Efficient, Collision-Free Multi-Robot Navigation in an Environment Abstraction Framework

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Adrian Böckenkamp

Dortmund

2023

Tag der mündlichen Prüfung: 10.11.2023

Dekan:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter:	Prof. Dr. Heinrich Müller Prof. Dr. Dr. h. c. Michael ten Hompel

Dedication

This thesis is dedicated to my beloved parents, sister, and family for their endless support, encouragement, and sacrifices.

Preface

At this point, I would like to express my sincere gratitude to everyone who supported me in the creation of this work. I would particularly like to thank my supervisor, Prof. Dr. Heinrich Müller, for his valuable support, the numerous technical discussions, his flexibility in finding time slots for our regular meetings, and a large amount of liberties. I would also like to thank Prof. Dr. Michael ten Hompel for his willingness to write the second evaluation as well as Prof. Dr. Günter Rudolph and Prof. Dr. Peter Buchholz for their membership in the board of examiners.

I would like to thank my former colleagues at the Department of Computer Science VII as well as my current colleagues at the Fraunhofer IML for the support and excellent working atmosphere. I especially thank my former colleague and friend Dr. Daniel Hegels for many technical discussions and his willingness to help. Similarly, I would like to thank my friend Samir Mahmalat for proofreading, lengthy discussions and valuable feedback on the topic. I also thank Dr. Jana Jost and Dr. Denis Fisseler for proofreading, editorial feedback and organizational support. Finally, I would like to say a special thank you to my parents, who always supported and motivated me during my studies and doctorate.

Dortmund, November 12, 2023 Adrian Böckenkamp

Abstract

Industrial automation deploys a continuously increasing amount of mobile robots in favor of classical linear conveyor systems for material flow handling in manufacturing and intralogistics. This increases flexibility by handling a larger variety of goods, improves scalability by adapting the fleet size to varying system loads, and enhances fault tolerance by avoiding single points of failure. However, it also raises the need for efficient, collision-free multi-robot navigation. According to the current state of research, there is a need for further investigation focusing on guaranteed viable and collision-free motion while avoiding deadlocks and being able to cope with non-deterministic disturbances, e. g., unexpected obstacles and delays caused by computation and communication. These challenges are addressed in this thesis through the following contributions.

The core problem, namely collision-free *multi-robot navigation*, is first precisely modeled in a form that differs from existing approaches specifically in terms of application relevance and simultaneous structured algorithmic treatability. Collision-free trajectories for the mobile robots between given start and goal locations are sought so that the number of goals reached per time is as high as possible. Based on this, a decoupled solution is designed and implemented, which, in contrast to existing solutions, aims at avoiding deadlocks with the greatest possible concurrency. Moreover, this solution includes the handling of dynamic inputs consisting of both moving and non-moving robots.

Based on an existing, exchangeable planning algorithm, global paths are planned independently by each robot in the known semi-static environment. The developed *Collaborative Local Planning Framework* then coordinates the movement of the robots along these fixed, pre-planned paths. The framework defines the communication protocol, the behavioral model by a finite state machine, the graph-based conflict representation and the management of local knowledge on each robot. Robots exchange messages collaboratively to share knowledge about their current path and state. The pairwise conflicts occurring between the robots are geometrically detected and communicated to finally determine the Right-of-Way. Two algorithms were developed for this purpose, which are used in the framework's core: the *Incremental Coordination-Space Path Scheduler* which proceeds heuristically, and the *Optimal Multi-Robot Path Scheduler* which calculates optimal solutions but has exponential runtime complexity. After the Right-of-Way has

been determined for all conflicts and acknowledgments have been exchanged between all non-conflicting pairs of robots, robots are allowed to move while respecting the Right-of-Way. This also takes care of non-deterministic events. The framework guarantees collision-free movements and allows renegotiations with non-moving robots that request to move as well.

For testing, performance analysis, and optimization, due to the complexity of multi-robot systems, the use of simulation is common. However, this also creates a gap between real and simulated robots. Even worse, simulation speed and both physical and graphical fidelity are usually a trade-off in robotic simulation and have to be balanced depending on the investigated problem. These issues can be reduced by using several different simulators—albeit with the disadvantage of further increasing complexity. For this purpose, the *Robotic Experimentation Framework* is introduced to write robotic experiments with a unified interface that can be run on multiple simulators and also on real hardware. It is based on the Robot Operating System (ROS) and offers an easy-to-use Python API that facilitates the creation of experiments for performance assessment, (parameter) optimization and runtime analysis. The framework has proven its effectiveness throughout this thesis.

Lastly, experimental proof of the viability of the solution is provided based on a case study of a complete (simulated) assembly system of decentralized autonomous agents for the production of highly individualized automobiles. This integrates all the concepts presented into a holistic application of industrial automation. The modeling includes the definition of a product lineup and the dispatching of customer orders on the one hand and their fully autonomous assembly on the other hand.

Detailed evaluations of more than 800 000 solved scenarios with more than 5 700 000 processed goals have experimentally proven the robustness and reliability of the developed concepts. Up to 50 robots within a single environment were simulated on a single computer to demonstrate scalability. The execution times of the heuristic solver algorithm were typically much smaller than one second for up to 20 pairwise conflicts per input on a multi-core PC. Moreover, the system was able to increase its throughput while simultaneously incrementing the number of robots. Robots have never crashed into each other in any of the conducted experiments, empirically proving the claimed safety guarantees. A fault-tolerance analysis of the decentralized assembly system has experimentally proven its resilience to failures at workstations and, thus, specifically revealed an advantage over linear conveyor systems. Finally, an exemplary comparison with existing local planning algorithms also showed that these were either unsuitable or achieved a 10x lower throughput of customer orders while even causing collisions.

Zusammenfassung

In der industriellen Automatisierung werden für die Materialflussabwicklung in der Fertigung und Intralogistik zunehmend mobile Roboter anstelle klassischer linearer Fördersysteme eingesetzt. Dies erhöht die Flexibilität durch die Handhabbarkeit einer größeren Warenvielfalt, verbessert die Skalierbarkeit durch Anpassung der Flottengröße an unterschiedliche Systemlasten und erhöht die Fehlertoleranz durch Vermeidung von Single Points of Failure. Es steigert jedoch auch den Bedarf und die Anforderungen an eine effiziente, kollisionsfreie Navigation von Robotern. Nach aktuellem Stand der Wissenschaft besteht hier weiterer Forschungsbedarf, der speziell auf die garantierte Kollisionsfreiheit mit Ausführbarkeit bei gleichzeitiger Behandlung von nicht-deterministischen Störungen und der Deadlockvermeidung fokussiert. Diese Herausforderungen werden in dieser Arbeit durch die folgenden Beiträge behandelt.

Das Kernproblem, die kollisionsfreie *Multi-Roboter-Navigation*, wird zunächst präzise in einer Form modelliert, welche sich von existierenden Vorgehensweisen besonders durch Anwendungsrelevanz und gleichzeitige strukturell methodisch-algorithmische Behandelbarkeit unterscheidet. Gesucht sind dabei kollisionsfreie Trajektorien für die mobilen Roboter zwischen gegebenen Start- und Zielpunkten, so dass die Anzahl der erreichten Ziele pro Zeit möglichst hoch ist. Darauf aufbauend wird ein entkoppeltes Lösungskonzept entworfen und realisiert, welches im Unterschied zu existierenden Lösungen neben den eingangs genannten Herausforderungen auch darauf abzielt, Deadlocks bei möglichst hoher Nebenläufigkeit zu vermeiden. Zudem umfasst diese Lösung die Behandlung von dynamischen Eingaben, bestehend aus fahrenden und nicht fahrenden Robotern.

Basierend auf einem existierenden, austauschbaren Planungsalgorithmus werden von jedem Roboter unabhängig globale Pfade in der ihnen bekannten semi-statischen Umgebung geplant. Das entwickelte *Collaborative Local Planning Framework* koordiniert dann die Bewegung der Roboter auf diesen festen, vorgeplanten Pfaden. Das Framework definiert das Kommunikationsprotokoll, das Verhaltensmodell durch einen endlichen Automaten, die graphbasierte Konfliktrepräsentation und die Verwaltung von lokalem Wissen. Roboter tauschen kollaborativ Nachrichten aus, um ihren aktuellen Pfad und Zustand zu teilen. Die zwischen den Robotern auftretenden paarweisen Konflikte werden geometrisch erkannt und kommuniziert, um schließlich Vorfahrtsrechte zu bestimmen. Dazu wurden zwei Algorithmen entwickelt, die im Kern des Frameworks genutzt werden:

der *Incremental Coordination-Space Path Scheduler*, welcher heuristisch vorgeht, und der *Optimal Multi-Robot Path Scheduler*, welcher optimale Lösungen berechnet, jedoch exponentielle Laufzeit hat. Nachdem die Vorfahrt für alle Konflikte bestimmt wurde und Bestätigungen zwischen allen nicht im Konflikt stehenden Roboterpaaren ausgetauscht wurden, dürfen sich Roboter unter Einhaltung der Vorfahrt bewegen. Dies berücksichtigt auch nicht-deterministische Ereignisse.

Um aufgrund der Komplexität von Multi-Robotersystemen die Entwicklung, Tests und Optimierungen zu vereinfachen, ist der Einsatz von Simulation üblich. Dadurch entsteht jedoch auch eine Abweichung zwischen realen und simulierten Robotern. Simulationsgeschwindigkeit und sowohl physikalische als auch grafische Genauigkeit sind in der Regel zudem ein Kompromiss bei der Robotersimulation und abhängig von der zu untersuchenden Fragestellung. Durch den Einsatz mehrerer verschiedener Simulatoren kann dieses Problem vermindert werden – allerdings mit dem Nachteil weiter steigender Komplexität. Hierzu wird das *Robotic Experimentation Framework* vorgestellt, um Roboterexperimente mit einer einheitlichen Schnittstelle zu beschreiben, die auf mehreren Simulatoren und auch auf realer Hardware laufen. Es basiert auf dem Robot Operating System (ROS) und bietet eine einfach zu benutzende Python API, die die Erstellung von Experimenten etwa zur Leistungsanalyse, (Parameter-) Optimierung und Laufzeitbestimmung erleichtert. Das Framework hat sich durch intensive Nutzung im Rahmen dieser Arbeit bewährt.

Abschließend erfolgt ein experimenteller Nachweis der Tragfähigkeit der Lösung anhand einer Fallstudie eines vollständigen (simulierten) Montagesystems aus dezentral agierenden autonomen Agenten zur Herstellung hochindividualisierter Automobile. Diese integriert alle vorgestellten Konzepte in einen ganzheitlichen Anwendungsfall industrieller Automatisierung. Die Modellierung umfasst einerseits die Definition einer Produktpalette und die Einlastung von Kundenaufträgen sowie andererseits deren vollständig autonome Fertigung.

Detaillierte Auswertungen von mehr als 800 000 gelösten Eingabeszenarien mit mehr als 5 700 000 verarbeiteten Zielen haben die Robustheit und Zuverlässigkeit der entwickelten Konzepte experimentell nachgewiesen. Dabei wurde auf einem einzelnen Multi-Core-PC mit bis zu 50 Robotern innerhalb einer Umgebung simuliert, um die Skalierbarkeit zu zeigen. Bis zu einer Anzahl von 20 paarweisen Konflikten pro Eingabe lagen die Ausführungszeiten des heuristischen Lösungsalgorithmus typischerweise deutlich unter einer Sekunde. Darüber hinaus konnte das System seinen Durchsatz steigern, während gleichzeitig die Anzahl der Roboter erhöht wurde. Die Roboter sind in keinem der durchgeführten Experimente miteinander kollidiert, was die Zuverlässigkeit empirisch belegt. Eine Fehlertoleranzanalyse des dezentralen Montagesystems hat seine Widerstandsfähigkeit gegenüber Ausfällen an Arbeitsstationen nachgewiesen und damit insbesondere einen Vorteil gegenüber linearen Fördersystemen aufgezeigt. Ein exemplarischer Vergleich mit existierenden lokalen Planungsalgorithmen zeigte zudem, dass diese entweder gänzlich ungeeignet waren oder einen etwa 10x geringeren Durchsatz an Kundenaufträgen erzielten und dabei zudem Kollisionen auftraten.

Contents

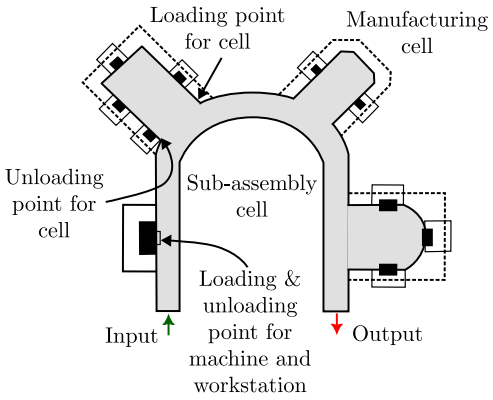
Chapter 1

Introduction

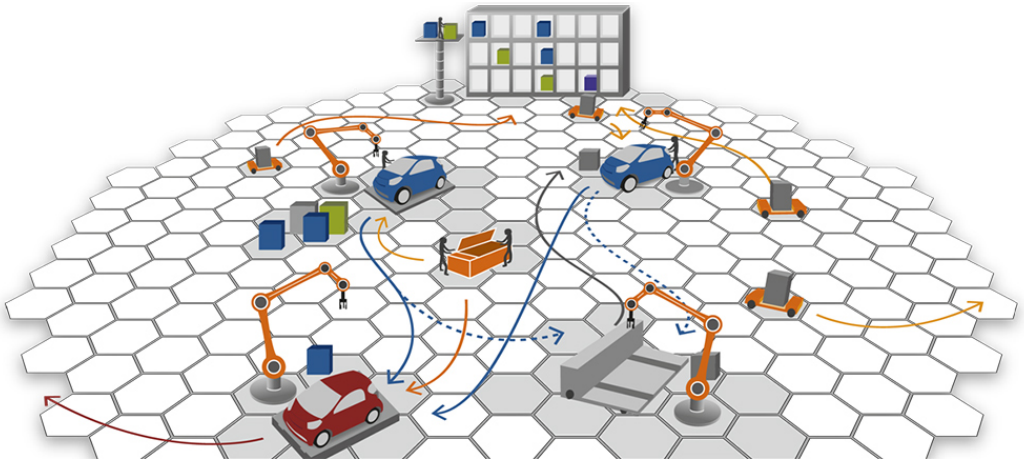
Robotics continuously gains momentum in the industry and massively changes the way commodities are being produced, and materials are being transported, improving quality and reducing costs. For instance, the mobile robot market, mainly consisting of sales of *Automated Guided Vehicles* (AGVs) and *Autonomous Mobile Robots* (AMRs), increased by 53 % in 2022, with over 4 million robots expected to be installed until 2027 [61]. Especially mobile robotics can be seen as the connecting link between the processing and transport of goods, parts or materials (production- and intralogistics). In contrast to fixed linear conveyor systems, a particular advantage of mobile robots is their variability, scalability and, thus, flexibility. They will therefore continue to supersede classic structures in logistics and manufacturing [31, 37, 52].

Generally, there are three main advantages of employing concepts of mobile robotics. First, classic linear conveyor systems are typically designed for a required throughput. By replacing them with mobile robots, the system can more easily be adapted to varying loads by adding or removing robots (up to a certain limit). That is, by avoiding fixed structures, *scalability* is increased. Second, transporting different goods or materials may be easier by different types of robots, empowering higher *flexibility* [6]. If all types of materials must be transported by the same conveyor system, it must be designed to always handle the most complex goods which may be fed into the system—even though this happens only rarely. Additionally, classic linear systems typically require some fixed topology while not allowing to dynamically skip or detour to another area or drop-off point. Contrarily, in a complex manufacturing process, decentralized operating mobile robots also allow for selecting a different assembly station if the originally targeted station is congested or failed. Third and finally, all segments of a linear conveyor system or an assembly line constitute a single point of failure. If an incident occurs, the entire

(a) Parcel sorting with the LoadRunner at the distribution center of DPD in Cologne [16]; 4 LoadRunner robots (black arrows) and 2 drop-off targets (red arrows) can be seen in the hall.



(b) Outline of a U-shaped cellular manufacturing system (based on [60]); products to be produced are loaded (green arrow), processed at the 4 cells and unloaded (red arrow).



(c) Sketch of fully decentralized assembly stations as a "Smart Factory" (from the SMART FACE project [25]); there are four workstations equipped with industrial robot arms and a warehouse (buffering parts and materials for assembly) in the background. Arrows indicate possible material flow.

Figure 1.1: Different use cases for decentralized mobile robotics; in (a) parcels are picked up by mobile robots, called LoadRunner, and they move their loaded parcel to designated dropoff locations. In (b), the U-shaped cellular manufacturing system is both supplied with materials and may even itself be driven by mobile robots. The sketch in (c) illustrates decentrally organized workstations spread over an industrial hall. Mobile robots are responsible for supplying materials and use them for assembly.

system must typically be stopped which incurs high losses. In contrast, mobile robots allow for selecting a different target (if any), increasing *fault tolerance*.

Three exemplary use cases are shown in Figure 1.1. Figure 1.1(a) depicts parcel sorting at the distribution center of DPD in Cologne, Germany, using the so-called LoadRunner (black arrows) developed at the Fraunhofer IML [56]. By relinquishing the classic conveyor belt for sorting parcels, robots exhibit much more flexibility in targeting different locations for parcel drop-off (red arrows). Conceptually, by joining multiple LoadRunners together via electromagnets, they can even transport larger objects in a team.

Figure 1.1(b) visualizes a U-shaped cellular manufacturing system with four "attached" cells. Such a system is similar to an assembly line because emerging products are typically loaded into the cell at the input (green arrow) and unloaded at the output point (red arrow). Similarly to the previous sorting example, such systems may not only be supplied with parts and materials by mobile robots, which is already common sense, but they also move the emerging product through the cellular manufacturing system itself.

As a last use case, consider the sketch in Figure 1.1(c), showing a decentralized assembly scenario for the production of automobiles from the SMART FACE project [25]. Up until now, this is normally done in highly optimized assembly lines, but recent trends tend to replace them with mobile robots as well [37, 41]. Fully or semi-automated assembly stations (or workstations for short) are distributed across the working environment in such a way that robots piggyback the cars to be produced while others are responsible for supplying required parts and materials for work steps conducted at workstations.

All explained use cases share the same theme: the transition from fixed linear structures to more flexible robots. Thus, based on rising demands for deploying mobile robotics and motivated by the plethora of possible applications, this thesis deals with efficient and collision-free multi-robot navigation—an active field of research. Basically and on the one hand, existing approaches can be categorized into coupled and decoupled. Coupled approaches [19, 23, 47, 57, 63, 66, 67] combine the path finding with the resolving of conflicts. Decoupled approaches [14, 27, 34, 39, 50] separate the problem into two phases: path finding and conflict resolution based on previously computed paths. On the other hand, approaches may also be categorized into centralized and decentralized. Centralized approaches [12, 14, 34, 48, 57, 63, 66] have access to all information relevant for the problem and consider all robots as a whole composite system (global perspective). In contrast, decentralized approaches [9, 11, 13, 19, 42] treat every robot as an individual and plan the paths independently (local perspective). In particular, many approaches [9, 11, 13, 14, 34] directly calculate the trajectories of the robots involved, but require strict adherence to the calculated target velocities, which can otherwise lead to collisions.

Therefore, according to the current state of research, there is a need for further investigations focusing on guaranteed viable and collision-free motion while avoiding deadlocks and being able to cope with non-deterministic disturbances, e. g., unexpected obstacles and delays caused by computation and communication. This also allows accounting

for simplifications and uncertainties in the underlying modeling of the robots and the environment. The proposed concept retains flexibility by coordinating pre-planned paths from almost arbitrary existing and interchangeable global planning algorithms. In contrast to many existing approaches based on graphs [12, 23, 57, 63, 66], this includes planning in free space. The approach focuses not only on avoiding but completely preventing collisions, making it specifically applicable for industrial automation where autonomous long-term operation is targeted.

The details of the approached problems of this thesis are described in the next Section 1.1. A thorough explanation of all contributions follows in Section 1.2. Section 1.3 continues with an outline of the entire thesis. Section 1.4 classifies the author's publications and Section 1.5 completes this chapter with a mandatory acknowledgment.

1.1 Problem Statements

This section explains the specific problems approached in this thesis. Section 1.1.1 deals with the main problem, namely multi-robot navigation. Section 1.1.2 addresses the issue of increased complexity when using robotic simulation. Both serve as the foundation for the contributions described afterwards in Section 1.2.

1.1.1 Multi-Robot Navigation

As motivated in the introduction, many industrial use cases can be tackled with mobile robots requiring multi-robot navigation as a fundamental ingredient. The problem is defined as follows.

Let a finite number N, $N \geq 0$, of inhomogeneous robots at predefined locations in the plane (2D poses) be given. Their size and shape is approximated by their smallest enclosing circle, i. e., every robot can have a differently sized circle. The environment is considered semi-static, that is, there are static obstacles (like walls) known to the robots. Some of these robots may also be temporarily inactive (idle), represented as known semi-static obstacles. In other words, idle robots are known as semi-static obstacles to all other robots, but such obstacles are being removed when they become active again. Note that unexpected obstacles like humans are not modeled as known semi-static obstacles, especially because they are unknown and exhibit much higher dynamics.

Goals are being assigned to idle robots from some external entity at any time. A *goal* specifies the 2D pose (position and orientation) in the environment, a robot should move to. Informally, a *path* defines the continuous series of locations from a robot's start location to its current goal. A *trajectory* additionally specifies the time when a robot must have reached the locations on its path (yielding velocities, accelerations, etc.). Once a robot has reached its assigned goal, it remains at the goal and becomes idle (semi-static obstacle). It may then be assigned another goal, reactivating the robot (making it active again).

Initially unknown disturbances may happen along the paths of the robots, occurring with a low frequency such that there are longer phases of normal operation without any disturbances. For instance, such disturbances may be humans, occasionally appearing in the robot's environment, detected by sensors on the robot (like laser scanners). Moreover, a small spatial deviation between a robot's actual and target trajectory is tolerated by adding safety margins to a robot's size. This effectively increases the space requirements of every robot along its path.

Additionally, it is expected that there are bidirectional communication links (e. g., via Wi-Fi) between all pairs of robots. As a result, in a system with a total of $N > 1$ active and inactive robots, each robot knows about all other $N - 1$ robots. Communication may have arbitrary delays but no data loss (as ensured by, e. g., TCP). There may be certain reaction times w. r. t. starting a robot's motion, e. g., due to delays caused by computation and communication.

The number N of simultaneously existing robots may vary over time. That is, new robots may be added as semi-static obstacles in the environment, effectively increasing N. Similarly, idle robots may be removed from the environment, effectively reducing N.

Within the previously explained specifications, multi-robot navigation eventually aims at finding collision-free trajectories for all active robots between their start and goal locations in the plane such that the number of goals per time is as high as possible. This is similar to the Multi-Agent Path-Finding (MAPF) problem from the literature [40] while also distinguishing between moving and non-moving robots (dynamic inputs).

1.1.2 Robotic Simulation

Apart from the advantages mentioned in the beginning, the industrial application of mobile robots becomes more and more complex due to an increased demand for product individualization up to lot size one, the wish for flexibility w. r. t. the employed robots (fleet size, inhomogeneous types, etc.), and the requirement to fully integrate robots seamlessly into the existing supply chains and manufacturing processes. Therefore, a potential disadvantage is the resulting complexity in developing and integrating such systems in practice. A common approach to overcome or at least reduce these challenges is applying simulation, e. g. using the Stage or NVIDIA Isaac simulator [24, 38]. Without the need to actually build a robotic system, simulation allows for testing, optimizing and evaluating in a "sandbox". Ideally, simulation allows to create a so-called digital twin of all relevant structures, components and goods, i. e., a virtual copy of the real environment.

