, \mathcal{O}_{N_{obs}}$;
- a set of N robots $r^{(i,k)}$, $i \in \{1, \ldots, N\}$ and $k \in \{1, \ldots, M\}$;
- start locations $\mathbf{s}^{(i,k)}$ and goal locations $\mathbf{g}^{(i,k)}$ for each robot;
- the robot-environment collision model $\mathcal{R}_{\mathcal{E}}^{k}(\cdot)$ for each robot type k and the inter-robot collision model $R_{\mathcal{R}}^{(k,l)}(\cdot)$ for each robot type-pair (k, l); and
- the maximum velocity $v_{\max}^k$ and acceleration limits $a_{\max}^k$ for each robot type k;

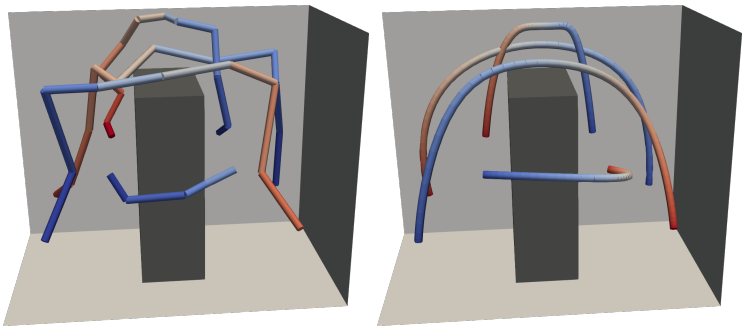
our goal is to compute T and a kinodynamically feasible trajectory $f^{(i,k)}$ that is collision-free according to Eq. (9.2) for each robot $r^{(i,k)}$ such that $f^{(i,k)}(0) = \mathbf{s}^{(i,k)}$ and $f^{(i,k)}(T) = \mathbf{g}^{(i,k)}$. This trajectory planning problem is often referred to as the *labeled* case, because each robot has its goal assigned a priori. We also consider the *multi-color* case, where robots of the same type are interchangeable and allowed to swap goal assignments.

9.2.3 Overview of Approach

As described previously, the high level components of the presented method share the same structure as in Chapter 8. To extend the work to the heterogeneous case, several modifications to the discrete scheduling and trajectory optimization stages are necessary. In particular, the roadmap generation and conflict annotation phases are extended to account for the different free space definitions and velocity limits of each type of robot. Additionally, the construction of safe corridors in the trajectory optimization stage is generalized to account for the different collision volumes for each robot and robot type-pair. The following sections detail these modifications.



(a) Initial configuration. **(b) Roadmap for large/fast UAVs.** **(c) Roadmap for small/slow UAVs.**



(d) Discrete trajectories. **(e) Continuous trajectories.**

Figure 9.3: Example with two small and two large UAVs, one of each type on each side of the obstacle. The UAVs are tasked with moving to goal locations on the opposite side of the obstacle.

9.3 Roadmap Generation, Conflict Annotation, Discrete Planning

Recall that the first phase of the hybrid planning method is to generate collision free *discrete schedules* for every robot in the team. A discrete schedule assigns each robot a line segment to traverse for a specific timestep; the segment might consist of a single point if the robot should be stationary during that timestep. The discrete schedule guarantees collision-free execution if the robots move along their segments with arbitrary velocity profiles.

For homogeneous robot teams, a discrete schedule can be computed by first generating a roadmap and then solving a multi-agent path-finding problem on that roadmap. When applying this approach to motion planning for heterogeneous teams, three problems arise. First, the free space definition for each type of robot is different because of varying physical extent or different methods of locomotion. Second, the inter-robot conflicts are specific to the types of robots that are interacting, e.g., a small quadrotor cannot fly closely below a large quadrotor but it is able to fly closely to static obstacles or ground robots. Third, each robot type has different dynamic limits such as maximum velocities.

We address all three problems by constructing a *super roadmap*, which is the disjoint union of the roadmaps for the individual robot types. We show that the super roadmap can be used as a drop-in replacement for a regular roadmap as input to existing planning algorithms.

9.3.1 Super Roadmap Generation

A roadmap for robot type $k \in \{1, \ldots, M\}$ is an undirected connected graph of the environment $\mathcal{G}^k = (\mathcal{V}^k, \mathcal{E}^k)$, where each vertex $v \in \mathcal{V}^k$ corresponds to a location in $\mathcal{F}^k$ and each edge $(u, v) \in \mathcal{E}^k$

denotes that there is a linear path in $\mathcal{F}^k$ connecting u and v.

We generate the roadmap $\mathcal{G}^k$ given a representation of the environment and the shape of $\mathcal{R}^k_{\mathcal{E}}(\cdot)$ using the SPARS algorithm [41]. SPARS generates dense and sparse roadmaps such that the sparse roadmap is a subgraph of the dense one, while keeping any-pair shortest distances within a user-specified sub-optimality factor. Another parameter, Δ^k, controls the visibility radius and roughly corresponds to the average edge length of the generated sparse roadmap. We choose Δ^k to be smaller for slower robots, reflecting that they can travel shorter distances compared to faster ones in the same amount of time. Specifically, we generate $\mathcal{G}^k$ using:

$$\Delta^k = \alpha v^k_{\max}, \quad \forall k \in \{1, \dots, M\}, \tag{9.3}$$

where α corresponds to the sparsity of all roadmaps.

After executing SPARS, we add additional vertices that correspond to $\mathbf{s}^{(i,k)}$ and $\mathbf{g}^{(i,k)}$ to $\mathcal{V}^k$ for all $i \in \{1, \dots, N\}$. We connect those vertices to neighboring vertices by adding additional edges to $\mathcal{E}^k$.

Consider an example with two small and two large quadrotors in an environment with an obstacle (see Fig. 9.3a), where each quadrotor has to move to the opposite side of the obstacle. The physical extent of the large UAVs forces them to fly over the obstacle (see roadmap in Fig. 9.3b), while the small UAVs can fly over or in front of the obstacle (see roadmap in Fig. 9.3c). In this example we also assume that the large UAVs can fly twice as fast as the small ones and select the Δ^k values accordingly. The resulting roadmap for the large UAV has fewer edges that are longer on average (166 edges with average length of 0.47 m) compared to the roadmap for the small UAVs (1712 edges with average length of 0.28 m).

For ground robots, we limit the roadmap generation to two dimensions. We can combine the separate roadmaps into a *super roadmap* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ by computing the disjoint union or graph sum:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}) = \left(\bigcup_{k=1}^{M} \mathcal{V}^k, \bigcup_{k=1}^{M} \mathcal{E}^k \right). \tag{9.4}$$

Each vertex $v \in \mathcal{V}$ is associated with a position $\mathbf{q} \in \mathbb{R}^3$ and we denote this relationship by $\mathbf{q} = loc(v)$. The vertex set $\mathcal{V}$ is surjective to $\mathbb{R}^3$, i.e., a point in Euclidean space might correspond with multiple vertices (up to one for each robot type). Each vertex $v \in \mathcal{V}$ and edge $e \in \mathcal{E}$ have their respective robot type k associated and we refer to it by $k = type(v)$ and, with a slight abuse of notation, $k = type(e)$.

9.3.2 Conflict Annotation

The super roadmap $\mathcal{G}$ can be directly used for path planning for a single robot to avoid robot-obstacle collisions. When using multiple robots of either the same or different types, inter-robot constraints must be considered. An offline pre-processing step annotates the roadmap with potential inter-robot conflicts, by adding the following types of constraints:

Vertex-Vertex Constraints Two robots may not concurrently occupy two vertices which are in close proximity to each other. We can compute the collision set for each vertex $v \in \mathcal{V}$ by checking if a point is in the respective collision cylinder:

$$\begin{aligned} conVV(v) = \{u \in \mathcal{V} | \\ loc(u) \in \mathcal{R}^{(type(v), type(u))}_{\mathcal{R}}(loc(v))\}. \end{aligned} \tag{9.5}$$

Edge-Vertex Constraints One robot may not traverse an edge if a collision could occur with another stationary robot. A robot may become stationary during discrete planning if it is

determined that to avoid collisions, it should wait at its current vertex rather than traverse an edge. Let $\hat{\mathcal{R}}_{\mathcal{R}}^{(k,l)}(e)$ be the convex hull that is created when $\mathcal{R}_{\mathcal{R}}^{(k,l)}(\cdot)$ is swept along edge e. We can compute the collision set for edge e by checking if the location associated with a vertex lies within that convex hull:

$$conEV(e) = \{v \in \mathcal{V} | \\ loc(v) \in \hat{\mathcal{R}}_{\mathcal{R}}^{(type(e),type(v))}(e)\}. \tag{9.6}$$

Edge-Edge Constraints Two robots may not concurrently traverse two edges if a collision could occur during the traversal. Let $\hat{d}$ be the set of points comprising edge d. We can compute the collision set for each edge e by checking for intersection between the convex hull and the line segment that is defined by edge d:

$$conEE(e) = \{d \in \mathcal{E} \,|\, \hat{d} \subset \hat{\mathcal{R}}_{\mathcal{R}}^{(type(e),type(d))}(e)\}. \tag{9.7}$$

9.3.3 Discrete planning

The discrete path planning problem can now be formulated as an instance of *Multi-Agent Path-Finding with Generalized Conflicts* (MAPF/C), see Section 6.3. The inputs are the annotated roadmap and start and goal vertices for the individual robots. The output is a discrete trajectory for each robot such that all constraints are obeyed. A discrete trajectory $p^{(i,k)}$ for each robot $r^{(i,k)}$ is composed of a sequence of $K+1$ locations:

$$p^{(i,k)} = \mathbf{x}_0^{(i,k)}, \mathbf{x}_1^{(i,k)}, \ldots, \mathbf{x}_K^{(i,k)}, \tag{9.8}$$

where robots are synchronized in time and have to arrive at location $\mathbf{x}_n^{(i,k)}$ at timestep n. We define $\ell_n^{(i,k)}$ as the line segment between locations $\mathbf{x}_n^{(i,k)}$ and $\mathbf{x}_{n+1}^{(i,k)}$. Example discrete trajectories are shown in Fig. 9.3d. Each edge is color-coded by the timestep n in which it will be traversed: blue at $n = 0$, gradually changing to red at $n = K$. The small UAV in the back flies on top of a large one (which is safe according to our collision model), while the small one in front has to keep a large vertical safety distance in order to pass below the other large UAV.

A robot is restricted to move on the roadmap for its own type by construction of the super roadmap using the disjoint union of the per-type roadmaps. As our solver, we use a variant of *Enhanced Conflict-Based Search* (ECBS) [11] that takes the generalized conflicts into account.

In the multi-color case, robots of the same type may swap their goals. The super roadmap allows us to use existing task assignment algorithms, because there is no path between any vertices that belong to different robot types. For example, minimizing the maximum duration over all robots can be achieved by running the Threshold algorithm [22] prior to the ECBS execution.

9.4 Trajectory Optimization

Trajectory optimization is done using the same general process as for homogeneous teams (see Chapter 8). First, collision-free corridors are generated from the discrete schedule for each robot for the entire duration of the plan. Second, independent trajectory optimizations occur for each robot within its corridor. Third, continuous refinement of the initial trajectories is done by sampling the trajectories, recomputing the safe corridors, and re-optimizing the trajectories inside the new corridors. Finally, the trajectories are post-processed by uniformly scaling their duration in order to ensure the dynamic limits of all robots are obeyed. While the top-level process is similar, modifications to the construction of the corridors and the method for trajectory scaling are necessary to extend the process to heterogeneous teams. We start by introducing the definitions needed to describe these modifications.

9.4.1 Definitions

We assign a time $t_n = n\Delta t$ to every step in the schedule determined by the discrete solver. Each of the K steps from the discrete solution specifies a time interval $[t_{n-1}, t_n]$. The parameter Δt is user defined and specifies an initial guess for the duration of the entire trajectory $T = K\Delta t$. Let $\mathcal{F}_n^{(i,k)}$ be the set of points defined by the trajectory of a robot $r^{(i,k)}$ during timestep n:

$$\mathcal{F}_n^{(i,k)} = \left\{ f^{(i,k)}(t) \mid t_n \leq t \leq t_{n+1} \right\}. \tag{9.9}$$

During both the initial trajectory optimization and continuous refinement stages, collision-free *safe corridors* are computed. The safe corridor for robot $r^{(i,k)}$ is defined as a sequence of convex polyhedra $\mathcal{P}_n^{(i,k)}, n \in \{1, \ldots, K\}$, such that if every robot travels in their respective $\mathcal{P}_n^{(\cdot,\cdot)}$ during timestep n they are guaranteed to be collision free for that timestep. The polyhedra $\mathcal{P}_n^{(i,k)}$ for a robot $r^{(i,k)}$ are specified as the intersection of $N-1$ half-spaces that separate $r^{(i,k)}$ from all other robots and N_{obs} half-spaces separating $r^{(i,k)}$ from all obstacles.

Let $\hat{\mathcal{R}}_{\mathcal{R}}^{(l,k)}(\mathcal{F}_n^{(j,l)})$ be the set of points defined by sweeping the cylinder specified by $\mathcal{R}_{\mathcal{R}}^{(l,k)}$ along $\mathcal{F}_n^{(j,l)}$. Also let the half-space separating $r^{(i,k)}$ from $r^{(j,l)}$ be denoted as $\mathcal{H}_{\mathcal{R}n}^{(i,j)}$. $\mathcal{H}_{\mathcal{R}n}^{(i,j)}$ is defined such that

$$\mathcal{H}_{\mathcal{R}n}^{(i,j)} \cap \mathcal{F}_n^{(i,k)} = \mathcal{F}_n^{(i,k)} \tag{9.10}$$

$$\mathcal{H}_{\mathcal{R}n}^{(i,j)} \cap \hat{\mathcal{R}}_{\mathcal{R}}^{(l,k)}(\mathcal{F}_n^{(j,l)}) = \emptyset. \tag{9.11}$$

Let $\mathcal{H}_{\mathcal{E}n}^{(i,h)}$ be a half-space separating $r^{(i,k)}$ from an obstacle $\mathcal{O}_h$ at step n. $\mathcal{H}_{\mathcal{E}n}^{(i,h)}$ is defined such that

$$\mathcal{H}_{\mathcal{E}n}^{(i,h)} \cap \mathcal{F}_n^{(i,k)} = \mathcal{F}_n^{(i,k)} \tag{9.12}$$

$$\mathcal{H}_{\mathcal{E}n}^{(i,h)} \cap \left(\mathcal{O}_h \oplus \mathcal{R}_{\mathcal{E}}^{k}(\mathbf{0}) \right) = \emptyset, \tag{9.13}$$

where $\oplus$ denotes the Minkowski sum.

With the above definitions $\mathcal{P}_n^{(i,k)}$ is specified as

$$\mathcal{P}_n^{(i,k)} = \left(\bigcap_{j \neq i}^{N} \mathcal{H}_{\mathcal{R}n}^{(i,j)} \right) \bigcap \left(\bigcap_{h}^{N_{obs}} \mathcal{H}_{\mathcal{E}n}^{(i,h)} \right). \tag{9.14}$$

9.4.2 Safe Corridors

In the prior homogeneous planning work, the half-spaces defining the safe corridors were computed using a modified SVM in a stretched coordinate system to account for the uniform ellipsoidal collision models. In the heterogeneous case this method can no longer be used because of the non-uniform volumes that each robot occupies. Instead, we directly enumerate a vertex cloud specifying the time swept hull of the cylinders defined in Section 9.2.1, then compute the half-spaces using a standard SVM.

Consider robots $r^{(i,k)}$, $r^{(j,l)}$, and a corresponding half-space $\mathcal{H}_{\mathcal{R}n}^{(i,j)}$. $\mathcal{H}_{\mathcal{R}n}^{(i,j)}$ is computed by first specifying two vertex sets $\mathcal{V}_n^{(i,k)}$ and $\mathcal{V}_n^{(j,l)}$ for each robot and then separating those vertex sets with a linear SVM. The set $\mathcal{V}_n^{(i,k)}$ is composed of the trajectory points sampled from $\mathcal{F}_n^{(i,k)}$, and the set $\mathcal{V}_n^{(j,l)}$ is constructed by generating a conservative approximation of the swept hull specified by $\hat{\mathcal{R}}_R^{(l,k)}(\mathcal{F}_n^{(j,l)})$.

To generate the swept hull vertices for $\mathcal{V}_n^{(j,l)}$ we first compute a polytope approximation of the cylinder specified by $R_R^{(l,k)}$ by computing the vertices of a circumscribed polygon with radius r and

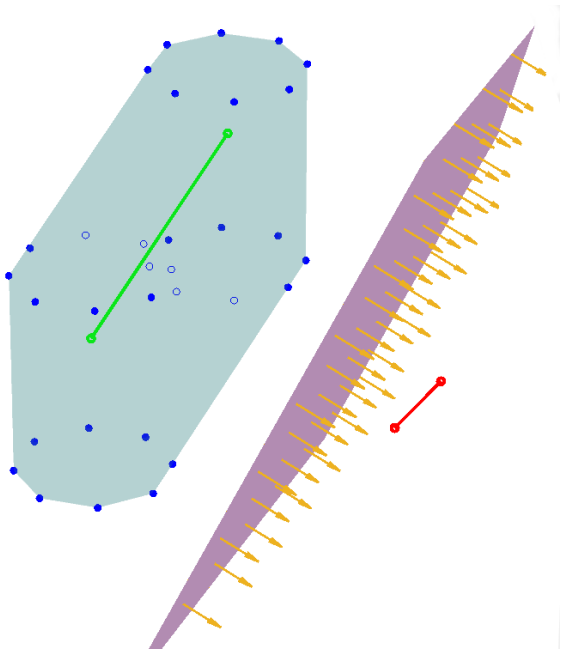


Figure 9.4: An example of the the half-space computation for a robot $r^{(i,k)}$. The trajectory of $r^{(i,k)}$ is displayed in red on the right side, and another robot $r^{(j,l)}$'s trajectory is in green. The swept hull approximation for $\hat{\mathcal{R}}_R^{(l,k)}(\mathcal{F}_n^{(j,l)})$ is shown in blue. The separating hyper-plane is in purple with the arrows indicating the direction of the half-space

placing those vertices on both the top and bottom ends of the cylinder. $\mathcal{V}_n^{(j,l)}$ is then constructed by enumerating these cylinder approximations at points sampled from $\mathcal{F}_n^{(j,l)}$. In the first step of optimization, the points sampled from $\mathcal{F}_n^{(i,k)}$ and $\mathcal{F}_n^{(j,l)}$ are the endpoints of $\ell_n^{(i,k)}$ and $\ell_n^{(j,l)}$ from the discrete solution. During continuous refinement the samples from $\mathcal{F}_n^{(\cdot,\cdot)}$ are uniform evaluations of $f_n^{(\cdot,\cdot)}(t)$ over the corresponding interval.

Finally, $\mathcal{H}_{\mathcal{R}n}^{(i,j)}$ is computed by a linear SVM such that $\mathcal{F}_n^{(i,k)}$ lies entirely on the positive side of the separating hyper-plane and $\hat{\mathcal{R}}_R^{(l,k)}(\mathcal{F}_n^{(j,l)})$ lies entirely on the negative side. This process is repeated for every robot pair at every timestep to specify the components of the corridor that partition the free space for every robot with respect to the rest of the team. Construction of half-spaces separating the robots from the environment follows the same procedure described in the prior work with the addition that every robot can have a separate specified size. A visualization of an approximated swept cylinder hull and a separating hyper-plane defining a half-space are given in Fig. 9.4.

9.4.3 Optimization

Trajectory optimization is identical to the previous chapter, where we construct independent quadratic programs for each robot. In the following we outline this method. For each robot, we formulate a quadratic program with an optimization objective that minimizes the sum of integrated squared derivatives of the trajectories. The decision variables for the optimization are the control points of K sequential Beziér curves. We specify linear inequality constraints to bound the control points within the safe corridors and equality constraints to enforce starting and goal positions as well as continuity between each of the Beziér curves. The optimization can be repeated using the

Table 9.1: Robot Properties.

Robot	Radius [m]	Height [m]	Weight [kg]	$v_{\max}$ [m/s]	$a_{\max}$ [m/s^2]
Small	0.08	0.06	0.033	1.7	6.2
Medium	0.14	0.12	0.124	2.0	8.5
Large	0.21	0.15	0.491	1.8	7.2
Ground	0.25	0.45	6.3	0.5	0.5

Table 9.2: Robot Interactions. Minimal required horizontal (r) and vertical distances (v) in meters denoted as $\langle r, v \rangle$.

top / bottom	Small	Medium	Large	Ground
Small	$\langle 0.2, 0.6 \rangle$	$\langle 0.3, 1.4 \rangle$	$\langle 0.35, 2.0 \rangle$	$\langle 0.33, 0.26 \rangle$
Medium	$\langle 0.3, 0.1 \rangle$	$\langle 0.3, 0.5 \rangle$	$\langle 0.4, 0.3 \rangle$	$\langle 0.39, 0.29 \rangle$
Large	$\langle 0.35, 0.2 \rangle$	$\langle 0.4, 0.2 \rangle$	N.A.	$\langle 0.46, 0.3 \rangle$
Ground	$\langle 0.33, 0.26 \rangle$	$\langle 0.39, 0.29 \rangle$	$\langle 0.46, 0.3 \rangle$	$\langle 0.5, 0.45 \rangle$

previous results as input for iterative cost improvement.