Nonetheless, using simulation also creates a so-called reality gap between real and simulated robots [36]. The only way to overcome this issue is testing on real hardware which is tedious and costly. Furthermore, simulation speed and both physical and graphical fidelity are usually a trade-off in robotic simulation that needs to be balanced depending on the investigated problem. For instance, if a large fleet of robots needs to be

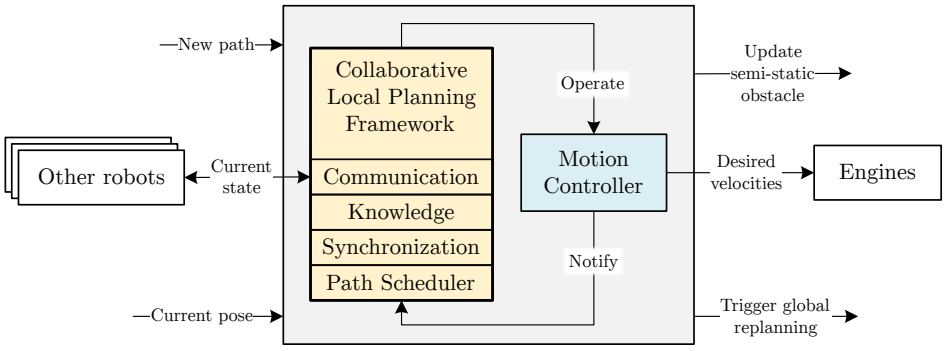


Figure 1.2: Conceptual overview of the new Collaborative Local Planning Framework (CLPF) with the main components communication, robot-local knowledge, state synchronization and the path scheduler (solver algorithm)

analyzed, fidelity is typically lower to prevent performance degradation of the simulation. In contrast, if only a few robots are being simulated, physical and graphical fidelity may be higher, yielding results that are closer to the behavior of real robots. Such issues can be reduced by using several different simulators—albeit with the disadvantage of further increasing complexity because different simulators [18, 24, 33, 38] typically provide different interfaces.

1.2 Overview of Contributions

This thesis provides the following contributions to the field of mobile robotics and its application in industrial automation.

First, the core problem, namely collision-free *multi-robot navigation*, is precisely modeled in a form that differs from existing approaches specifically in terms of application relevance and simultaneous structured algorithmic treatability (algorithm engineering). This was already sketched in Section 1.1.1.

Second, based on this modeling and according to the outlined taxonomy, a *decoupled* solution is designed and implemented, which, in contrast to existing solutions, specifically aims at avoiding deadlocks with the greatest possible concurrency in addition to the handling of unknown disturbances. Moreover, this solution includes the handling of dynamic inputs consisting of both moving and yet non-moving robots. Global paths are planned independently by each robot in the known semi-static environment between given start and goal locations using an existing, exchangeable planning algorithm. A map of the environment is provided by the so-called Vector Map Server (VMS).

The developed *Collaborative Local Planning Framework* (CLPF) then coordinates the movement of the robots along these fixed, pre-planned paths. The framework defines the communication protocol, the behavioral model by a finite state machine, the graph-based conflict representation and the management of local knowledge on each robot,

see Figure 1.2. Robots exchange messages collaboratively to share knowledge about their current path and state. The pairwise conflicts occurring between the robots are geometrically detected and communicated to finally determine the *Right-of-Way* (RoW).

Briefly speaking, a conflict is present if the occupied spaces around the paths of two robots intersect each other. In such a situation, we also speak of *intersecting paths*. The RoW at a given conflict specifies the order in which the two associated robots are allowed to pass through the conflict area: a non-eligible robot (not getting the RoW) may only enter the conflict once the eligible robot (getting the RoW) has completely passed through and left the conflict area. Determining the RoW at all conflicts of intersecting paths is termed *Path Scheduling*. That is, assume that a finite number of possibly intersecting paths in a 2D plane with robots located at their start locations is given. A robot is only allowed to move forward and has a defined maximum velocity whereby the magnitude of acceleration is assumed to be infinite. For all pairwise conflicts, a RoW is sought such that collision-free motion to the associated goals is possible for all robots while respecting the RoW and keeping the required time as low as possible.

Two algorithms were developed for path scheduling, which are used in the framework's core: the *Incremental Coordination-Space Path Scheduler* (ICSPS) which proceeds heuristically, and the *Optimal Multi-Robot Path Scheduler* (OMRPS) which calculates optimal solutions but has exponential runtime complexity. Basically, ICSPS is based on incrementally constructed so-called coordination spaces that are spawned by the paths of conflicting robots. By representing conflicts in the coordination spaces, the RoW at every conflict can be deduced by computing the shortest path in every step. OMRPS uses full search space enumeration of all possible decision vectors encoding the RoW. Every decision vector is scored and the best-scoring vector is returned. It turns out that OMRPS is quite applicable for small-sized inputs. ICSPS achieves practicable efficiency for larger-sized inputs but may not generally find a solution if one exists and may also return suboptimal solutions. Both solvers assume a static input, i. e., all robots are (assumed to be) non-moving and located at known start locations.

After the RoW has been determined for all conflicts and acknowledgments have been exchanged between all non-conflicting pairs of robots, the robots are allowed to move while respecting the RoW. CLPF operates the robot's motion controller (which controls the robot's engines, cf. Figure 1.2) and gets notified about events like reaching a goal. It guarantees collision-free movements and allows renegotiations with non-moving robots that request to move as well (dynamic input), effectively solving the multi-robot navigation problem. By computing RoWs and enforcing the explicit release of negotiated conflicts by CLPF's communication protocol, the handling of unknown disturbances is made possible which constitutes a major advantage of the proposed concept. Detailed evaluations of more than 800 000 solved scenarios with more than 5 700 000 processed goals have experimentally proven the robustness and reliability of the developed concepts. Up to 50 robots within a single environment were simulated on a single computer to demonstrate scalability. The execution times of the heuristic solver algorithm were typically much smaller than one second for up to 20 pairwise conflicts per input on a multi-

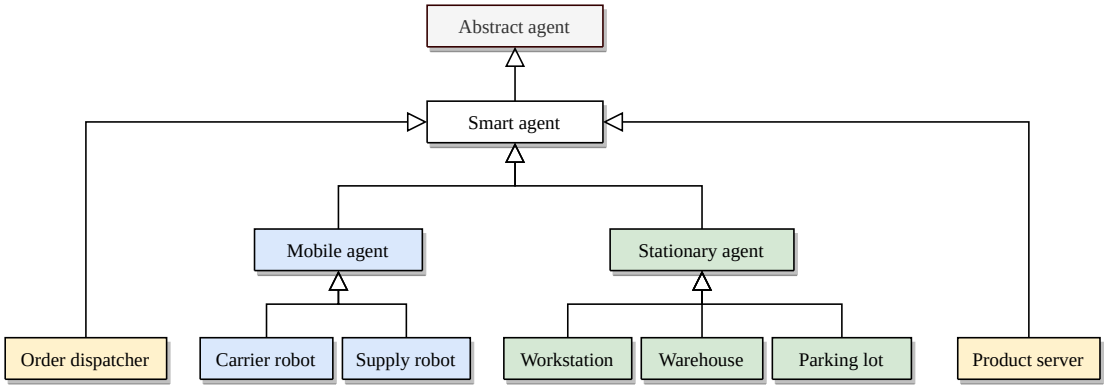


Figure 1.3: Modeling of the decentralized assembly system: concrete agents are shown on the lowest level in the hierarchy representing all autonomously acting entities

core PC. Moreover, the system was able to increase its throughput while simultaneously incrementing the number of robots. Robots have never crashed into each other in any of the conducted experiments, empirically proving the claimed safety guarantees.

Third and due to the increasing complexity justified in Section 1.1.2, the *Robotic Experimentation Framework (REF)* is introduced to write robotic experiments with a unified interface that can be run on multiple simulators and also on real hardware. It is based on the Robot Operating System (ROS) and offers an easy-to-use Python Application Programming Interface (API) that facilitates the creation of simulator-agnostic experiments for performance assessment, (parameter) optimization and runtime analysis. To the knowledge of the author, there is no such framework yet. Full mature support for the Stage [24] simulator and basic support for Gazebo [33] and MORSE [18] have already been implemented as part of this thesis. The framework has proven its effectiveness throughout this thesis.

Fourth and finally, an *experimental proof of the viability* of the solution is provided based on a case study of a complete (simulated) assembly system of decentralized autonomous agents for the production of highly individualized automobiles [49, 36]. This integrates all the concepts presented into a holistic application of industrial automation according to the use case visualized in Figure 1.1(c). The modeling includes the definition of a product lineup and the dispatching of customer orders on the one hand and their fully autonomous assembly on the other hand. The component hierarchy of the agents is sketched in Figure 1.3. Basically, a carrier robot transports the products to be assembled at workstations and supply robots provide materials from warehouses required during the assembly. Information about the product lineup, the required work steps for every product, the materials for every work step and the capabilities required by every agent are provided by the product server. Conceptually, the system can handle an unlimited number of different products and variants. In a throughput analysis, the system experimentally proved its scalability by continuously increasing the number of completed customer orders while incrementing the robot count. An exemplary comparison with

existing local planning algorithms also showed that these were either unsuitable or achieved a tenfold lower throughput of customer orders while even causing collisions. Notably, the assembly system was analyzed w. r. t. fault tolerance and experimentally proved its resilience against failures injected into workstation during operation. This specifically reveals an advantage over linear conveyor systems.

1.3 Outline of This Thesis

According to Figure 1.4, this thesis is organized as follows. Chapter 2 reviews related work from the literature and classifies the proposed concepts regarding existing state-of-the-art approaches. In addition, a taxonomy is presented and all reviewed research is categorized accordingly. Chapter 3 then presents the new *Robotic Experimentation Framework* (REF) based on ROS and an easy-to-use Python API which serves the general purpose of abstracting and simplifying robotic experiments. Chapter 4 presents and formalizes the underlying problem statements of collision-free multi-robot scheduling. The two novel solver algorithms, namely the *Incremental Coordination-Space Path Scheduler* (ICSPS) and the *Optimal Multi-Robot Path Scheduler* (OMRPS), are then described in detail and evaluated to compute a schedule guiding involved robots of a given input scenario to their goals. This guarantees collision-free motion given that the required assumptions are met. Chapter 5 introduces the new *Collaborative Local Planning Framework* (CLPF) that uses the previously mentioned solvers at its core in order to handle a fleet of dynamically moving robots in a shared environment, effectively preventing collisions between them. The framework contains the communication logic that is employed to negotiate conflicts between robots. After parameter optimization, it is experimentally evaluated regarding runtime performance, throughput, scalability and safety. Based on the concepts of all previous chapters, Chapter 6 presents the case study which comprises a fully operational (simulated) assembly system for the production of highly individualized automobiles based on decentrally and autonomously acting agents (with mobile robots in particular). Its evaluation targets throughput, fault tolerance and a comparison with other planners. Finally, Chapter 7 concludes the thesis with a brief summary and an outlook regarding further research.

1.4 Author's Publications

This section demarcates the author's publications relevant for the content of this thesis.

The design and implementation of *roslaunch2* (RL2) is presented in "roslaunch2: Versatile, Flexible and Dynamic Launch Configurations for the Robot Operating System" [5]. It is an essential part of the Robotic Experimentation Framework from Chapter 3. roslaunch2 was completely designed, implemented and published by the author of this thesis.

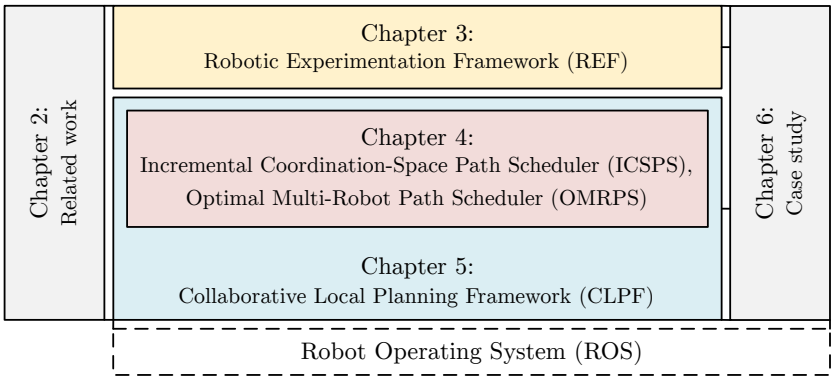


Figure 1.4: Illustration of the thesis' structure and its relation to the main chapters with their primary content: the concepts of Chapter 4 are used inside Chapter 5. Chapter 3 is conceptually independent of Chapter 5, and Chapter 6 uses the content of Chapters 3-5.

The paper "Towards Autonomously Navigating and Cooperating Vehicles in Cyber-Physical Production Systems" [7] deals with initial work on the topic of using mobile robots in production systems based on ROS. The author contributed the majority of the paper except for the section on collision avoidance and parts of the discussion and conclusion. In contrast to the concepts of this thesis, the paper focused on sensor based reactive collision detection and avoidance. It is only used as a fallback in this thesis in case of nondeterministic obstacles like humans. In such a case, a robot simply stops and waits until the obstacle disappears. The resulting delays are properly handled by CLPF from Chapter 5. Parts of the proposed design from [7] have influenced the concepts of this thesis while others have been partly abandoned (e. g., hierarchy of collision handling) or considered out-of-scope (e. g., hardware level).

The joint work "Towards Decentralized Production: A Novel Method to Identify Flexibility Potentials in Production Sequences Based on Flexibility Graphs" [4] with Bochmann et al. focuses on analyzing and leveraging the flexibility potentials of parallelizable production steps of the assembly of automobiles at Volkswagen. The author contributed in writing the sections related to graph theory, i. e., the modeling and visualization of flexibility graphs as well as the graph analysis of flexibility potentials. This served as input for the case study from Chapter 6, especially the modeling of product dependencies (Section 6.3).

Finally, the paper "A Versatile and Scalable Production Planning and Control System for Small Batch Series" [6] jointly written with Mertens et al. continued on the topic of transitioning from linear assembly lines to decentralized production, motivated by product individualization and to exploit flexibilities. The author primarily contributed to the section of flexibility potentials. The topic also relates to the motivation of the case study from Chapter 6.

1.5 Acknowledgment

Parts of this thesis have been realized in the context of the interdisciplinary research project “Smart Micro Factory für Elektrofahrzeuge mit schlanker Produktionsplanung” (SMART FACE) [25]. The author thanks the German Federal Ministry for Economic Affairs and Climate Action (BMWK) for the financial support.

Chapter 2

Related Work

This chapter reviews related approaches for solving the so-called *Multi-Agent Path-Finding* (MAPF) problem from the literature and explains the major differences to the proposed methods of this thesis (cf. Chapters 4 and 5) without explaining the math. In Section 2.1, we give a brief overview of the MAPF problem including a taxonomy of related approaches from the research field. The proposed methods from this thesis are then classified in Section 2.2 according to the taxonomy of Section 2.1. Finally, four major clusters of related approaches are identified, presented and delimited in Section 2.3.

2.1 Overview and Taxonomy

All possible states of the robots within the environment are typically called the *configuration space* $\mathcal{C}$, or C-space for short [40]. It typically comprises its position and orientation in the environment. The input for the MAPF problem is given by an initial configuration and a goal configuration for every involved robot. The output (solution) is a *trajectory* which specifies the sequence of configurations in time for all involved robots to reach their goals while avoiding (or even guaranteeing the prevention of) collisions. Note that a robot's shape is also represented differently in the various approaches, e. g., as a circle [42] or as a convex polygon [34], which can have a crucial impact on the performance. Every solution to the problem needs to compute a *path* for every robot. Approaches can therefore be categorized as follows:

- *Coupled* (aka direct or integrated) approaches [57, 63, 47, 66, 67, 23, 19] combine the path finding (aka routing) with the resolving of conflicts. A conflict is given by the intersection of two or more paths in space and time. That is, paths are

determined while avoiding conflicts between them. Notice that the notion of a conflict is sometimes separated into (first) space and (second) time for simplicity because an intersection in space is always a necessary condition for a conflict.

- *Decoupled* approaches [34, 27, 50, 39, 14] separate the problem into two phases, namely path finding and conflict resolution. In the first phase, paths are computed for all robots *independently* and in the second phase, robots are coordinated along their path (w. r. t. to time) to resolve all conflicts. The process may be iterative meaning that both phases are being repeated until an adequate solution was found. Additionally, the paths may be altered in the second phase to account for information about detected conflicts.

Based on the amount of information available in the path planner, approaches can also be classified as follows [40, 11]:

- *Centralized* approaches [34, 57, 63, 12, 48, 66, 14] consider all robots "to be a composite robot system, to which a classical single-robot path planning algorithm is applied", e. g., A* [40]. In particular, all information is available for all involved robots as from a *global perspective*. Centralized approaches normally provide the possibility of solving the problem optimally and completely (see below).
- *Decentralized* approaches [42, 13, 11, 9, 19] treat every robot as a separated individual and its motion is planned independently by treating other robots as moving obstacles. Notably, robots only have a *local perspective* on the entire system. Such approaches can also be cooperative by means of communication between the robots [42]. However, even if robots execute an algorithm on their own, sharing *all* information that a centralized solver would have (equaling a global perspective) makes the approach itself a centralized one [48]. The spectrum of decentralized approaches ranges from algorithms that solely rely on local sensor readings (fully decentralized) to those that build upon communication, cooperation or even collaboration (less decentralized).

Another distinguishing characteristic is how the approaches deal with the size of the C-space. For N robots, the combined C-space is given by $\mathcal{C}_1 \times \ldots \times \mathcal{C}_N$ which becomes huge already for small values of N. Thus, some approaches [57, 63, 12, 23, 14] discretize the C-space (or parts of it) by means of a so-called *roadmap*, typically represented as a *graph*. This effectively reduces the problem to the given graph. Related approaches are presented in Section 2.3.1.

Completeness refers to the property of a path planner of being able to find a solution, if one exists [40]. A *resolution-complete* path planner [12] must provide a solution if one exists, given a specific C-space discretization that the planner is operating on. Thus, it might not be complete in the general case. Noteworthy, sampling-based path planners generally only provide *probabilistic completeness*, e. g., the Rapidly Exploring Random Trees (RRT) algorithm [32].

Optimality refers to achieving the best "quality" of all possible solutions returned by an approach whereby quality is measured given some specific criterion, e. g., minimizing the sum of all execution times until a given set of robots has reached their goals (total travel time, throughput) [14, 23]. Another widely used criterion is minimizing the time it takes for the longest executing robot to reach its goal (critical path time, makespan) [47, 63].

Completeness, optimality and run-time complexity are tightly coupled with each other. Since the MAPF problem is PSPACE-hard [40], solving the problem optimally and completely likely requires exponential run-time complexity. Many proposed algorithms therefore apply heuristics to overcome practical runtime limitations while sacrificing either completeness, optimality, or both.

Additional dimensions of the problem are the type of obstacles considered (e. g., static vs. dynamic obstacles), the handling of deadlocks (and possibly even livelocks), provided safety guarantees, dealing with uncertainty, model inaccuracies (e. g., w. r. t. motion dynamics, the robot's shape, etc.), and in general, assumptions made in the entire approach. We summarize this by the employed *Environmental and Planning Model* (EPM). Most approaches that aim to handle dynamic obstacles (like humans) are decentralized because such obstacles are typically incorporated by means of sensor observations (e. g., laser scanners or cameras) [9, 26, 62, 2, 17]. However, in industrial settings and due to the overall complexity, dynamic obstacles are often neglected by assuming that robots drive in an isolated environment. Nevertheless, every robot still poses a dynamic obstacle to all other robots in the same environment.

2.2 Classification of Contributions

The proposed solvers of Chapter 4, namely the ICSPS and the OMRPS as part of the CLPF presented in Chapter 5, are mainly centralized (with the possibility of distributing the computations among all robots of the input) and decoupled. This is because the CLPF establishes a communication and interaction layer for robots of the system allowing them to share all required information. Computed trajectories from an instantiated solver within this framework are then executed ensuring safety (collision prevention) by explicitly releasing negotiated conflicts. Due to restricting the negotation of conflicts to smaller groups (so-called subgraphs of the intersection graph), the overall approach (CLPF) adds decentralized flavors. In particular, until distributed knowledge is synchronized, robots do not share an equal view on the system (hybrid approach). Within the category of decoupled approaches, the suggested approach falls into the category of fixed path coordination (cf. [11, 34, 27, 50, 39, 14] for related approaches) because both solver algorithms, the ICSPS and the OMRPS, aim at deciding the RoW for all conflicts given previously computed paths that remain fixed.

It is worth noting that a solver expects a *static* input while the CLPF extends this to the considerably more complex problem of handling *dynamic* inputs. A static input refers to a set of robots that are assumed to be non-moving with known start locations.

The solver then determines the RoW while assuming robots are at their start locations. Contrary to this, a dynamic input refers to the situation where a subset of the robots is already moving while others are standing still but requesting to move as well. In such a situation, the CLPF provides the methodologies to integrate the yet non-moving robots into the set of moving robots whereby non-moving robots may request integration at any time. The CLPF also handles possible uncertainty in the EPM on a conceptual level.

With respect to the general MAPF problem and especially given the assumption of fixed (that is, non-modifiable or re-plannable) input paths, none of the presented approaches are optimal or complete. However, note that this also accounts for increased practical computability. With respect to the path scheduling problem, the OMRPS can be considered optimal and complete (given the required assumptions) because it checks all possible solutions (see Section 4.5.1). Generally, the aforementioned properties depend on the actual instantiated solver within the CLPF. For instance, employing the ICSPS and considering all possible permutations of the order of the robots should guarantee completeness. Extending this by also taking into account all possible paths in coordination space (CS) should theoretically guarantee optimal solutions.

2.3 Range of Related Subjects

Within this section, four main clusters of topics will be presented which are related to the topic of this thesis, mainly the MAPF problem as introduced previously in Section 2.1. They contain a brief review and disinction of associated related work. Note that the clusters are naturally neither distinct nor disjoint.

Section 2.3.1 presents the topic of discretizing the C-space via roadmaps which is one of the most widely used method for tackling the MAPF problem. Section 2.3.2 then continues with decoupled approaches employing fixed path coordination. Inspired by work from the train scheduling domain, Section 2.3.3 presents approaches using Linear Programming techniques like Mixed Integer Linear Programming (MILP), Integer Linear Programming (ILP) and Quadratic Linear Programming (QLP) to solve the related Train Timetabling and MAPF problems. Finally, Section 2.3.4 reviews decentralized approaches.

2.3.1 C-Space Reduction via Roadmaps

The following papers perform some sort of discretizing the C-space in order to reduce its size and complexity and then apply different approaches on the simplified configuration space. Many of the approaches are coupled [57, 63, 12, 66, 23] and therefore also include the computations of the paths itself. In contrast, the approaches from Chapters 4 and 5 use fixed path coordination and consider the path for every robot to be part of the input which is a major conceptual difference. Generally, operating on a reduced C-space can only provide optimality and completeness within the given resolution of the roadmaps.

Additionally, the roadmap topology must be defined previously and is subject to have a huge performance impact. Clearly, all approaches are incomplete w. r. t. the general problem (without roadmaps).

Ter Mors et al. present a centralized, coupled algorithm for planning optimal and resolution-complete paths [57]. They integrate routing (planning) and conflict resolution in a "free path approach" while the environment is being modeled as a roadmap which is termed "infrastructure" where resources (like vertices in a graph) have a given capacity and a travel time. Reachability between resources is defined via a successor relation on the resources (similar to edges). A path is then given by a sequence of (free) resources through the roadmap. With the definitions of free time windows and reachability, the approach plans on a graph of such free time windows, similar to Dijkstra or A*. Robots can reserve resources up to the given capacity to prevent collisions. It is important to note that the authors consider the specific case where a set of robots have already planned paths and an additional one should plan its path as well while respecting the paths of the existing robots (not precisely equal to MAPF). Thus, this approach depends on the order of jobs and both optimality and completeness only applies to the single lastly planned robot. Nonetheless, the approach scales well and provides fast solutions for a large number of resources (1000) and existing paths (3000).

Yu and LaValle present a centralized, coupled and resolution-complete planning approach which is optimal on the employed graph model and under the imposed assumptions [63]. Based on their graph model, an ILP formulation of the multi-commodity network flow problem is presented which can be used to solve MAPF problems, resulting in their Time Optimal Multi-robot Path Planning (TOMPP) and Distance Optimal Multi-robot Path Planning (DOMPP) algorithms. However, the authors assume unit speeds and meet and head-on collision types only. Additionally, the formulation does not work on multigraphs, i. e., there cannot be more than one edge between two vertices and scenarios focus on grid-based environments. The output is a "path" on the graph, i. e., a valid series of vertices on the graph from the start to the goal for every robot which is either time optimal (TOMPP) or distance optimal (DOMPP). They evaluate their approach for up to 150 robots but only in grids up to size 32×32, requiring up to hundreds of seconds depending on the number of obstacles, robots and the actual grid size. After a specified cutoff time, the ILP solver was terminated for an input without having a solution which also indicates the complexity of the problem.

De Wilde et al. suggest the "Push and Rotate" algorithm, an improvement of the previously published "Push and Swap" [12]. It is a centralized, coupled path planning approach which is proven to be resolution-complete for two unoccupied locations on a connected graph. Methodologically, the authors transfer the problem of finding conflict-free paths to the *pebble motion problem* which, briefly summarized, deals with coordinating motions of pebbles placed on vertices in a graph. The pebbles (aka robots) must be moved from a given source to a goal location in the graph while not violating capacity constraints on the vertices. The output is a sequence of assignments (like a "move" in a board game) telling the robots to move from one vertex to another adjacent

vertex. The first assignment in this sequence equals the start and the last assignment the goal assignment. The basic idea is to plan a path in the graph for a single robot, identify conflicts with exiting paths and then try to fix the conflicts by a set of operations (`push`, `clear`, `resolve`, etc.) that move robots around on adjacent vertices in order to resolve the conflict. The authors identify graph structures that are the foundation for applying the various operations. The algorithm improves "Push and Swap" by fixing the handling of so-called isthmus (i. e., vertices with degree two in the underlying graph). Clearly, the algorithm respects robots (pebbles) already blocking vertices and thus, it depends on the order of how robots are planned similar to the work of [57].

Sharon et al. propose a centralized and resolution-optimal algorithm, called Conflict Based Search (CBS) [48], operating on a graph and optimizing the sum over all robots of the number of time steps required to reach their goals. The authors consider their approach as being "a continuum of coupled and decoupled approaches". They define the MAPF problem to be on *graphs* which is not equal to the common general case [40] with an undiscretized C-space. Sharon et al. summarize the idea of CBS as follows: "CBS is a two-level algorithm where the high level search is performed in a constraint tree whose nodes include constraints on time and location for a single agent. At each node in the constraint tree a low-level search is performed to find new paths for all agents under the constraints given by the high-level node. Unlike A*-based searches where the search tree is exponential in the number of agents, the high-level search tree of CBS is exponential in the number of conflicts encountered during the solving process." Similar to the intersection graph of Chapter 5, the approach employs what the authors call "Independence detection" to group robots having conflicts with each other in distinct groups to solve them independently.

Based on a QLP model, Digani et al. suggest a centralized, decoupled, non-optimal approach for AGV coordination on predefined paths of predefined roadmaps [14]. This can be considered as a transfer of the linear programming methods applied in the use case of the Train Timetabling Problem discussed in Section 2.3.3 to fixed-path coordination, cf. Section 2.3.2. The world is modelled as a given roadmap with "sectors" that robots need to travel whereby velocity is assumed to be constant on every sector. The authors consider robots having (geometric) intersections inside a sector as the input for the QLP. The output are velocities for the involved robots to coordinate them in a way to avoid collisions. The paper is based on several assumptions; however, as the most important distinction to this work (and although some assumptions are equal), the "collision avoidance" is based on adherence to these assumptions, especially in the QLP formulation. If this is not the case (as always in practice), collisions can occur which is not desired in an industrial context. Additionally and as already stated at the introduction, the approach requires "the roadmap" as well as the partitioning into sectors and that highly affects the performance.

Finally, Fransen et al. propose a centralized and coupled dynamic path planning approach [23]. The grid-based layout is represented as a graph with dynamically updated vertex weights. A vertex represents the center point of a so-called "zone" (a cell of the grid) and

an edge is present if two neighboring zones are connected. Edge weights are constant and initialized to the minimum travel time. All robots are assumed to be identical and the graph is directed, i. e., path segments are unidirectional only. Vertex weights are initialized to zero and updated based on the waiting time of a robot in a zone (occupied vertex), following by exponential smoothing after every update. Updates are performed in every "simulation step" (evaluated in MATLAB), so authors seem to assume clocked operation of their centralized system. It is considered coupled due to the dynamic updating of vertex weights. Replanning itself is done via A* based on the updated vertex and edge weights to minimize a cost function. The approach is entitled "dynamic" because replanning is triggered frequently, e. g., after traveling a given number of zones. The authors combine deadlock avoidance (deadlocks can still occur though) with deadlock recovery which is possible "as long as an alternative path to the same destination exists for at least one of the AGVs waiting in deadlock [...]" [23]. The authors consider a fair notion of dynamics by means of second order kinematics. However, they do not provide any information about optimality or completeness.

Notably, the papers [34, 66, 19] also employ C-space reduction but will be discussed in subsequent sections to elaborate on their central themes.

2.3.2 Fixed Path Coordination

There are decoupled approaches that consider the computed paths to be *fixed*. The resulting problem of planning motions (velocities) along the given fixed paths is therefore termed *path coordination* (or path scheduling). Approaches of this class will be presented in this section and are therefore specifically related to the proposed methods of Chapters 4 and 5.

LaValle and Hutchinson provide useful theoretical foundations for algorithms without evaluating them in simulations or real-world setups [34]. In particular, their centralized and decoupled fixed path coordination algorithm uses the "coordination space" which is also employed in the suggested approach of this thesis (cf. Section 4.4). However, in contrast to our approach, they seem to discretize the coordination space (grid-like voxels). The authors also present an approach for planning on roadmaps and a centralized motion planning approach without any C-space discretization but this is out of the scope of this section. The formulated optimization problems can be considered as a "black box": when no solution is found, it is unknown why and how this can be fixed. In their path coordination approach, similar to C-space discretization, the authors perform time discretization while sacrificing completeness for resolution-completeness. The work assumes that "a robot is capable of switching between a fixed, maximum speed and remaining motionless [...]". Although this is common in geometric motion planning to make the underlying problem more tractable, it renders the proposed approaches difficult to be applied in practice or voids collision avoidance guarantees if no additional care is taken.

Guo and Parker propose a decentralized, decoupled and presumably non-optimal (in a global sense) and incomplete motion planning approach for multiple mobile robots [27]. Incompleteness is inferred from its decentralized nature just as non-optimality, regardless of the fact that the approach optimizes a cost function for each robot individually and a "global measurement function" for the entire team of robots. The authors also do not explain in detail what information is actually shared between the robots making it difficult to judge whether the approach may even be rather centralized. For example, it is not explained how the final solution is selected based on what every robot has computed decentrally and is broadcasting afterwards. However, since it is stated that the approach can handle unknown terrain which is most probably integrated via additional sensors (being integrated in the computations), we consider it decentralized. The approach itself relies on a grid-based sampled environment and paths are being computed independently of other robots via D*. Robots share information to optimize a cost function and computations are distributed among the robots. Similar to the proposed approach of Section 4.4, the approach employs a (possibly discretized) "coordination diagram" to plan collision-free motion for the given input (termed "coordination space" in this thesis). The overall performance has only been evaluated for up to three robots with computation times in the magnitude of minutes, making dynamic replanning questionable. It is therefore also unclear how it scales. Results and experiments are not reproducible since used weights and software are not stated. In particular, the authors drop all assumptions in their practical experiments and still finally state that their approach is optimal. Given the stated computation times, it is hard to believe that this works in a "dynamic outdoor environment"; especially because they even state that their "approach is exponential in the number N of robots [but] efficient for a fixed N". It is unclear why this is described as "efficient". Due to its decentralized nature, it cannot guarantee safety. Finally, it should be noted that deadlocks are not considered at all which are particularly relevant in narrow environments.

The resolution-complete approach published by Simeon et al. is also based on finding a path in a so-called coordination diagram (CD) [50] which is similar to the CS representation of the proposed method from Section 4.4. The authors lack to classify their approach, however, we consider it to be centralized because all robots must be incorporated into a single CD. Simeon et al. define a path as a sequence of line segments (S) and arcs (A), different to the definition of this thesis (considering a path as a series of line segments only). The paper explains how the CD is defined and then presents an algorithm to do the path coordination based on searching in a decomposed N-dimensional generalized coordination diagram. This has the advantage of avoiding the explicit computation of the CD (exponential in the number of robots N). The authors also explain how CDs have been used in recent work to coordinate the motion of two robot arms only, indicating the general complexity of the path coordination problem. Similarly to the proposed *intersection graph* from Chapter 5, they also use such a concept to tackle only subsets of the problem by searching for connected components in the graph. All robots in the same connected component of the intersection graph have a spatial intersection of their paths so that they require conflict resolution. In contrast, this is not necessary

when considering two robots from different connected components. The work also uses so-called "coordination configurations" which conceptually correspond to the proposed *intersection guards* from Section 4.2.1. A coordination configuration represents obstacles as bounding boxes (simplification) to not explicitly compute them. This is different to the proposed work of Chapter 4 which makes use of halt and release points, precisely representing the last and first valid positions on a path respectively. The use of a simplification [50] may be justified by the more complex path definition (SA). The authors do not consider deadlocks and there is also no discussion of corner cases, i. e., if the start or goal configuration is located *on* the path of another robot. Another important difference to the work of this thesis is that Simeon et al. allow backwards motion of robots along their (fixed) paths, requiring non-monotonic paths in the CD. This may result in a more complex path planning compared to the monotonic paths in the CS required in this thesis (cf. Section 4.4).

Given a central instance that already generated deadlock-free paths for all robots, Olmi et al. have published a centralized and decoupled roadmap-based approach maximizing the "advancement of the [robot] fleet" [39]. The paper presents an approach for coordinating the planned path and also allows for deviations from the paths which are then re-coordinated. Coordination is also done in a coordination diagram (CD), as in [34, 50]. However, cells in a CD represent entire segments on the paths, being a specific artifact of the employed roadmaps. Colliding segments are termed "blocks". If a robot deviates from a path, the coordination diagram is updated. The central instance performs the coordination on a regular period and incrementally adjusts the reserved segments of the robots accordingly. These are major differences to the approach proposed in Chapters 4 and 5, especially the clocked re-coordination, updating the CD and generally the way, updated paths are incorporated. Not assuming that robots adhere to the computed "nominal velocities" on reserved segments can be perceived as a similarity to this work. With respect to Section 4.4, we achieve this by requiring that the bounding box of every segment of the path in the CD (aka coordination space) is empty. Although improving upon an existing proprietary AGV system, the authors did not evaluate on real robots and results are compared against the proprietary system only. The periodic execution of the central coordinator with a fixed period seems to require the explicit modeling of "starving vehicles" (that is, robots coming to a full stop and waiting for the central instance). In contrast, this thesis (see Chapter 5) suggests reacting to events that require an update of the coordination. Finally, given the segment granularity of the CD, the algorithm of Olmi et al. further approximates this by "convex polygonal regions".

Finally, given fixed paths for a set of robots, the centralized and decoupled coordination approach by Cui et al. [11] constructs a probabilistic roadmap (PRM) in the coordination space with a cost map equaling the "repulsive potential energy" between the involved robots. The repulsive potential energy is computed based on the distance between two robots' potential collision points (smaller distance yields higher energy). The authors also provide an estimation method for determining such "potential collision points". Finally, a search heuristic on the PRM is proposed (based on Dijktra incorporating motion and safety costs) to obtain "pareto-optimal solutions" in the coordination space

for the robots. The output are velocities for the given paths of all involved robots. The authors do not provide generalization of their approach to more than three robots (only applied for up to three robots). Notably, evaluation was only done for two real and three simulated robots, so scalability and general performance in higher dimensions is unknown. Expanding the approach and all proposed parts of it is assumed to be challenging. The paper suggests building a so-called "coordination roadmap" which is then used to find the resulting trajectories. However, that coordination roadmap is constructed within the coordination space and therefore depends on the current involved robot paths. Thus, it must be reconstructed whenever one robot gets a new goal. The paper does not deal with deadlocks or even the possibility of not finding a solution for an input at all. The authors further claim that a non-intersecting path in the coordination space is sufficient for collision avoidance between robots (despite the fact that they are incorporating "safety costs" in their search as this provides no guarantees). This only holds true if they precisely adhere to the (resulting) trajectories, or w. r. t. the addressed paper, if the estimated "potential collision points", distances and costs are sufficiently correct. All this may not be true in practice. This is an important distinction to the method proposed in this thesis which facilitates *arbitrary* velocity changes (e. g., due to unknown obstacles).

The work by Digani et al. [14] is also a (centralized) path coordination approach but has already been presented in Section 2.3.1 because it is also based on a roadmap.

2.3.3 Train Timetabling Problem

This section deals with related approaches employing any kind of Linear Programming (LP) models as its method for solving the underlying problem. During the review of relevant work, train scheduling, or more formally, the *Train Timetabling Problem* (TTP) has been found to be similar to the MAPF problem. The TTP aims to find a timetable that must respond both to commercial needs and certain capacity and security related constraints [8]. It may include the "routing" of trains, i. e., to also determine the route through the (fixed) railway network. Determining a timetable only refers to finding appropriate arrival and departure times for all involved trains while servicing desired stations, adhering to safety constraints, etc. Basically, this just adjusts the velocities (timings, including full stops with delays) of the trains on their tracks while avoiding collisions. The majority of approaches model the TTP via graphs [8, 47, 66, 67]. Notice that the TTP exhibits many variations like the platform design (single- vs. multi-platform models), the model of tracks (single- vs. multi-tracks which itself can be multi-, bi- or just unidirectional), the consideration of external constraints (w. r. t. safety, commercial, service level, etc.), and the requirement of computing periodic vs. aperiodic schedules. Since the train scheduling use case is only partially related, not exactly equal to the MAPF problem and many of the papers pursue a similar approach, the following review is kept brief. Practical feasibility is highly dependent on the resulting complexity of the LP models and computing time is normally in the magnitude of many minutes [66, 67].

As already presented in Section 2.3.1, there are approaches from the MAPF problem domain that employ similar methods, namely QLP [14] and ILP [63].

Generally, the reviewed work in the TTP domain always includes the routing of trains (similar to what is surveyed in Section 2.3.1) making it difficult to compare against the proposed methods of this thesis. In addition, neither the resulting timetable information nor the abundance of constraints are relevant within the MAPF problem because they are a direct artifact of the train transportation context. Nonetheless, the underlying ideas may be used to solve the problem addressed in Section 4.2.2 via some sort of LP model as well. However, this is out of the scope of this work.

Samà et al. aim to solve periodic routing and scheduling for the *Real-time Train Scheduling and Routing Problem* with the objective to minimize train delays. The approach can be classified as centralized and coupled, like many approaches for the TTP [47]. The authors apply MILP via the well-known commercial CPLEX solver with a relaxation of some constraints based on an "alternative graph", a directed multigraph with edges being alternatives. The relaxation is used to compute a lower bound which is then transformed via a "constructive metaheuristic" into a valid schedule, representing a "good upper bound" to the problem.

The publication by Zhou et al. aims to "simultaneously optimize operation periods, arrival times, and departure times of all period types of trains on a double-track rail network" [66]. Their method is based on "the construction of a weighted directed graph, [...] a 0-1 linear programming model is [then] built to minimize the total travel time of all period-types of trains subject to many operational constraints." Being centralized and coupled, this work reflects the typical approach in the area of the TTP.

Finally, Zhou et al. published additional work for the simultaneous multi-periodic train timetabling and routing problem by using a MILP model [67] to minimize the total travel time of trains. The railway network is modeled as a directed multigraph based on stations and railway tracks; stations are represented as (multiple) vertices, tracks are edges. They also used CPLEX with a heuristic strategy to simplify the model.

2.3.4 Decentralized Approaches

Within this last section, we focus on decentralized approaches tackling the MAPF problem. The level of decentralism is also briefly discussed.

Purwin et al. suggest a cooperative decentralized, decoupled, non-optimal and incomplete path planning algorithm and communication scheme by negotiation between the robots [42]. The underlying idea is that robots have to first request and then reserve areas in order to move strictly inside such areas, aiming to guarantee collision safety. Robots interact by communicating wirelessly via an Ad-hoc network and the authors design a protocol that is robust to communication delays. Intersections are computed geometrically based on rectangles enclosing fitted line segments to the robot's path (approximation). Reserved areas are computed based on the rectangle approximation

and requested areas are determined based on the convex hull of relevant points. If an intersection of two requested areas (B) is detected, robots exchange information via a handshake procedure to have recent information about each other. This allows them to check if also the reserved area (A) intersects ($B \subseteq A$). The concept of having requested and reserved areas is motivated by allowing robots to change their paths/goals (e. g., due to a new goal, a detour or obstacle avoidance, etc.). However, as the authors also reflect, having such concepts along with (even small) reserved areas A (requiring to be updated over time as the robot progresses on its path) requires a lot more communication between the robots. Once reserved areas of two robots are detected to have an intersection, a priority is determined solely between the two involved robots (decentrally) by evaluating a scalar cost function. This involves further communication as both robots evaluate their cost function and the "priority assigner" robot then decides which robot gets priority, i. e., the Right-of-Way, based on lower costs. The cost function is a heuristic that aims at minimizing waiting times. Its underlying idea gives robots lower cost which are closer to an intersection. The approach is presented for a pair (i, j) of agents and just simply extended by applying it to all pairs $i \neq j$. This does not guarantee any form of deadlock consideration, completeness or optimality. The intent of the suggested approach appears to be partial reservations (until the goal is reached) as also indicated by the reasoning for distinguishing between requested and reserved areas, although the authors state that requested and reserved areas should include the final destination "for performance reasons". The approach is extended by the concept of heuristically selected waypoints. The paper does not address deadlocks; especially given the proposed cost function, they are assumed to be even more likely (cf. Figure 4.5(a) where all robots are close to their first intersection, getting priority here). Even though the authors state that "[...] an extension to the basic algorithm is proposed, which [...] prevents deadlocks", they lack to provide justification why deadlocks cannot occur. Given the decentralized nature of how two agents solve a conflict suggests that deadlocks can happen and are not being handled at all. Although the authors claim that the system scales well, it has only been evaluated for up to five robots. Nonetheless, results always depend on numerous factors which makes them impossible to compare. All robots must be able to communicate with all other robots and the number of robots must be fixed a priori. However, an ad-hoc network may not guarantee pairwise communication between all entities of the network which could somewhat void the claimed safety guarantees.

Desaraju and How have published a cooperative decentralized, coupled and non-optimal path planning algorithm based on communication between the robots [13]. The authors present the (Cooperative) *Decentralized Multi-Agent Rapidly-exploring Random Tree* (DMA-RRT) algorithm "which introduces a cooperation strategy that allows an agent to modify its teammates' plans in order to select paths that reduce their combined cost". The DMA-RRT algorithm introduces a merit-based coordination strategy that takes advantage of the RRT trees to ensure global feasibility. It is non-optimal due to the employed *Closed-Loop-RTT* (CL-RTT) which focuses on speed and "uses a closed-loop simulation of the system dynamics to grow a tree of feasible paths. As a result, it is able to handle the types of complex and constrained dynamics [...] by embedding the

complexity in the simulation." The approach relies on robots to precisely adhere to their computed velocities (and the underlying models and assumptions to be correct) and cannot guarantee safety otherwise. The approach only "avoids certain common deadlock scenarios", but cannot generally do so or detect them.

The work of Chen et al. aims to solve the MAPF problem with Deep Reinforcement Learning [9] via their *Collision Avoidance with Deep RL* (CADRL) algorithm. As a result, it is decentralized (because every robot executes its own learned model based on sensor data), coupled (because the applied learned model directly yields velocities specifiying the paths to travel), non-optimal and incomplete (both due to its decentralized nature). The authors model the problem as a partially-observable sequential decision-making problem and apply Deep Reinforcement Learning to learn velocities (output) to move robots to their goals while avoiding collisions. Thus, the approach cannot guarantee collision-free motions. The CADRL algorithm incorporates sensor updates (input during motion) of every robot (approximated as a circle) and queries a previously learned "value network" while not requiring any communication between the robots (fully decentralized). However, they do not explain any details about the employed sensors. Sensor data seem to "report" information about others (robots, obstacles) close to a robot. The authors claim to do "simulation" but it seems that they are just "computing" the output of their models without an independent simulation backend. However, a real (physics-based) simulator would therefore provide a more realistic feedback of how the approach works. In particular, they have used their algorithm on a real robot (Clearpath Jackal with a Lidar sensor) but only evaluate for up to two dynamic obstacles (humans or other robots). Thus, its scaled applicability in practice is assumed to be limited.

Finally, the recent work of Eran et al. is presented [19]. The token-based negotiation approach for the MAPF problem is decentralized as robots share a local view via communication, non-optimal, and a mixture of coupled and decoupled because initial paths are planned individually but executing the proposed *Token-Based Alternating Offers Protocol* (TAOP) is being influenced by replanning with A* based on received offers (aka bids) from other robots nearby. However, we consider it to be rather coupled because the eventually travelled path is based on communication and replanning. The approach uses a grid (see Section 2.3.1) which represents the environment as a graph whereby every cell is a vertex which can be a position of a robot. Edges only connect the 4-neighborhood (i. e., no diagonal moves supported). The planning algorithm (A*) minimizes the sum of all path lengths whereby an edge has unit length, thus, it basically counts the number of edges. A field of view (FoV) concept specifies a local region (in the graph) around a robot's current position (occupied vertex) specifying to what other robots it can communicate and share information. Although beneficial in terms of performance, this causes the approach to be unable to guarantee collision-free motions, by design, because communication may be delayed or limited causing a lack of information in the planner of a robot. This is a major difference to the suggested approach from Chapters 4 and 5. Robots broadcast their paths to others within the FoV. If a conflict is being detected, robots execute the proposed TAOP to negotiate the conflict. The costs of planned paths are also used to decide how tokens should be used and whether to accept

or reject an "offer" of another robot. The main contribution of the paper by Eran et al. is based on improving the TAOP compared to previous token-(bid-)based protocols; it is not focusing on sophisticated path planning algorithms. Similar to Chapter 5, negotiations must be completed before the next one can start. However, we always communicate with all other (affected) robots to ensure safety.