9.4.4 Dynamic Limits

Many nonholonomic mobile robots are differentially flat in position outputs, including quadrotors and differential drive robots [52, 103, 107]. That means the control input to move the robots along a trajectory can be computed using the trajectory itself. Polynomial trajectories can be scaled in time: a longer trajectory has lower velocities and accelerations. Therefore, dynamic limits can be enforced by scaling all trajectories by a constant factor. We compute a trajectory stretching factor for each robot using a binary search approach. The maximum of all those factors is then applied uniformly to all trajectories and guarantees that the dynamic limits are fulfilled.

An example of the generated continuous trajectories is shown in Fig. 9.3e; the trajectories are color-coded by time (blue means $t = 0$ and red means $t = T$). The small quadrotor in the back flies directly above the large one.

9.5 Experiments

For the discrete planning we implement roadmap generation and annotation in C++ using the SPARS implementation from the OMPL library [142] and an OctoMap [71] data structure. The half-space computation is implemented in Matlab using libSVM [28]. The remaining code (ECBS solver, trajectory optimization) is only slightly modified compared to our previous homogeneous planning work in Chapter 8. All simulations were executed on a PC running Ubuntu 16.04, with a Xeon E5-2630 2.2 GHz CPU and 32 GB RAM.

9.5.1 Robot Characterization

We use four different types of robots in our experiments. The *small* robots are Bitcraze Crazyflie 2.0 nano-quadrotors, see Section 4.2 for details. We use off-the-shelf components for frame, motor, and power distribution to build two larger kinds of quadrotors (*medium* and *large*). We use the Crazyswarm (see Chapter 4), our open-source solution to use many Crazyflie 2.0 quadrotors simultaneously. Each of the quadrotors uses the same extended custom firmware that runs a

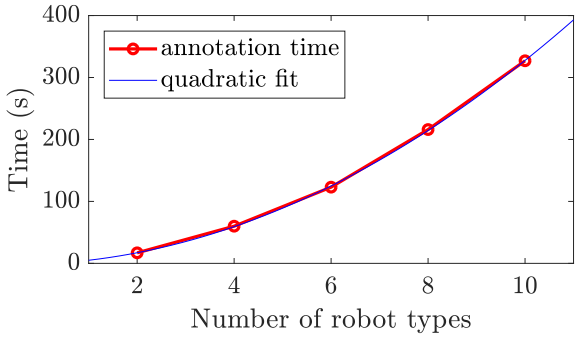


Figure 9.5: Runtime of roadmap annotation for different numbers of robot types.

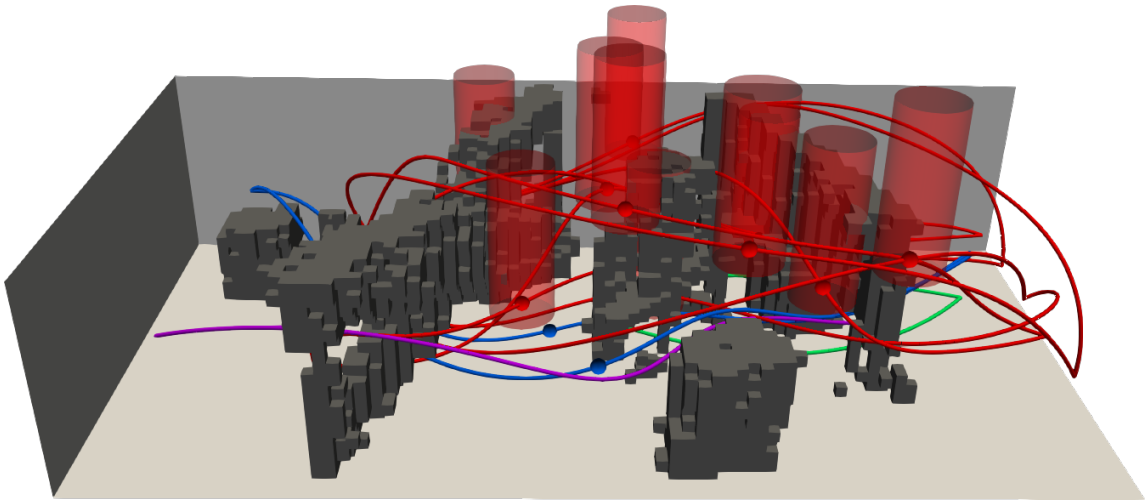


Figure 9.6: Trajectories and small/medium collision-model for our physical experiment. The robot types are color-coded (red: small, blue: medium, purple: large, green: ground). The red cylinders denote $\mathcal{R}_{\mathcal{R}}^{(small,medium)}(\cdot)$ at a time during a simulated execution. There is no medium/small collision, because the blue spheres marking the position of the medium quadrotors are outside of all red cylinders.

non-linear trajectory tracking controller, extended Kalman filter for state estimation, and trajectory evaluation on-board at 500 Hz. As *ground* robot we use the Turtlebot2 platform equipped with a single-board embedded computer running Ubuntu 16.04 and ROS Kinetic. We implement a trajectory-tracking controller that runs on-board [79] at 50 Hz. A summary of the robot properties is given in Table 9.1.

All experiments are conducted indoors using a motion-capture system to provide state estimates to the robots. All robots fuse their external state estimate at approximately 100 Hz using an on-board EKF, but otherwise only receive high-level commands such as "start trajectory execution". The execution is distributed and clocks are initially synchronized using low-latency wireless broadcasts.

We use a figure-8 trajectory that can be stretched in time to empirically determine physically safe maximum velocity and acceleration limits for each robot platform. In each case, we stop if the Euclidean position error worsens significantly. All UAVs have similar dynamic limits of $v_{\max} \approx 1.8\,\mathrm{m/s}$, reaching roll/pitch angles of around 30°. The ground robot is significantly slower with $v_{\max} = 0.5\,\mathrm{m/s}$, see Table 9.1 for details.

For pairwise interaction, we find that two quadrotors hovering next to and on top of each other

are worst-case scenarios and conducted experiments with all robot pairs except large/large, see Table 9.2. For example, if the small UAV hovers on top of a medium UAV a vertical safety distance of 0.1 m is required, while in the opposite ordering the downwash effect necessitates a vertical safety distance of 1.4 m. This results in a inter-robot collision model $R_{\mathcal{R}}^{(small,medium)}(\cdot) = \langle 0.3, 1.4, 0.1 \rangle$ or equivalently $R_{\mathcal{R}}^{(medium,small)}(\cdot) = \langle 0.3, 0.1, 1.4 \rangle$. This asymmetric effect has been noted in the literature [104] and does not occur with all UAV types. For example, the interaction between our medium and large quadrotors is nearly symmetric with 0.3 m and 0.4 m required vertical safety distances.

9.5.2 Scalability

We analyze the scalability of our approach in two experiments: first with respect to the number of robot types M and second with respect to the maximum ratio of velocity limits. We use $N = 50$ robots in a 28 m × 12 m × 12 m environment with obstacles where 25 robots start on each side and are tasked with swapping sides. We use an asymmetric collision model similar to the one we observed on real quadrotors, i.e. smaller UAVs can fly above bigger ones, but require a large vertical safety distance to fly below bigger ones (ranging from 0.02 m to 1.8 m). We use up to ten different types, where the smallest one has radius 0.02 m and height 0.01 m and the biggest has radius 0.2 m and height 0.1 m.

In the first experiment, all robots have the same velocity limits. We vary the number of robot types $M \in \{2, 4, 6, 8, 10\}$ and attempt to keep the difficulty of the problem similar by always using the full range of robot types, e.g., in case of $M = 2$, we use 25 robots of the smallest size and 25 robots of the largest size distributed equally. We find that the roadmap annotation step varies roughly quadratically with M (see Fig. 9.5), while the other steps stay mostly constant (ECBS: 2 s, two iterations of optimization: 60 s). Each of the roadmaps belonging to a single robot type has about 800 vertices and 2500 edges, because the desired Δ^k is constant. Thus, the super roadmap contains approximately a factor of M more vertices and edges compared to a single roadmap. The annotation step requires checking every pair of entities in the roadmap, resulting in a quadratic runtime in M. However, this computation is a preprocessing step – robots can be assigned new start/goal locations without the need to re-run the roadmap annotation. The planning time itself only depends on the number of robots and the "hardness" of the problem. Here, "hardness" refers to the number of generalized conflicts described in Section 9.3.2. The number of conflicts might increase for large collision cylinders, but this is difficult to quantify.

In the second experiment, we use two robot types (the largest and smallest) and reduce the maximum velocity limit of the smallest robot type using factors $\{1, 2, 3, 4\}$. Lower velocities necessitate the generation of more vertices and edges for that roadmap type, e.g., the roadmap for the case where the small robot is four times slower has 28 times more vertices and 7 times more edges compared to the large robot. As before, this results in a runtime increase during the roadmap annotation (23 s to 518 s for the two extreme cases). The resulting plans require more discrete timesteps because of the smaller robots moving slower, resulting in longer runtimes for both ECBS (3 s to 7 s) and optimization (43 s to 460 s).

9.5.3 Physical Experiment

We set up an obstacle course in a space of 9 m × 4.5 m × 2 m where robots must swap positions using ten small, two medium, and one large quadrotors, and two ground robots. We create an OctoMap representation of the environment using an RGB-D camera that is tracked by our Vicon motion-capture system. The roadmap generation step takes 30 s to 120 s per roadmap, where each roadmap has 140 to 330 vertices and 260 to 1300 edges, depending on robot size, dynamic limits, and roadmap dimensions (2D or 3D). We merge the individual roadmaps to our super roadmap and annotate it with potential vertex-vertex, vertex-edge, and edge-edge conflicts in 4 s, resulting in 820

vertices and 2700 edges. Most conflicts are edge-edge conflicts with a mean of 77 conflicting edges per edge due to the large swept volumes created by our inter-robot collision model. Our discrete planner can compute discrete trajectories in less than 1 s, and the four iterations of spatial partitioning and optimization takes about 15 s. After stretching the trajectories according to our dynamic limits (2 s), the resulting trajectories are 16 s long. The trajectories and the small/medium collision model are shown in Fig. 9.6. The robots safely execute the generated trajectories simultaneously, see Fig. 9.1 for a snapshot and the supplemental video for the full execution.

9.6 Remarks

We present a motion planning method that can compute safe trajectories for heterogeneous robot teams. Our approach considers asymmetric inter-robot spatial constraints as well as the different dynamic limits for each robot type in the team. Prior alternative methods either do not scale well for large teams, may get stuck in non-trivial environments, or have not been shown to work in 3D. In contrast, we have shown that our method scales well for at least 50 robots in simulation and have physically demonstrated trajectories for 15 robots in a 3D obstacle-rich environment. To our knowledge, our method is the first method that efficiently solves heterogeneous trajectory planning problems for large robot teams.

Part III

Persistent Operation and Robust Execution

CHAPTER 10

Ground Robots in Warehouses

Executing MAPF schedules on physical robot teams remains challenging, because most efficient MAPF formulations make unrealistic simplifying assumptions. In this section, we will discuss these shortcomings as they pertain to a challenging industrial warehouse planning problem [175].

Consider the example domain in Fig. 10.1, where robots are tasked with delivering shelves to pack stations. At each station, a human worker picks one or more items from each delivered shelf. Upon completion, the robot then returns the shelf to a storage location in the warehouse. Most MAPF formulations make two significant assumptions that cannot be ignored on real robots. First, they assume that robots can act synchronously, executing exactly one action per timestep. In practice, warehouse robots are subject to at least second-order dynamic constraints; for example, moving 3 m forward continuously without stopping will be faster than stopping each meter of movement (a typical action in MAPF planners) due to finite acceleration constraints. Additionally, exact execution times may vary due to unforeseen necessary slow-downs or control inaccuracies. Second, MAPF formulations assume that the planning problem is single-shot, i.e., robots move from their current position to a goal and remain there. In real world applications, the planning problem is likely persistent and evolving (also sometimes referred to as life-long [97]). For example, robots have to move shelves continuously in the warehouse scenario, because new orders arrive regularly.

Our approach addresses these shortcomings by introducing an execution framework that is agnostic of the underlying MAPF solver. We first introduce the *Action Dependency Graph* (ADG). The ADG is a graph that captures the action-precedence relationships of a MAPF solution and can be used to enforce these relationships on real robots with higher-order dynamics. Second, we show that this data structure enables efficient and persistent performance where (re-)planning and execution occur simultaneously, avoiding robot idle time during planning. We demonstrate our approach in simulation and in a mixed reality experiment with physical differential drive robots.

10.1 Related Work

Some work introduces more realism to the common MAPF formulation. Execution robustness can be improved by avoiding $k-$delay conflicts, which guarantees collision-free operation if robots are delayed up to k timesteps [7]. Another formulation considers delay probabilities [96], where robots might stay at their current location with a given probability when tasked with a move action. In both cases, robustness is increased, but, unlike our work, newly appearing obstacles are not considered.

This chapter is based on **Wolfgang Hönig**, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Persistent and Robust Execution of MAPF Schedules in Warehouses". In: *IEEE Robotics and Automation Letters (RA-L)*. vol. 4. 2. 2019, pp. 1125–1131. DOI: 10.1109/LRA.2019.2894217.

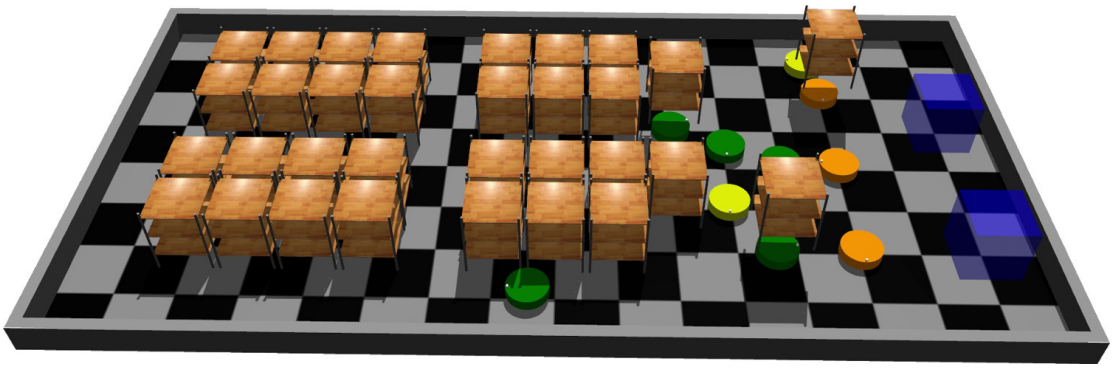


Figure 10.1: An example of the warehouse domain with 32 shelves, 12 robots, and two stations (transparent squares on the right). A task requires any robot to pick up a particular shelf, bring it to a specified station, and return the shelf to another location. The objective is to keep the stations utilized with as few robots as possible.

More realistic robot collision models can be considered when using MAPF with generalized conflicts (MAPF/C), which allows planning on roadmaps rather than grids [172]. Another generalization introduces edge weights and capacity limits [12]. In both cases, the generalization enables a wider range of robot and environment types, but does not improve persistence or robustness.

The post-processing step MAPF-POST (see Chapter 7) can be used to execute MAPF schedules on robots with varying velocity constraints. MAPF-POST leverages the fact that that precedence relations of a schedule can be extracted in polynomial time. A simple temporal network is constructed based on the precedence relation and is updated continuously in an attempt to avoid re-planning. The approach in this chapter is based on the same key insight, but uses the precedence relation on actions rather than states. This method is more robust than MAPF-POST, because it requires less communication (robots only need to communicate when an action is finished instead of broadcasting their position continuously) and has stronger guarantees on collision-free operation (robots can have arbitrary dynamic limits). We also demonstrate persistence, which was not demonstrated with MAPF-POST.

RMTRACK uses a similar idea for robustness, but does so as control law, rather than a post-processing step [60]. Robust plan-execution policies use the key idea of MAPF-POST at runtime, similar to our work, by sending messages whenever an agent enters a new state (Fully Synchronized Policy), or only for some important state changes (Minimal Communication Policy) [96]. Unlike our work, RMTRACK and robust plan-execution policies only consider delaying disturbances, while our approach addresses persistence and newly appearing obstacles as well.

MAPF formulations can be used for persistent planning of delivery tasks [97]. However, that work assumes perfect execution. In contrast, our work focuses on robust execution on real robots and allows us to efficiently and safely overlap planning and execution.

One of the key ideas of our approach, using the partial order of a schedule to deal with robustness, has been applied in operations research before [174]. We extend this approach to work in multi-robot settings, provide persistence, and show how to construct such a partial order schedule in polynomial time from an existing MAPF schedule.

Robust execution is related to cooperative obstacle avoidance, but the objective is to stay as close as possible to the pre-planned schedule. In contrast, existing obstacle avoidance techniques such as reciprocal velocity obstacles [14], buffered Voronoi cells [182], and safety barrier certificates [161] do not consider the complete pre-planned schedule. By staying close to the pre-planned schedule, robust execution reduces the risk that a collision occurs in the future. Our distributed approach to robust trajectory execution (described in Chapter 11) considers pre-planned trajectories, but

requires significantly more computation than the method outlined in this chapter.

10.2 Problem Description

We now formulate our persistent warehouse problem. Consider the map of a warehouse as a four-connected grid. Each cell in the map can either contain an obstacle, contain a station, be free space, or be a shelf-storage location. Shelf-storage cells may or may not contain a shelf at any given time, but they may not be traversed other than to attach or detach a shelf. There are P shelves with known locations either in one of the shelf-storage cells in the map or on top of a robot. There are R robots with known locations and orientations, as well as S stations at fixed known locations. A task requires a shelf to be carried to a particular station, yield there for a given estimated time, then return to a given shelf-storage location. We assume that the map fulfills the *well-known infrastructure* requirement [25] when considering potential shelf locations as only valid start and stop locations (so-called endpoints). In a well-known infrastructure, any robot waiting at one of the endpoints cannot prevent other robots from moving between any other two endpoints. Thus, in our warehouse setting robots are never obstructed from moving, even if other robots are stationary at potential shelf locations. This assumption allows our approach to provide completeness and liveness guarantees even in a persistent setting. We focus on a two-tiered objective function. The primary objective is to maximize the utilization of all stations (i.e., minimizing the human worker idle time), and the secondary objective is to minimize the number of required robots.

10.2.1 Robot Model

The proposed execution framework does not rely on a specific robot movement model, however, differential drive robots are used in our experiments. We assume each robot is circular with diameter d_r and each grid cell is large enough to contain at least one robot. Each robot can turn-in-place by 90 degrees, move forward to the next cell, attach to a shelf, detach from a shelf, and yield at a station. We denote the set of actions as $\mathcal{A} = \{\circlearrowleft, \circlearrowright, \uparrow, \mathbf{A}ttach, \mathbf{D}etach, \mathbf{Y}ield\}$. Each robot is able to localize itself in the warehouse and execute its actions autonomously using an on-board controller. While a time estimate for each action is known, the actual execution might differ. However, we assume that a robot will not diverge significantly from its path spatially and that it will eventually finish its action. Furthermore, a robot can signal, in a timely manner, when it has finished an action. A robot's ability to accelerate can be significantly dampened by carrying the heavy load of a shelf. To address this, each robot has a command queue and can combine sequential actions in its queue. For example, if three "move forward" actions are in a robot's command queue, the robot can accelerate, move three units, and decelerate in a smooth continuous motion. This results in faster and smoother execution compared to one where the robot must accelerate and decelerate for each move action. Feedback signals for each individual edge traversed are still reported.

10.2.2 Warehouse Planning Problem

We are given the map of the warehouse as an undirected graph $\mathcal{G}_E = (\mathcal{V}_E, \mathcal{E}_E)$, where vertices correspond to locations arranged in a grid and edges correspond to straight lines between locations that can be traversed by the robot without colliding with a static obstacle. A subset of the vertices $\mathcal{V}_{\hat{P}} = \{v_{p^1}, \ldots, v_{p^{\hat{P}}}\} \subset \mathcal{V}_E$ is the set of $\hat{P}$ possible shelf storage locations, arranged such that the well-known infrastructure property is fulfilled if considering those locations as the only endpoints. A different subset $\{v_{s^1}, \ldots, v_{s^S}\} \subset \mathcal{V}_E$ describes the location of the S stations. There are $R \leq \hat{P}$ robots, each of which is initially located at a vertex in $\mathcal{V}_{\hat{P}}$. Furthermore, there are $P \leq \hat{P}$ shelves, each of which is initially located at a vertex in $\mathcal{V}_{\hat{P}}$ (where it is possible that shelves and robots are co-located). Shelves are assumed to be square with a side length of d_p.