Chapter 3

Robotic Experimentation Framework

Robotics typically starts with simulation. This is justified by the overall topic's complexity, the need for rapid prototyping and simplified testing. Because eventually, such a system needs to be tested and used with real robots, a modular execution of algorithms both in simulation and on real hardware is desirable. Choosing an appropriate simulator is challenging because they all have their advantages and weaknesses–more details will be explained in Section 3.2. Even worse, working in simulation normally imposes the so-called *reality gap* [46]. That is, briefly speaking, a real robot may behave slightly or even totally different compared to its simulated counterpart. For these reasons, using multiple simulation environments with different degrees of accuracy or speed may be beneficial—typically a trade-off. However, this adds even more complexity since simulators have different interfaces when interacting with the simulated scene.

Based on the *Robot Operating System* (ROS) middleware [43], this chapter proposes a framework for abstracting from different simulators to allow switching between variants more easily while still encouraging rapid prototyping and optimizations. Given the way how ROS works, it should be noted that ROS nodes (also known to be forming the *computation graph*) are running in a somewhat decoupled way from simulation because the simulator processes inputs for (simulated) actuators and generates output for (simulated) sensors based on ROS topics. However, this resembles the behavior on real hardware. The framework has been used efficiently in the experimental evaluations of Chapters 5 and 6.

The chapter is organized as follows. In Section 3.1, the new framework for abstracting different existing robotic simulators (and even real hardware) is motivated and presented

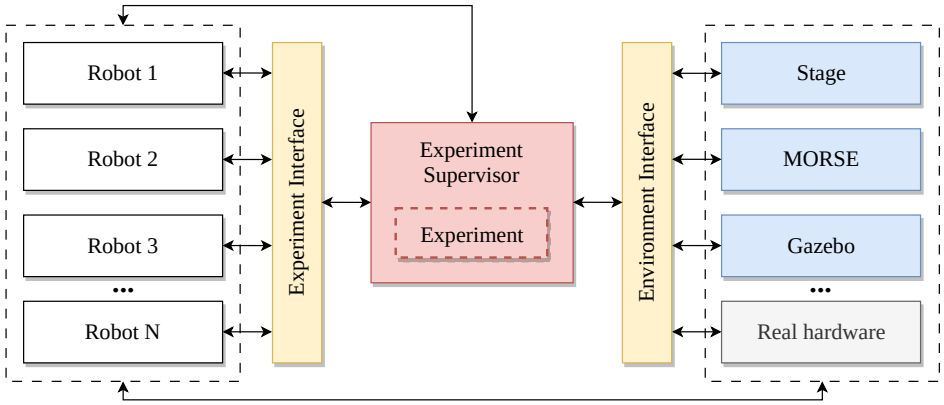


Figure 3.1: Components of the Robotic Experimentation Framework (REF) and their interaction; blue boxes represent robotic simulators and "real hardware" is added for completeness. The Environment Interface (EIF) (yellow) provides an interface to different simulators for the Experiment (Supervisor) (red). On the left side, the software stacks comprising the (simulated) robots are shown as white boxes, communicating with the Experiment Interface (EPI) (yellow) which provides unified access to certain ROS functionality for convenience. Arrows indicate communication.

in detail. It allows for writing robotic experiments that can be used to test and quantify certain aspects of a system. Section 3.2 describes three simulators that are completely or at least partially supported by the framework. Finally, Section 3.3 explains the (slightly simplified) Python code of an exemplary experiment.

3.1 Experiments in Abstracted Environments

The general idea is to have a framework that abstracts from a specific simulator (and even real hardware). This allows for switching between different simulators to trade off, e. g., precise and complex physics against performance. Another advantage of being simulator-agnostic is the independence to proprietary or abandoned software.[1]

The main components of the so-called *Robotic Experimentation Framework* (REF) are (cf. Figure 3.1)

- the *Environment Interface* (EIF),
- the *Experiment Interface* (EPI),
- the *Experiment Supervisor* (ESV) and

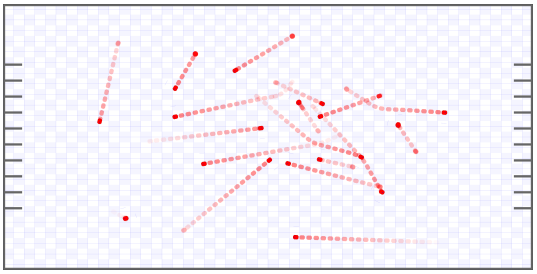
[1] In fact, there have been many discontinued robotic simulator projects in the past, e. g., MORSE or the STDR Simulator.

- *roslaunch2* (RL2) [5].

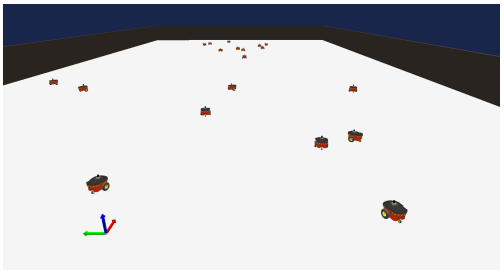
They are all based on the ROS middleware and written in Python. The Python language was chosen because it is simple, flexible (due to dynamic typing, reflection and introspection) and provides a common basis for accessing all components' APIs in a single codebase used to write the experiment. Additionally, Python is one of the two major supported programming language in ROS and unified access to ROS functionality is provided by the EPI. The EIF is responsible for abstracting from the interface of different simulators. The ESV simplifies specifying (writing) a concrete scenario (experiment) to be evaluated, that is, writing experiments for testing, training, and optimizing robotic algorithms and parameters in a setup defined and controlled by the experiment logic. Lastly, RL2 provides versatile, flexible and dynamic launch configurations based on ROS' *roslaunch*.

For example, consider the feature of pausing, resuming and resetting the simulation which is typically needed when setting up a specific scenario and repeating the execution of that scenario. A "scenario" might consist of a set of robots at predefined start locations and the goal of the experiment is to command them to some predefined goals while testing for collisions in the underlying planner. For the well-known Adaptive Monte Carlo Localization (AMCL), it is required to provide an initial pose before a robot can localize itself. This can be accomplished by providing that pose first, resume the simulation for some time to let the algorithm settle based on simulated sensor inputs, and pause it again to continue with the goal assignment. Once all robots are supplied with goals, the experiment can be started by resuming the simulation again. Registering collisions is also provided by the EIF which simply invokes a callback provided by the experiment code when a collision has been detected. Repeating the same experiment just requires resetting all robots to their initial start locations to start all over again. Starting ROS components (nodes) dynamically inside an experiment is provided by roslaunch2, even on different machines in the network.

Figure 3.1 shows the components and their interaction. All arrows in the drawing represent (possible) flows of information. Robotic simulators are shown as blue boxes on the right. They are considered as independent software blocks. The EIF (yellow) provides an unified interface to all supported simulators while the EPI on the left (yellow as well) provides convenience access to ROS functionality, e. g. waiting for a coordinate transformation to become available or a message to be delivered. Note that the drawing is centered around the experiment and experiment supervisor (red) because this is the Python code that needs to be written using the provided APIs to define the experiment *logic*. Experiments can be written from scratch which means that they *contain* both the experiment and supervisor logic. In other words, an experiment defines what kind of tasks the (simulated) robots should do and the supervisor controls and monitors the execution. Alternatively, experiments may also only define the tasks and some criterion to be computed once everything has finished, e. g. the total experiment time or the number of collisions that occurred during the experiment. More details will be presented



(a) Exemplary scene in Stage (top view)



(b) Exemplary scene in MORSE

(c) Simulated "Mini-Demonstrator" in MORSE

(d) Real setup of the "Mini-Demonstrator" [7]

Figure 3.2: From simulation to reality: (a) shows the simulated scene in Stage with 18 robots moving around randomly (see traces in red). (b) shows a similar scene in MORSE with 20 robots based on the state-of-the-art Bullet physics engine. Similarly, (c) shows a simulated setup in MORSE of the so-called "Mini-Demonstrator" (with 4 robots) developed in the SMART FACE projects with its real setup depicted in (d).

in Section 3.3. Finally, note that parallel experiment execution is possible by employing systematic ROS namespacing and remapping.

This concludes the overview of the developed concepts, i. e., how simulation environments can be abstracted and how this can be used to write robotic experiments against that abstraction. To the knowledge of the author, there is no such concept up to date based on existing simulation software. In the next section, we are going to discuss three (at least partially) supported simulators briefly and in Section 3.3 we will explain the possible experimental designs and give a concrete example.

3.2 Simulators

Initially, various simulators from the field of robotic simulation have been surveyed to identify the most appropriate ones for this work. Within this section, three simulators (supported by the REF) will briefly be presented along with their main features and properties. Figure 3.2 shows some simulated scenes for Stage (cf. (a)) and the Modular OpenRobots Simulation Engine (MORSE) ((b) and (c)) as well as a real hardware setup (cf. (d)) which was modeled in simulation (cf. (c)).

Stage [24] is a 2.5D multi-robot simulator which has been developed at the University of Southern California back in 2000. Stage provides rather simple robot models allowing to build large environments with up to hundreds of robots. Due to its high simulation speed, open-sourced codebase and mature ROS support, this has been the main simulation environment used throughout this thesis. As an extension to the pure text based world description format provided by Stage, a Python-based API has been developed to procedurally generate environments. This has the benefit of allowing more complex simulations because scenes can automatically be generated for different evaluation scenarios (and even inside an experiment). The footprints of generated static objects can be queried from the simulator and send to the map server, making them available to localization and path planning. Figure 3.2(a) shows a top view of a simulated scene with 18 robots (red) including their last motion traces (transparent red).

The second simulator to be presented is the *Modular OpenRobots Simulation Engine* (MORSE) [18]. It aims at simulating physics by means of the state-of-the-art *Bullet* physics engine [10] and provides a 3D visualization using the Blender Game Engine (BGE). Unfortunately, though still available, BGE was removed and dropped from Blender in 2019 and MORSE was discontinued in 2020. Because MORSE is written in Python as well, simulated scenes are also described with Python code. Limited support for MORSE in the EIF is provided. As shown in the two simulated scenes in Figure 3.2(b) and (c), MORSE provides a high-quality 3D visualization of the scene due to the BGE. Unfortunately, simulation speeds are much lower compared to Stage which is most probably be justified by the more complex physics. Requesting footprints from scene objects seems to be unsupported.

Third and finally, *Gazebo* [33] is mentioned for completeness. It is well-known in the robotics community, has excellent ROS support and is well integrated in the EIF although requesting footprints from scene objects is not implemented yet. Similar to MORSE, simulated robots need an attached "collision sensor" (aka contact sensor or bumper) to allow detecting collisions. Simulated scenes have to be described with XML and, unlike Stage and MORSE, would require more efforts to allow procedural generation.

3.3 Design of Experiments

It remains to present the experimental design. As already briefly outlined in Section 3.1, experiments can either also include the supervisor logic or may only specify the underlying conditions (robot tasks, criterion to be assessed) such that the proposed framework executes the experiment "automatically".

Because the former case is the more complex and flexible variant of writing an experiment, we will now discuss an example for it. Listing 3.1 shows the code for a complete experiment (including the supervisor logic) based on a configuration file (not shown here for brevity) written in the *Yet Another Markup Language* (YAML). The file specifies the number of robots, their starting positions and goals respectively, as well as other parameters. Line 7

```python
import experiment_supervisor.environment_interface as eif
import experiment_supervisor.experiment_interface as epi
import roslaunch2 as rl2
import rospy, yaml

# Statistics about the experiment:
collisions, reachables, unreachables = 0, 0, 0
def onCollision(state, timestamp, name):
    if state:
        rospy.logwarn(f"{name} has collided at {timestamp}!")
        collisions += 1

# Step 1: declare and setup the experiment.
this_exp = epi.ExperimentInterface('my_experiment')
launch_package = init_param('launch_package', 'experiment_supervisor')
launch_file = init_param('launch_file', 'experiment_base.pyl')
# Be able to interact with the "environment" (= simulator or "hardware"):
this_env = eif.get_environment_interface()
# Load the scenario stored as a .yaml file:
with open(init_param('data_path', None), 'r') as stream:
    (robots, scenario_data) = preprocess(yaml.safe_load(stream))

    # Step 2: start the world (robots) to be simulated.
    pkg = rl2.Package(launch_package)
    world = pkg.use(pkg.find(launch_file), robots=robots, instance_id=...)
    # Start the simulator and wait for the clock server (= simulator):
    launch = rl2.start_async(world, silent=True)
    this_exp.wait_for_non_zero_sim_time(max_timeout)

    # Step 3: setup test case.
    path_topic = init_param('path_topic', 'path')
    global_frame_id = init_param('fixed_frame_id', 'map')
    goal_action = init_param('goal_action', 'move_base')
    base_frame_id_suffix = init_param('base_frame_id_suffix', 'base_link')
    for robot in robots:
        # Register collision callback (called when a crash was observed or resolved):
        this_env.register_collision_callback(robot['name'], onCollision, robot['name'])
        # Wait for robots to start their software stack (path planning and localization):
        topic = rl2.ros_join(instance_id, rl2.ros_join(robot['name'], path_topic), True)
        rospy.loginfo(f"Waiting for navstack at {topic} ...")
        this_exp.wait_for_topic(topic, None, max_timeout)
        rospy.loginfo(f"Waiting for localization of {robot['name']} ...")
        this_exp.wait_for_transform(tf_join(robot['name'], base_frame_id_suffix),
                                    global_frame_id, max_timeout)

    # Step 4: let all robots start to move simultaneously.
    repeats = scenario_data['repeat'] if 'repeat' in scenario_data else 1
    while not rospy.is_shutdown() and iteration < repeats:
        iteration += 1 # number of repeats is a parameter
        this_env.pause_experiment(True)
        for robot in robots: # assign all goals while simulation is paused
            sendGoal(robot, robot['goal'])
        this_env.pause_experiment(False)
        # Wait until all goals have been processed and evaluate the result:
        for robot in robots:
            robot['client'].wait_for_result()
            if robot['client'].get_state() == GoalStatus.SUCCEEDED:
                reachables += 1 # robot successfully reached its goal
            else:
                unreachables += 1
    rl2.terminate(launch) # exit the simulation
```

Listing 3.1: Example for a complete experiment in Python based on the EIF, the EPI and RL2. Basically, the (slightly simplified) code builds up a simulation with a set of robots (step 1 and 2), waits for their software components to start (step 3), assigns goals to all robots and waits until all of them have been processed (reached or considered unreachable), see step 4. Some imports, parameters, error handling and minor helper functions have been removed for brevity.

declares variables recording the number of collisions between participating robots as well as reachable and unreachable goals respectively. Collisions are detected by the callback function `onCollision()` defined in Line 8 (and registered in Line 37 using the EIF). After declaring the experiment in Line 14 and retrieving a reference to the EIF in Line 18, the code reads the experiment description YAML file (Line 20), and loads and starts the simulator (Lines 23-28). The simulator and the simulated world are a dedicated launch module written and started using RL2. Lines 30-44 set up the experiment by waiting until the path planning and localization modules have been started using the EPI.

Lines 46-60 contain the logic for assigning goals to the robots and waiting until they have been processed. The YAML file may contain a parameter "repeat" specifying the number of repetitions, i. e., how often goals should be assigned to the robots. After pausing the simulator in Line 50, every robot is assigned its goal in Line 52. The simulation is then resumed in Line 53 such that all robots start to move simultaneously. The experiment logic then waits until all the goals have been reached or if robots report them as unreachable. Both outcomes are used to update the variables defined in Line 7. Once all repetitions have been processed, the experiment ends by terminating the simulator in Line 61. It should be noted that, depending on the desired experiment logic and software stack, one could also call `this_env.reset_simulation()` at the end of the `while`-loop (right before Line 61) to reset the simulator's state to the startup state. Because some simulators (like Stage) reset an object's velocity to zero after detecting a collision, additional actions may be required upon detecting a collision (e. g., moving a stalled robot to some free location by calling `this_env.teleport_entity()`) to avoid waiting infinitely in Line 56.

As an extension to Listing 3.1, consider the task of optimizing some parameter of the local planning algorithm. That is, after processing a set of goals, a parameter should be changed at run-time to retry the processing for the same set of goals while measuring the required total processing time for different parameterizations. For example, a new `value` for a parameter name stored in `param_name` can be set dynamically with:

```
for robot in robots:
  this_exp.reconfigure(f"{robot['name']}/path_planner/CollabLocalPlannerPlugin",
                       **{param_name: value})
```

Putting the code in Lines 46-60 in its own loop (iterating over different values for `param_name`), the previous call to `reconfigure()` would be part of that new loop.

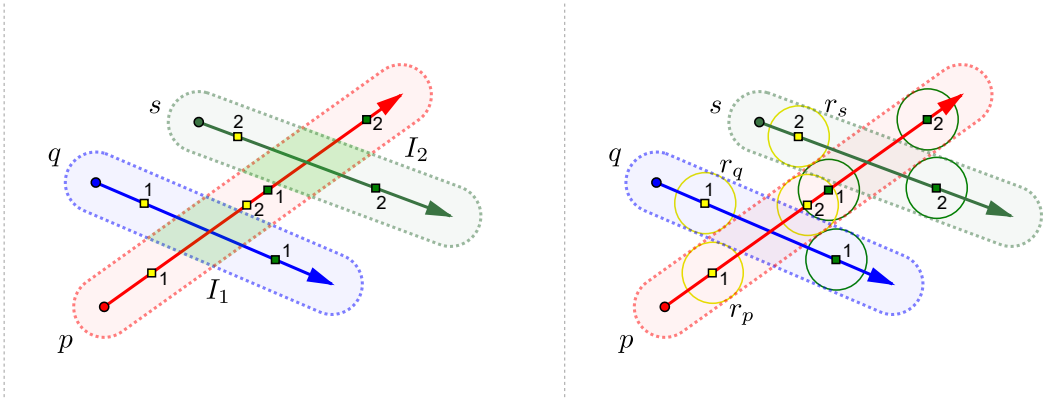
In anticipation of upcoming chapters, it is remarked that the REF is used throughout the evaluations of Chapters 5 and 6. For example, an experiment was written to assess the fault tolerance of the decentralized production system (which will be presented in Chapter 6). Basically, the experiment makes use of RL2's asynchronous starting capability to launch a dedicated process generating customer orders (system load). While the simulation is running and customer orders are being assembled, the experiment injects failures in workstations at defined time intervals. Meanwhile, the experiment also collects statistics like the number of detected collisions (if any), the achieved throughput and the number of completed and failed orders.

Chapter 4

Collision-Free Multi-Robot Scheduling

This chapter presents two novel approaches for the collision-free scheduling of a given set of robots with pre-planned paths. That is, given a set of robots with a path for every robot connecting its start (current position) with its respective goal, the algorithms described in this chapter either aiming at computing a schedule for every robot that is guaranteed to be collision-free or detecting infeasibility if there is no such schedule. With respect to Chapter 6 where robust and safe operation of autonomous mobile robots is needed in an industrial production system, these algorithms perfectly fit the needs of such a system. However, to make them applicable in such a dynamic environment, the concepts of the next Chapter 5 are needed to establish the communication logic between the robots required to provide synchronized and consistent inputs to the algorithms presented here. Both algorithms can thus be perceived as "building blocks" that can be invoked in the communication and synchronization framework described in Chapter 5.

The chapter is structured as follows. Section 4.1 briefly overviews the general underlying problems tackled here without detailing the math. Section 4.2 then explains the problems in more detail from a mathematical point of view, namely computing intersections geometrically and finding a schedule if one exists. The idea of conflict detection between robots of an input set is detailed in Section 4.3. The two novel solver algorithms are subsequently explained in Sections 4.4 and 4.5. Notice that related approaches from the literature have already been presented and delimited thematically in Chapter 2.



(a) Exemplary input scenario with 3 robots (b) Circles indicating intersection guards

Figure 4.1: Introductory example showing robots p (red), q (blue) and s (green) with paths consisting of a single segment each. Start points are indicated with a dot and goals are visualized by arrows. This scenario has two conflicts I_1 and I_2 (green intersecting areas) in total. The boundary around a path is determined by the radius of a robot's smallest enclosing circle. For both I_1 and I_2, the "guarding points" (squared numbered dots) indicate where robots would enter (green) and leave (yellow) an intersection (see (b)).

4.1 Introduction

An introductory example for the problem of collision-free multi-robot scheduling is visualized in Figure 4.1. It shows three robots p, q and s with their paths given as a single line segment for simplicity. Since all robots occupy space in the environment, their smallest enclosing circle with a given radius is shown by the rounded (inflated) polygons around the paths. The start of each segment is the (current) starting position of the associated robot and its endpoint denotes the goal. Within this particular example, p, q and s have two so-called *pairwise conflicts*, indicated as I_1 and I_2. The input is *feasible* because there exists an order of how these robots can be scheduled w. r. t. I_1 and I_2 without causing any collision or deadlock (cf. Section 4.2.2); for instance, p can get the RoW both at I_1 and I_2. When executing this schedule practically, it requires q to not pass the conflict area at I_1 until p has passed it completely and s to not pass through I_2 until p has left it as well. This might require q and s to actually wait (for p's motion) or to just reduce their velocities in order to adhere to the given schedule. That is, a schedule specifies how p, q, and s actually move along their paths. Note that the words "robot" and "path" are used interchangeably here (1:1 relationship).

This chapter deals with two problems. First, given the paths of all involved robots, the pairwise conflicts must be determined along with their exact so-called *guards* specifying where a conflict area begins and ends. Details of this problem are described in Section 4.2.1 and the new solution proposed to this problem is presented in Section 4.3. Second,

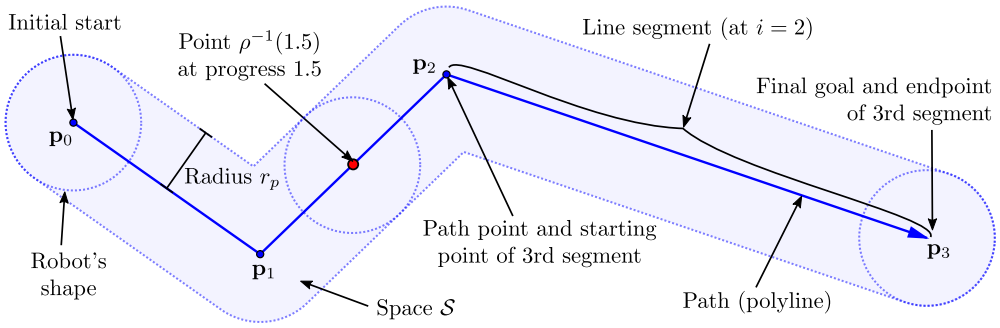


Figure 4.2: Visualization of basic terms like the robot's shape (blue circle), radius r_p (size approximation), path (solid blue, with $N = 4$ points), line segment (black curly bracket), start point $\boldsymbol{p}_0$, goal point (arrow at $\boldsymbol{p}_3$), space $\mathcal{S}$ (shaded in light blue), and progress (exemplarily depicted as red dot with occupied space visualized as blue circle).

once all conflicts are determined, either a Right-of-Way decision must be computed for all or a subset of all pairwise conflicts, or the input must be detected to be infeasible. This is explained and formalized in Section 4.2.2, and the proposed solutions are given in Sections 4.4 and 4.5. These solutions can not only be applied w. r. t. the case of Chapter 6 but are also generally applicable in contexts where safe multi-robot operation is needed.