A task $T^q \in \mathcal{T}$ is a tuple $(\text{shelf}^i, \text{station}^k, \delta, v_{p^j})$, describing that shelf i must be picked up by a single robot from its current location, delivered to station k where it will approximately yield for δ seconds (during which a human can pick items from the shelf), and returned to a possibly different location v_{p^j}. When initially issued, a task is not bound to a robot and thus robots can freely be assigned to any task. New tasks may be added to $\mathcal{T}$ at any time.

Time is continuous and flows forward unabated; at each instant a robot can either wait at its current vertex or begin executing one of its actions. Let $\text{loc}(r^i, t) \in \mathbb{R}^2$ be the location of robot r^i at time t and $\text{loc}(\text{shelf}^i, t) \in \mathbb{R}^2$ the location of shelf^i at time t. Note that shelves are either at a shelf storage location or on top of a robot. A robot can drive under a shelf if it is currently not carrying another shelf. To avoid collisions, we must ensure that: i) robots never collide with each other, i.e., $\| \text{loc}(r^i, t) - \text{loc}(r^j, t) \|_2 > d_r, i \neq j, \forall t$; and ii) shelves never collide with each other, i.e., $\| \text{loc}(\text{shelf}^i, t) - \text{loc}(\text{shelf}^j, t) \|_\infty > d_p, i \neq j, \forall t$. Even if i) is enforced, ii) can occur if a robot attempts to drive over a shelf location when it still has a shelf attached.

Whenever a robot fulfills a task that brings a shelf to a station and yields, the station is considered utilized during the yield's duration. We denote the total utilization duration of station k from time 0 to time t as $dur(k, t)$. Our goal is to maximize the average station utilization over the time interval $[0, t]$, i.e., $\max u(t) = \frac{1}{tS} \sum_{k=1}^{S} dur(k, t)$. Typically, we are interested in maximizing $u(t)$ over a long time horizon, e.g., a work shift.

10.2.3 Persistent and Robust Execution

We consider an execution *persistent* if robots continue to fulfill tasks and avoid unnecessary wait times. Unnecessary wait times occur if robots cannot execute any action because they are waiting for the planner to finish; a formal definition is given in Section 10.4.1.

We consider an execution *robust* if no collision occurs even in the event of varying execution times of robot actions. Such time variations may arise due to varying dynamic limits, temporary robot malfunction, or unforeseen obstacles (e.g., items that fell from a shelf and are now blocking the robot's path).

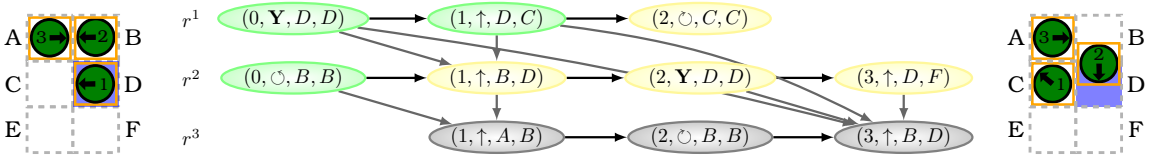
10.3 Approach

First, we simplify the planning stage to operate in discrete time and ignore higher-order dynamics, which allows us to use existing single-shot MAPF planners. Second, we leverage an action dependency graph (ADG) for robust continuous-time execution. Third, we demonstrate how the ADG can be used for persistent execution by overlapping execution and planning.

10.3.1 Single-Shot MAPF Formulation

We define the state s of a robot to be a tuple $s = (location, heading, task, stage)$, where $location \in \mathcal{V}_E$ is the current location of the robot, $heading \in \{\mathbf{S}outh, \mathbf{N}orth, \mathbf{E}ast, \mathbf{W}est\}$ is its current heading, $task \in \{None\} \cup \mathcal{T}$ the currently assigned task, and $stage \in \{\mathbf{I}dle, \mathbf{S}helf\mathbf{A}ttached, \mathbf{Y}ielded\}$ keeps track of the task progress. The possible state transitions can now be defined based on the available robot actions (see Section 10.2.1). For example, the *Attach* action can only be executed if the shelf and robot are co-located and its execution will change the *stage* variable in the robot's state from *Idle* to *ShelfAttached*. This state-action model can be used in single-shot MAPF solving frameworks with a few modifications.

In Conflict-Based Search (CBS) [131], a conflict occurs if two robots are at the same location at the same timestep (vertex conflict) or if two robots traverse the same edge at the same timestep (edge conflict). The conflict resolution of CBS is almost identical in the larger warehouse state space, but we consider an additional conflict if a robot that is carrying a shelf attempts to occupy a potential shelf location. We use ECBS-TA (see Section 6.2), a variant of Conflict-Based Search



(a) Initial state. **(b) Action Dependency Graph.** **(c) Current state.**

Figure 10.2: Example of an action dependency graph. (a) initial state next to one of the stations with locations $A, \ldots, F$ and three robots. M(b) $\mathcal{G}_{ADG}$ as constructed by Algorithm 10.1. Black horizontal edges are Type 1 edges and all other edges are Type 2 edges. (c) current state where some actions in the ADG are finished and enqueued. Green vertices in the ADG are finished, yellow vertices are enqueued, and gray vertices are staged.

that can compute a bounded suboptimal solution to simultaneously assign tasks and find action sequences. ECBS-TA takes an assignment matrix as input and thus also works with cases where some robots already have a task assigned (for example, because they already picked up a shelf), while other robots are idle.

CBS is not the only algorithm that can be used for MAPF in this setting. The well-known infrastructure property also allows us to apply prioritized planning with completeness guarantees [25]. In this case, an algorithm such as SIPP [117] can be used, but requires separating the state from the location. Specifically, all safe intervals are defined for locations only, while actions chosen change the whole state (including the location). In the prioritized planning case, task assignment is done greedily, in the order in which agents are planning their actions.

Other existing single-shot MAPF solvers, such as reduction-based solvers [179], might also be used. However, solver-specific changes are required, similar to the changes presented for CBS and SIPP. In particular, additional constraints need to be added to avoid the case that a robot that is carrying a shelf occupies a potential shelf location. The task assignment can be done independently (as in SIPP) or integrated in the MAPF solver (as in ECBS-TA). Independent of the MAPF solver used, the input of a single-shot planner is the current state of all robots $(s^1, \ldots, s^R)$. Let the output of a planner be a sequence of n^i tuples for each robot i: $p^i = [(t^i_1, a^i_1, s^i_1, g^i_1), \ldots, (t^i_{n^i}, a^i_{n^i}, s^i_{n^i}, g^i_{n^i})]$, where a^i_k denotes the kth action that should be executed starting at timestep t^i_k and that changes the robot's location from s^i_k to g^i_k. A MAPF planner computes outputs that are collision-free with respect to the criteria in Section 10.2, when considering t at fixed timesteps.

An example is shown in Fig. 10.2a. The current state of the three robots is $((D, \mathbf{W}, T^4, \mathbf{SA}), (B, \mathbf{W}, T^2, \mathbf{SA}), (A, \mathbf{E}, T^9, \mathbf{SA}))$. A valid MAPF plan is: $p^1 = [(0, \mathbf{Y}, D, D), (1, \uparrow, D, C), (2, \circlearrowleft, C, C)]$, $p^2 = [(0, \circlearrowleft, B, B), (1, \uparrow, B, D), (2, \mathbf{Y}, D, D), (3, \uparrow, D, F)]$, and $p^3 = [(1, \uparrow, A, B), (2, \circlearrowleft, B, B), (3, \uparrow, B, D)]$.

10.3.2 Action Dependency Graph: Basics

We make use of the idea that a multi-agent plan implicitly encodes dependencies between robots, for example, defining which robot should move first through a narrow passage way. Such dependencies can be extracted in polynomial time in a post-processing step, similar to MAPF-POST [168]. In MAPF-POST, the dependencies are created between states and a simple temporal network is used to create a smooth schedule with guaranteed safety distances between robots. In our approach, we define the dependencies on the robots' actions instead.

10.3.2.1 Construction

We create an action dependency graph (ADG) $\mathcal{G}_{ADG} = (\mathcal{V}_{ADG}, \mathcal{E}_{ADG})$ where $p_k^i \in \mathcal{V}_{ADG}$ and p_k^i refers to the kth tuple in plan p^i. Edges in the ADG represent inter-action dependencies. If $(p_k^i, p_{k'}^{i'}) \in \mathcal{E}_{ADG}$, then a robot is only allowed to start executing $a_{k'}^{i'}$ after a_k^i has been completed. The ADG construction is a two-step process. First, we create all vertices based on all p_k^i and connect subsequent actions for robot i with so-called Type 1 edges. Second, we find dependencies between different robots, indicating temporal precedences between actions (so-called Type 2 edges). The ADG construction pseudo code is shown in Algorithm 10.1. The construction is accomplished in $O(R^2T^2)$, where $T = \max_i n^i$. An example ADG is shown in Fig. 10.2b.

Algorithm 10.1: Action Dependency Graph Construction

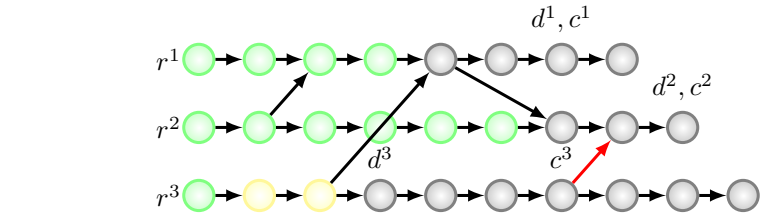
Input: Plan p^i for each robot.
Result: $\mathcal{G}_{ADG}$

```
/* create vertices and Type 1 edges */
for i ← 1 to R do
    Add vertex p_1^i to V_ADG
    p ← p_1^i
    for k ← 2 to n^i do
        Add vertex p_k^i to V_ADG
        Add edge (p, p_k^i) to E_ADG
        p ← p_k^i
/* create Type 2 edges */
for i ← 1 to R do
    for k ← 1 to n^i do
        for i' ← 1 to R do
            if i ≠ i' then
                for k' ← 1 to n^i' do
                    if s_k^i = g_k'^i' and t_k^i ≤ t_k'^i' then
                        Add edge (p_k^i, p_k'^i') to E_ADG
                        break
```

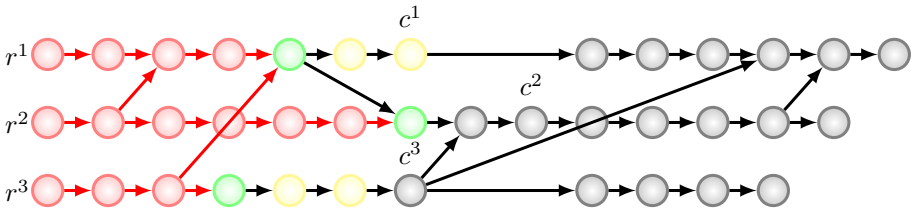
Not all standard MAPF plans can be executed robustly and collision-free with an arbitrary robot model. For example, consider four robots that move in a 2×2 grid in a circular motion. While a planner such as CBS would produce a plan, there is no safe way for robots to execute such a plan because it requires precise synchronous execution, a property that our robots do not have. Such an unsafe state transition is easily detectable as a cycle in the constructed ADG. We can also avoid such cycles during planning, for example by disallowing that a robot moves out of a cell in the perpendicular direction of another robot moving into that cell. In the CBS framework this translates to an additional edge conflict, while in SIPP the computation of the earliest arrival time can be adjusted accordingly.

10.3.2.2 Execution

At execution time, we keep track of the completion status of each vertex (action) in $\mathcal{V}_{ADG}$. Each vertex can either be staged, enqueued, or finished. We only enqueue actions into a robot's command queue if i) the previous vertex (that is connected by an incoming Type 1 edge) is already enqueued or finished, and ii) all vertices associated with incoming Type 2 edges are finished. We mark a



(a) Current action dependency graph.



(b) Updated action dependency graph after re-planning.

Figure 10.3: Overlapping of execution and re-planning. Green vertices are finished, yellow vertices are enqueued, and gray vertices are staged. Robot 2 is going to finish its current plan within the next three actions and therefore re-planning is desired. The desired set of vertices we want to commit to are labeled with d^1, d^2, d^3. The computed commit cut vertices are labeled with c^1, c^2, c^3, and are not the same as the desired vertices, because of the red dependency between robot 3 and robot 2. (b) Updated ADG after planning with shifted start times for new vertices. Completed vertices (red) can be deleted.

vertex as finished once the robot notifies the execution monitor of the successful execution of the associated action.

This approach guarantees that a robot will only move into a location after the previous robot has completely moved out of that location. While this implies coarser safety distances than MAPF-POST (the safety distance is a single cell), it requires less communication at runtime and works with arbitrary dynamic limits. In particular, we only need to track finished actions rather than the current position of all robots at all times.

If a robot detects an unforeseen obstacle in its path, the robot stops autonomously, empties its command queue, and notifies the planner of the new obstacle and the aborted command queue. New actions will be enqueued, once the planner has finished re-planning.

Consider the example in Fig. 10.2c and colored vertices in Fig. 10.2b. Robot 1 finished two actions and has one more action in its command queue, robot 2 finished its turning action and has three more actions in its command queue, and robot 3 has no action in its queue. Robot 3 cannot enqueue its next move action, until robot 2 finishes its move action first.

10.3.3 Lifelong Planning

A typical approach for dynamic scenarios is to continuously re-plan with a finite time horizon, as, for example, in model predictive control. However, this requires fast planning and state estimation. The coupling of robots in multi-robot systems makes planning typically too slow for this kind of re-planning. Another method is to plan for the next goal, once one goal is reached [97]. While this can work in decentralized settings, it does not work for a centralized planner because it neglects the finite runtime of the planner and would cause all robots to stall while a new plan is computed. Thus, it is desirable to overlap planning and execution, such that there is no delay when re-planning occurs.

Our approach is based on action dependency graphs. We detect cases when re-planning is required: either if a robot senses an obstacle on its current path or if at least one robot has an estimated fixed duration of execution remaining in the ADG. In order to overlap planning and execution, we need to find a set of *committed vertices* in $\mathcal{V}_{ADG}$ that defines the actions that the robots will execute before switching to the new plan. We use the term *commit cut* as the set of the last actions, one for each robot, that is a subset of the committed vertices. We allow continued execution of the old plan up until the commit cut. In parallel, we re-plan by constructing the start state for our single-shot MAPF planner from the final state that would be reached after the commit cut. If desired, re-planning can use the old plan as seed to find a new solution quicker.

In order to ensure a valid transition between the old and the new plan, we need to find the commit cut in the old plan, such that the old committed plan is consistent with its dependencies. We compute the commit cut in four steps, see Algorithm 10.2: First, we define a *desired set* of vertices we want to commit to, one for each robot. These vertices should be chosen such that the remaining execution time to finish those actions is larger than the expected planning time. Such a measure might require domain specific tuning, which is encapsulated in the helper function `ComputeDesiredSet`, see Line 1. In Fig. 10.3 we chose the desired set to be the actions that will be finished in three MAPF schedule timesteps. Second, we compute the reverse graph of $\mathcal{G}_{ADG}$, where the direction of all edges is reversed, see Line 2. Third, we find the reachable set of vertices by executing an exhaustive search on the reverse graph of $\mathcal{G}_{ADG}$ starting with the set of desired vertices, see Lines 3 to 10. The reachable set of vertices is a superset of the desired vertices and defines the set of committed vertices. Fourth, we find the latest occurring action for each robot in the set of committed vertices, which defines the robot's commit cut, see Lines 11 to 12.

Algorithm 10.2: Compute Commit Cut

Input: $\mathcal{G}_{ADG}$
Result: commit cut $c^i \in \mathcal{V}_{ADG}$ for $i = 1, \ldots, R$

1 $\{d^1, \ldots, d^R\} \leftarrow ComputeDesiredSet(\mathcal{G}_{ADG})$
2 $\mathcal{G}'_{ADG} \leftarrow (\mathcal{V}_{ADG}, \mathcal{E}'_{ADG})$ where $\mathcal{E}'_{ADG} = \{(u, v) | (v, u) \in \mathcal{E}_{ADG}\}$
3 $reachable \leftarrow \emptyset$
4 $q \leftarrow Queue(\{d^1, \ldots, d^R\})$
5 **while** q *not empty* **do**
6 $p^i_k \leftarrow Dequeue(q)$
7 $reachable \leftarrow reachable \cup \{p^i_k\}$
8 **for** $(p^i_k, u) \in \mathcal{E}'_{ADG}$ **do**
9 **if** $u \notin reachable$ **then**
10 $Enqueue(q, u)$
11 **for** $j \leftarrow 1$ **to** R **do**
12 $c^j \leftarrow \arg\max_k \{p^i_k | p^i_k \in reachable \wedge i = j\}$

We use the robots' states after the commit cut as starting point for our single-shot MAPF planner, with one small adjustment: we synchronize the time, using the maximum of all timesteps t^j_i of the commit cut vertices. This ensures that there will be no dependencies from the new plan to the old plan. The new plan can be added to the ADG, and dependencies computed according to Algorithm 10.1. Dependencies may exist from the old plan to the new plan, but the construction of our commit cut disallows dependencies from the new plan to the old plan. Finally, finished vertices can be safely deleted, to keep memory usage small. An example of our approach is shown in Fig. 10.3.

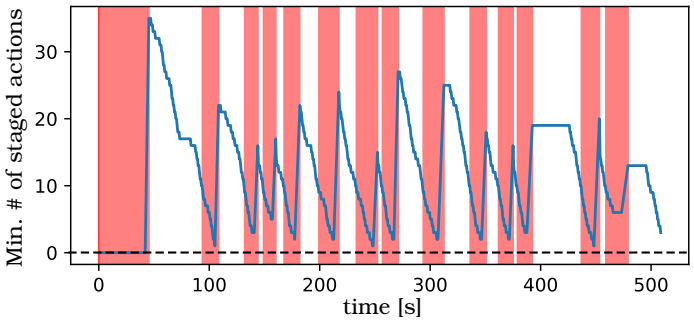


Figure 10.4: Overlap of planning and execution. The red parts mark timespans used for re-planning (an average of 15 s). The blue line shows the minimum number of staged actions over all 50 agents. This line never drops to zero after the initial plan was found, indicating that re-planning and execution always overlapped.

10.4 Evaluation

We implement our approach in C++, using our ECBS-TA implementation (see Section 6.2) and boost graph for our ADG data structure. In our experiments, we demonstrate persistence and robustness in simulation and Mixed Reality. Our method also provides a significant performance gain over a baseline implementation. A video of our experiments is provided in the supplemental material.

10.4.1 Simulation

We evaluate our approach in simulation using HARMONIES[1], a simulator developed at Amazon Robotics specifically for quantifying academic results in warehouse-like environments. HARMONIES implements the robot model as discussed in Section 10.2.1, where robots have acceleration limits (which are not modeled in most MAPF planners, including ours). The simulator runs on Amazon Web Services, executes actions in real-time, and provides a RESTful API. Simulator client applications can enqueue actions and receive errors/statistics during the execution, including the number of occurred collisions and the station utilization $u(t)$. Our client was executed on a laptop (i7-4600U 2.1 GHz and 12 GB RAM).

We evaluate persistence by counting the number of staged actions in the ADG per robot at a fixed time interval of 0.5 s. The unnecessary wait time for a robot is the cumulative time while the robot had no actions staged; the total unnecessary wait time is the sum of all per-robot wait times. We test our approach on the "small_1" scenario in HARMONIES (50 robots, 600 shelves, 8 stations). We use ECBS-TA with a bounded suboptimal factor of 2; trigger re-planning if there are fewer than 10 actions in a robot's queue; and use a lookahead of 10 actions for the selection of the desired set of commit cut vertices. These settings were found empirically to be sufficiently large to overlap re-planning and execution, see Fig. 10.4. Re-planning takes on average 15 s and is thus a significant factor when minimizing wait times. In our experiment, there was no unnecessary wait time and we achieved a station utilization of $u(500) = 0.24$. We also evaluated robustness by measuring the number of reported collisions from the HARMONIES simulator, which for our experiments was zero.

[1] **H**igh-fidelity **A**utonomous-agent **R**esearch in **M**otion-planning and **O**rganization over a **N**etwork at **I**ndustrial **E**xhibited **S**cale; For more information contact harmonies@amazon.com.