4.2 Problem Statements

In the previous section, we already used terms like "path", "segment", etc. without defining them. Within this section, such terms are formally defined and the associated problems of this chapter are specified along with their in- and outputs.

Geometrically, a *path* is an ordered set $\mathcal{P} = \langle P_0, \dots, P_{N-1} \rangle$ of N poses and $N - 1$ *line segments* whereby $P_i = (\boldsymbol{p}_i, \boldsymbol{o}_i)$ is a tuple denoting the location $\boldsymbol{p}_i \in \mathbb{R}^2$ and orientation $\boldsymbol{o}_i \in \mathbb{R}^4$ (quaternion). The starting pose is given by P_0 and the goal pose by P_{N-1}. A path is assumed to be *clean* in a sense that there are no recurring elements, i. e., $\forall \boldsymbol{p}_i : \nexists \boldsymbol{p}_j : i \neq j \wedge \boldsymbol{p}_i = \boldsymbol{p}_j$. Additionally, we require that there are *no self-intersections*, thus

$$\forall i, j \in \{1, ..., N-1\}, i \neq j, i \neq j-1, j \neq i-1 : \overline{\boldsymbol{p}_{i-1}\boldsymbol{p}_i} \cap \overline{\boldsymbol{p}_{j-1}\boldsymbol{p}_j} = \emptyset, \tag{4.1}$$

or verbalized, for any two distinct, non-adjacent segments at indices $(i-1, i)$, $(j-1, j)$ of points on the path, their intersection must be empty. This property enforces that every point on a path is uniquely parameterizable (as explained subsequently along with the term "progress").

Given a robot p, its size $r_p > 0$ specifies the radius of its smallest enclosing circle which is used throughout this chapter to represent a robot's shape. For simplicity, we may also

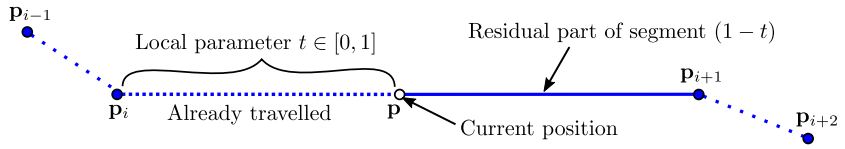


Figure 4.3: Illustration of how a progress value $i + t$ (with $t = 0.5$ here) relates to a point on a robot's path $\langle \ldots, \boldsymbol{p}_{i-1}, \boldsymbol{p}_i, \boldsymbol{p}_{i+1}, \boldsymbol{p}_{i+2}, \ldots \rangle$ for a line segment $\overline{\boldsymbol{p}_i \boldsymbol{p}_{i+1}}$ according to Equation (4.3). The dotted path indicates the part that the robot has already travelled.

subsequently refer to a path $\mathcal{P} = \langle \boldsymbol{p}_0, \ldots, \boldsymbol{p}_{N-1} \rangle$, $\boldsymbol{p}_i \in \mathbb{R}^2$ as being a *polyline in 2D* space only, i. e., ignoring orientation completely because given that robots are represented as circles, their orientations does not matter in subsequent concepts.

The set

$$\mathcal{S} \coloneqq \{\, \boldsymbol{p} \in \mathbb{E} \mid \exists i \in [1, N-1], \exists \boldsymbol{q} \in \overline{\boldsymbol{p}_i \boldsymbol{p}_{i+1}} : \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}) < r_p \,\} \tag{4.2}$$

specifies the *space* occupied by (the path of) p in the *environment* $\mathbb{E} \subset \mathbb{R} \times \mathbb{R}$ whereby $\operatorname{dist}(\boldsymbol{p}, \boldsymbol{q})$ is the Euclidean distance from $\boldsymbol{p}$ to $\boldsymbol{q}$.

It is convenient to globally parameterize a point $\boldsymbol{p}$ on a robot's path $\mathcal{P}$ in order to fully specify it with a single scalar value. This way, such points can simply be compared and stored efficiently by their global parameter (cf., e. g., Figure 4.1). Let $i \in [0, N-1] \subset \mathbb{N}_0$ be the index of the segment $\overline{\boldsymbol{p}_i \boldsymbol{p}_{i+1}}$ that the robot is currently driving along, that is, where $\boldsymbol{p}$ is located on. The resulting progress $\rho(\boldsymbol{p})$ is given by (cf. Figure 4.3)

$$\rho(\boldsymbol{p}) \coloneqq i + \frac{|\boldsymbol{p} - \boldsymbol{p}_i|}{|\boldsymbol{p}_{i+1} - \boldsymbol{p}_i|} \in [0, N-1]\,. \tag{4.3}$$

since

$$\boldsymbol{p} = \boldsymbol{p}_i + t \cdot (\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) \iff t \cdot (\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) = \boldsymbol{p} - \boldsymbol{p}_i \implies t = \frac{|\boldsymbol{p} - \boldsymbol{p}_i|}{|\boldsymbol{p}_{i+1} - \boldsymbol{p}_i|}. \tag{4.4}$$

Figure 4.2 schematically depicts the computation of progress as stated in Equation (4.3). For instance, the progress value 1.5 (red) refers to the second path segment and the robot has already reached the half of that segment. In other words, the progress denotes the completion status of a path, represented as a single scalar.

Vice versa, given a progress value $\rho(\boldsymbol{p})$, the associated point $\boldsymbol{p} \in \mathbb{R}^2$ on the robot's path can efficiently be computed by linear interpolation:

$$\boldsymbol{p} = (1 - \beta)\, \boldsymbol{p}_i + \beta \boldsymbol{p}_{i+1} \quad \text{with } i = \lfloor \rho(\boldsymbol{p}) \rfloor \text{ and } \beta = \rho(\boldsymbol{p}) - i. \tag{4.5}$$

In this equation, i denotes the index of the affected segment and β denotes the percental completion status of that segment. For improved readability, the notion "point" and "progress" will be used interchangeably throughout this text when referring to locations on a robot's path.

Finally, two robots p, q are said to be in *conflict* $\chi(p, q)$ (binary predicate), also (geometrically) termed *intersection*, if and only if there is at least one pair of points $\boldsymbol{p} \in \mathcal{P}$, $\boldsymbol{q} \in \mathcal{Q}$ on the paths with a distance less than the sum of both radii, i. e.

$$\exists \boldsymbol{p} \in \mathcal{P}, \boldsymbol{q} \in \mathcal{Q} : \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}) < r_{pq} \text{ with } r_{pq} = r_p + r_q. \tag{4.6}$$

As already indicated by Figure 4.3, the notion of a conflict somewhat depends on the starting point of the robot. For the remainder of this chapter, we assume that robots are always located at their initial start and take the entire path into consideration (e. g., w. r. t. Equation (4.6)). However, if this is not applicable (cf. Chapter 5) because robots may have moved already, the path must be truncated at the beginning and the remainder is considered to be $\mathcal{P}$ to adhere to this assumption.

In the next Section 4.2.1, intersection guards of a conflict are defined, followed by Section 4.2.2 which presents the detail of scheduling pre-planned paths.

4.2.1 Intersection Guards

This section introduces the notion of *pairwise intersections* based on Equation (4.6) along with so-called *guards* computed for every pairwise intersection. Intersection guards have already been shown exemplarily in Figure 4.1(b) as yellow and green squared markers on the involved robot paths. As an intuition, they "protect" the associated intersection in a sense that they indicate where robots would enter or leave the intersecting space required by the robots.

Let N robots with paths $\mathcal{P}_i$ and radius r_i be given as input. For every pair of conflicting paths $(\mathcal{P}, \mathcal{Q})$ of robots p and q in this input (cf. Equation (4.6)), all $k \in \{1, \ldots, K\}$ *pairwise intersections*

$$\begin{aligned} I_{p,q}^k \coloneqq & \left(\left[p_{\min}^k, p_{\max}^k \right], \left[q_{\min}^k, q_{\max}^k \right] \right) \text{ with} \\ & p_{\min}^k \in \left[0, p_{\max}^k \right) \subset \mathbb{R}, \quad p_{\max}^k \in \left(p_{\min}^k, N_p - 1 \right] \subset \mathbb{R}, \\ & q_{\min}^k \in \left[0, q_{\max}^k \right) \subset \mathbb{R}, \quad q_{\max}^k \in \left(q_{\min}^k, N_q - 1 \right] \subset \mathbb{R} \end{aligned} \tag{4.7}$$

of associated robots p, q must be determined (output). For every $I_{p,q}^k$, the *completeness property*

$$\begin{aligned} & \exists \boldsymbol{p} \in \mathcal{P}, \boldsymbol{q} \in \mathcal{Q} : \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}) < r_{pq} \implies \exists k \in \{1, \ldots, K\} : \\ & \rho(\boldsymbol{p}) \in \left(p_{\min}^k, p_{\max}^k \right) \wedge \rho(\boldsymbol{q}) \in \left(q_{\min}^k, q_{\max}^k \right) \end{aligned} \tag{4.8}$$

must hold, or in other words, there is no point on the path fulfilling Equation (4.6) that is not covered by at least one $I^k_{p,q}$. In addition, the *connectedness property*

$$\begin{aligned}
&\forall k \in \{1,\dots,K\} : \forall \boldsymbol{p} \in \mathcal{P} : \rho(\boldsymbol{p}) \in \left[p^k_{\min}, p^k_{\max}\right] \\
&\quad \implies \exists \boldsymbol{q} \in \mathcal{Q} : \rho(\boldsymbol{q}) \in \left[q^k_{\min}, q^k_{\max}\right] \wedge \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}) \leq r_{pq} \quad \text{and} \qquad (4.9) \\
&\forall k \in \{1,\dots,K\} : \forall \boldsymbol{q} \in \mathcal{Q} : \rho(\boldsymbol{q}) \in \left[q^k_{\min}, q^k_{\max}\right] \\
&\quad \implies \exists \boldsymbol{p} \in \mathcal{P} : \rho(\boldsymbol{p}) \in \left[p^k_{\min}, p^k_{\max}\right] \wedge \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}) \leq r_{pq} \qquad (4.10)
\end{aligned}$$

must apply, that is, for all points in an interval $\left[p^k_{\min}, p^k_{\max}\right]$ of an intersection $I^k_{p,q}$, the distance to the (partial) segments within the other robot's associated interval $\left[q^k_{\min}, q^k_{\max}\right]$ must be less than or equal to r_{pq} (likewise for q in Equation (4.10)).

Note that k depends on the robot that the pairwise intersection is referring to (p here), that is, for a given $I^k_{p,q}$ w. r. t. p, there $\exists k' \in \{1,\dots,K'\} \subset \mathbb{N}_0 : I^{k'}_{q,p} \mathrel{\widehat{=}} I^k_{p,q}$ regarding q and it does not necessarily hold that $k = k'$. We write $I^k_{p,q}$ if we are referencing the k-th intersection along the path of robot p. Likewise, we omit the subscript in I^k if it is clear from the context what robots are involved and it is the k-th intersection along the path of the robot that is lexicographically smaller.

The quadruplet in Equation (4.7) specifies a closed interval $[p^k_{\min}, p^k_{\max}]$ on path $\mathcal{P}$ (a *subpath*) and another closed interval $[q^k_{\min}, q^k_{\max}]$ on $\mathcal{Q}$. Given the left and right interval boundaries, we can define the *halt progress*

$$h(I^k_{p,q}, p) \coloneqq \begin{cases} p^k_{\min} & \text{if } \operatorname{dist}\left(\rho^{-1}(p^k_{\min}), \mathcal{Q}_{[q^k_{\min}, q^k_{\max}]}\right) = r_{pq} \\ -\infty & \text{otherwise} \end{cases} \qquad (4.11)$$

and *release progress*

$$r(I^k_{p,q}, p) \coloneqq \begin{cases} p^k_{\max} & \text{if } \operatorname{dist}\left(\rho^{-1}(p^k_{\max}), \mathcal{Q}_{[q^k_{\min}, q^k_{\max}]}\right) = r_{pq} \\ +\infty & \text{otherwise} \end{cases} \qquad (4.12)$$

for a given $I^k_{p,q}$ and an involved robot p. In both equations, dist() is defined as follows

$$\operatorname{dist}\left(\boldsymbol{p}, \mathcal{Q}_{[a,b]}\right) \coloneqq \min\left\{\operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}) \mid \forall \boldsymbol{q} \in \mathcal{Q} : \rho^{-1}(\boldsymbol{q}) \in [a,b]\right\} \qquad (4.13)$$

and specifies the minimum distance of $\boldsymbol{p}$ to the subpath $\mathcal{Q}_{[a,b]}$. Both boundary values are commonly termed *intersection guards*. Halt progresses are also named *halt points* or just *halts* interchangeably throughout this text (likewise for releases). For example, $r(I^k_{p,q}, q)$ is the release progress for robot q regarding the k-th intersection counted along the path of p.

The completeness property is important to ensure safety since all conflicting points $\boldsymbol{p}$ are then covered in at least one of the pairwise intersections $I^k_{p,q}$ which will finally be

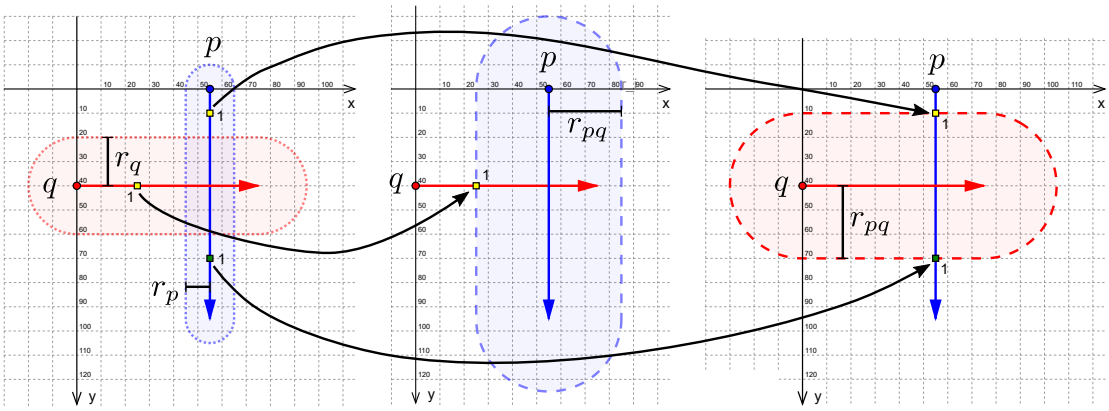


Figure 4.4: Different perspectives of how robot sizes can be interpreted w. r. t. intersection guards (yellow and green squared markers). All three plots show the same three intersection guards between the two robots p (blue) with size $r_p = 10$ and q (red) with size $r_q = 20$. Note that $r_{pq} = r_p + r_q = 30$. On the left, both paths are enlarged by their size respectively (dotted lines). In the middle, only p and on the right, only q is enlarged by r_{pq} (dashed lines).

negotiated by a solver algorithm (cf. Section 4.2.2), ensuring exclusive (space) resource assignments, that is, only one of the two robots p, q will be in intervals of $I^k_{p,q}$ simultaneously. The connectedness property together with Equations (4.11) and (4.12) provides the foundation for computing pairwise intersections because the property implies that only the guards (interval boundaries) have a distance of exactly r_{pq} to the segments within the subpaths defined by the intervals.

Figure 4.4 depicts the possible perspectives on how the locations of intersection guards are related to the robots' sizes. In the leftmost plot, both robots are shown with their space requirements based on their individual sizes. In the mid-plot, only p (blue) is enlarged by the sum r_{pq} of both radii while in the rightmost plot, only q is enlarged by r_{pq}. Basically, to identify the guards for q (cf. mid plot), it is sufficient to find the intersection points (yellow squared marker) of the inflated polygon around p (blue) with the polyline of q (red). Likewise, to identify the guards for p (cf. yellow and green squared marker in the right plot), we intersect the inflated polygon around q (red) with the polyline of p (blue).

This justifies the assumption that all perspectives are equal w. r. t. finding such guard points. All pairwise intersections together with their defining guards comprise the output which serves as the input for the problem described in Section 4.2.2.

We will finally review some useful insights and properties of halt and release points. Assume two robots p, q, the sum of both radii r_{pq} and an (pairwise) intersection $I^k_{p,q} = \left(\left[p^k_{\min}, p^k_{\max}\right], \left[q^k_{\min}, q^k_{\max}\right]\right)$ are given.

(i) If a halt doesn't exist for robot p, it means p is already *inside* that associated intersection I (blocked at the starting position).

(ii) Equally, if a release doesn't exist for a robot p, it means p will not be able to leave the associated intersection I (blocked at the goal).

(iii) If both $h(I^k_{p,q}, p) \neq -\infty$ and $r(I^k_{p,q}, p) \neq \infty$, it always holds that $h(I^k_{p,q}, p) < r(I^k_{p,q}, p)$. More precisely, this holds even if only $r(I^k_{p,q}, p)$ exists.

(iv) Theoretically, a release must always be larger than (and cannot be equal to) the start progress (and less or equal to the goal progress). Assume there is a release point on the start. For that release $r(I^k_{p,q}, p)$, it must hold that it has a distance exactly equal to r_{pq} to the subpath $[q^k_{\min}, q^k_{\max}]$ of robot q. Since there is the intersection I associated with this release, there must be a point $\boldsymbol{p} \in \mathcal{P}$ with a smaller progress than $r(I^k_{p,q}, p)$ (i. e., $\rho(\boldsymbol{p}) < r(I^k_{p,q}, p)$) that has a distance less than r_{pq} to path $\mathcal{Q}$. However, since $r(I^k_{p,q}, p)$ is equal to the start, there cannot be any point $\boldsymbol{p}$ satisfying the previously mentioned condition and, thus, a contradiction to the assumption that the release point is located on the start.

(v) Likewise, a halt must always be less than (and cannot be equal to) the goal progress (and must be greater or equal to the starting progress). The justification of the previous property can also be applied here similarly.

(vi) If both $h(I^k_{p,q}, p) = -\infty$ and $r(I^k_{p,q}, p) = \infty$, it means that p will never "leave" the intersection $I^k_{p,q}$ at all (infeasible).

(vii) If both halts $h(I^k_{p,q}, p)$ and $h(I^k_{p,q}, q)$ do not exist, both robots p and q already crash (or at least are too close) at their starting positions (infeasible). This is considered invalid input.

(viii) If both releases $r(I^k_{p,q}, p)$ and $r(I^k_{p,q}, q)$ do not exist, both robots have goals that are too close—so-called a *goal conflict* (infeasible).

4.2.2 Scheduling Pre-planned Paths

As described in the previous section, for any given input scenario, the set of pairwise intersections I^k is computed which, along with the maximum velocity $v_{\max}$ per robot, serves as the input for the problem that is detailed in this section. Basically, the underlying idea is to find a way of how all robots can safely move on their precomputed paths to their goals by determining an ordering in which they are allowed to pass "critical areas" (pairwise intersections) and enforcing this ordering temporally without sacrificing a particular robot (i. e., all robots are considered equal). More specifically, for every $I^k_{p,q}$ a *Right-of-Way* (RoW) decision

$$\delta\left(I^k_{p,q}\right) \in \{p, q, \emptyset\} \tag{4.14}$$

is determined specifying which robot (p, q) is allowed to pass its interval defined in $I^k_{p,q}$ first or if $I^k_{p,q}$ is considered infeasible ($\emptyset$). For example, an intersection can be infeasible if it is deadlocked (as detailed shortly). When combining all decisions for all pairwise intersections, a partial order of subpaths of all involved robots is obtained. Such a subpath is the partition of the underlying robot path and defined

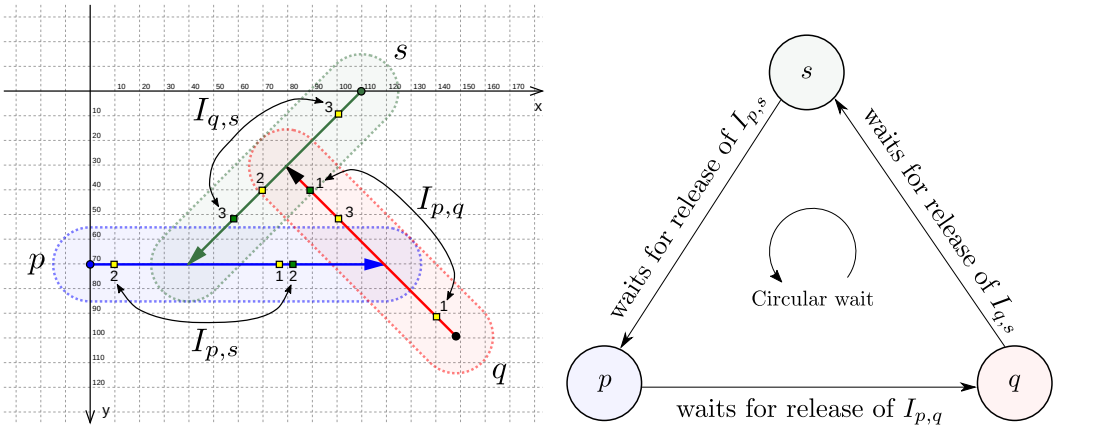
- from a robot's starting point to the first relevant[1] halt point,
- from a relevant halt to the next relevant halt point and finally,
- from the last relevant halt point to a robot's goal.

These are the sections along a robot's path where it can move forward freely without any synchronization. The schedule ensures safe motion of the involved robots provided all robots adhere to their RoW and never leave their allocated space. Alternatively, an input might be classified as fully or partially infeasible if there is no such schedule, i. e., $\exists k : \delta(I^k_{p,q}) = \emptyset$. Note that such a schedule contains the RoW for all pairwise intersections occurring in the entire scenario from a robot's start until its goal.

A robot p getting RoW at $I^k_{p,q}$ can simply ignore the halt point $h(I^k_{p,q}, p)$ but must report when it has passed its release $r(I^k_{p,q}, p)$. This ensures that the other robot q, not getting RoW at $I^k_{p,q}$, gets notified when it is allowed to enter the intersection. Until then, q is not allowed to drive past its halt $h(I^k_{p,q}, q)$. The underlying communication logic is explained in Chapter 5. Also note that robots are not allowed to move backwards on their paths.

Inputs for this problem can have deadlocks which will be justified and explained next. To begin with, Figure 4.5 illustrates an example showing a deadlocked scenario. The black arrows in Figure 4.5(a) highlight how guards form the interval as part of a pairwise intersection. Note that all intersections miss a release point (cf. Equation (4.12)) for one particular robot. Two observations are important. First, all decisions at intersections are already predetermined because there is always one robot involved that would block an intersection forever. Second, every robot passing through its first intersection to give RoW at its second intersection will continue to block the first while waiting for release. Without sacrificing one of the robots, the RoW assignments would be $\delta(I_{p,q}) = q$, $\delta(I_{p,s}) = p$ and $\delta(I_{q,s}) = s$. The predeterminism might even be enforced if the robots would have already started inside their first intersection (same $\delta()$). For instance, $\delta(I_{p,q}) = q$ (cf. the three intersection guards labeled "1" in Figure 4.5(a)) is required because p would block $I_{p,q}$ forever once reached causing q to not be able to pass it ($I_{p,s}$ and $I_{q,s}$ are justified alike). This schedule imposes the waiting dependencies shown in Figure 4.5(b), effectively creating a circular waiting relation. Since there is no other possible schedule (only the deadlocking one), this scenario is considered infeasible.

[1] "relevant" means that the associated robot does not get RoW requiring it to possibly come to a full stop at the halt.



(a) Deadlock situation between p, q and s (b) Waiting dependencies for the robots in (a)

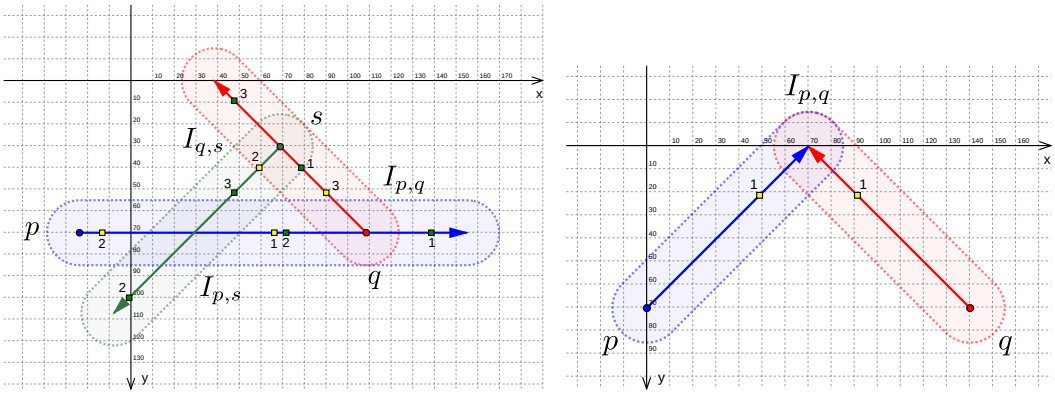
Figure 4.5: Example scenario with three robots p, q and s ($r = 15$) exhibiting deadlock behavior because the decisions for all three pairwise intersections $\{I_{p,q}, I_{p,s}, I_{q,s}\}$ are predetermined in a sense that the decisions $\delta(\cdot)$ are implied by the scenario itself, i. e., $\delta(I_{p,q}) = q$, $\delta(I_{p,s}) = p$ and $\delta(I_{q,s}) = s$ without "sacrificing" one specific robot. These yield the waiting dependencies depicted in (b).

According to [55], there are four so-called *Coffmann conditions* which are sufficient for a deadlock to take effect:

- *Mutual exclusion:* areas (spaces) in the environment are blocked by robots, and they must be used exclusively to prevent collisions. Thus, they are the essential "resource" that is reserved and used mutually exclusive.

- *Hold and wait:* a robot is always occupying an area around its current position (hold) and in terms of driving to a goal, areas without any intersections are also reserved exclusively for that robot (which makes sense since no other robot demands them). Thus, all areas that do not have an intersection are blocked by that particular robot and are part of the "hold". Basically, this treats these areas as if they are the same (or have the same property). These areas (resources) are held until the associated area is passed through by driving along the path (release of resources). Conceptually, a resource is (partially) freed when a robot reaches its next release point within the given scenario. The "wait" applies for intersection areas where a robot may not get RoW and, thus, needs to wait until another robot (with RoW) has passed through that area.

- *No preemption:* because robots move on their own (will) and safety must be guaranteed at any time, assigned resources (reserved areas) can never ever be transferred to others. In particular, a robot always voluntarily frees segments of its path during motion.



(a) Feasible input with deadlock possibility (b) Infeasible input due to a goal conflict

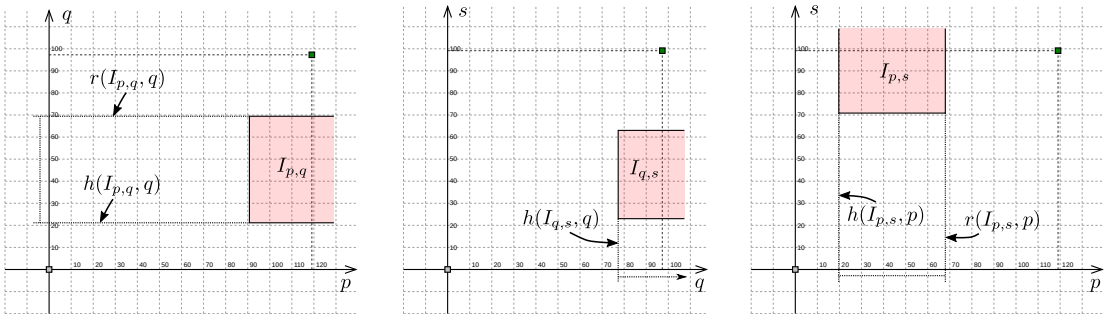
Figure 4.6: Examples for (a) a feasible input with deadlock possibility and (b) an infeasible input due to a goal conflict between robots p and q (no infinite wait, goals are not reachable for both). In (a), the assignment $\delta(I_{p,q}) = q, \delta(I_{q,s}) = s, \delta(I_{p,s}) = p$ would be a deadlock because none of the robots would be able to reach their goals (infinite wait). In contrast, with $\delta(I_{p,s}) = s$, the input is feasible without deadlocking behavior.

- *Circular wait:* since robots block areas of the path ahead and wait for subsequent areas at halts of intersections, a circular waiting situation can occur which will never terminate, see Figure 4.5(b).

If all conditions are met, an input can (but does not have to) have a deadlock. Of course, this depends on the actual schedule. One needs to decide between two very distinct and important cases. On the one hand, a deadlock that cannot be resolved since it is part of the scenario and there is no deadlock-free schedule at all (cf. Figure 4.5). On the other hand, there can be inputs where not only deadlocking (invalid) but also deadlock-free schedules exist (and a planning approach that aims to be complete must find it, cf. Section 2.1). An example for this is presented in Figure 4.6(a). The assignment $\delta(I_{p,q}) = q$, $\delta(I_{q,s}) = s$, $\delta(I_{p,s}) = p$ would be a deadlock. However, changing $\delta(I_{p,s}) = s$ would solve the scenario without any deadlocking behavior because s would always get RoW. For comparisons, Figure 4.6(b) also shows a scenario that is infeasible due to a goal conflict between p and q (not a deadlock).

Another important observation regarding the use, definition and notion of a "resource" (area) is that (parts of) resources may also never be freed since robots will always occupy space at their current position and finally, at the goal. This is somewhat different to resources in the context of processes in an operating system where resources are typically used for a fixed amount of time.

Pairwise intersections can also be interpreted in a (spatial) *coordination space* (CS) [34] as depicted in Figure 4.7 for the previous deadlock example from Figure 4.5. A coordination space is spanned by the two involved robot paths defining the horizontal and vertical axis respectively. More precisely, all segment endpoints of a robot's path



(a) CS for robots p and q (b) CS for robots q and s (c) CS for robots p and s

Figure 4.7: Pairwise CS representation for the deadlock scenario shown in Figure 4.5(a). Each axis represents the length of the associated robot path. The dashed lines indicate where the path ends, that is, all paths start at $(0, 0)$ (gray marker) and end at the green squared marker in the upper right. Pairwise intersections are visualized by their associated red rectangles whose boundaries are given by the halt and release progresses of the associated robots (if existing).

(polyline) are projected onto its axis (in Figure 4.7(a): p onto the horizontal and q onto the vertical axis) so that distances are being preserved and any location on that axis represents a position on that robot's path w. r. t. its starting point (indicated by the gray squared markers in the lower left at $(0, 0)$). Any 2D coordinate in such a coordination space then represents a specific position of *both* robots on their paths. In Figure 4.7, the (axis-parallel) dashed lines at the end of the axes represent the entire path length, i. e., the goal of the respective robot. The green squared marker in the upper right therefore represents the goal of both robots. Thus, locations with negative coordinates or coordinates above (ordinate) and on the right of (abscissa) the dashed lines are invalid because robots are assumed to just move inside their paths.

Because a pairwise intersection has associated halts and releases according to Equations (4.11) and (4.12), it can be represented in the CS by so-called *collision rectangles*. For instance, the halt $h(I_{p,q}, q)$ and the release progress $r(I_{p,q}, q)$ in Figure 4.7(a) specify points on the associated robot path q (on the ordinate) as indicated by the two dotted horizontal lines. Both $h(I_{p,q}, q)$ and $r(I_{p,q}, q)$ define an interval on the q-axis which equals the height of the collision rectangle (likewise for the p-axis in Figure 4.7(c) representing the collision rectangle's width). If a halt or release does not exist, the associated collision rectangle boundary does not exist and strives towards infinity as exemplary shown in Figure 4.7(b) where q is never releasing $I_{q,s}$; that is, $r(I_{q,s}, q) = \infty$ and the right boundary of its collision rectangle does not exist, striving to positive infinity on the q-axis. The CS allows for a useful visualization and intuition of pairwise intersections by representing intersections as collision rectangles.[2]

[2]It also provides the foundation of the Incremental Coordination-Space Path Scheduler algorithm in Section 4.4 which solely operates in CS.

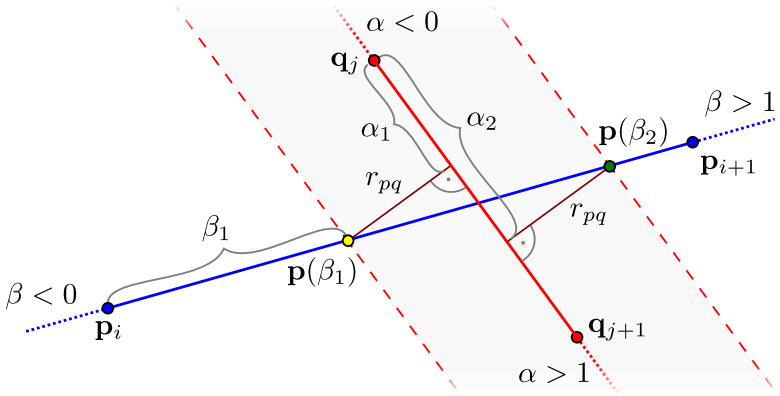


Figure 4.8: Visualization of the inputs (line segments $\mathcal{S}_p = \overline{\boldsymbol{p}_i\boldsymbol{p}_{i+1}}$, $\mathcal{S}_q = \overline{\boldsymbol{q}_j\boldsymbol{q}_{j+1}}$, sum of radii r_{pq}) as part of two given robot paths for the segment-hull intersection problem in search of the points $\boldsymbol{p}(\beta_k)$ on segment $\mathcal{S}_p$ (blue) having exactly a distance of r_{pq} (black) to segment $\mathcal{S}_q$ (red). The actual solution depends on how the two segments are located to each other and the value of r_{pq}. This example has two valid solutions $\boldsymbol{p}(\beta_1)$ and $\boldsymbol{p}(\beta_2)$ which represent the halt (yellow) and release points (green) for p respectively. Here, $\beta_k, \alpha_k \in [0, 1], \forall k \in \{1, 2\}$ (i. e., corresponds to case (i) as explained subsequently).

4.3 Conflict Detection

Within this section, two new approaches for computing pairwise intersections and their associated halt and release points (cf. Section 4.2.1) are presented. After presenting the fundamentals in Section 4.3.1 for both approaches, the first algorithm, named *Smallest Guarded Segments* (SGS) is explained in Section 4.3.2 which aims at finding the smallest possible intersections based on a segment level. Section 4.3.3 then continues with the *Merged Guarded Subpaths* (MGS) algorithm which aims at merging adjacent intersections to create a smaller number of larger connected intersections. Both algorithms are then compared in Section 4.3.4.

4.3.1 Segment-Hull Intersections

A fundamental issue that arises in both algorithms presented in Sections 4.3.2 and 4.3.3 is the *segment-hull intersection problem* which serves as the foundation for computing halt and release progresses. That is, given two line segments $\mathcal{S}_p \coloneqq \overline{\boldsymbol{p}_i\boldsymbol{p}_{i+1}}$, $\mathcal{S}_q \coloneqq \overline{\boldsymbol{q}_j\boldsymbol{q}_{j+1}}$ from robot paths $\mathcal{P}$, $\mathcal{Q}$ and the sum r_{pq} of both radii, we are interested in finding the (locally parameterized) intersections $\beta_k \in [0, 1]$ on the first segment $\mathcal{S}_p$ with the hull of the second segment $\mathcal{S}_q$ whereby the hull is obtained by sliding a circle with radius r_{pq} along the second segment $\mathcal{S}_q$ and taking the convex hull of the union of all such circles. Note that this problem is not commutative.

4.3.1.1 Derivation of a Closed-Form Solution

We will now deduce the closed-form solution for this problem, i. e., for finding all parameterized points on an infinite line given by segment $\mathcal{S}_p$ with fixed distance r_{pq} to another line defined by segment $\mathcal{S}_q$. This is sketched in Figure 4.8.

Therefore, let the coefficient $\beta \in [0, 1] \subset \mathbb{R}$ denote a point

$$\boldsymbol{p}(\beta) \coloneqq \boldsymbol{p}_i + \beta \cdot (\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) \in \mathbb{R}^2 \tag{4.15}$$

on $\mathcal{S}_p$ (linear interpolation) and, based on $\boldsymbol{p}(\beta)$, let

$$\alpha(\beta) \coloneqq \frac{(\boldsymbol{p}(\beta) - \boldsymbol{q}_j) * (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)}{\operatorname{dist}(\boldsymbol{q}_{j+1}, \boldsymbol{q}_j)^2} \in \mathbb{R} \tag{4.16}$$

denote the parameter for the corresponding point $\boldsymbol{q}_\alpha$ on $\mathcal{S}_q$ if $\alpha(\beta) \in [0, 1]$ whereby

$$\operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}) \coloneqq \|\boldsymbol{p} - \boldsymbol{q}\| = \sqrt{\left(p^{(x)} - q^{(x)}\right)^2 + \left(p^{(y)} - q^{(y)}\right)^2} \in \mathbb{R}_{\geq 0}. \tag{4.17}$$

Equation (4.16) is deduced as follows. Again, refer to Figure 4.8 for a visualization of these terms (whereby $\alpha_1 \coloneqq \alpha(\beta_1)$ and $\alpha_2 \coloneqq \alpha(\beta_2)$ for brevity). First, we can express an arbitrary point $\boldsymbol{q}_\alpha \in \mathcal{S}_q$ as

$$\boldsymbol{q}_\alpha = \boldsymbol{q}_j + \alpha \cdot (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)\,. \tag{4.18}$$

Since the line through that point $\boldsymbol{q}_\alpha$ and the given point $\boldsymbol{p}(\beta)$ must be orthogonal to $\mathcal{S}_q$, its dot product must be zero:

$$(\boldsymbol{q}_{j+1} - \boldsymbol{q}_j) * (\boldsymbol{q}_\alpha - \boldsymbol{p}(\beta)) \stackrel{!}{=} 0. \tag{4.19}$$

By substituting Equation (4.18) into (4.19) and rearranging, we obtain Equation (4.16):

$$
\begin{aligned}
& (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j) * (\boldsymbol{q}_j + \alpha \cdot (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j) - \boldsymbol{p}(\beta)) = 0 && (4.20)\\
\iff & \left(q_{j+1}^{(x)} - q_j^{(x)}\right) \cdot \left(q_j^{(x)} + \alpha \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) - p(\beta)^{(x)}\right) + && (4.21)\\
& \left(q_{j+1}^{(y)} - q_j^{(y)}\right) \cdot \left(q_j^{(y)} + \alpha \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) - p(\beta)^{(y)}\right) = 0 && (4.22)\\
\iff & \left(q_j^{(x)} - p(\beta)^{(x)}\right) \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) + \alpha \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) + && (4.23)\\
& \left(q_j^{(y)} - p(\beta)^{(y)}\right) \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) + \alpha \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) = 0 && (4.24)\\
\iff & \alpha \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) + \alpha \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) = && (4.25)\\
& - \left(q_j^{(x)} - p(\beta)^{(x)}\right) \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) - \left(q_j^{(y)} - p(\beta)^{(y)}\right) \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) && (4.26)\\
\iff & - \left(q_j^{(x)} - p(\beta)^{(x)}\right) \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) - \left(q_j^{(y)} - p(\beta)^{(y)}\right) \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right) && (4.27)\\
& = \alpha \cdot \left(\left(q_{j+1}^{(x)} - q_j^{(x)}\right)^2 + \left(q_{j+1}^{(y)} - q_j^{(y)}\right)^2\right) && (4.28)\\
\implies & \alpha = \frac{\left(p(\beta)^{(x)} - q_j^{(x)}\right) \cdot \left(q_{j+1}^{(x)} - q_j^{(x)}\right) + \left(p(\beta)^{(y)} - q_j^{(y)}\right) \cdot \left(q_{j+1}^{(y)} - q_j^{(y)}\right)}{\left(q_{j+1}^{(x)} - q_j^{(x)}\right)^2 + \left(q_{j+1}^{(y)} - q_j^{(y)}\right)^2} && (4.29)\\
& = \frac{(\boldsymbol{p}(\beta) - \boldsymbol{q}_j) * (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)}{\mathrm{dist}(\boldsymbol{q}_{j+1}, \boldsymbol{q}_j)^2}. && (4.30)
\end{aligned}
$$

With these definitions, we can specify the equation for finding all points $\boldsymbol{p}(\beta)$ on segment $\mathcal{S}_p$ that have an Euclidean distance of exactly $r_{pq} \in \mathbb{R}_{>0}$ to its corresponding point $\boldsymbol{q}_\alpha$ on $\mathcal{S}_q$ (see Figure 4.8):

$$
\mathrm{dist}\left(\boldsymbol{p}(\beta), \boldsymbol{q}_j + \alpha(\beta) \cdot (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)\right) \stackrel{!}{=} r_{pq}. \tag{4.31}
$$

Solving this equation for β yields all solution points on $\mathcal{S}_p$ according to Equation (4.15); note that α is defined using β.

Using the definition of dist() (cf. Equation (4.17)) and rearranging terms appropriately, we obtain:

$$\iff \|\boldsymbol{p}(\beta) - (\boldsymbol{q}_j + \alpha(\beta) \cdot (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j))\| = r_{pq} \tag{4.32}$$

$$\iff \|\boldsymbol{p}(\beta) - \boldsymbol{q}_j - \alpha(\beta) \cdot (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)\| = r_{pq} \tag{4.33}$$

$$\iff \left\| \boldsymbol{p}_i + \beta \cdot (\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) - \boldsymbol{q}_j - \frac{\overbrace{(\boldsymbol{p}_i + \beta \cdot (\boldsymbol{p}_{i+1} - \boldsymbol{p}_i) - \boldsymbol{q}_j) * (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)}^{\gamma}}{\|\boldsymbol{q}_{j+1} - \boldsymbol{q}_j\|^2} \right\| = r_{pq} \tag{4.34}$$

$$\iff \left\| \begin{pmatrix} p_i^{(x)} + \beta p_{i+1}^{(x)} - \beta p_i^{(x)} - q_j^{(x)} - \gamma q_{j+1}^{(x)} + \gamma q_j^{(x)} \\ p_i^{(y)} + \beta p_{i+1}^{(y)} - \beta p_i^{(y)} - q_j^{(y)} - \gamma q_{j+1}^{(y)} + \gamma q_j^{(y)} \end{pmatrix} \right\| = r_{pq} \tag{4.35}$$

$$\iff \sqrt{\begin{array}{l} \left(p_i^{(x)} + \beta p_{i+1}^{(x)} - \beta p_i^{(x)} - q_j^{(x)} - \gamma q_{j+1}^{(x)} + \gamma q_j^{(x)}\right)^2 + \\ \left(p_i^{(y)} + \beta p_{i+1}^{(y)} - \beta p_i^{(y)} - q_j^{(y)} - \gamma q_{j+1}^{(y)} + \gamma q_j^{(y)}\right)^2 \end{array}} = r_{pq} \tag{4.36}$$

$$\begin{aligned} \overset{r_{pq}>0}{\iff} & \left(p_i^{(x)} + \beta p_{i+1}^{(x)} - \beta p_i^{(x)} - q_j^{(x)} - \gamma q_{j+1}^{(x)} + \gamma q_j^{(x)}\right)^2 + \\ & \left(p_i^{(y)} + \beta p_{i+1}^{(y)} - \beta p_i^{(y)} - q_j^{(y)} - \gamma q_{j+1}^{(y)} + \gamma q_j^{(y)}\right)^2 - r_{pq}^2 = 0 \end{aligned} \tag{4.37}$$

$$\begin{aligned} \iff & \left(p_i^{(x)} + \beta \Delta p^{(x)} - q_j^{(x)} - \gamma \Delta q^{(x)}\right)^2 + \\ & \left(p_i^{(y)} + \beta \Delta p^{(y)} - q_j^{(y)} - \gamma \left(q_{j+1}^{(y)} - q_j^{(y)}\right)\right)^2 - r_{pq}^2 = 0 \end{aligned} \tag{4.38}$$

with

$$\Delta \boldsymbol{p} \coloneqq \begin{pmatrix} \Delta p^{(x)} \\ \Delta p^{(y)} \end{pmatrix} = \begin{pmatrix} p_{j+1}^{(x)} - p_j^{(x)} \\ p_{j+1}^{(y)} - p_j^{(y)} \end{pmatrix} \tag{4.39}$$

and likewise for $\Delta \boldsymbol{q}$. To continue solving Equation (4.38), we rearrange γ as follows in order to isolate constants c_i (not depending on β):

$$\gamma \coloneqq \frac{\left(p_i^{(x)} + \beta \Delta p^{(x)} - q_j^{(x)}\right) \cdot \Delta q^{(x)} + \left(p_i^{(y)} + \beta \Delta p^{(y)} - q_j^{(y)}\right) \cdot \Delta q^{(y)}}{{\Delta q^{(x)}}^2 + {\Delta q^{(y)}}^2} \tag{4.40}$$

$$= \frac{\beta \cdot \overbrace{(\Delta \boldsymbol{p} * \Delta \boldsymbol{q})}^{c_1} + \overbrace{\Delta q^{(x)} p_i^{(x)} - \Delta q^{(x)} q_j^{(x)} + \Delta q^{(y)} p_i^{(y)} - \Delta q^{(y)} q_j^{(y)}}^{c_2}}{\underbrace{\|\Delta \boldsymbol{q}\|^2}_{c_3}} \tag{4.41}$$

$$= \frac{\beta c_1 + c_2}{c_3}. \tag{4.42}$$

Substituting these abbreviations, we can further rearrange Equation (4.38):

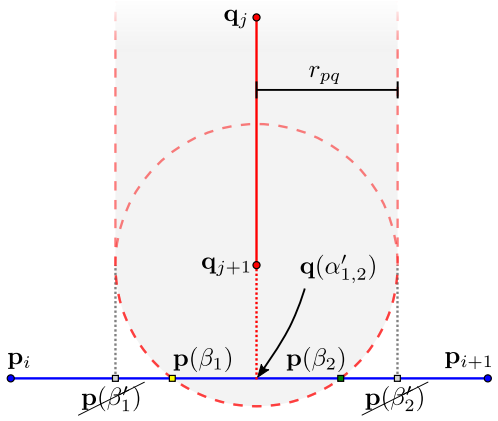
$$\begin{aligned}
&\iff \left(p_i^{(x)} + \beta\Delta p^{(x)} - q_j^{(x)} - \frac{\beta c_1 \Delta q^{(x)} + c_2 \Delta q^{(x)}}{c_3}\right)^2 + \\
&\quad \left(p_i^{(y)} + \beta\Delta p^{(y)} - q_j^{(y)} - \frac{\beta c_1 \Delta q^{(y)} + c_2 \Delta q^{(y)}}{c_3}\right)^2 - {r_{pq}}^2 = 0 \qquad (4.43)\\
&\iff \left(\beta \overbrace{\left(\Delta p^{(x)} - \frac{c_1 \Delta q^{(x)}}{c_3}\right)}^{c_4} + \overbrace{p_i^{(x)} - q_j^{(x)} - \frac{c_2 \Delta q^{(x)}}{c_3}}^{c_5}\right)^2 + \\
&\quad \left(\beta \underbrace{\left(\Delta p^{(y)} - \frac{c_1 \Delta q^{(y)}}{c_3}\right)}_{c_6} + \underbrace{p_i^{(y)} - q_j^{(y)} - \frac{c_2 \Delta q^{(y)}}{c_3}}_{c_7}\right)^2 - {r_{pq}}^2 = 0 \qquad (4.44)\\
&\iff (\beta c_4 + c_5)^2 + (\beta c_6 + c_7)^2 - {r_{pq}}^2 = 0 \qquad (4.45)\\
&\iff \underbrace{(c_4 + c_5)}_{a} \cdot \beta^2 + \underbrace{(2c_4c_5 + 2c_6c_7)}_{b} \cdot \beta + \underbrace{{c_5}^2 + {c_7}^2 - {r_{pq}}^2}_{c} = 0 \qquad (4.46)
\end{aligned}$$

which is the standard form of a quadratic equation $a\beta^2 + b\beta + c = 0$, having up to two solutions

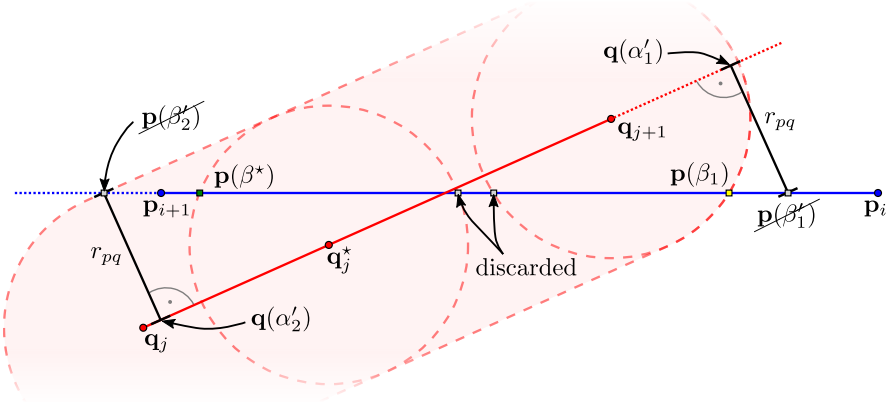
$$\beta_{1,2} = \frac{-(2c_4c_5 + 2c_6c_7) \pm \sqrt{D}}{2\left({c_4}^2 + {c_6}^2\right)} \qquad (4.47)$$

based on its discriminant $D := (2c_4c_5 + 2c_6c_7)^2 - 4\left({c_4}^2 + {c_6}^2\right) \cdot \left({c_5}^2 + {c_7}^2 - {r_{pq}}^2\right)$. The $\beta_{1,2}$ are a (local) parameterization of the associated points on the given segment $\mathcal{S}_p$ while the corresponding $\alpha_{1,2}$ parameterize the associated points on $\mathcal{S}_q$. However, this only holds true if such an ideal situation is given as depicted in Figure 4.8 where $\beta_{1,2}, \alpha_{1,2} \in [0, 1]$. In practice, $\mathcal{S}_p$ and $\mathcal{S}_q$ can be located arbitrarily to each other so that the derivation for computing $\beta_{1,2}$ may not be applicable.