10.4.2 Mixed Reality Experiment

We implement our approach in a mixed reality experiment (see Chapter 5), which allows us to show robustness with respect to newly appearing obstacles and unmodeled dynamics. We use Gazebo as our virtual environment as well as our robotics simulator. Our custom Gazebo world plugin has three types of differential drive robot models: i) simulated agents that do not use the physics engine and move perfectly with a constant velocity; ii) simulated robots that use a physics engine and are modeled after the iRobot Create2 robots; and iii) physical iRobot Create2 robots. All three robot types have different (and in the discrete planning problem, unmodeled) dynamics. Shelves and stations are visualized only and not modeled using the physics engine for all robots.

Our iRobot Create2 robots are equipped with one of ODROID C1+ or ODROID XU4 single-board computers that run Ubuntu 16.04 with ROS Kinetic. Controller and command queues are executed on-board the robots. State estimation is done using a motion-capture system. The robots communicate with the simulator using ROS services. These services are used to enqueue new actions and to send notifications of successfully completed actions.

In our experiment, we use a total of 12 robots: 6 physical robots, 2 simulated robots, and 4 simulated agents. The robots have different dynamics based on their type. For physical robots, battery level also affects their dynamics. None of the dynamics were explicitly modeled in our MAPF solver. During the run, we introduce an unknown (virtual) obstacle, which robots can detect in our mixed reality setting. Furthermore, we artificially change the maximum speed of one of the robots during a time period. No collisions occurred during our experiment, showing that our approach is robust to varying and unforeseen dynamics. We demonstrate persistence by executing our experiment for several minutes such that each robot finished multiple tasks without any execution delays caused by re-planning. A screenshot is shown in Fig. 10.1.

10.4.3 Baseline

We compare our approach to a simple baseline that, like our approach, is an execution framework relying on existing MAPF solvers. Our baseline approach executes the MAPF schedule synchronously, that is, we only enqueue one action per timestep per robot, and wait until all robots finished executing their current action before enqueuing actions for the next timestep. This baseline is comparable to the ALLSTOP strategy in previous work [60]. If an unforeseen obstacle is detected, or at least one robot finishes its current schedule, synchronous re-planning is triggered. This baseline provides persistence and robustness like our approach, but causes robots to spend a significant amount of time waiting rather than executing actions.

We use the HARMONIES simulator on the same scenario with the same settings as in Section 10.4.1. Using the baseline, the achieved station utilization is $u(500) = 0.09$ — over 2.5 times lower compared to our approach. The utilization of the baseline and our approach over time is shown in Fig. 10.5.

10.5 Remarks

We present an execution framework that can be used to execute MAPF plans on physical robots persistently and robustly. We demonstrate both properties in a mixed reality experiment and in simulation. For persistence, we show that planning and execution can be overlapped such that robots do not have to wait until the planner finds a new solution. For robustness, we test with unknown, time-varying dynamic limits as well as a randomly appearing obstacle.

We believe that our approach closes the gap between recent advances in multi-agent path finding algorithms from the artificial intelligence community and practical applications in robotics. Our approach can be used with existing MAPF planners with slight modification and requires little additional computation to ensure persistent and robust execution. It also uses significantly less

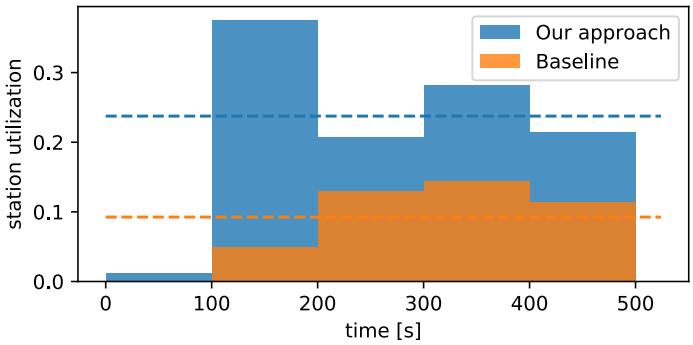


Figure 10.5: Station utilization of our approach compared to a baseline over time on the "small_1" scenario in HARMONIES. Bars represent the station utilization for a time period, e.g., our approach reached a station utilization of 0.37 during $t \in [100, 200]$, while the baseline reached a station utilization of 0.05 during the same time. The dashed lines show the utilization after 500 seconds, i.e., $u(500)$, for our approach and the baseline.

communication than other existing execution frameworks, such as MAPF-POST. In the future, we hope that our method will allow researchers and practitioners to apply and study MAPF planners in additional realistic persistent applications.

CHAPTER 11

Distributed Approach for Differentially-Flat Robots

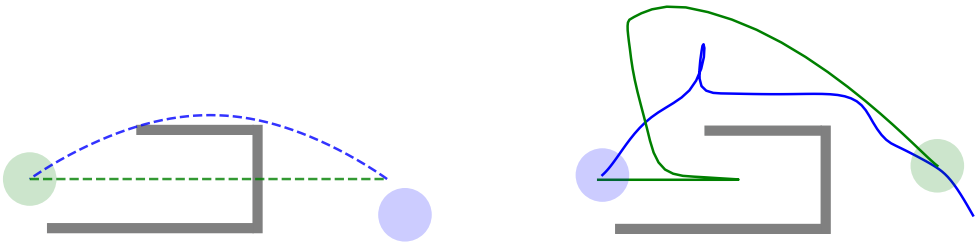
The robust execution solution in the previous chapter is fully centralized, which limits the applicability to scenarios with good network connectivity. This chapter discusses a fully decentralized approach to robust execution, where robots can sense each others' position but do not communicate.

One existing solution to compensate for changes in the environment or imperfect execution is to apply cooperative collision avoidance strategies, such as ORCA [16], at runtime. However, such algorithms often operate locally and do not take the pre-planned trajectories into account. Robust trajectory execution, on the other hand, avoids future collision more effectively because it directly considers pre-planned trajectories. Consider the example in Fig. 11.1(a), where two robots must swap positions. The pre-planned trajectories are collision-free, but they do not consider the newly introduced obstacle and the blue robot does not start at its correct location. However, the pre-planned trajectories can be used as guidance for replanning. In this example, robots can get stuck if a local cooperative collision avoidance strategy is applied. Using our method, the robots can successfully swap positions, while staying as close as possible to the pre-planned trajectories, as in Fig. 11.1(b). Our method is fully distributed and requires no communication. The robots only need to know their own trajectories and be able to sense other robots' positions and the obstacles around them.

Robust trajectory execution is an extension of cooperative collision avoidance where the objective is to stay as close to the originally planned trajectories as possible. In contrast, traditional collision avoidance methods frequently only take a desired velocity, desired goal state, or desired action as input (we discuss these in more detail in Section 11.1). Our method relies on *Buffered Voronoi Cells* (BVC) [182] as the underlying cooperative collision avoidance strategy and retains the same theoretical guarantees. We employ a novel combination of trajectory optimization and discrete search-based planning using a dynamic receding horizon approach. The discrete search allows us to avoid local minima effectively even in difficult scenarios, while the trajectory optimization generates smooth trajectories that are collision-free.

This chapter is based on Baskın Şenbaşlar, **Wolfgang Hönig**, and Nora Ayanian. "Robust Trajectory Execution for Multi-robot Teams Using Distributed Real-time Replanning". In: *International Symposium on Distributed Autonomous Robotic Systems (DARS)*. vol. 9. Springer Proceedings in Advanced Robotics. Springer, 2018, pp. 167–181. DOI: 10.1007/978-3-030-05816-6_12.

Baskın Şenbaşlar was a Master's student whom I supervised. He implemented data structures and hyperplane optimization and I implemented the discrete search and construction of the quadratic programs. Experiments were executed jointly.



(a) Initial plan. **(b) Executed trajectory.**

Figure 11.1: (a) Two robots (green and blue circles) are tasked with following their pre-planned trajectories (green and blue dashed lines). The initial plans were created without the knowledge of the obstacle (gray) and the blue robot does not start at its planned initial position. (b) Our approach computes smooth trajectories in real-time, avoiding both the new obstacle and other robots while staying close to the pre-planned trajectory.

The main contribution of this work is a novel distributed algorithm for robust trajectory execution that considers higher-order dynamic limits. It also compensates for a variety of dynamic changes, including imperfect motion execution of robots, newly appearing obstacles, robots breaking down, or external disturbances. We show in simulations that our method avoids deadlocks better than ORCA [16]. Furthermore, we implement and test our approach on a team of six differential drive robots with several dynamic environmental changes.

11.1 Related Work

Our method is closely related to cooperative collision avoidance such as reciprocal velocity obstacles, buffered Voronoi cells, and safety barrier certificates. Methods based on reciprocal velocity obstacles (RVO) [14] assume that robots continue with constant velocity and compute the safe configuration space such that no other robot might collide for the time horizon. Many extensions of the RVO method have been proposed, see [3] for an extensive overview. Buffered Voronoi Cells (BVC) [182] compute the safe configuration space for a robot by its Voronoi cell shifted by the physical extent of the robot. Safety barrier certificates achieve collision-free operation by modifying a user-specified controller such that no collision can occur [161]. Our robust trajectory execution approach uses cooperative collision avoidance at its core (specifically BVC), while extending it to minimize the difference to the original trajectories (rather than just a preferred velocity as in [3], preferred control input as in [161], or difference over a fixed time horizon as in [182].)

Our method is inspired by our previous work on offline planning for robotic teams (see Chapter 8) and uses the same optimization framework based on Bézier curves to generate trajectories, although with a different cost function. Similar to our previous work we use discrete search to quickly get out of local minima, but do so in a distributed manner.

While our approach naturally works in multi-robot settings, some of the methods are inspired by single-robot optimization and collision avoidance. Local collision avoidance for single robots such as UAVs can be formulated as optimization problems [111, 153]. In both cases collisions are considered as a soft constraint in the cost function using a Euclidean (Signed) Distance Field. In contrast, our formulation uses a hard constraint allowing us to easily detect infeasible trajectories. The optimization can use a discrete plan as an initial guess [111] or shift the existing trajectory based on newly appearing obstacles [153]. In contrast, our approach shifts the existing trajectory whenever possible, while falling back to an efficient discrete planner with dynamic receding horizon to avoid local minima.

11.2 Problem Formulation

The general problem we would like to solve can be formulated as follows. Consider a group of m robots. Each robot i is given the following:

$$\begin{aligned}
\mathbf{o}_i(t):&\ \text{original trajectory } (\mathbb{R} \to \mathbb{R}^n) \text{ of } i^{\text{th}} \text{ robot where time } t \in [0, T_i],\\
c:&\ \text{order of derivative up to which smoothness is required,}\\
R(\mathbf{p}):&\ \text{convex collision shape of any robot at position } \mathbf{p},\\
\gamma_k:&\ \text{dynamic limit of the robot for the } k^{\text{th}} \text{ derivative of its trajectory.}
\end{aligned}$$

Each robot i can sense the positions $\{\mathbf{p}_1, \ldots, \mathbf{p}_m\}$ of other robots as well as the current occupied space $\mathcal{O}_i$ around it[1]. We represent $\mathcal{O}_i$ as a set of θ convex obstacles. Robots are unaware of the other robots' planned trajectories, and cannot communicate with each other. Each robot i must execute a trajectory $\mathbf{f}_i(t)$, where $\mathbf{f}_i(t)$ is a solution to the following optimization problem:

$$\begin{aligned}
\text{minimize } & \int_0^{T_i} \left\|\mathbf{f}_i(t) - \mathbf{o}_i(t)\right\|^2 dt\\
\text{subject to } & \\
& \mathbf{f}_i(t) \text{ is continuous up to degree } c,\\
& \frac{d^j \mathbf{f}_i}{dt^j}(0) = \frac{d^j \mathbf{p}_i}{dt^j}(0) \text{ for } j \in \{0, 1, ..., c\}\\
& \mathbf{f}_i(t) \text{ is collision-free, and}\\
& \left\|\frac{d^k \mathbf{f}_i(t)}{dt^k}\right\| \leq \gamma_k \text{ for all desired } k,\\
& \text{where } t \in [0, T_i].
\end{aligned} \tag{11.1}$$

We solve this problem approximately, using a dynamic receding horizon approach iteratively. At every iteration K, robot i plans a trajectory $\mathbf{f}_i^K(t)$ that starts at the robots' current position and is safe to execute up to the user-provided period δt. We set $R(\mathbf{p})$ to a sphere with radius r_s centered at $\mathbf{p}$.

11.3 Preliminaries

This section introduces important mathematical concepts and notations that will be used in this chapter.

11.3.1 Buffered Voronoi Cell

Given a set of m robots with positions $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m \in \mathbb{R}^n$ and radii $r_s \in \mathbb{R}$, the buffered Voronoi cell $\mathcal{V}_i$ of robot i is defined as [182]:

$$\mathcal{V}_i = \left\{\mathbf{p} : \forall_{j \neq i} \frac{\mathbf{p}_j - \mathbf{p}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|} \cdot \mathbf{p} - \frac{\mathbf{p}_j - \mathbf{p}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|} \cdot \frac{\mathbf{p}_j + \mathbf{p}_i}{2} + r_s \leq 0\right\}, \tag{11.2}$$

where $\|\mathbf{p}\|$ is the L^2-norm of the vector $\mathbf{p}$.

[1] Since our approach can accommodate many sensing modalities, we do not provide a specific sensing capability in the general problem.

The inequality inside Eq. (11.2) defines a hyperspace $\mathcal{S}_i^j$ bounded by hyperplane $\mathcal{H}_i^j$ that separates point $\mathbf{p}_i$ from $\mathbf{p}_j$. $\mathcal{H}_i^j$ has normal $\alpha_i^j \in \mathbb{R}^n$ and distance $\beta_i^j \in \mathbb{R}$ along α_i^j such that

$$\alpha_i^j = \frac{\mathbf{p}_j - \mathbf{p}_i}{\|\mathbf{p}_j - \mathbf{p}_i\|} \text{ and } \beta_i^j = \alpha_i^j \cdot \left(\frac{\mathbf{p}_i + \mathbf{p}_j}{2}\right) - r_s. \tag{11.3}$$

For a given buffered Voronoi decomposition of the space, any point $\mathbf{p} \in \mathbb{R}^n$ can be inside at most one of the buffered Voronoi cells. We use this property in order to avoid robot-to-robot collisions.

Using the hyperspaces $\mathcal{S}_i^j$ we can reformulate $\mathcal{V}_i$ as follows:

$$\mathcal{V}_i = \bigcap_{j \neq i} \mathcal{S}_i^j \text{ , where } \mathcal{S}_i^j = \left\{\mathbf{p} : \alpha_i^j \cdot \mathbf{p} - \beta_i^j \leq 0\right\}. \tag{11.4}$$

Thus, we can compute the buffered Voronoi cell of any robot i as the set of the hyperplanes $\mathcal{H}_i^j$ in $O(m)$ time.

11.3.2 Bézier Curve

A degree d Bézier curve $\mathbf{f}(t)$ implicitly parameterized by duration T is defined by $d+1$ control points $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_d \in \mathbb{R}^n$ such that

$$\mathbf{f}(t) = \sum_{i=0}^{d} \mathbf{P}_i \binom{d}{i} \left(\frac{t}{T}\right)^i \left(1 - \frac{t}{T}\right)^{d-i}, \quad 0 \leq t \leq T. \tag{11.5}$$

The curve starts at $\mathbf{P}_0$ and ends at $\mathbf{P}_d$, however does not interpolate other control points. A Bézier curve lies completely inside the convex hull of its control points [48]; we leverage this property to avoid robot-to-obstacle collisions.

We use splines as trajectories with user-specified number of pieces (l) and degree (d), where each piece is a degree d Bézier curve. Given a trajectory $\mathbf{f}_i^K(t)$ for robot i with l pieces and duration T_i^K at iteration K, $T_{i,j}^K$ denotes the duration of the j^{th} piece where $j \in \{1, 2, \ldots, l\}$. $\mathbf{f}_{i,j}^K(t)$ denotes the j^{th} piece of the trajectory with implicit duration parameter $T_{i,j}^K$ where $t \in [0, T_{i,j}^K]$. $\mathbf{P}_{i,j,\rho}^K$ denotes the ρ^{th} control point of the j^{th} piece where $\rho \in \{0, 1, \ldots, d\}$.

11.3.3 Trajectory Optimization using Quadratic Programming

Our replanning approach utilizes *quadratic programming* (QP) for trajectory optimization. The decision variables $\mathbf{x}$ are the concatenated Bézier curve control points. The overall structure of our quadratic optimization problem is as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\mathbf{x}^T H \mathbf{x} + \mathbf{x}^T \mathbf{g} \\ \text{subject to} \quad & \mathbf{lbA} \leq A\mathbf{x} \leq \mathbf{ubA}. \end{aligned} \tag{11.6}$$

A quadratic cost function is represented using the matrix H and the vector $\mathbf{g}$. The quadratic cost function we use is described in Section 11.4.2.

The constraints are represented using the matrix A with vectors $\mathbf{lbA}$ and $\mathbf{ubA}$. Notice that all constraints should be linear in the decision variables. There are three types of constraints we impose on the curves: *initial point constraints*, *continuity constraints*, and *hyperspace constraints*. An initial point constraint on a Bézier curve requires the initial point of the curve to be equal to a given vector in a specified degree of differentiation. This translates to n linear constraints on control points, where n is the dimension we are working in. A continuity constraint between curve j and curve $j+1$ requires the end of curve j to be equal to the beginning of curve $j+1$ at any order of

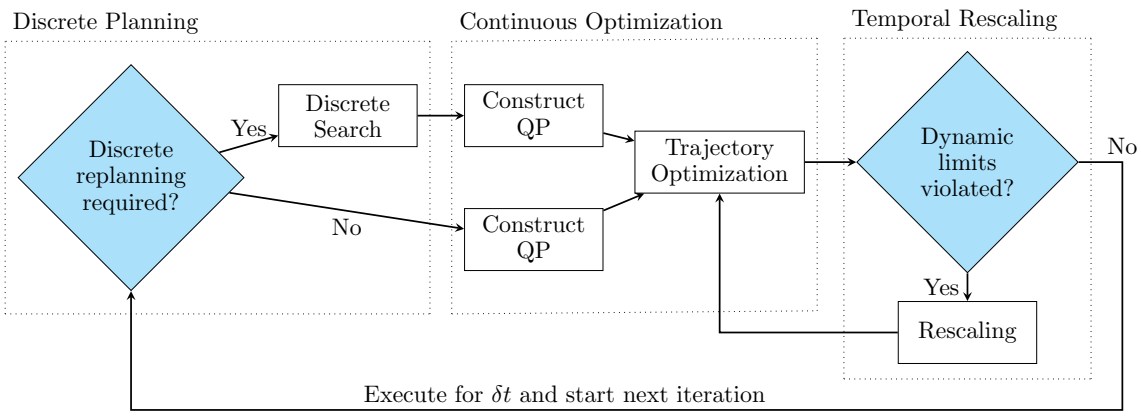


Figure 11.2: Overview of the replanning pipeline.

differentiation. We take the vector difference of those values and require it to be equal to $\mathbf{0}$. This translates to n linear constraints on control points. A hyperspace constraint requires all control points of a curve to be on a specific side of a hyperplane. If a curve has $d+1$ control points, this translates to $d+1$ constraints on control points. All three types of constraints are linear in control points. The exact construction is discussed in Chapter 8.

11.4 Approach

Replanning is done at a fixed period of δt. In each iteration K, we sense the other robots' positions to compute the buffered Voronoi cell $\mathcal{V}_i$, update our current representation of the occupied space ($\mathcal{O}_i$), and compute a trajectory $\mathbf{f}_i^K(t)$. The planning horizon τ' is automatically adjusted, but the desired planning horizon τ can be provided.

We execute the following three major components iteratively: *discrete planning* that is used to efficiently plan around new obstacles, *trajectory optimization* to generate smooth and collision-free trajectories, and *temporal rescaling* to enforce the dynamic limits of the robot (see Fig. 11.2).

In the beginning of each iteration, we check several conditions to decide if discrete planning is required. If discrete planning is required, we execute a discrete search that results in a discrete path that is collision-free but not smooth. We use this discrete path as an initial guess in trajectory optimization. If discrete planning is not required, we use the control points of the previous plan as the initial guess.

We construct a QP with hard constraints for trajectory optimization in a slightly different way depending on whether discrete planning was executed or not. In both cases, buffered Voronoi cells are used to ensure collision-free operation for time δt and collisions with static obstacles are avoided for the planning horizon.

Dynamic limits cannot be represented as linear constraints in our QP. Thus, we check dynamic limit violations in the temporal rescaling stage that runs after optimization. While dynamic limits are violated, we increase the durations of all trajectory pieces uniformly, and since the initial point constraints are violated when the durations are increased, we re-solve the QP.

At the end of each iteration, each robot has its trajectory $\mathbf{f}_i^K(t)$ that is guaranteed to be collision-free up to time δt; is continuous up to the c^{th} derivative; obeys the dynamic limits of the robot; tries to stay close to the original trajectory; and is a good starting point for the next iteration. We execute this trajectory for a period of δt and replan for the next iteration.

11.4.1 Discrete Planning

Robot i executes discrete planning if any of the following conditions are true, where $\psi = K\delta t$ is the current time:

1. The original trajectory is not collision-free for the desired time horizon τ, i.e., $\exists t \in [\psi, \psi + \tau] : R(\mathbf{o}_i(t)) \cap \mathcal{O}_i \neq \emptyset$,
2. The first piece of the previously planned trajectory is outside the robot's buffered Voronoi cell, i.e., $\exists t \in [0, T_{i,1}^{K-1}] : \mathbf{f}_{i,1}^{K-1}(t) \notin \mathcal{V}_i$, or
3. The previously planned trajectory is not collision-free for the desired time horizon τ, i.e., $\exists t \in [0, \tau] : R(\mathbf{f}_i^{K-1}(t)) \cap \mathcal{O}_i \neq \emptyset$.

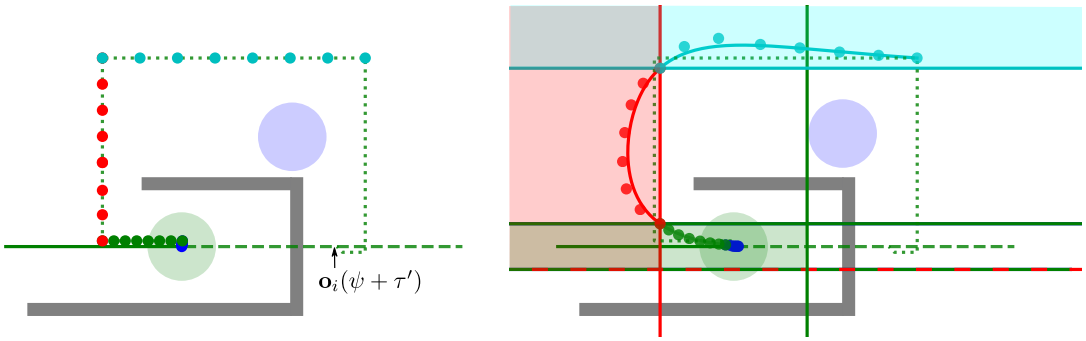
The first condition handles cases where previously unknown obstacles block the pre-planned path of a robot. The second condition handles cases where previously unknown robots appear and cases where robots are close and moving towards each other. The third condition handles dynamic obstacles, and also infeasibilities and numerical issues that resulted in a trajectory with collisions in the previous iteration. Sections 11.4.4 and 11.5.1 detail reasons for infeasibilities and numerical issues, respectively.

Discrete planning uses a dynamic receding horizon approach. First, we find the earliest time $\tau' \in [\min(\tau, T_i - \psi), T_i - \psi]$ where the original trajectory is collision-free at time $\psi + \tau'$ with respect to both obstacles and other robots. Second, we use a discrete graph search to find a path from the robot's current location to $\mathbf{o}_i(\psi + \tau')$ that avoids both static obstacles and other robots. If τ' does not exist or no solution path exists, we skip the discrete planning stage and construct the QP as if discrete planning was not required. Third, we use the first l segments of the discrete path to uniformly place the new estimated control points on top of those segments. In case the discrete path has fewer than l segments, the last discrete segment is shared between multiple Bézier curves. Finally, we adjust $T_{i,j}^K$ relative to the segment lengths and scale by τ', such that we would arrive at time $\psi + \tau'$ at $\mathbf{o}_i(\psi + \tau')$ if we followed the discrete path with constant speed. To guarantee collision-free operation, we ensure that $T_{i,1}^K \geq \delta t$ in any case.

An example is shown in Fig. 11.3 (a), with parameters $l = 4$ and $d = 7$. Discrete planning is executed because the original trajectory (green dashed line) passes through an obstacle. We find the earliest time τ' such that $R(\mathbf{o}_g(\psi + \tau'))$ does not intersect with the obstacle and the blue robot where g is the green robot. The discrete planner is then used to find a path (green dotted line) that avoids both the static obstacle and the other (blue) robot, with a total of six path segments. The first four segments ($l = 4$) are used to place new guesses of Bézier control points (blue, green, red, and cyan circles). Notice that since $d = 7$, each curve has eight control points, while some of the points are overlapping. The duration $T_{i,j}^K$ for each Bézier curve is adjusted linearly according to its segment length; for example, the duration of the segment with the red control points ($T_{i,3}^K$) is approximately twice as long as the duration of the segment with the green control points ($T_{i,2}^K$).

11.4.2 Continuous Optimization

We compute a new trajectory by formulating a quadratic program where the decision variables are the concatenated control points of the pieces. The parameters d and l of the pieces (see Section 11.3.2) are provided by the user. If discrete planning is not performed, initial guesses of the decision variables are copied from the previous iteration, and the durations $T_{i,j}^K$ are set uniformly to $\frac{\min(\tau, T_i - \psi)}{l}$. If discrete planning is performed, initial variables and durations are calculated from



(a) Discrete path. **(b) Continuous trajectory.**

Figure 11.3: Example at $t = 3.9$ s. (a) Discrete path around an obstacle and other robot back to the original trajectory. (b) Continuous trajectory split into four pieces and respective hyperspaces.

the discrete path as explained in Section 11.4.1. The objective that we minimize is defined

$$\sum_{e=1}^{\mathcal{E}} \lambda_e \int_0^{T_i^K} \left\| \frac{d^e \mathbf{f}_i^K(t)}{dt^e} \right\|^2 dt + \sum_{j=1}^{l} \theta_j \left\| \mathbf{P}_{i,j,d}^K - \chi_j \right\|^2, \tag{11.7}$$

where χ_j is the point where we want piece j to end. The first term of the objective minimizes the energy along the trajectory, and is the combination of integrated squared derivatives up to user provided degree $\mathcal{E}$ with weights λ_e [121, 172]. The second term of the objective penalizes deviation from the given end points for each piece of the trajectory with different weights θ_j. In case discrete planning is performed, we attempt to get to the position of the last guessed control point, i.e., we set θ_l to a positive value, enforcing $\mathbf{P}_{i,l,d}^K = \chi_l$, and $\theta_j = 0, \forall j < l$. If discrete planning is not performed, we attempt to stay as close as possible to the original trajectory, i.e., we set $\theta_1 = 0$, and $\theta_j, \forall j \geq 2$ to positive values (increasing with j) and $\chi_j = \mathbf{o}_i(\psi + \sum_{u=1}^{j} T_{i,u}^K)$. The matrix H (see Section 11.3.3) for the first term of our objective can be constructed as in Chapter 8. The second term is a quadratic function of the control points; hence it is straightforward to construct the H matrix and the $\mathbf{g}$ vector.

For robot-to-robot collision avoidance, the buffered Voronoi hyperplanes are computed according to Eq. (11.3) and $m - 1$ hyperspace constraints are added for the first piece. These constraints ensure that the first piece stays inside $\mathcal{V}_i$ because of the convexity of $\mathcal{V}_i$ and the convex hull property of Bézier curves. As long as $T_{i,1}^K \geq \delta t$ and all other robots stay inside their Voronoi cells up to time δt, we can be sure that no robot-to-robot collision will occur up to time δt.

For robot-to-obstacle collision avoidance we compute separating hyperplanes between convex obstacles $\mathcal{O}_i$ for each curve piece j. Let M_j^b be the hyperplane that separates the initially guessed control points of the j^{th} piece from the b^{th} convex obstacle obtained from $\mathcal{O}_i$ (these can be computed, e.g., using support vector machines [36]). We shift each hyperplane towards its obstacle and then shift it back using the radius r_s to account for the physical extent of the robot. We add hyperspace constraints as before, requiring control points of the j^{th} piece in the non-occupied side of each hyperplane M_j^b. These constraints ensure that no robot-to-obstacle collision will occur up to time τ'. In case discrete planning was executed, we additionally treat other robots as static obstacles. Figure 11.3(b) shows the effective set of hyperspaces for our example.

Moreover, we add continuity constraints that enforce the continuity requirements between pieces and initial point constraints that enforce continuity requirements between iterations.

All constraints are linear and matrix A and its bounds can be constructed as in Section 11.3.3. The number of decision variables in our QP is $l(d+1)n$. Let θ' describe the number of considered

static obstacles, i.e., θ' is equal to $\theta + (m-1)$ if discrete planning was performed and θ otherwise. We add $(m-1)(d+1) + \theta' l(d+1) + (c+1)nl$ linear constraints, where the terms refer to the Voronoi hyperspace, obstacle hyperspace, and continuity constraints, respectively. For our example in Fig. 11.3, we have $n = 2$, $m = 2$, $d = 7$, $l = 4$, and $c = 2$. Thus, we have 64 decision variables and $8 + 128 + 24 = 160$ linear constraints.

11.4.3 Temporal Rescaling

Since we use fixed durations of the pieces and do not account for the dynamic limits of the robot during optimization, the resulting trajectory may violate the dynamic limits of the robot. After trajectory optimization, we calculate the maximum magnitudes Γ_k of the k^{th} derivatives of the curve, and check if there exists a k such that $\Gamma_k > \gamma_k$, where γ_k is the dynamic limit of the robot in the k^{th} derivation degree. If that is the case, we uniformly scale the piece durations $T_{i,j}^K$, and re-run the trajectory optimization with the same exact constraints using the previous result as the initial guess. If the dynamic limits are not violated, no temporal rescaling is needed and the trajectory is feasible.

11.4.4 Theoretical Guarantees

For robot-to-robot collision avoidance our approach uses buffered Voronoi cells which has the following theoretical guarantee: if robots start in a collision-free configuration (that is, $\|\mathbf{p}_i - \mathbf{p}_j\| \geq 2r_s, \forall i \neq j$), then all future configurations are collision-free. However, this guarantee has only been proven for the case of synchronous robot execution, if robots have first-order integrator dynamics ($c = 0$), and if they execute their trajectories perfectly [182]. The QP in our formulation has additional higher-order continuity constraints that can cause it to be infeasible. However, in this case, one can simply fallback to the QP formulation of the BVC approach [182] to retain the same theoretical guarantee.

Formal guarantees under arbitrary disturbances and higher order dynamics cannot be provided. In fact, our QP can fail if it is not feasible to satisfy all safety and continuity constraints under the given dynamic limits. However, our empirical evaluation presented in Section 11.5.1 shows that the QP rarely fails and even if it does, the robots do not collide with each other and the obstacles since the QP becomes feasible in the following iterations. In addition, our QP formulation allows us to easily detect failure cases because we model all safety-critical parts as hard constraints.

Similar to other work, there are no formal liveness guarantees and there might be deadlocks [182]. Nevertheless, our approach works in practice for robots with higher-order dynamics, if robot execution is asynchronous, or trajectories are not executed perfectly.

11.5 Evaluation

We implement our approach in C++. We use an occupancy grid as the environment representation, because previous work has shown that such data structures can be updated in real-time on robots that are equipped with a LIDAR sensor or an RGB-D camera. In particular, OctoMap [71] is an octree-based 3D occupancy grid that can be run on unmanned aerial vehicles with at least 4 Hz update rate [111]. OctoMaps are memory efficient, but update operations can show high execution time variance. For local replanning, occupancy grids using ring buffers as data structures have been shown to achieve near constant execution time [153]. Our implementation uses a simple pre-initialized 2D occupancy grid.

We use the CVXGEN-package [102] to generate small QPs to find separating hyperplanes between control points and obstacles. We test with qpOASES [51] and OSQP [140] as QP solvers; both are open source and have been shown to work well in model predictive control scenarios.

Table 11.1: Runtime with varying curve count l.

#	l	θ	m	t_{avg} [ms]
1	4	0	4	10
2	8	0	4	15
3	10	0	4	13
4	12	0	4	27
5	16	0	4	107

Table 11.2: Runtime with varying occupied cells θ.

#	l	θ	m	t_{avg} [ms]
6	4	4	4	9
7	4	62	4	28
8	4	196	4	47
9	4	213	4	69
10	4	1250	4	253

Table 11.3: Runtime with varying robot count m.

#	l	θ	m	t_{avg} [ms]
11	4	5	4	9
12	4	5	8	10
13	4	5	16	13
14	4	5	32	15
15	4	5	64	14

Table 11.4: Comparison of our method, ORCA, and DS+ORCA with respect to average computation time (t_{avg}), the number of robots that reach their destinations (s), and the percentage of time that our QP fails.

			ORCA		DS+ORCA		Our Method		
#	m	θ	t_{avg} [ms]	s	t_{avg} [ms]	s	t_{avg} [ms]	s	QP failures [%]
16	2	4	< 1	0	< 1	2	7	2	0.00
17	4	12	< 1	0	< 1	4	10	4	0.30
18	8	30	< 1	4	< 1	8	13	8	0.00
19	16	9	< 1	13	< 1	16	12	16	0.08
20	32	30	< 1	23	< 1	32	16	32	0.09

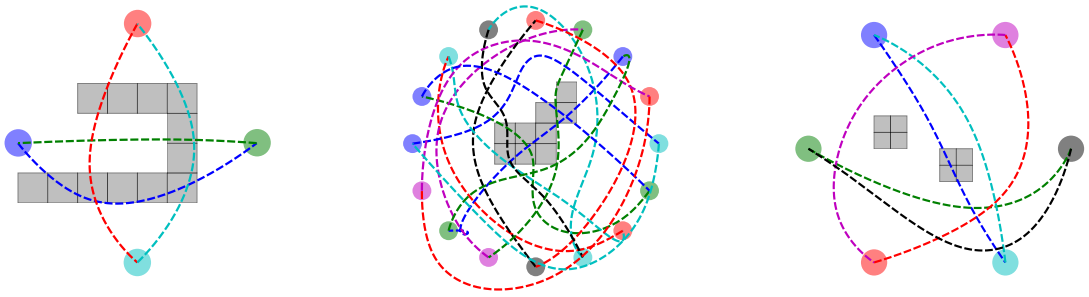
A supplemental video containing some of our simulations and physical experiments is available at https://youtu.be/LbWRvLfdwTA.

11.5.1 Simulation

We test our algorithm in a simulation running on a laptop computer (i7-4700MQ 2.4 GHz, 16 GB) with Ubuntu 16.04 as the operating system.

In the first set of experiments, we test the scalability of our method in terms of the number of pieces l we plan for, the number of occupied cells θ in the occupancy grid and the number of robots m. Our results are summarized in Tables 11.1 to 11.3, where t_{avg} is the average time that qpOASES takes per iteration. Our algorithm scales well with the number of robots. In terms of number of curves, our algorithm has almost the same performance up to $l = 10$. In our simulations and physical experiments, we use $l = 4$. The bottleneck of our algorithm is the number of occupied cells in the occupancy grid. However, as it can be seen in Table 11.2, our algorithm still has real-time capability when considering hundreds of occupied cells, assuming a 10 Hz execution. When we use OSQP instead of qpOASES, our implementation takes significantly more time if we consider many obstacles. For example, when we do experiment 7 using OSQP, it takes 297 ms on average.

We also compare our method to two ORCA variants in the second set of experiments. In the first ORCA variant, we use the RVO2 library [16] and set the preferred velocities at time ψ to $\mathbf{o}_i'(\psi)$ if $\mathbf{p}_i \approx \mathbf{o}_i(\psi)$ or to $\frac{\mathbf{o}_i(\psi)-\mathbf{p}_i}{\delta t}$ otherwise. In the second ORCA variant, we combine ORCA and our discrete planning method with a dynamic receding horizon approach (denoted as DS+ORCA). We demonstrate that this variant resolves deadlocks better than the first variant. For our method we use $\delta t = 0.1\,\text{s}$, $l = 4$, and $d = 7$ and for the ORCA variants we use $\delta t = 0.01\,\text{s}$. The results are summarized in Table 11.4. All robots using our method or DS+ORCA reach their destinations, while robots using ORCA can easily get stuck around obstacles. Our method takes more time in computation compared to the ORCA variants, but produces smooth curves up to a user-defined smoothness. We use $c = 2$ in our experiments meaning that the generated trajectories are continuous in position, velocity, and acceleration. The ORCA variants, on the other hand, provide smoothness



(a) Experiment 17. **(b) Experiment 19.** **(c) Physical experiment.**

Figure 11.4: The original trajectories and the occupancy grids in the simulation experiments 17 (a), 19 (b), and the physical experiment (c).

guarantees up to $c = 0$ only, i.e., velocities can jump between iterations. Furthermore, the ORCA variants must sense the other robots' velocities and positions while our approach relies on positions only.

We also report the percentage of time our QP fails, which was less than 0.3 % in our experiments. Notice that even if our QP fails, robots do not collide with each other and the obstacles, because the QP becomes feasible in the following iteration after 100 ms. There are two reasons for QP failures: infeasibilities, which are explained in Section 11.4.4, and numerical issues. The numerical issues stem from separating hyperplane calculations between robots and obstacles. We use hard-margin SVMs to calculate separating hyperplanes. When robots get too close to obstacles, small epsilon values in SVM optimization may result in invalid hyperplanes, and hence the QP fails. The original trajectories and the occupancy grids in some experiments are shown in Fig. 11.4 and the supplemental video contains selected simulations.

11.5.2 Physical Robots

We implement our approach on six differential drive robots (iRobot Create2) that are equipped with one of ODROID C1+ or ODROID XU4 single-board computers. Those computers run Ubuntu 16.04 with ROS Kinetic, but C1+ has very limited computation capabilities (ARM Cortex-A5, max. 10 W). The robots are arranged in a circle (2 m radius) and are tasked with swapping sides (Fig. 11.4c). We plan the original trajectories with one static obstacle using the centralized planner described in Chapter 9. Each robot receives the position information of all other robots using a motion-capture system. A trajectory tracking controller and our algorithm run on-board at a frequency of 10 Hz.

We conduct several experiments and add an additional obstacle, change the robots initial position, disturb the robots during run-time, or artificially stop one of the robots. In all cases robots successfully avoid collisions and in many cases they reach their final destination within the originally planned durations. In a few cases a livelock between two or more robots occurred, which we attribute to the fact that the robots, unlike the simulation, cannot execute very low velocity commands. The supplemental video includes recordings of our experiment.

11.6 Remarks

We present a method for robust trajectory execution that takes pre-planned trajectories as input and compensates for a variety of dynamic changes, including imperfect motion execution, newly

appearing obstacles, robots breaking down, or external disturbances. Our approach does not require communication between the robots. We use a novel planning strategy employing both discrete planning and trajectory optimization with a dynamic receding horizon approach. We demonstrate in simulation and on physical robots that we can generate smooth trajectories in real-time, while avoiding deadlocks successfully. In comparison, ORCA neither generates smooth trajectories nor avoids deadlocks in our test cases.

CHAPTER 12

Conclusions & Future Directions

This chapter provides a summary of the thesis and presents potential future research directions.

12.1 Conclusions

Motion coordination is an important foundation for mobile multi-robot teams that plans and executes robot motions. In real-world applications, such as search-and-rescue, mining, or warehouse automation, robots need to operate in environments that contain many obstacles as well as other robots. In these applications, it is also important to operate safely in dynamically changing environments, and for some tasks, persistently without interruptions. In the past, motion coordination was split into the subproblems of motion planning and motion execution. For motion planning, no solutions exists that can plan for hundreds of robots in a few minutes on commodity hardware. For motion execution, existing algorithms can cause livelocks in the environments with many unforeseen obstacles. The motion coordination problem is solved by coupling novel motion planning techniques with robust execution.

For motion planning, we develop a method that can compute trajectories for hundreds of heterogeneous robots in obstacle-rich environments within minutes. Our approach combines ideas from artificial intelligence and robotics. We use AI planners with simplistic agent models, perform a computationally efficient post-processing step to satisfy robot-specific constraints, and execute the resulting plan on physical robots with safety guarantees. For ground robots, our post-processing step is MAPF-POST, an algorithm based on simple temporal networks that can take into account simple kinematic constraints (such as speed limits) and a safety distance. The post-processing step for UAVs is based on trajectory optimization for each robot within a safe corridor. The optimization is independent from other robots and can easily be executed in parallel.

For robust execution, we develop a centralized method that works very well in warehouse domains, and a decentralized method for differentially-flat robots. In both cases, we combine the execution and motion planning by adding an additional feedback term. For the warehouse scenario, we use one of the key insights of MAPF-POST, and construct a partial order of actions given a schedule. This partial order can then be used at runtime to achieve robustness (with respect to newly appearing obstacles and higher-order dynamics) and persistence (by overlapping planning and execution). For the decentralized method, we combine an idea from collaborative collision avoidance with a distributed version of our trajectory optimization method. We use a dynamic receding horizon approach to escape local minima at runtime.