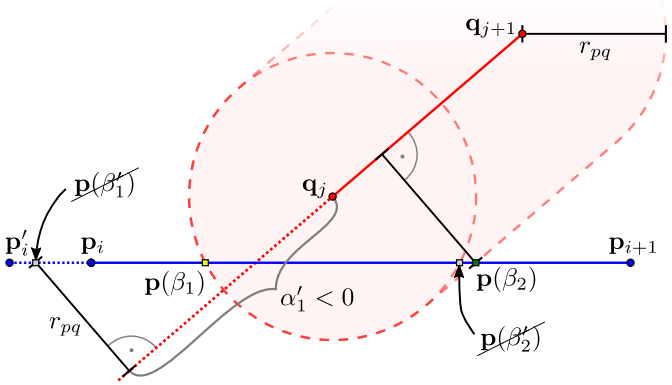
To simplify the subsequent case analysis, we assume $\text{dist}(\mathcal{S}_p, \mathcal{S}_q) < r_{pq}$ because if $\text{dist}(\mathcal{S}_p, \mathcal{S}_q) \geq r_{pq}$, there are no conflicting points according to Equation (4.6) anyway and the segments can be ignored. The case analysis yields an algorithm to compute a solution for the introduced segment-hull intersection problem which can be outlined as follows: first, if $\mathcal{S}_p$ and $\mathcal{S}_q$ are non-parallel, Equation (4.47) is evaluated and the resulting β_k and corresponding α_k are filtered based on whether they are inside or outside the $[0, 1]$-range. Second, if the input segments are parallel, circle-segment intersections must be computed because Equation (4.47) is not applicable. This also yields β_k-values describing the locations of the intersection of a given circle with the segment $\mathcal{S}_p$. Third and finally, the set $\mathcal{B}$ of all determined β_k undergoes a final postprocessing to only retain valid solutions.



(a) Case (ii): $\beta_{1,2}, \beta'_{1,2} \in [0,1]$ and $\alpha'_{1,2} \notin [0,1]$ (circle-segment intersection: $\beta_{1,2}$)



(b) Case (iii): $\beta'_2 \notin [0,1]$ and $\alpha'_2 \in [0,1]$ (β'_2 is invalid, no solution for the left part)



(c) Case (iv): $\beta'_1 \notin [0,1]$ and $\alpha'_1 \notin [0,1]$ (circle-segment intersection: $\beta_{1,2}$)

Figure 4.9: Schematic of all possible cases for the computation of $\beta_{1,2}$ according to Equation (4.47) if input segments $\mathcal{S}_p = \overline{\boldsymbol{p}_i \boldsymbol{p}_{i+1}}$ and $\mathcal{S}_q = \overline{\boldsymbol{q}_j \boldsymbol{q}_{j+1}}$ are neither parallel nor equal to each other. Note that all illustrations refer to locations ($\beta_{1,2}$) computed on the segment $\mathcal{S}_p$ only. The figures assume that $\mathcal{S}_p$ and $\mathcal{S}_q$ have a distance less than r_{pq}.

4.3.1.2 Non-parallel Segments

First, let us consider the case where $\mathcal{S}_p$ is not parallel or equal to $\mathcal{S}_q$ as visualized by Figure 4.9. Generally, the discriminant D in Equation (4.47) must be ≥ 0 because it would only be < 0 if there is no solution at all which applies only if $\text{dist}(\mathcal{S}_p, \mathcal{S}_q) \geq r_{pq}$. Additionally, $D = 0$ is impossible as well because mathematically, the deduced formula in Equation (4.47) assumes infinite lines, not (finite) segments and non-parallel lines will always yield two solutions $\beta_{1,2}$ (although not necessarily in $[0, 1]$).

We can therefore focus on $D > 0$ which yields the following cases when computing β_k and its corresponding α_k, $k \in \{1, 2\}$, according to Equations (4.47) and (4.16):

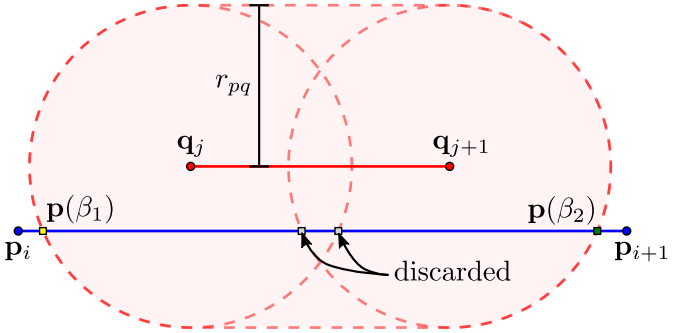
(i) $\beta_k \in [0, 1] \wedge \alpha_k \in [0, 1]$: This is the simple case which has already been depicted in Figure 4.8. It also applies for β_2 in Figure 4.9(c). Because both parameters are valid, β_k is added to $\mathcal{B}$ for further processing as outlined below.

(ii) $\beta_k \in [0, 1] \wedge \alpha_k \notin [0, 1]$: This case is visualized by Figure 4.9(a). All β-parameters on $\mathcal{S}_p$ are valid but all α-parameters on $\mathcal{S}_q$ are outside $[0, 1]$. This case shows that it is insufficient to only consider the β values. To find the correct intersection points, a *circle-segment intersection* must be computed with radius r_{pq} and against segment $\mathcal{S}_p$. More specifically, if $\alpha_k < 0$ we know that the intersection is located somewhere before $\boldsymbol{q}_j$ (as in Figure 4.9(c) for α'_1) and if $\alpha_k > 1$, we are certain that the intersection is behind $\boldsymbol{q}_{j+1}$ (as in Figure 4.9(a) for $\alpha'_{1,2}$). We therefore select the circle's center to be $\boldsymbol{q}_j$ if $\alpha_k < 0$ and $\boldsymbol{q}_{j+1}$ otherwise. However, for Figure 4.9(a), this would cause two identical circle-segment intersection tests due to $\alpha'_{1,2} > 1$. Evaluating the signum of the involved α-values avoids performing these computations twice, requiring $\text{sign}(\alpha_1) \neq \text{sign}(\alpha_2)$. Basically, this case adds all resulting circle-segment intersections to $\mathcal{B}$ for further processing.

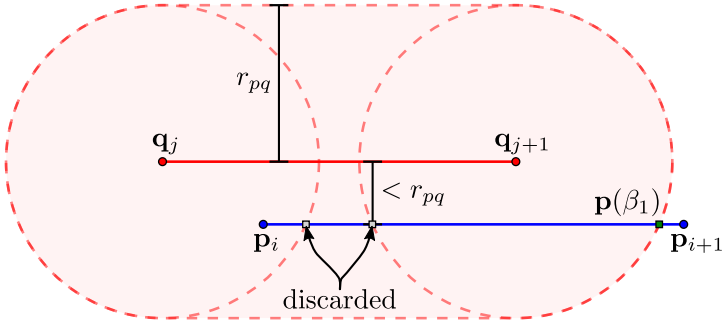
(iii) $\beta_k \notin [0, 1] \wedge \alpha_k \in [0, 1]$: The lower left part of Figure 4.9(b) visualizes this case (via β'_2, α'_2) when focusing on $\mathcal{S}_q = \overline{\boldsymbol{q}_j \boldsymbol{q}_{j+1}}$ (i. e., ignoring $\boldsymbol{q}^\star_j$ and the entire right side next to $\boldsymbol{q}_{j+1}$ and $\boldsymbol{p}_i$ for this case because that is discussed later). It becomes obvious that there is no valid solution β because $\mathcal{S}_p$ (blue) is completely contained in the r_{pq}-hull of $\mathcal{S}_q$ (red).

(iv) $\beta_k \notin [0, 1] \wedge \alpha_k \notin [0, 1]$: This case is shown in Figure 4.9(c) which is somewhat equal to case (ii) for β_1 requiring a circle-segment intersection as well. A subtle difference is that the endpoints $\boldsymbol{q}_j$ and $\boldsymbol{q}_{j+1}$ are swapped w. r. t. $\mathcal{S}_p$ so that the circle is centered around $\boldsymbol{q}_j$ here. Like in case (ii), the circle-segment intersections are added to $\mathcal{B}$.

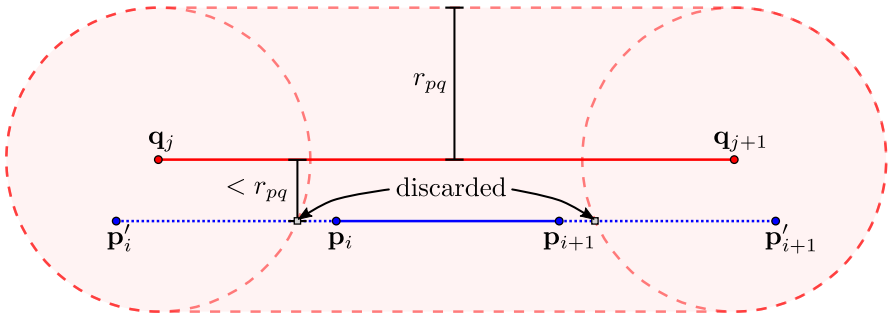
As already implied by the gray squared markers in Figure 4.9, some invalid β_k might have been added to $\mathcal{B}$. Their removal is discussed after the parallel case since it is required therein as well.



(a) Parallel case with two valid and two invalid solutions; the two solutions labeled as "discarded" will be removed by non-min/max suppression. The two invalid solutions in the middle arise from the two circle-segment intersection tests around $\boldsymbol{q}_j$ and $\boldsymbol{q}_{j+1}$ against the segment $\mathcal{S}_p$ (blue) having four solutions here.



(b) Parallel case with only one valid and two invalid solutions; non-min/max suppression is insufficient here showing the necessity of r_{pq}-equality filtering. This schematic also represents the case where a β would be found next to $\boldsymbol{p}_i$, i. e., as an entering into the r_{pq}-sized $\mathcal{S}_q$ hull (red) if $\boldsymbol{p}_i$ would be located further left (like in (a)).



(c) Parallel case without ($\mathcal{S}_p = \overline{\boldsymbol{p}_i\boldsymbol{p}_{i+1}}$) or just invalid ($\mathcal{S}_p = \overline{\boldsymbol{p}'_i\boldsymbol{p}'_{i+1}}$) solutions only. Clearly, both solutions (labeled as "discarded") would only be rejected by r_{pq}-equality filtering as indicated by the "$< r_{pq}$". This situation occurs if $\mathcal{S}_p$ (blue) is fully contained in the r_{pq}-sized hull of $\mathcal{S}_q$ (red) and there is no entering or exiting of $\mathcal{S}_p$ into that hull.

Figure 4.10: Cases for the computation of $\beta_{1,2}$ if input segments $\mathcal{S}_p$ and $\mathcal{S}_q$ are parallel or equal to each other. If one of these cases applies, Equation (4.47) is *not* applicable. Note that all illustrations refer to locations ($\beta_{1,2}$) computed on the segment $\mathcal{S}_p$ only.

4.3.1.3 Parallel Segments

Let us therefore now assume that $\mathcal{S}_p$ is parallel or equal to $\mathcal{S}_q$. As indicated by Figure 4.10, the parameterized intersections of the two circles centered around $\boldsymbol{q}_j$ and $\boldsymbol{q}_{j+1}$ with radius r_{pq} are computed against segment $\mathcal{S}_p$. This results in two (see Figure 4.10(a)), only one (see Figure 4.10(b)) or no valid solution (see Figure 4.10(c)) for β_k to be added to $\mathcal{B}$ based on the segments' lengths and how they are located to each other. Algorithmically, all circle-segment intersections are added to $\mathcal{B}$ preliminarily.

Thus far, the set $\mathcal{B}$ is composed of all β_k computed by distinguishing parallel vs. non-parallel input segments, and in the latter case, by further distinguishing between the cases (i) to (iv).

4.3.1.4 Postprocessing

The required circle-segment intersection tests from the parallel and non-parallel cases have also caused "invalid" β_k to be added to $\mathcal{B}$ (visualized via gray squared markers in Figures 4.9 and 4.10). A β_k is considered invalid if the following *invalidity property* holds (cf. Equation (4.16)):

$$\operatorname{dist}(\boldsymbol{p}(\beta_k), \mathcal{S}_q) \neq r_{pq} \tag{4.48}$$

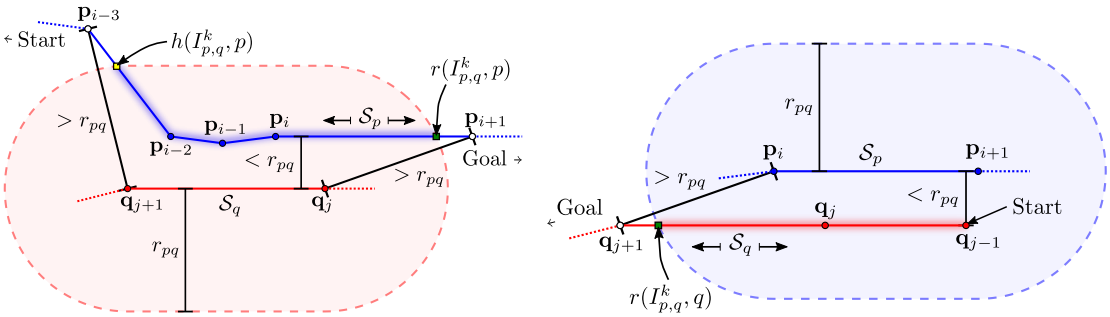
with

$$\operatorname{dist}(\boldsymbol{p}, \mathcal{S}_q) \coloneqq \begin{cases} \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}_j) & \text{if } \alpha < 0 \\ \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}_{j+1}) & \text{if } \alpha > 1 \\ \operatorname{dist}(\boldsymbol{p}, \boldsymbol{q}_j + \alpha\,(\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)) & \text{otherwise,} \end{cases} \tag{4.49}$$

$$\alpha = \frac{(\boldsymbol{p} - \boldsymbol{q}_j) * (\boldsymbol{q}_{j+1} - \boldsymbol{q}_j)}{\operatorname{dist}(\boldsymbol{q}_j, \boldsymbol{q}_{j+1})^2} \text{ and } \mathcal{S}_q = \overline{\boldsymbol{q}_j \boldsymbol{q}_{j+1}}. \tag{4.50}$$

The following two-stage filtering is proposed to efficiently remove them from $\mathcal{B}$. First, a *non-min/max suppression* only retains the smallest and largest β_k from $\mathcal{B}$ if $|\mathcal{B}| > 2$. Second, all β_k are removed from $\mathcal{B}$ not satisfying Equation (4.48), termed r_{pq}*-equality filtering*. Note that non-min/max suppression is always applied first to already remove as many invalid β_k values as possible because it is faster since it only requires comparisons of scalars. r_{pq}-equality filtering would be sufficient on its own but it is slower due to the involved distance computations.

For instance, in the absence of $\boldsymbol{q}_j^\star$ in Figure 4.9(b), the only (rightmost) gray marker is removed by r_{pq}-equality filtering. However, if we assume that $\mathcal{S}_q$ starts at $\boldsymbol{q}_j^\star$ (not at $\boldsymbol{q}_j$), both gray markers are already rejected more efficiently by non-min/max suppression because $\beta^\star$ and β_1 are the valid (retained) extremes. Similarly, in Figure 4.9(c), the gray marker at $\boldsymbol{p}(\beta_2')$ is removed by non-min/max suppression as well. For the parallel cases in Figure 4.10, both gray markers in (a) and the rightmost one in (b) is rejected by non-min/max suppression while the remaining ones are rejected by r_{pq}-equality filtering.



(a) Perspective of robot p (blue) on the intersection with interval $[p^k_{\min}, p^k_{\max}] = [i-3+0.35, i+0.82]$ (highlighted) for $\mathcal{S}_p$ against $\mathcal{S}_q$

(b) Perspective of robot q (red) on the intersection with interval $[q^k_{\min}, q^k_{\max}] = [0, 1.81]$ (highlighted) for $\mathcal{S}_q$ against $\mathcal{S}_p$ (since $j = 1$ due to "Start")

Figure 4.11: Strategy of how the SGS algorithm moves along a path both towards the start and goal to identify a pairwise intersection $I^k_{p,q} = ([p^k_{\min}, p^k_{\max}], [q^k_{\min}, q^k_{\max}])$ along with the halts and releases (if any) for both robots based on two conflicting segments $\mathcal{S}_p$, $\mathcal{S}_q$. (a) shows the perspective of robot p (blue) while (b) shows the same for q (red). For example in (a), both the halt $h(I^k_{p,q}, p) = p^k_{\min}$ and release $r(I^k_{p,q}, p) = p^k_{\max}$ exist which "protect" the segment $\mathcal{S}_p = \overline{\boldsymbol{p}_i \boldsymbol{p}_{i+1}}$ of interest against $\mathcal{S}_q = \overline{\boldsymbol{q}_j \boldsymbol{q}_{j+1}}$. Note that other segments of the path $\mathcal{Q}$ in (a) are *not* relevant for the computation here.

This postprocessing ensures that $|\mathcal{B}| \leq 2$. As for the final output, the ordered set

$$\mathcal{B}^\star = \langle i + \beta_k \mid \forall \beta_k \in \mathcal{B} \rangle \tag{4.51}$$

is returned by the algorithm, converting all segment-local parameters β_k to a path-global parameterization (see also Equation (4.5)).

4.3.2 Smallest Guarded Segments

Within this section, the *Smallest Guarded Segments* (SGS) algorithm is presented which solves the problem described in Section 4.2.1, that is, given two conflicting input paths $\mathcal{P}$ and $\mathcal{Q}$, determine all pairwise intersections according to Equation (4.7) along with their halt and releases (cf. Equations (4.11) and (4.12)). It is based on the solution of the segment-hull intersection problem, see Section 4.3.1, and its output serves as the input of the solver algorithms for computing a valid schedule (cf. Section 4.2.2). The proposed approach identifies the smallest possible pairwise intersections based on a segment-level while still ensuring validity according to the properties (4.8) to (4.10).

Figure 4.11 illustrates the underlying idea of this algorithm. For all pairs $\mathcal{S}_p = \overline{\boldsymbol{p}_i, \boldsymbol{p}_{i+1}}$, $\mathcal{S}_q = \overline{\boldsymbol{q}_j, \boldsymbol{q}_{j+1}}$ of conflicting segments, i. e., $\operatorname{dist}(\mathcal{S}_p, \mathcal{S}_q) < r_{pq}$ with

$$\operatorname{dist}(\mathcal{S}_p, \mathcal{S}_q) := \begin{cases} 0 & \text{if } \mathcal{S}_p \cap \mathcal{S}_q \neq \emptyset \\ \min\{\operatorname{dist}(\boldsymbol{p}_i, \mathcal{S}_q), \operatorname{dist}(\boldsymbol{p}_{i+1}, \mathcal{S}_q), & \\ \quad \operatorname{dist}(\boldsymbol{q}_j, \mathcal{S}_p), \operatorname{dist}(\boldsymbol{q}_{j+1}, \mathcal{S}_p)\} & \text{otherwise,} \end{cases} \tag{4.52}$$

it follows the path $\mathcal{P}$ towards the robot's initial start until it encounters the first *entering segment* having a starting point $\boldsymbol{p}_k$, $k \leq i$, with $\text{dist}(\boldsymbol{p}_k, \mathcal{S}_q) \geq r_{pq}$. In Figure 4.11(a), this is $\boldsymbol{p}_{i-3}$ (white) and the entering segment is therefore $\overline{\boldsymbol{p}_{i-3}\boldsymbol{p}_{i-2}}$. Similarly, it also follows $\mathcal{P}$ towards the robot's goal until it encounters the first *leaving segment* having an endpoint $\boldsymbol{p}_k$, $k \geq i$, with $\text{dist}(\boldsymbol{p}_k, \mathcal{S}_q) \geq r_{pq}$. Again in Figure 4.11(a), this is $\boldsymbol{p}_{i+1}$ (white) and the leaving segment is $\mathcal{S}_p$ itself. Such segments (if any) are then used as the input for the segment-hull intersection problem which defines the interval boundaries $[p^k_{\min}, p^k_{\max}]$ for this specific pairwise intersection $I^k_{p,q}$. If the entering segment does not exist, the left interval boundary is set to the progress start, $p^k_{\min} = 0$. Likewise, if the leaving segment does not exist, the right boundary is set to the progress of the goal, $p^k_{\max} = N_p - 1$. As depicted in Figure 4.11(a), if such segments exist, the interval boundaries also equal the halt $h(I^k_{p,q}, p)$ (yellow) and release $r(I^k_{p,q}, p)$ (green) respectively, according to Equations (4.11) and (4.12). Notice that this only computes the guards on path $\mathcal{P}$. To compute the associated interval $[q^k_{\min}, q^k_{\max}]$, the roles of p and q are swapped, i. e., the r_{pq}-hull of $\mathcal{S}_p$ is intersected with segments of path $\mathcal{Q}$, see Figure 4.11(b) (with $j = 1$). On the right side, $\mathcal{Q}$ (red) starts inside the r_{pq}-hull of $\mathcal{S}_p$ (blue) so that no halt exists for robot q and this particular intersection, thus $q^k_{\min} \neq h(I^k_{p,q}, q)$.