One key insight in this dissertation is that continuous (time and space) problems can be solved efficiently by first discretizing the problem in time and space, then solving the resulting combinatorial

problem, and finally using the result as an initial solution for a continuous optimization problem. Such a solution technique can still provide resolution-completeness and safety guarantees, but loses guarantees of optimality. However, our empirical results show that the generated solutions are of high quality, because both discrete and continuous steps individually provide optimality guarantees. We believe that this general framework can be useful in other related optimization problems, in particular for multi-robot coordination.

All of our approaches on motion coordination have been verified on teams of physical robots (up to 32 robots in a single experiment). To facilitate these and similar experiments, we also contributed the Crazyswarm, a widely used open-source framework for flying a swarm of quadrotors indoors; and we show that Mixed Reality is useful for both development and verification of multi-robot systems.

12.2 Future Directions

In the future, many collaborative robots should be able to operate persistently and safely in the real world, pursuing useful tasks such as finding survivors after an earthquake. The work presented thus far has been demonstrated on real robots, but only in indoor lab settings. Possible future directions would widen the utility of this research, as outlined in the following.

Support for robots that are not differentially flat The robots used in this thesis, differential-drive ground robots and quadrotors, are differentially flat, a property that we rely on in our motion planning and execution frameworks. Many other kinds of robots, for example boats, planes, and spacecraft, do not have this property and require a modified method.

Support for higher-level objectives The tasks considered in this thesis are for reaching certain goal regions. However, some real-world applications, such as information gathering or search, have different task specifications. As shown in Section 6.2, joint motion and task planning is very important in obstacle-rich environments. Conversely, finding new ways of combining motion planning with other higher-level objectives might improve the performance of the overall system.

Consider uncertainty during planning Our approach assumes perfect knowledge of the environment, either a priori (for motion planning) or at runtime through sensors (for motion execution). Such an assumption is not true in reality; thus, some single robot motion planners take various forms of uncertainty into account during planning [82]. In the future, we will need to directly consider such uncertainty for multi-robot systems as well.

Motion coordination with communication constraints The work presented in the thesis spans from fully centralized solutions (motion planning) to distributed solutions without direct communication (motion execution). In practice, communication might be available only intermittently, but robots should still attempt to coordinate their motions as well as possible. In some applications, for example for exploration or mapping tasks, motion coordination might need to directly reason about where, which, and when communication should be used.

Scalability to thousands of robots We demonstrated our motion planning approach on teams consisting of hundreds of robots. In some cases, for example warehouses or urban traffic, thousands of robots need to coordinate. Combining the presented motion coordination ideas with ideas from swarm robotics, hierarchical planning, and mobility-on-demand solutions, might provide a solution that can scale to much larger problem instances.

Better theoretical understanding of hybrid discrete/continuous planning In motion planning, spatial discretization is very common for search- or sampling-based planning methods. The tradeoff between discretization and completeness can be described using the existing

definitions of resolution-completeness or probabilistic-completeness [90]. Similarly, sampling based methods have been shown to converge to an optimal solution [81]. A key aspect of the approaches presented in this dissertation is that they discretize in both spatial and temporal domains. An important open question is whether we can formally define spatiotemporal discretization and its trade-offs as well, in particular in terms of optimality guarantees.

Bibliography

[1] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. "The Transitive Reduction of a Directed Graph". In: *SIAM Journal on Computing* 1.2 (1972), pp. 131–137. DOI: 10.1137/0201008 (cit. on p. 78).

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 978-0-13-617549-0 (cit. on p. 48).

[3] Javier Alonso-Mora, Paul A. Beardsley, and Roland Siegwart. "Cooperative Collision Avoidance for Nonholonomic Robots". In: *IEEE Transactions on Robotics (T-RO)* 34.2 (2018), pp. 404–420. DOI: 10.1109/TRO.2018.2793890 (cit. on p. 125).

[4] Javier Alonso-Mora, Andreas Breitenmoser, Martin Rufli, Roland Siegwart, and Paul A. Beardsley. "Image and animation display with multiple mobile robots". In: *International Journal of Robotics Research (IJRR)* 31.6 (2012), pp. 753–773. DOI: 10.1177/0278364912442095 (cit. on p. 10).

[5] Javier Alonso-Mora, Tobias Naegeli, Roland Siegwart, and Paul A. Beardsley. "Collision avoidance for aerial vehicles in multi-agent scenarios". In: *Autonomous Robots* 39.1 (2015), pp. 101–121. DOI: 10.1007/s10514-015-9429-0 (cit. on p. 100).

[6] Brandon Araki, John Strang, Sarah Pohorecky, Celine Qiu, Tobias Naegeli, and Daniela Rus. "Multi-robot path planning for a swarm of robots that can both fly and drive". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 5575–5582. DOI: 10.1109/ICRA.2017.7989657 (cit. on p. 100).

[7] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. "Robust Multi-Agent Path Finding". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2018, pp. 2–9. URL: https://aaai.org/ocs/index.php/SOCS/SOCS18/paper/view/17954 (cit. on p. 113).

[8] Federico Augugliaro, Angela P. Schoellig, and Raffaello D'Andrea. "Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2012, pp. 1917–1922. DOI: 10.1109/IROS.2012.6385823 (cit. on pp. 8, 85).

[9] Vincenzo Auletta, Angelo Monti, Mimmo Parente, and Pino Persiano. "A Linear-Time Algorithm for the Feasibility of Pebble Motion on Trees". In: *Algorithmica* 23.3 (1999), pp. 223–245. DOI: 10.1007/PL00009259 (cit. on p. 6).

[10] Ronald Azuma, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier, and Blair MacIntyre. "Recent Advances in Augmented Reality". In: *IEEE Computer Graphics and Applications (CG&A)* 21.6 (2001), pp. 34–47. DOI: 10.1109/38.963459 (cit. on p. 28).

[11] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. "Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS14/paper/view/8911 (cit. on pp. 44, 50, 60, 79, 105).

[12] Roman Barták, Jiri Svancara, and Marek Vlk. "A Scheduling-Based Approach to Multi-Agent Path Finding with Weighted and Capacitated Arcs". In: *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 2018, pp. 748–756. URL: http://dl.acm.org/citation.cfm?id=3237494 (cit. on p. 114).

[13] Maren Bennewitz, Wolfram Burgard, and Sebastian Thrun. "Optimizing Schedules for Prioritized Path Planning of Multi-Robot Systems". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2001, pp. 271–276. DOI: 10.1109/ROBOT.2001.932565 (cit. on p. 8).

[14] Jur P. van den Berg, Ming C. Lin, and Dinesh Manocha. "Reciprocal Velocity Obstacles for real-time multi-agent navigation". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2008, pp. 1928–1935. DOI: 10.1109/ROBOT.2008.4543489 (cit. on pp. 114, 125).

[15] Jur P. van den Berg and Mark H. Overmars. "Prioritized motion planning for multiple robots". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2005, pp. 430–435. DOI: 10.1109/IROS.2005.1545306 (cit. on p. 8).

[16] Jur van den Berg, Stephen J. Guy, Ming C. Lin, and Dinesh Manocha. "Reciprocal n-Body Collision Avoidance". In: *International Symposium on Robotics Research (ISRR)*. Vol. 70. Springer Tracts in Advanced Robotics. Springer, 2009, pp. 3–19. DOI: 10.1007/978-3-642-19457-3_1 (cit. on pp. 2, 9, 124, 125, 132).

[17] Paul J. Besl and Neil D. McKay. "A Method for Registration of 3-D Shapes". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 14.2 (1992), pp. 239–256. DOI: 10.1109/34.121791 (cit. on p. 18).

[18] Paul T Boggs and Jon W Tolle. "Sequential quadratic programming". In: *Acta numerica* 4 (1995), pp. 1–51. DOI: 10.1017/S0962492900002518 (cit. on p. 92).

[19] Mark T. Bolas and Scott S. Fisher. "Head-coupled remote stereoscopic camera system for telepresence applications". In: *Proceedings of Society of Photo-Optical Instrumentation Engineers (SPIE)* 1256 (1990), pp. 113–123. DOI: 10.1117/12.19896 (cit. on p. 28).

[20] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. "ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 2015, pp. 740–746. URL: http://ijcai.org/Abstract/15/110 (cit. on p. 44).

[21] Stephen J. Buckley. "Fast motion planning for multiple moving robots". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 1989, pp. 322–326. DOI: 10.1109/ROBOT.1989.100008 (cit. on p. 8).

[22] Rainer E. Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics (SIAM), 2009. ISBN: 978-0-89871-663-4. DOI: 10.1137/1.9781611972238 (cit. on pp. 61, 105).

[23] Jonathan Butzke, Kalin Gochev, Benjamin Holden, Eui-Jung Jung, and Maxim Likhachev. "Planning for a ground-air robotic system with collaborative localization". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 284–291. DOI: 10.1109/ICRA.2016.7487146 (cit. on p. 100).

[24] Richard H. Byrd, Jean Charles Gilbert, and Jorge Nocedal. "A trust region method based on interior point techniques for nonlinear programming". In: *Mathematical Programming* 89.1 (2000), pp. 149–185. DOI: 10.1007/PL00011391 (cit. on p. 92).

[25] Michal Čáp, Peter Novák, Alexander Kleiner, and Martin Selecký. "Prioritized Planning Algorithms for Trajectory Coordination of Multiple Mobile Robots". In: *IEEE Transactions on Automation Science and Engineering (T-ASE)* 12.3 (2015), pp. 835–849. DOI: 10.1109/TASE.2015.2445780 (cit. on pp. 8, 115, 117).

[26] Michal Čáp, Peter Novák, Jiří Vokřínek, and Michal Pěchouček. "Multi-agent RRT*: sampling-based cooperative pathfinding". In: *International conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*. 2013, pp. 1263–1264. URL: http://dl.acm.org/citation.cfm?id=2485174 (cit. on p. 9).

[27] Michal Čáp, Jiří Vokřínek, and Alexander Kleiner. "Complete Decentralized Method for On-Line Multi-Robot Trajectory Planning in Well-formed Infrastructures". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 2015, pp. 324–332. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/view/10504 (cit. on p. 8).

[28] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm, 27:1–27:27 (cit. on p. 108).

[29] Chandra R. Chegireddy and Horst W. Hamacher. "Algorithms for Finding K-best Perfect Matchings". In: *Discrete Applied Mathematics* 18.2 (1987), pp. 155–165. DOI: 10.1016/0166-218X(87)90017-5 (cit. on pp. 47, 48).

[30] Ian Yen-Hung Chen, Bruce A. MacDonald, and Burkhard Wünsche. "Mixed reality simulation for mobile robots". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2009, pp. 232–237. DOI: 10.1109/ROBOT.2009.5152325 (cit. on p. 28).

[31] Yu Fan Chen, Mark Cutler, and Jonathan P. How. "Decoupled multiagent path planning via incremental sequential convex programming". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 5954–5961. DOI: 10.1109/ICRA.2015.7140034 (cit. on p. 8).

[32] Timothy H. Chung, Michael R. Clement, Michael A. Day, Kevin D. Jones, Duane Davis, and Marianna Jones. "Live-fly, large-scale field experimentation for large numbers of fixed-wing UAVs". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1255–1262. DOI: 10.1109/ICRA.2016.7487257 (cit. on p. 17).

[33] Marcello Cirillo, Tansel Uras, and Sven Koenig. "A lattice-based approach to multi-robot motion planning for non-holonomic vehicles". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2014, pp. 232–239. DOI: 10.1109/IROS.2014.6942566 (cit. on p. 9).

[34] Liron Cohen, Tansel Uras, and Sven Koenig. "Feasibility Study: Using Highways for Bounded-Suboptimal Multi-Agent Path Finding". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2015, pp. 2–8. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/11301 (cit. on p. 44).

[35] Liron Cohen, Tansel Uras, T. K. Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. "Improved Solvers for Bounded-Suboptimal Multi-Agent Path Finding". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI/AAAI Press, 2016, pp. 3067–3074. URL: http://www.ijcai.org/Abstract/16/435 (cit. on p. 44).

[36] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine Learning* 20.3 (1995), pp. 273–297. DOI: 10.1007/BF00994018 (cit. on p. 130).

[37] Raffaello D'Andrea. *Meet the dazzling flying machines of the future*. TED Talk. 2016. URL: https://www.ted.com/talks/raffaello_d_andrea_meet_the_dazzling_flying_machines_of_the_future (cit. on p. 17).

[38] Mark Debord, **Wolfgang Hönig**, and Nora Ayanian. "Trajectory Planning for Heterogeneous Robot Teams". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 7924–7931. DOI: 10.1109/IROS.2018.8593876 (cit. on pp. 3, 99).

[39] Rina Dechter, Itay Meiri, and Judea Pearl. "Temporal Constraint Networks". In: *Artificial Intelligence* 49.1-3 (1991), pp. 61–95. DOI: 10.1016/0004-3702(91)90006-6 (cit. on pp. 73, 77).

[40] Vishnu Desaraju and Jonathan P. How. "Decentralized path planning for multi-agent teams with complex constraints". In: *Autonomous Robots* 32.4 (2012), pp. 385–403. DOI: 10.1007/s10514-012-9275-2 (cit. on p. 9).

[41] Andrew Dobson and Kostas E. Bekris. "Sparse roadmap spanners for asymptotically near-optimal motion planning". In: *International Journal of Robotics Research (IJRR)* 33.1 (2014), pp. 18–47. DOI: 10.1177/0278364913498292 (cit. on pp. 87, 104).

[42] Kurt M. Dresner and Peter Stone. "A Multiagent Approach to Autonomous Intersection Management". In: *Journal of Artificial Intelligence Research (JAIR)* 31 (2008), pp. 591–656. DOI: 10.1613/jair.2502 (cit. on p. 10).

[43] John Enright and Peter R. Wurman. "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems". In: *AAAI Workshop on Automated Action Planning for Autonomous Mobile Robots*. Vol. WS-11-09. AAAI Workshops. AAAI, 2011. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW11/paper/view/3917 (cit. on p. 1).

[44] David Eppstein. "k-Best Enumeration". In: *Encyclopedia of Algorithms*. Springer, 2016, pp. 1003–1006. ISBN: 978-1-4939-2864-4. DOI: 10.1007/978-1-4939-2864-4_733 (cit. on p. 47).

[45] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. "A General Formal Framework for Pathfinding Problems with Multiple Agents". In: *AAAI Conference on Artificial Intelligence*. AAAI Press, 2013. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6293 (cit. on p. 6).

[46] Michael A. Erdmann and Tomás Lozano-Pérez. "On Multiple Moving Objects". In: *Algorithmica* 2 (1987), pp. 477–521. DOI: 10.1007/BF01840371 (cit. on p. 8).

[47] Matthias Faessler, Flavio Fontana, Christian Forster, Elias Mueggler, Matia Pizzoli, and Davide Scaramuzza. "Autonomous, Vision-based Flight and Live Dense 3D Mapping with a Quadrotor Micro Aerial Vehicle". In: *Journal of Field Robotics (JFR)* 33.4 (2016), pp. 431–450. DOI: 10.1002/rob.21581 (cit. on p. 18).

[48] Rida T. Farouki. "The Bernstein polynomial basis: A centennial retrospective". In: *Computer Aided Geometric Design* 29.6 (2012), pp. 379–419. DOI: 10.1016/j.cagd.2012.03.001 (cit. on p. 127).

[49] Ariel Felner, Roni Stern, Solomon Eyal Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan R. Sturtevant, Glenn Wagner, and Pavel Surynek. "Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2017, pp. 29–37. URL: https://aaai.org/ocs/index.php/SOCS/SOCS17/paper/view/15781 (cit. on pp. 6, 7).

[50] Andrew W. Feng, Yuyu Xu, and Ari Shapiro. "An example-based motion synthesis technique for locomotion and object manipulation". In: *Symposium on Interactive 3D Graphics and Games (I3D)*. ACM, 2012, pp. 95–102. DOI: 10.1145/2159616.2159632 (cit. on p. 32).

[51] Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. "qpOASES: a parametric active-set algorithm for quadratic programming". In: *Mathematical Programming Computation* 6.4 (2014), pp. 327–363. DOI: 10.1007/s12532-014-0071-1 (cit. on p. 131).

[52] Michel Fliess, Jean Lévine, Philippe Martin, and Pierre Rouchon. "Flatness and defect of non-linear systems: introductory theory and examples". In: *International Journal of Control* 61.6 (1995), pp. 1327–1361. DOI: 10.1080/00207179508921959 (cit. on p. 108).

[53] Julian Förster. "System Identification of the Crazyflie 2.0 Nano Quadrocopter". BA Thesis. ETH Zurich, 2015. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/214143 (cit. on p. 17).

[54] Eckhard Freund and Jürgen Rossmann. "Projective virtual reality: bridging the gap between virtual reality and robotics". In: *IEEE Transactions on Robotics and Automation* 15.3 (1999), pp. 411–422. DOI: 10.1109/70.768175 (cit. on p. 28).

[55] Abraham Prieto Garcia, Gervasio Varela Fernandez, Blanca Maria Priego Torres, and Fernando López-Peña. "Mixed reality educational environment for robotics". In: *IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems (VECIMS)*. 2011, pp. 119–124. DOI: 10.1109/VECIMS.2011.6053845 (cit. on p. 28).

[56] Fabrizio Ghiringhelli, Jerome Guzzi, Gianni A. Di Caro, Vincenzo Caglioti, Luca Maria Gambardella, and Alessandro Giusti. "Interactive Augmented Reality for understanding and analyzing multi-robot systems". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2014, pp. 1195–1201. DOI: 10.1109/IROS.2014.6942709 (cit. on p. 28).

[57] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan R. Sturtevant, Robert C. Holte, and Jonathan Schaeffer. "Enhanced Partial Expansion A*". In: *Journal of Artificial Intelligence Research (JAIR)* 50 (2014), pp. 141–187. DOI: 10.1613/jair.4171 (cit. on p. 6).

[58] Oded Goldreich. "Finding the Shortest Move-Sequence in the Graph-Generalized 15-Puzzle Is NP-Hard". In: *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*. Vol. 6650. Lecture Notes in Computer Science. This work was completed in July 1984, and later appeared as Technical Report No. 792 of the Computer Science Department of the Technion (Israel). Springer, 2011, pp. 1–5. DOI: 10.1007/978-3-642-22670-0_1 (cit. on p. 5).

[59] Gilad Goraly and Refael Hassin. "Multi-Color Pebble Motion on Graphs". In: *Algorithmica* 58.3 (2010), pp. 610–636. DOI: 10.1007/s00453-009-9290-7 (cit. on pp. 5, 6).

[60] Jean Gregoire, Michal Cáp, and Emilio Frazzoli. "Locally-optimal multi-robot navigation under delaying disturbances using homotopy constraints". In: *Autonomous Robots* 42.4 (2018), pp. 895–907. DOI: 10.1007/s10514-017-9673-6 (cit. on pp. 9, 114, 122).

[61] Steve R. Gunn. *Support Vector Machines for Classification and Regression*. Tech. rep. University of Southhampton, 1998. URL: http://users.ecs.soton.ac.uk/srg/publications/pdf/SVM.pdf (cit. on p. 89).

[62] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: http://www.gurobi.com (cit. on p. 54).

[63] Stephen J. Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming C. Lin, Dinesh Manocha, and Pradeep Dubey. "ClearPath: highly parallel collision avoidance for multi-agent simulation". In: *ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA)*. ACM, 2009, pp. 177–187. DOI: 10.1145/1599470.1599494 (cit. on pp. 9, 10).

[64] Michel Habib, Michel Morvan, and Jean-Xavier Rampon. "On the calculation of transitive reduction — closure of orders". In: *Discrete Mathematics* 111.1-3 (1993), pp. 289–303. DOI: 10.1016/0012-365X(93)90164-O (cit. on pp. 78, 80).

[65] Arno Hartholt, David R. Traum, Stacy C. Marsella, Ari Shapiro, Giota Stratou, Anton Leuski, Louis-Philippe Morency, and Jonathan Gratch. "All Together Now - Introducing the Virtual Human Toolkit". In: *Intelligent Virtual Agents (IVA)*. Vol. 8108. Lecture Notes in Computer Science. Springer, 2013, pp. 368–381. DOI: 10.1007/978-3-642-40415-3_33 (cit. on p. 32).

[66] Sabine Hauert, Severin Leven, Maja Varga, Fabio Ruini, Angelo Cangelosi, Jean-Christophe Zufferey, and Dario Floreano. "Reynolds flocking in reality with fixed-wing robots: Communication range vs. maximum turning rate". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2011, pp. 5015–5020. DOI: 10.1109/IROS.2011.6095129 (cit. on p. 17).

[67] Karol Hausman, Jörg Müller, Abishek Hariharan, Nora Ayanian, and Gaurav S. Sukhatme. "Cooperative multi-robot control for target tracking with onboard sensing". In: *International Journal of Robotics Research (IJRR)* 34.13 (2015), pp. 1660–1677. DOI: 10.1177/0278364915602321 (cit. on p. 34).

[68] Karol Hausman, James A. Preiss, Gaurav S. Sukhatme, and Stephan Weiss. "Observability-Aware Trajectory Optimization for Self-Calibration With Application to UAVs". In: *IEEE Robotics and Automation Letters (RA-L)* 2.3 (2017), pp. 1770–1777. DOI: 10.1109/LRA.2017.2647799 (cit. on p. 20).

[69] Robert A. Hearn and Erik D. Demaine. "PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation". In: *Theoretical Computer Science* 343.1-2 (2005), pp. 72–96. DOI: 10.1016/j.tcs.2005.05.008 (cit. on p. 7).

[70] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. "On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE- Hardness of the "Warehouseman's Problem"". In: *International Journal of Robotics Research (IJRR)* 3.4 (1984), pp. 76–88. DOI: 10.1177/027836498400300405 (cit. on p. 7).

[71] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. "OctoMap: an efficient probabilistic 3D mapping framework based on octrees". In: *Autonomous Robots* 34.3 (2013). Software available at http://octomap.github.com, pp. 189–206. DOI: 10.1007/s10514-012-9321-0 (cit. on pp. 93, 108, 131).

[72] Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. "VRPN: a device-independent, network-transparent VR peripheral system". In: *Symposium on Virtual Reality Software and Technology (VRST)*. ACM, 2001, pp. 55–61. DOI: 10.1145/505008.505019 (cit. on p. 32).

[73] Intel. *2018 CES: Intel Sets Guinness World Records Title, Flies 110 Intel Shooting Star Mini Drones Indoors*. https://newsroom.intel.com/news/intel-sets-guinness-world-records-title-flies-100-intel-shooting-star-mini-drones-indoors/. 2018 (cit. on p. 17).

[74] Intel and Ars Electronica Futurelab. *Drone 100*. world record for the most UAVs airborne simultaneously, http://www.spaxels.at/show_drone100.html. 2015 (cit. on pp. 10, 17).

[75] iRobot Corporation. *iRobot Create2 Open Interface (OI) Specification based on the iRobot Roomba600*. http://www.irobotweb.com/~/media/MainSite/PDFs/About/STEM/Create/iRobot_Roomba_600_Open_Interface_Spec.pdf?la=en. Website accessed: Nov. 2016 (cit. on p. 12).

[76] Kenneth I. Joy. "Bernstein polynomials". In: *On-Line Geometric Modeling Notes* (2000). URL: http://www.idav.ucdavis.edu/education/CAGDNotes/Bernstein-Polynomials.pdf (cit. on p. 90).

[77] Rahul Kala. "Multi-robot path planning using co-evolutionary genetic programming". In: *Expert Systems with Applications* 39.3 (2012), pp. 3817–3831. DOI: 10.1016/j.eswa.2011.09.090 (cit. on p. 8).

[78] Mina Kamel, Javier Alonso-Mora, Roland Siegwart, and Juan I. Nieto. "Robust collision avoidance for multiple micro aerial vehicles using nonlinear model predictive control". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 236–243. DOI: 10.1109/IROS.2017.8202163 (cit. on p. 9).

[79] Y. Kanayama, Y. Kimura, F. Miyazaki, and T. Noguchi. "A stable tracking control method for an autonomous mobile robot". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 1990, pp. 384–389. DOI: 10.1109/ROBOT.1990.126006 (cit. on pp. 14, 109).

[80] Kamal Kant and Steven W. Zucker. "Toward Efficient Trajectory Planning: The Path-Velocity Decomposition". In: *International Journal of Robotics Research (IJRR)* 5.3 (1986), pp. 72–89. DOI: 10.1177/027836498600500304 (cit. on p. 8).

[81] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *International Journal of Robotics Research (IJRR)* 30.7 (2011), pp. 846–894. DOI: 10.1177/0278364911406761 (cit. on pp. 9, 87, 137).

[82] Lydia E. Kavraki and Steven M. LaValle. "Motion Planning". In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Springer, 2016, pp. 139–162. DOI: 10.1007/978-3-319-32552-1_7 (cit. on p. 136).

[83] Mokhtar M. Khorshid, Robert C. Holte, and Nathan R. Sturtevant. "A Polynomial-Time Algorithm for Non-Optimal Multi-Agent Pathfinding". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2011. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/view/4039 (cit. on p. 7).

[84] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. "RoboCup Rescue: search and rescue in large-scale disasters as a domain for autonomous agents research". In: *IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 6. 1999, 739–743 vol.6. DOI: 10.1109/ICSMC.1999.816643 (cit. on p. 10).

[85] Nathan P. Koenig and Andrew Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2004, pp. 2149–2154. DOI: 10.1109/IROS.2004.1389727 (cit. on p. 35).

[86] Daniel Kornhauser, Gary L. Miller, and Paul G. Spirakis. "Coordinating Pebble Motion on Graphs, the Diameter of Permutation Groups, and Applications". In: *Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1984, pp. 241–250. DOI: 10.1109/SFCS.1984.715921 (cit. on pp. 5, 43).

[87] Harold W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97. ISSN: 1931-9193. DOI: 10.1002/nav.3800020109 (cit. on pp. 48, 61).

[88] Aleksandr Kushleyev, Daniel Mellinger, Caitlin Powers, and Vijay Kumar. "Towards a swarm of agile micro quadrotors". In: *Autonomous Robots* 35.4 (2013), pp. 287–300. DOI: 10.1007/s10514-013-9349-9 (cit. on p. 16).

[89] Benoit Landry. "Planning and Control for Quadrotor Flight through Cluttered Environments". MA thesis. Massachusetts Institute of Technology, 2015. URL: http://groups.csail.mit.edu/robotics-center/public_papers/Landry15.pdf (cit. on p. 17).

[90] Steven M. LaValle. *Planning algorithms*. Cambridge University Press, 2006. ISBN: 978-0-521-86205-9. URL: http://planning.cs.uiuc.edu/ (cit. on pp. 8, 10, 87, 98, 137).

[91] Anton Ledergerber, Michael Hamer, and Raffaello D'Andrea. "A robot self-localization system using one-way ultra-wideband communication". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 3131–3137. DOI: 10.1109/IROS.2015.7353810 (cit. on p. 18).

[92] Florian Leutert, Christian Herrmann, and Klaus Schilling. "A spatial augmented reality system for intuitive display of robotic data". In: *ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. Ed. by Hideaki Kuzuoka, Vanessa Evers, Michita Imai, and Jodi Forlizzi. 2013, pp. 179–180. URL: http://dl.acm.org/citation.cfm?id=2447626 (cit. on p. 28).

[93] Ryan Luna and Kostas E. Bekris. "Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI/AAAI, 2011, pp. 294–300. DOI: 10.5591/978-1-57735-516-8/IJCAI11-059 (cit. on p. 7).

[94] Sergei Lupashin, Markus Hehn, Mark W Mueller, Angela P Schoellig, Michael Sherback, and Raffaello D'Andrea. "A platform for aerial robotics research and demonstration: The Flying Machine Arena". In: *Mechatronics* 24.1 (2014), pp. 41–54. DOI: https://doi.org/10.1016/j.mechatronics.2013.11.006 (cit. on pp. 16, 18, 20, 23).

[95] Hang Ma and Sven Koenig. "Optimal Target Assignment and Path Finding for Teams of Agents". In: *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. ACM, 2016, pp. 1144–1152. URL: http://dl.acm.org/citation.cfm?id=2937092 (cit. on pp. 7, 44, 45, 52, 56, 59).

[96] Hang Ma, T. K. Satish Kumar, and Sven Koenig. "Multi-Agent Path Finding with Delay Probabilities". In: *AAAI Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 3605–3612. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14984 (cit. on pp. 113, 114).

[97] Hang Ma, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. "Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks". In: *Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. ACM, 2017, pp. 837–845. URL: http://dl.acm.org/citation.cfm?id=3091243 (cit. on pp. 113, 114, 119).

[98] Hang Ma, Craig A. Tovey, Guni Sharon, T. K. Satish Kumar, and Sven Koenig. "Multi-Agent Path Finding with Payload Transfers and the Package-Exchange Robot-Routing Problem". In: *AAAI Conference on Artificial Intelligence*. AAAI Press, 2016, pp. 3166–3173. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12437 (cit. on pp. 5, 6, 43, 54).

[99] Masoumeh Mansouri, Fabien Lagriffoul, and Federico Pecora. "Multi vehicle routing with nonholonomic constraints and dense dynamic obstacles". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 3522–3529. DOI: 10.1109/IROS.2017.8206195 (cit. on p. 10).

[100] Ellips Masehian and Azadeh Hassan Nejad. "Solvability of multi robot motion planning problems on Trees". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2009, pp. 5936–5941. DOI: 10.1109/IROS.2009.5354148 (cit. on pp. 5, 6).

[101] Ellips Masehian and Davoud Sedighizadeh. "An Improved Particle Swarm Optimization Method for Motion Planning of Multiple Robots". In: *International Symposium on Distributed Autonomous Robotic Systems (DARS)*. Vol. 83. Springer Tracts in Advanced Robotics. Springer, 2010, pp. 175–188. DOI: 10.1007/978-3-642-32723-0_13 (cit. on p. 8).

[102] Jacob Mattingley and Stephen Boyd. "CVXGEN: a code generator for embedded convex optimization". In: *Optimization and Engineering* 13.1 (2012). Software available at https://cvxgen.com, pp. 1–27. ISSN: 1573-2924. DOI: 10.1007/s11081-011-9176-9 (cit. on pp. 94, 131).

[103] Daniel Mellinger and Vijay Kumar. "Minimum snap trajectory generation and control for quadrotors". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2011, pp. 2520–2525. DOI: 10.1109/ICRA.2011.5980409 (cit. on pp. 20, 22, 85, 92, 108).

[104] Daniel Mellinger, Aleksandr Kushleyev, and Vijay Kumar. "Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2012, pp. 477–483. DOI: 10.1109/ICRA.2012.6225009 (cit. on pp. 8, 99, 110).

[105] Nathan Michael, Daniel Mellinger, Quentin Lindsey, and Vijay Kumar. "The GRASP Multiple Micro-UAV Testbed". In: *IEEE Robotics & Automation Magazine* 17.3 (2010), pp. 56–65. DOI: 10.1109/MRA.2010.937855 (cit. on pp. 16, 18, 20, 85).

[106] P. Milgram and F. Kishino. "A Taxonomy of Mixed Reality Visual Displays". In: *IEICE Transactions on Information and Systems* E77-D.12 (1994), pp. 1321–1329. URL: https://search.ieice.org/bin/summary.php?id=e77-d_12_1321 (cit. on p. 28).

[107] Javier Minguez, Florent Lamiraux, and Jean-Paul Laumond. "Motion Planning and Obstacle Avoidance". In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Springer, 2016, pp. 1177–1202. DOI: 10.1007/978-3-319-32552-1_47 (cit. on p. 108).

[108] Robert Morris, Corina S. Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. "Planning, Scheduling and Monitoring for Airport Surface Operations". In: *AAAI Workshop on Planning for Hybrid Systems*. Vol. WS-16-12. AAAI Workshops. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12611 (cit. on p. 10).

[109] Jörg Müller and Gaurav S. Sukhatme. "Risk-aware trajectory generation with application to safe quadrotor landing". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2014, pp. 3642–3648. DOI: 10.1109/IROS.2014.6943073 (cit. on p. 20).

[110] Katta G. Murty. "Letter to the Editor – An Algorithm for Ranking all the Assignments in Order of Increasing Cost". In: *Operations Research* 16.3 (1968), pp. 682–687. DOI: 10.1287/opre.16.3.682 (cit. on pp. 47, 48).

[111] Helen Oleynikova, Michael Burri, Zachary Taylor, Juan I. Nieto, Roland Siegwart, and Enric Galceran. "Continuous-time trajectory optimization for online UAV replanning". In: *IEEE/RSJ International Conference on Intelligent Robots (IROS)*. 2016, pp. 5332–5339. DOI: 10.1109/IROS.2016.7759784 (cit. on pp. 125, 131).

[112] Jia Pan, Sachin Chitta, and Dinesh Manocha. "FCL: A general purpose library for collision and proximity queries". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Software available at https://github.com/flexible-collision-library/fcl. IEEE, 2012, pp. 3859–3866. DOI: 10.1109/ICRA.2012.6225337 (cit. on pp. 87, 93).

[113] Mike Peasgood, Christopher M. Clark, and John McPhee. "A Complete and Scalable Strategy for Coordinating Multiple Robots Within Roadmaps". In: *IEEE Transactions on Robotics (T-RO)* 24.2 (2008), pp. 283–292. DOI: 10.1109/TRO.2008.918056 (cit. on p. 7).

[114] Federico Pecora, Henrik Andreasson, Masoumeh Mansouri, and Vilian Petkov. "A Loosely-Coupled Approach for Multi-Robot Coordination, Motion Planning and Control". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 2018, pp. 485–493. URL: https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17746 (cit. on p. 9).

[115] Jufeng Peng and Srinivas Akella. "Coordinating Multiple Robots with Kinodynamic Constraints Along Specified Paths". In: *International Journal of Robotics Research (IJRR)* 24.4 (2005), pp. 295–310. DOI: 10.1177/0278364905051974 (cit. on p. 8).

[116] Thai Phan, **Wolfgang Hönig**, and Nora Ayanian. "Mixed Reality Collaboration Between Human-Agent Teams". In: *IEEE Conference on Virtual Reality and 3D User Interfaces, (VR)*. 2018, pp. 659–660. DOI: 10.1109/VR.2018.8446542 (cit. on p. 27).

[117] Mike Phillips and Maxim Likhachev. "SIPP: Safe interval path planning for dynamic environments". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2011, pp. 5628–5635. DOI: 10.1109/ICRA.2011.5980306 (cit. on p. 117).

[118] Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron D. Ames, Eric Feron, and Magnus Egerstedt. "The Robotarium: A remotely accessible swarm robotics research testbed". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1699–1706. DOI: 10.1109/ICRA.2017.7989200 (cit. on p. 17).

[119] James A. Preiss*, **Wolfgang Hönig***, Gaurav S. Sukhatme, and Nora Ayanian. "Crazyswarm: A large nano-quadcopter swarm". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Star (*) refers to equal contribution. 2017, pp. 3299–3304. DOI: 10.1109/ICRA.2017.7989376 (cit. on pp. 3, 16).

[120] Daniel Ratner and Manfred K. Warmuth. "Finding a Shortest Solution for the N × N Extension of the 15-PUZZLE Is Intractable". In: *National Conference on Artificial Intelligence*. Morgan Kaufmann, 1986, pp. 168–172. URL: http://www.aaai.org/Library/AAAI/1986/aaai86-027.php (cit. on p. 5).

[121] Charles Richter, Adam Bry, and Nicholas Roy. "Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments". In: *International Symposium on Robotics Research (ISRR)*. Vol. 114. Springer Tracts in Advanced Robotics. Springer, 2013, pp. 649–666. DOI: 10.1007/978-3-319-28872-7_37 (cit. on pp. 90, 92, 130).

[122] D. Reed Robinson, Robert T. Mar, Katia Estabridis, and Gary A. Hewer. "An Efficient Algorithm for Optimal Trajectory Generation for Heterogeneous Multi-Agent Systems in Non-Convex Environments". In: *IEEE Robotics and Automation Letters (RA-L)* 3.2 (2018), pp. 1215–1222. DOI: 10.1109/LRA.2018.2794582 (cit. on p. 100).

[123] Gabriele Röger and Malte Helmert. "Non-Optimal Multi-Agent Pathfinding is Solved (Since 1984)". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5407 (cit. on pp. 5, 43).

[124] Eric Rohmer, Surya P. N. Singh, and Marc Freese. "V-REP: A versatile and scalable robot simulation framework". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2013, pp. 1321–1326. DOI: 10.1109/IROS.2013.6696520 (cit. on p. 33).

[125] Lorenzo Sabattini, Valerio Digani, Cristian Secchi, and Cesare Fantuzzi. "Optimized simultaneous conflict-free task assignment and path planning for multi-AGV systems". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 1083–1088. DOI: 10.1109/IROS.2017.8202278 (cit. on p. 8).

[126] Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. "Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5385 (cit. on p. 7).

[127] Claude Samson, Pascal Morin, and Roland Lenain. "Modeling and Control of Wheeled Mobile Robots". In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Springer, 2016, pp. 1235–1266. DOI: 10.1007/978-3-319-32552-1_49 (cit. on p. 14).

[128] T. Schouwenaars, B. De Moor, E. Feron, and J. How. "Mixed integer programming for multi-vehicle path planning". In: *European Control Conference (ECC)*. 2001, pp. 2603–2608. DOI: 10.23919/ECC.2001.7076321 (cit. on p. 8).

[129] Baskın Şenbaşlar, **Wolfgang Hönig**, and Nora Ayanian. "Robust Trajectory Execution for Multi-robot Teams Using Distributed Real-time Replanning". In: *International Symposium on Distributed Autonomous Robotic Systems (DARS)*. Vol. 9. Springer Proceedings in Advanced Robotics. Springer, 2018, pp. 167–181. DOI: 10.1007/978-3-030-05816-6_12 (cit. on pp. 3, 124).

[130] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. "Conflict-Based Search For Optimal Multi-Agent Path Finding". In: *AAAI Conference on Artificial Intelligence*. AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5062 (cit. on p. 47).

[131] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66. DOI: 10.1016/j.artint.2014.11.006 (cit. on pp. 7, 43, 45, 47, 49, 116).

[132] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. "The increasing cost tree search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 195 (2013), pp. 470–495. DOI: 10.1016/j.artint.2012.11.006 (cit. on p. 7).

[133] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library - User Guide and Reference Manual*. C++ in-depth series. Software available at http://www.boost.org. Pearson / Prentice Hall, 2002. ISBN: 978-0-201-72914-6 (cit. on p. 78).

[134] David Silver. "Cooperative Pathfinding". In: *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. AAAI Press, 2005, pp. 117–122. URL: http://www.aaai.org/Papers/AIIDE/2005/AIIDE05-020.pdf (cit. on pp. 6, 10).

[135] Kiril Solovey and Dan Halperin. "k-color multi-robot motion planning". In: *International Journal of Robotics Research (IJRR)* 33.1 (2014), pp. 82–97. DOI: 10.1177/0278364913506268 (cit. on p. 9).

[136] Kiril Solovey and Dan Halperin. "On the hardness of unlabeled multi-robot motion planning". In: *International Journal of Robotics Research (IJRR)* 35.14 (2016), pp. 1750–1759. DOI: 10.1177/0278364916672311 (cit. on p. 7).

[137] Kiril Solovey, Oren Salzman, and Dan Halperin. "Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning". In: *International Journal of Robotics Research (IJRR)* 35.5 (2016), pp. 501–513. DOI: 10.1177/0278364915615688 (cit. on p. 9).

[138] Ryan Spicer, Edgar Evangelista, Raymond New, Julia Campbell, Todd Richmond, Christopher McGroarty, and Brian Vogt. "Innovation and Rapid Evolutionary Design by Virtual Doing: Understanding Early Synthetic Prototyping". In: *Proceeding of Simulation Interoperability Workshop*. 2015. URL: http://ict.usc.edu/pubs/Innovation%20and%20Rapid%20Evolutionary%20Design%20by%20Virtual%20Doing-Understanding%20Early%20Synthetic.pdf (cit. on p. 36).

[139] Trevor Scott Standley and Richard E. Korf. "Complete Algorithms for Cooperative Pathfinding Problems". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI/AAAI, 2011, pp. 668–673. DOI: 10.5591/978-1-57735-516-8/IJCAI11-118 (cit. on p. 6).

[140] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. "OSQP: An Operator Splitting Solver for Quadratic Programs". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1711.08013 [math.OC] (cit. on p. 131).

[141] Nathan R. Sturtevant and Michael Buro. "Improving Collaborative Pathfinding Using Map Abstraction". In: *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. The AAAI Press, 2006, pp. 80–85. URL: http://www.aaai.org/Library/AIIDE/2006/aiide06-017.php (cit. on p. 6).

[142] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. "The Open Motion Planning Library". In: *IEEE Robotics & Automation Magazine* 19.4 (2012). Software available at http://ompl.kavrakilab.org, pp. 72–82. DOI: 10.1109/MRA.2012.2205651 (cit. on pp. 93, 108).

[143] Pavel Surynek. "A novel approach to path planning for multiple robots in bi-connected graphs". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2009, pp. 3613–3619. DOI: 10.1109/ROBOT.2009.5152326 (cit. on p. 7).

[144] Pavel Surynek. "An Optimization Variant of Multi-Robot Path Planning Is Intractable". In: *AAAI Conference on Artificial Intelligence*. AAAI Press, 2010. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1768 (cit. on pp. 5, 43).

[145] Pavel Surynek. "Reduced Time-Expansion Graphs and Goal Decomposition for Solving Cooperative Path Finding Sub-Optimally". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 2015, pp. 1916–1922. URL: http://ijcai.org/Abstract/15/272 (cit. on pp. 5, 6).

[146] Pavel Surynek. "Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving". In: *Pacific Rim International Conference on Artificial Intelligence (PRICAI)*. Vol. 7458. Lecture Notes in Computer Science. Springer, 2012, pp. 564–576. DOI: 10.1007/978-3-642-32695-0_50 (cit. on p. 6).

[147] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. "Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective". In: *European Conference on Artificial Intelligence (ECAI)*. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, 2016, pp. 810–818. DOI: 10.3233/978-1-61499-672-9-810 (cit. on p. 6).

[148] Sarah Tang and Vijay Kumar. "Safe and complete trajectory generation for robot teams with higher-order dynamics". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 1894–1901. DOI: 10.1109/IROS.2016.7759300 (cit. on pp. 91, 95).

[149] Dave Thomas and Andy Hunt. "Mock Objects". In: *IEEE Software* 19.3 (2002), pp. 22–24. DOI: 10.1109/MS.2002.1003449 (cit. on p. 36).

[150] C. Tomlin, G. J. Pappas, and S. Sastry. "Conflict resolution for air traffic management: a study in multiagent hybrid systems". In: *IEEE Transactions on Automatic Control* 43.4 (1998), pp. 509–521. ISSN: 0018-9286. DOI: 10.1109/9.664154 (cit. on p. 10).

[151] Nikolas Trawny and Stergios I. Roumeliotis. *Indirect Kalman Filter for 3D Attitude Estimation*. Tech. rep. 2005-002. University of Minnesota, Department of Computer Science & Engineering, 2005. URL: http://mars.cs.umn.edu/tr/reports/Trawny05b.pdf (cit. on p. 19).

[152] Matthew Turpin, Nathan Michael, and Vijay Kumar. "CAPT: Concurrent assignment and planning of trajectories for multiple robots". In: *International Journal of Robotics Research (IJRR)* 33.1 (2014), pp. 98–112. DOI: 10.1177/0278364913515307 (cit. on pp. 8, 9).

[153] Vladyslav C. Usenko, Lukas von Stumberg, Andrej Pangercic, and Daniel Cremers. "Real-time trajectory replanning for MAVs using uniform B-splines and a 3D circular buffer". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 215–222. DOI: 10.1109/IROS.2017.8202160 (cit. on pp. 125, 131).

[154] Gábor Vásárhelyi, Csaba Virágh, Gergo Somorjai, Norbert Tarcai, Tamás Szörényi, Tamás Nepusz, and Tamás Vicsek. "Outdoor flocking and formation flight with autonomous aerial robots". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2014, pp. 3866–3873. DOI: 10.1109/IROS.2014.6943105 (cit. on p. 17).

[155] Prasanna Velagapudi, Katia P. Sycara, and Paul Scerri. "Decentralized prioritized planning in large multirobot teams". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2010, pp. 4603–4609. DOI: 10.1109/IROS.2010.5649438 (cit. on p. 8).

[156] Glenn Wagner. "Subdimensional Expansion: A Framework for Computationally Tractable Multirobot Path Planning". PhD thesis. Carnegie Mellon University, 2015. URL: http://ri.cmu.edu/pub_files/2015/12/g_wagner_robotics_2015.pdf (cit. on p. 7).

[157] Glenn Wagner and Howie Choset. "M*: A complete multirobot path planning algorithm with performance bounds". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2011, pp. 3260–3267. DOI: 10.1109/IROS.2011.6095022 (cit. on pp. 7, 43).

[158] Glenn Wagner, Howie Choset, and Nora Ayanian. "Subdimensional Expansion and Optimal Task Reassignment". In: *Symposium on Combinatorial Search (SOCS)*. AAAI Press, 2012. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5390 (cit. on pp. 45, 46).

[159] Ko-Hsin Cindy Wang and Adi Botea. "Fast and Memory-Efficient Multi-Agent Pathfinding". In: *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2008, pp. 380–387. URL: http://www.aaai.org/Library/ICAPS/2008/icaps08-047.php (cit. on pp. 7, 10).

[160] Ko-Hsin Cindy Wang and Adi Botea. "MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees". In: *Journal of Artificial Intelligence Research (JAIR)* 42 (2011), pp. 55–90. DOI: 10.1613/jair.3370 (cit. on p. 7).

[161] Li Wang, Aaron D. Ames, and Magnus Egerstedt. "Safety Barrier Certificates for Collisions-Free Multirobot Systems". In: *IEEE Transactions on Robotics (T-RO)* 33.3 (2017), pp. 661–674. DOI: 10.1109/TRO.2017.2659727 (cit. on pp. 114, 125).

[162] C. W. Warren. "Multiple robot path coordination using artificial potential fields". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 1990, 500–505 vol.1. DOI: 10.1109/ROBOT.1990.126028 (cit. on p. 9).

[163] Stephan Weiss and Roland Siegwart. "Real-time metric state estimation for modular vision-inertial systems". In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2011, pp. 4531–4537. DOI: 10.1109/ICRA.2011.5979982 (cit. on p. 19).

[164] Adam Wiktor, Dexter Scobee, Sean Messenger, and Christopher Clark. "Decentralized and complete multi-robot motion planning in confined spaces". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2014, pp. 1168–1175. DOI: 10.1109/IROS.2014.6942705 (cit. on p. 7).

[165] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. "Push and Rotate: a Complete Multi-agent Pathfinding Algorithm". In: *Journal of Artificial Intelligence Research (JAIR)* 51 (2014), pp. 443–492. DOI: 10.1613/jair.4447 (cit. on p. 7).

[166] **Wolfgang Hönig**, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Conflict-Based Search with Optimal Task Assignment". In: *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. Software available at https://github.com/whoenig/libMultiRobotPlanning. 2018, pp. 757–765. URL: http://dl.acm.org/citation.cfm?id=3237495 (cit. on pp. 3, 42).

[167] **Wolfgang Hönig**, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Persistent and Robust Execution of MAPF Schedules in Warehouses". In: *IEEE Robotics and Automation Letters (RA-L)*. Vol. 4. 2. 2019, pp. 1125–1131. DOI: 10.1109/LRA.2019.2894217 (cit. on pp. 3, 113).

[168] **Wolfgang Hönig**, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. "Multi-Agent Path Finding with Kinematic Constraints". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. **Awarded Best Paper in robotics track.** AAAI Press, 2016, pp. 477–485. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13183 (cit. on p. 117).

[169] **Wolfgang Hönig**, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. "Path Finding for Multi-Robot Systems with Kinematic Constraints". In: *Journal of Artificial Intelligence Research (JAIR)* (2019). Accepted. In Revision. (cit. on pp. 3, 63).

[170] **Wolfgang Hönig**, T. K. Satish Kumar, Hang Ma, Sven Koenig, and Nora Ayanian. "Formation change for robot groups in occluded environments". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 4836–4842. DOI: 10.1109/IROS.2016.7759710 (cit. on pp. 10, 83, 85).

[171] **Wolfgang Hönig**, Christina Milanes, Lisa Scaria, Thai Phan, Mark T. Bolas, and Nora Ayanian. "Mixed reality for robotics". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 5382–5387. DOI: 10.1109/IROS.2015.7354138 (cit. on pp. 3, 27).

[172] **Wolfgang Hönig**, James A. Preiss, T. K. Satish Kumar, Gaurav S. Sukhatme, and Nora Ayanian. "Trajectory Planning for Quadrotor Swarms". In: *IEEE Transactions on Robotics, Special Issue on Aerial Swarm Robotics* 34.4 (2018), pp. 856–869. DOI: 10.1109/TRO.2018.2853613 (cit. on pp. 3, 42, 83, 114, 130).

[173] **Wolfgang Hönig**, Arash Tavakoli, and Nora Ayanian. "Seamless Robot Simulation Integration for Education: A Case Study". In: *Workshop on the Role of Simulation in Robot Programming at SIMPAR 2016, San Francisco, CA, December 2016*. 2016. URL: http://act.usc.edu/publications/Hoenig_SimRP2016.pdf (cit. on pp. 12, 14, 15).

[174] S. David Wu, Eui-Seok Byeon, and Robert H. Storer. "A Graph-Theoretic Decomposition of the Job Shop Scheduling Problem to Achieve Scheduling Robustness". In: *Operations Research* 47.1 (1999), pp. 113–124. DOI: 10.1287/opre.47.1.113 (cit. on p. 114).

[175] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses". In: *AI Magazine* 29.1 (2008), pp. 9–20. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2082 (cit. on pp. 9, 45, 113).

[176] Zhenyu Yang, Klara Nahrstedt, Yi Cui, Bin Yu, Jin Liang, Sang-Hack Jung, and Ruzena Bajcsy. "TEEVE: The Next Generation Architecture for Tele-immersive Environment". In: *IEEE International Symposium on Multimedia (ISM)*. 2005, pp. 112–119. DOI: 10.1109/ISM.2005.113 (cit. on p. 28).

[177] Derrick Yeo, Elena Shrestha, Derek A. Paley, and Ella Atkins. "An Empirical Model of Rotorcraft UAV Downwash Model for Disturbance Localization and Avoidance". In: *AIAA Atmospheric Flight Mechanics Conference*. 2015. DOI: 10.2514/6.2015-1685 (cit. on p. 85).

[178] Jingjin Yu and Steven M. LaValle. "Multi-agent Path Planning and Network Flow". In: *Workshop on the Algorithmic Foundations of Robotics (WAFR)*. Vol. 86. Springer Tracts in Advanced Robotics. Springer, 2012, pp. 157–173. DOI: 10.1007/978-3-642-36279-8_10 (cit. on pp. 6, 7, 44, 58, 59).

[179] Jingjin Yu and Steven M. LaValle. "Optimal Multirobot Path Planning on Graphs: Complete Algorithms and Effective Heuristics". In: *IEEE Transactions on Robotics (T-RO)* 32.5 (2016), pp. 1163–1177. DOI: 10.1109/TRO.2016.2593448 (cit. on pp. 5, 6, 10, 54, 58, 59, 117).

[180] Jingjin Yu and Steven M. LaValle. "Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs". In: *AAAI Conference on Artificial Intelligence*. AAAI Press, 2013. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6111 (cit. on pp. 6, 43, 52, 58).

[181] Jingjin Yu and Daniela Rus. "Pebble Motion on Graphs with Rotations: Efficient Feasibility Tests and Planning Algorithms". In: *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*. Vol. 107. Springer Tracts in Advanced Robotics. Springer, 2014, pp. 729–746. DOI: 10.1007/978-3-319-16595-0_42 (cit. on p. 5).

[182] Dingjiang Zhou, Zijian Wang, Saptarshi Bandyopadhyay, and Mac Schwager. "Fast, On-line Collision Avoidance for Dynamic Vehicles Using Buffered Voronoi Cells". In: *IEEE Robotics and Automation Letters (RA-L)* 2.2 (2017), pp. 1047–1054. DOI: 10.1109/LRA.2017.2656241 (cit. on pp. 9, 114, 124–126, 131).

APPENDIX A

Helper Functions

Algorithm A.1: addType2Edge

Input: TPG $\mathcal{G}_{TPG}$, vertex v_s, distance δ_s, vertex v_e, distance δ_e; The distance might be negative to indicate a location before the respective vertex and positive to indicate a location after the vertex.

Result: Updated $\mathcal{G}_{TPG}$, with possibly two added vertices (one δ_s from v_s and one δ_e from v_e) and an edge connecting those vertices.

```
1  v_1 ← getOrCreateVertex(G_TPG, v_s, δ_s)
2  v_2 ← getOrCreateVertex(G_TPG, v_e, δ_e)
3  Add Type 2 edge e = (v_1, v_2) to E_TPG
4  constraints(e) ← {v_s → v_e}
```

Algorithm A.2: getOrCreateVertex

Input: Graph $\mathcal{G}_{TPG}$, vertex v_s, distance δ_s.

Result: If the specified vertex exists, $\mathcal{G}_{TPG}$ is unchanged and the vertex is returned. Otherwise, a new vertex is added and returned.

```
1  if δ_s ≥ 0 then
2      d ← 0
3      u ← NULL
4      v ← v_s
5      while d < δ_s do
6          u ← v
7          Find v, where (u, v) is Type 1 edge in G
8          d ← d + dist(u, v)
9      /* Does vertex exists within epsilon boundary? */
10     if |d − δ_s| < ε then
11         return v
12     Create a new vertex (referenced by y) and add it to G
13     Remove edge (u, v) from G
14     Add Type 1 edge (u, y) to G
15     Add Type 1 edge (y, v) to G
16     constraints((u, y)) ← constraints((u, v))
17     constraints((y, v)) ← constraints((u, v))
18     return y
19 else
20     /* As before, but traverse Type 1 edges in the reverse direction. */
```

Algorithm A.3: addConstraint

Input: Graph $\mathcal{G}$, vertex v, distance δ_s, distance δ_e, constraint C.
Result: Updated $\mathcal{G}$, with the constraint added to the requested edge.

```
1 v_1 ← getOrCreateVertex(G, v, δ_s)
2 v_2 ← getOrCreateVertex(G, v, δ_e)
3 u ← v_1
4 for Type 1 edges e = (u, w) do
5 |   constraints(e) ← constraints(e) ∪ {C}
6 |   u = w
7 |   if w = v_2 then
8 |   |   break
```

APPENDIX B

Published and Submitted Papers

Book Chapters and Journal Publications

[J1] **Wolfgang Hönig**, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Persistent and Robust Execution of MAPF Schedules in Warehouses". In: *IEEE Robotics and Automation Letters (RA-L)*. Vol. 4. 2. 2019, pp. 1125–1131. DOI: 10.1109/LRA.2019.2894217.

[J2] **Wolfgang Hönig**, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. "Path Finding for Multi-Robot Systems with Kinematic Constraints". In: *Journal of Artificial Intelligence Research (JAIR)* (2019). Accepted. In Revision.

[J3] **Wolfgang Hönig**, James A. Preiss, T. K. Satish Kumar, Gaurav S. Sukhatme, and Nora Ayanian. "Trajectory Planning for Quadrotor Swarms". In: *IEEE Transactions on Robotics, Special Issue on Aerial Swarm Robotics* 34.4 (2018), pp. 856–869. DOI: 10.1109/TRO.2018.2853613.

[J4] Hang Ma, **Wolfgang Hönig**, Liron Cohen, Tansel Uras, Hong Xu, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. "Overview: A Hierarchical Framework for Plan Generation and Execution in Multirobot Systems". In: *IEEE Intelligent Systems* 32.6 (2017), pp. 6–12. DOI: 10.1109/MIS.2017.4531217.

[J5] **Wolfgang Hönig** and Nora Ayanian. "Flying Multiple UAVs Using ROS". In: *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Ed. by Anis Koubaa. Springer International Publishing, 2017, pp. 83–118. ISBN: 978-3-319-54927-9. DOI: 10.1007/978-3-319-54927-9_3.

Conference Publications

[C1] Mark Debord, **Wolfgang Hönig**, and Nora Ayanian. "Trajectory Planning for Heterogeneous Robot Teams". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 7924–7931. DOI: 10.1109/IROS.2018.8593876.

[C2] Thai Phan, **Wolfgang Hönig**, and Nora Ayanian. "Mixed Reality Collaboration Between Human-Agent Teams". In: *IEEE Conference on Virtual Reality and 3D User Interfaces, (VR)*. 2018, pp. 659–660. DOI: 10.1109/VR.2018.8446542.

[C3] Baskın Şenbaşlar, **Wolfgang Hönig**, and Nora Ayanian. "Robust Trajectory Execution for Multi-robot Teams Using Distributed Real-time Replanning". In: *International Symposium on Distributed Autonomous Robotic Systems (DARS)*. Vol. 9. Springer Proceedings in Advanced Robotics. Springer, 2018, pp. 167–181. DOI: 10.1007/978-3-030-05816-6_12.

[C4] **Wolfgang Hönig**. "Scalable Task and Motion Planning for Multi-Robot Systems in Obstacle-Rich Environments". In: *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 2018, pp. 1746–1751. URL: http://dl.acm.org/citation.cfm?id=3237962.

[C5] **Wolfgang Hönig**, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. "Conflict-Based Search with Optimal Task Assignment". In: *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. Software available at https://github.com/whoenig/libMultiRobotPlanning. 2018, pp. 757–765. URL: http://dl.acm.org/citation.cfm?id=3237495.

[C6] James A. Preiss, **Wolfgang Hönig**, Nora Ayanian, and Gaurav S. Sukhatme. "Downwash-aware trajectory planning for large quadrotor teams". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 250–257. DOI: 10.1109/IROS.2017.8202165.

[C7] James A. Preiss*, **Wolfgang Hönig***, Gaurav S. Sukhatme, and Nora Ayanian. "Crazyswarm: A large nano-quadcopter swarm". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Star (*) refers to equal contribution. 2017, pp. 3299–3304. DOI: 10.1109/ICRA.2017.7989376.

[C8] **Wolfgang Hönig**, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. "Summary: Multi-Agent Path Finding with Kinematic Constraints". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2017, pp. 4869–4873. DOI: 10.24963/ijcai.2017/684.

[C9] James A. Preiss*, **Wolfgang Hönig***, Gaurav S. Sukhatme, and Nora Ayanian. "Crazyswarm: A large nano-quadcopter swarm (Extended Abstract)". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (Late Breaking)*. Star (*) refers to equal contribution. 2016. URL: http://act.usc.edu/publications/Preiss_IROSLateBreaking2016.pdf.

[C10] **Wolfgang Hönig** and Nora Ayanian. "Dynamic multi-target coverage with robotic cameras". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 1871–1878. DOI: 10.1109/IROS.2016.7759297.

[C11] **Wolfgang Hönig**, T. K. Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. "Multi-Agent Path Finding with Kinematic Constraints". In: *International Conference on Automated Planning and Scheduling (ICAPS)*. **Awarded Best Paper in robotics track.** AAAI Press, 2016, pp. 477–485. URL: http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13183.

[C12] **Wolfgang Hönig**, T. K. Satish Kumar, Hang Ma, Sven Koenig, and Nora Ayanian. "Formation change for robot groups in occluded environments". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 4836–4842. DOI: 10.1109/IROS.2016.7759710.

[C13] **Wolfgang Hönig**, Christina Milanes, Lisa Scaria, Thai Phan, Mark T. Bolas, and Nora Ayanian. "Mixed reality for robotics". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 5382–5387. DOI: 10.1109/IROS.2015.7354138.

Workshops and Symposia

[W1] Mark Debord, **Wolfgang Hönig**, and Nora Ayanian. "Trajectory Planning for Heterogeneous Robot Teams". In: *International Symposium on Aerial Robotics (ISAR), Philadelphia, PA, USA, June 2018*. 2018. URL: http://act.usc.edu/publications/Debord_ISAR2018.pdf.

[W2] James A. Preiss, **Wolfgang Hönig**, Gaurav S. Sukhatme, and Nora Ayanian. "Downwash-Aware Trajectory Planning for Large Quadcopter Teams". In: *Southern California Robotics Symposium (SCR), Los Angeles, CA, April 2017*. 2017. URL: http://act.usc.edu/publications/Preiss_SCR2017.pdf.

[W3] James A. Preiss, **Wolfgang Hönig**, Gaurav S. Sukhatme, and Nora Ayanian. "Downwash-Aware Trajectory Planning for Large Quadrotor Swarms". In: *International Symposium on Aerial Robotics, Philadelphia, PA, USA, June 2017*. 2017. URL: http://act.usc.edu/publications/Preiss_ISAR2017.pdf.

[W4] Hang Ma, Sven Koenig, Nora Ayanian, Liron Cohen, **Wolfgang Hönig**, T. K. Satish Kumar, Tansel Uras, Hong Xu, Craig Tovey, and Guni Sharon. "Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios". In: *IJCAI-16 Workshop on Multi-Agent Path Finding (WOMPF), New York City, NY, July 2016*. 2016. URL: http://act.usc.edu/publications/Ma_IJCAIWorkshop2016.pdf.

[W5] **Wolfgang Hönig**, T. K. Satish Kumar, Liron Cohen, Hang Ma, Sven Koeng, and Nora Ayanian. "Path Planning With Kinematic Constraints For Robot Groups". In: *Southern California Robotics Symposium (SCR), San Diego, CA, April 2016*. 2016. URL: http://act.usc.edu/publications/Hoenig_SCR2016.pdf.

[W6] **Wolfgang Hönig**, Arash Tavakoli, and Nora Ayanian. "Seamless Robot Simulation Integration for Education: A Case Study". In: *Workshop on the Role of Simulation in Robot Programming at SIMPAR 2016, San Francisco, CA, December 2016*. 2016. URL: http://act.usc.edu/publications/Hoenig_SimRP2016.pdf.