More details of the SGS algorithm are given in Algorithm 4.1. After checking for the corner case of having robots that are already at their goals and having a distance less than r_{pq} in Lines 2-4, the algorithm first accounts for robots that have already moved previously. This will be explained within the context of Chapter 5 but for now it is sufficient to know that robots can be placed anywhere on their current paths. Such starting positions are denoted as σ_p and σ_q for p and q respectively. Because all progress values are a *global* parameterization of locations on the paths, the algorithm first updates the input paths of both robots in Line 5. For instance w. r. t. $\mathcal{P}$, the location (x, y) at index $\lfloor \sigma_p \rfloor$ is replaced with $\rho^{-1}(\sigma_p) \in \mathbb{R}^2$ which effectively shifts the start of that segment to the robot's actual start. These are just temporary modifications and reverted in Line 22. In the following, if any intersection guards are located on the updated start segments $\mathcal{S}_{\lfloor \sigma_p \rfloor}$ or $\mathcal{S}_{\lfloor \sigma_q \rfloor}$, they must be remapped to account for the (global) parameterization of the original (reverted) paths. For instance, if $\left\lfloor h(I^k_{p,q}, q) \right\rfloor = \lfloor \sigma_p \rfloor$ (remapping necessary), the halt point $h(I^k_{p,q}, q)$ must be recomputed as follows (for dist(), see Equation (4.17)):

$$h(I^k_{p,q}, q) = \frac{\text{dist}(\boldsymbol{w}, \boldsymbol{u})}{\text{dist}(\boldsymbol{v}, \boldsymbol{u})} + \lfloor \sigma_p \rfloor \quad \text{whereby} \tag{4.53}$$

$$\boldsymbol{u} \coloneqq \boldsymbol{p}^{\text{old}}_{\lfloor \sigma_p \rfloor}, \boldsymbol{v} \coloneqq \boldsymbol{p}_{\lfloor \sigma_p \rfloor + 1} \text{ and } \boldsymbol{w} \coloneqq \rho^{-1}\left(h\left(I^k_{p,q}, q\right)\right). \tag{4.54}$$

Within Equation (4.54), $\boldsymbol{p}^{\text{old}}_{\lfloor \sigma_p \rfloor}$ denotes the previous point in the path $\mathcal{P}$ at index $\lfloor \sigma_p \rfloor$ before UpdatePaths() takes effect (cf. Line 5).

Within the next two loops, the algorithm iterates over all pairs of segments and tests whether a given pair is in conflict (see Line 8). For every conflicting segment, a new intersection $I^k_{p,q}$ is created (cf. Line 9) and the associated halts and releases are computed

Algorithm 4.1 Pseudocode of the Smallest Guarded Segments algorithm requiring the paths $\mathcal{P}$, $\mathcal{Q}$, the start progresses σ_p, σ_q and the sum of both radii r_{pq} as input. The output is the set of all pairwise intersections $\mathcal{I} = \{I^k_{p,q}\}$ between robots p and q including all halt and release points (if any) for every $I^k_{p,q}$.

```
1:  procedure SmallestGuardedSegments(P, Q, σ_p, σ_q, r_pq)
2:      if ⌊σ_p⌋ = N_p − 1 ∧ ⌊σ_q⌋ = N_q − 1 ∧ dist(ρ(σ_p), ρ(σ_q)) < r_pq then
3:          return          ▷ Robots crash already at their starts (equaling their goals)
4:      end if
5:      UpdatePaths(P, σ_p, Q, σ_q)                          ▷ Reflect new starting points
6:      for i ← ⌊σ_p⌋ + 1 to N_p − 1 do        ▷ Loop through all pairs of input segments
7:          for j ← ⌊σ_q⌋ + 1 to N_q − 1 do
8:              if dist(p_{i−1}p_i, q_{j−1}q_j) < r_pq then          ▷ Test for conflicting segments
9:                  I^k_{p,q} ← CreateIntersection(i − 1, j − 1, I)
10:                 ▷ Move towards p's goal and start to find release and halt respectively:
11:                 r(I^k_{p,q}, p) ← TraceForwards(P, Q, i, j, r_pq)
12:                 h(I^k_{p,q}, p) ← TraceBackwards(P, Q, i, j, r_pq)
13:                 ▷ Likewise for q:
14:                 r(I^k_{p,q}, q) ← TraceForwards(Q, P, i, j, r_pq)
15:                 h(I^k_{p,q}, q) ← TraceBackwards(Q, P, i, j, r_pq)
16:             end if
17:         end for
18:     end for
19:     if ⌊σ_p⌋ = N_p − 1 ∨ ⌊σ_q⌋ = N_q − 1 then    ▷ Robot has already reached its goal?
20:         HandleCornerCases(P, Q, σ_p, σ_q, r_pq)
21:     end if
22:     RevertPathsAndRemap(P, Q, {I^k_{p,q}})                ▷ Undo changes from Line 5
23:     return all pairwise intersections {I^k_{p,q}}
24: end procedure
```

for p and q respectively (cf. Lines 11-15). For TraceForwards() in Line 11, we iterate over the path $\mathcal{P}$ towards the goal starting at segment $\overline{\boldsymbol{p}_{i-1}\boldsymbol{p}_i}$, to find the first point $\boldsymbol{p}_k$, $k > i$ (i. e., the starting point of the next segment), satisfying $\text{dist}(\boldsymbol{p}_k, \mathcal{S}_{j-i}) \geq r_{pq}$ ("guarding point" behind the conflict). The two segments $\mathcal{S}_{j-i}, \mathcal{S}_k$ are then fed into the solution for the segment-hull intersection problem as described in Section 4.3.1 which yields a set $\mathcal{B}$ of possible solutions. We return $\max(\mathcal{B})$ as release point because we know there is a release due to $\text{dist}(\boldsymbol{p}_k, \mathcal{S}_{j-i}) \geq r_{pq}$ and its progress must be the maximum among all values in $\mathcal{B}$. Similarly, for the case of TraceBackwards() in Line 12, we start iterating from segment $\mathcal{S}_{i-i}$ to $\mathcal{S}_{\lfloor\sigma_p\rfloor}$ towards the start ($i - 1 \geq \lfloor\sigma_p\rfloor$) to find the first point $\boldsymbol{p}_k$, $k \leq i - 1$ (i. e., the starting point of the previous segment), satisfying $\text{dist}(\boldsymbol{p}_k, \mathcal{S}_{j-i}) \geq r_{pq}$ ("guarding point" before the conflict). After solving the segment-hull intersection problem for $\mathcal{S}_{j-i}, \mathcal{S}_k$, we return $\min(\mathcal{B})$. Note that in both cases, if no such "guarding point" were found, the release and/or halt do not exist.

Finally, the special case is handled where only one of the two robots has already reached its goal (cf. Lines 19-21). The case must be handled separately because the last segment of a robot's path, already being at the goal, degenerates to a single point (that is, the goal itself). Note that in this case, the two nested for loops (Lines 6 and 7) are not entered at all. To identify all pairwise intersections for the special case, HandleCornerCases() traces the segments $\mathcal{S}$ of the non-degenerated path towards the goal as explained before and adds a pairwise intersection for the degenerated segment $\mathcal{S}'$ and every $\mathcal{S}$ if they are in conflict. Afterwards, halts and releases are computed based on a circle-segment intersection test whereby the circle is centered around the (degenerated) goal point with radius r_{pq} and tested against segment $\mathcal{S}$ (cf. Section 4.3.1).

4.3.3 Merged Guarded Subpaths

An obvious disadvantage of the SGS algorithm is that it always creates a huge amount of small pairwise intersections, although they may be close to each other. This raised the idea of merging such adjacent intersections, leading to the proposed *Merged Guarded Subpaths* (MGS) algorithm. It also computes valid pairwise intersections along with their halt and releases but also tries to combine as many close intersections as possible.

At first, this algorithm performs much like the SGS algorithm by determining all pairs of conflicting segments. Afterwards, it tries to merge as many segment pairs as possible based on a specific merge condition. This continues until there are no more segment pairs fulfilling the merge condition. That is, along the way, the two subpaths contained in a pairwise intersections can "grow" (due to a merge) while the total number decreases. Basically, a merge operation combines two intersections to one while effectively reducing the number of intersections by one. Merged intersections therefore always contain at least the same or a larger number of segments compared to the pairwise intersection of the input of the merge.

Algorithm 4.2 shows the pseudocode listing for the algorithm. The Lines 2-5 are equal to the SGS algorithm, see Section 4.3.2, and the corner case handling has been omitted because it is similar to SGS. Next, similiarly to the SGS algorithm, the two nested for loops (Lines 7 and 8) determine all pairwise intersections on a segment level, that is, the index tupel $((i-1,i-1),(j-1,j-1))$ is added to the set $\mathcal{I}$ whereby $(i-1,i-1)$ identifies the segment $\mathcal{S}_{i-1} = \overline{\boldsymbol{p}_{i-1}\boldsymbol{p}_i}$ on $\mathcal{P}$ and $(j-1,j-1)$ identifies the segment $\mathcal{S}_{j-1} = \overline{\boldsymbol{q}_{j-1}\boldsymbol{q}_j}$ on $\mathcal{Q}$. We abbreviate this by writing $((p_\triangleright, p_\triangleleft), (q_\triangleright, q_\triangleleft))$ (see Line 16) whereby $p_\triangleright$ and $q_\triangleright$ denote the indices of the *entering segments* into the intersection for p and q respectively. Similarly, $p_\triangleleft$ and $q_\triangleleft$ denote the *exiting segments* (i. e., the intersecting subpaths are in between "$\triangleright \cdots \triangleleft$"). Note that this is still only index-based, i. e., it is yet unknown what specific parts of the segments are actually part of the intersection.

The set $\mathcal{I}$ serves as the input for the Merge() operation (cf. Line 14) which is detailed in Algorithm 4.3. The outer loop iterates until there are no more merge operations possible inside the inner loop. The nested for loop inspects all possible distinct pairs (I^m, I^n) in the set $\mathcal{I}$ to check whether they can be merged (Lines 4-17). In Lines 6

Algorithm 4.2 Pseudocode of the Merged Guarded Subpaths algorithm requiring the paths $\mathcal{P}$, $\mathcal{Q}$, the start progresses σ_p, σ_q and the sum of both radii r_{pq} as input. The output is the set $\mathcal{I}$ of all pairwise intersections $I^k_{p,q}$ between robots p and q including all halt and release points (if any) for every $I^k_{p,q}$.

1: **procedure** MERGEDGUARDEDSUBPATHS($\mathcal{P}$, $\mathcal{Q}$, σ_p, σ_q, r_{pq})
2: **if** $\lfloor \sigma_p \rfloor = N_p - 1 \wedge \lfloor \sigma_q \rfloor = N_q - 1 \wedge \operatorname{dist}(\rho(\sigma_p), \rho(\sigma_q)) < r_{pq}$ **then**
3: **return** ▷ Robots crash already at their starts (equaling their goals)
4: **end if**
5: UPDATEPATHS($\mathcal{P}$, σ_p, $\mathcal{Q}$, σ_q) ▷ Reflect new starting points
6: $\mathcal{I} \leftarrow \emptyset$
7: **for** $i \leftarrow \lfloor \sigma_p \rfloor + 1$ **to** $N_p - 1$ **do** ▷ Loop through all pairs of input segments
8: **for** $j \leftarrow \lfloor \sigma_q \rfloor + 1$ **to** $N_q - 1$ **do**
9: **if** $\operatorname{dist}(\overline{\boldsymbol{p}_{i-1}\boldsymbol{p}_i}, \overline{\boldsymbol{q}_{j-1}\boldsymbol{q}_j}) < r_{pq}$ **then** ▷ Test for conflicting segments
10: CREATEINTERSECTION($i-1, j-1, \mathcal{I}$)
11: **end if**
12: **end for**
13: **end for**
14: MERGE($\mathcal{I}$, r_{pq}) ▷ See Algorithm 4.3
15: **for all** $I^k_{p,q} \in \mathcal{I}$ **do** ▷ Compute all halts and releases
16: Let $I^k_{p,q} \coloneqq ((p_\triangleright, p_\triangleleft), (q_\triangleright, q_\triangleleft))$
17: **if** EXISTS($h(I^k_{p,q}, p)$) **then**
18: $h(I^k_{p,q}, p) \leftarrow$ COMPUTEHALT($q_\triangleright, q_\triangleleft, p_\triangleright$)
19: **end if**
20: **if** EXISTS($r(I^k_{p,q}, p)$) **then**
21: $r(I^k_{p,q}, p) \leftarrow$ COMPUTERELEASE($q_\triangleright, q_\triangleleft, p_\triangleleft$)
22: **end if**
23: **if** EXISTS($h(I^k_{p,q}, q)$) **then**
24: $h(I^k_{p,q}, q) \leftarrow$ COMPUTEHALT($p_\triangleright, p_\triangleleft, q_\triangleright$)
25: **end if**
26: **if** EXISTS($r(I^k_{p,q}, q)$) **then**
27: $r(I^k_{p,q}, q) \leftarrow$ COMPUTERELEASE($p_\triangleright, p_\triangleleft, q_\triangleleft$)
28: **end if**
29: **end for**
30: REVERTPATHSANDREMAP($\mathcal{P}$, $\mathcal{Q}$, $\mathcal{I}$)
31: **return** $\mathcal{I}$
32: **end procedure**

Algorithm 4.3 Pseudocode of the MERGE() operation, as part of the MGS algorithm (see Algorithm 4.2). The input is the set of initial pairwise intersections $\mathcal{I}$ as well as the sum of both radii r_{pq}. Its output is a possibly modified set $\mathcal{I}$ where elements of the orignal set have been merged and deleted. Note that $\mathcal{I}$ contains all pairwise intersections of two robots p, q.

1: **procedure** MERGE($\mathcal{I}$, r_{pq})
2: **do**
3: **for all** $I^m, I^n \in \mathcal{I}, m \neq n$ **do** ▷ For all index-based subpath pairs
4: Let $I^m \coloneqq \left(\left(m^i_{\triangleright}, m^j_{\triangleleft}\right), \left(m^k_{\triangleright}, m^l_{\triangleleft}\right)\right)$ and $I^n \coloneqq \left(\left(n^i_{\triangleright}, n^j_{\triangleleft}\right), \left(n^k_{\triangleright}, n^l_{\triangleleft}\right)\right)$
5: ▷ Determine intersection of intervals:
6: $(d^+_p, d^-_p) \leftarrow (\max(m^i_{\triangleright}, n^i_{\triangleright}), \min(m^j_{\triangleleft} + 1, n^j_{\triangleleft} + 1))$
7: $(d^+_q, d^-_q) \leftarrow (\max(m^k_{\triangleright}, n^k_{\triangleright}), \min(m^l_{\triangleleft} + 1, n^l_{\triangleleft} + 1))$
8: **if** $d^+_p \leq d^-_p \wedge d^+_q \leq d^-_q$ **then** ▷ Intersection non-empty?
9: $\Delta p \leftarrow d^-_p - d^+_p + 1$ ▷ Number of elements in p-intersection
10: $\Delta q \leftarrow d^-_q - d^+_q + 1$
11: **if** $\Delta p \geq 2 \wedge \Delta q \geq 2$ **then** ▷ Segments overlap, ready to merge.
12: valid $\leftarrow$ **true**
13: **else** ▷ Adjacent segments, testing necessary.
14: $F_p \leftarrow$ VERIFY($\mathcal{Q}, m^k_{\triangleright}, m^l_{\triangleleft}, n^k_{\triangleright}, n^l_{\triangleleft}, \boldsymbol{p}_{d^+_p}, r_{pq}$)
15: $F_q \leftarrow$ VERIFY($\mathcal{P}, m^i_{\triangleright}, m^j_{\triangleleft}, n^i_{\triangleright}, n^j_{\triangleleft}, \boldsymbol{q}_{d^+_q}, r_{pq}$)
16: valid $\leftarrow F_p \wedge F_q$
17: **end if**
18: **if** valid **then** ▷ Is it valid to merge I^m with I^n?
19: $M \leftarrow ((\min(m^i_{\triangleright}, n^i_{\triangleright}), \max(m^j_{\triangleleft}, n^j_{\triangleleft})), (\min(m^k_{\triangleright}, n^k_{\triangleright}), \max(m^l_{\triangleleft}, n^l_{\triangleleft})))$
20: UPDATE(M, I^m, I^n, $\mathcal{I}$) ▷ Delete I^m, I^n from $\mathcal{I}$ and add M.
21: **break** ▷ Leave inner for-loop.
22: **end if**
23: **end if**
24: **end for**
25: **while** $\mathcal{I}$ has changed ▷ Lines 19 and 20 executed in previous iteration?
26: **end procedure**

and 7, the algorithm first computes the intersection d_p^+, d_p^- of the intervals, defined by the given indices I^m, I^n (cf. Line 4):

$$(d_p^+, d_p^-) \coloneqq (\max(m_\triangleright^i, n_\triangleright^i), \min(m_\triangleleft^j + 1, n_\triangleleft^j + 1)) \quad \text{and} \tag{4.55}$$

$$(d_q^+, d_q^-) \coloneqq (\max(m_\triangleright^k, n_\triangleright^k), \min(m_\triangleleft^l + 1, n_\triangleleft^l + 1)). \tag{4.56}$$

If such intervals are non-empty (Line 8), there are two possible cases w. r. t. the lengths $\Delta p, \Delta q \in \mathbb{N}$ of these intervals: they can either be both ≥ 2 (see Line 11) or at least one of them has length 1, i. e., $\Delta p = 1 \vee \Delta q = 1$ (see Line 13). If they have a length of at least two, the subpaths overlap in an entire segment and can thus be merged immediately. However, in case they just overlap at their endpoints, further checks are necessary (cf. VERIFY() in Lines 14 and 15). Without loss of generality, the check is executed w. r. t. p if $\Delta p = 1$ so that a merge is possible if either $\Delta p \geq 2$ or $\Delta p = 1$ and the check succeeds. First, the check tests whether there is at least one segment on $\mathcal{Q}_{[m_\triangleright^k, m_\triangleleft^l]}$ with a distance less than r_{pq} to the single shared endpoint $\boldsymbol{p}_{d_p^+}$ in the interval. Notice that $\Delta p = 1$ if and only if $d_p^+ = d_p^-$. Second, the check also tests whether there is at least one segment on $\mathcal{Q}_{[n_\triangleright^k, n_\triangleleft^l]}$ with a distance less than r_{pq} to the same endpoint $\boldsymbol{p}_{d_p^+}$. This way, a shared endpoint of two adjacent segments is ensured to be still part of the intersection (i. e., has a distance of $< r_{pq}$). The same conditions and checks must also hold for q and a merge is triggered if both succeed (Line 16). A merge is eventually indicated by setting the "valid" flag to true causing a merge in Lines 19-20. According to Line 19, the resulting pairwise intersection is computed by taking the minimum of the entering and the maximum of exiting segment indices. It directly follows that the result of the merge contains both inputs (and is therefore possibly larger). The set $\mathcal{I}$ is updated by removing I^m, I^n (inputs) and adding M (result); this also triggers another iteration of the outer loop. As the code indicates, the MERGE() operation still ensures that $p_\triangleright \leq p_\triangleleft$ and $q_\triangleright \leq q_\triangleleft$ (similarly for q). Termination is guaranteed because there are only two possible cases: either nothing can be merged (right at the beginning or after some iterations) or all can be merged to a single intersection. In the first cases, the set $\mathcal{I}$ will eventually not be changed (anymore) which terminates the loops. In the second case, once there is only one pairwise intersection remaining, the inner loop will also terminate without altering $\mathcal{I}$ causing the outer loop to terminate as well.

Once all possible merges have been performed, the loop in Lines 15-29 iterates over all remaining intersections $I_{p,q}^k$ with indices $((p_\triangleright, p_\triangleleft), (q_\triangleright, q_\triangleleft))$ and computes the halt and release for p and q respectively. For every intersection guard, we test if it actually exists. Without loss of generality, we describe this for p but the same applies for q with roles swapped. For the halt of p, we first check if $p_\triangleright = \lfloor \sigma_p \rfloor$, that is, the first conflicting

segment in $I^k_{p,q}$ is equal to the start segment of p's path. If this applies and the following condition

$$\{0, 1\} \subseteq \{\, R(\mathcal{S}_i, \overline{\boldsymbol{p}_{p_\triangleright}\boldsymbol{p}_{p_\triangleright+1}}) \mid i \in \{q_\triangleright, \ldots, q_\triangleleft\} \subset \mathbb{N}_0 \,\} \text{ whereby} \tag{4.57}$$

$$R(\overline{\boldsymbol{p}_0\boldsymbol{p}_1}, \overline{\boldsymbol{q}_0\boldsymbol{q}_1}) = \begin{cases} -1 & \text{if } \operatorname{dist}(\boldsymbol{q}_0, \overline{\boldsymbol{p}_0\boldsymbol{p}_1}) \geq r_{pq} \wedge \operatorname{dist}(\boldsymbol{q}_1, \overline{\boldsymbol{p}_0\boldsymbol{p}_1}) < r_{pq}, \\ 0 & \text{if } \operatorname{dist}(\boldsymbol{q}_0, \overline{\boldsymbol{p}_0\boldsymbol{p}_1}) < r_{pq} \wedge \operatorname{dist}(\boldsymbol{q}_1, \overline{\boldsymbol{p}_0\boldsymbol{p}_1}) < r_{pq}, \\ 1 & \text{if } \operatorname{dist}(\boldsymbol{q}_0, \overline{\boldsymbol{p}_0\boldsymbol{p}_1}) < r_{pq} \wedge \operatorname{dist}(\boldsymbol{q}_1, \overline{\boldsymbol{p}_0\boldsymbol{p}_1}) \geq r_{pq} \end{cases} \tag{4.58}$$

holds true, we know that $p^k_{\min} = \sigma_p$ and, thus, $h(I^k_{p,q}, p) = -\infty$ (cf. Equations (4.7) and (4.11)). The property $R(\overline{\boldsymbol{p}_0\boldsymbol{p}_1}, \overline{\boldsymbol{q}_0\boldsymbol{q}_1})$ determines the type of the two given segments, that is, whether the second segment $\overline{\boldsymbol{q}_0\boldsymbol{q}_1}$ enters (-1), exits (1) or is fully inside (0) the first segment $\overline{\boldsymbol{p}_0\boldsymbol{p}_1}$. Likewise, for the release of p, we first check if $p_\triangleleft = N_p - 1$, that is, the last conflicting segment in $I^k_{p,q}$ is equal to the last segment of p's path. If this applies and the following condition

$$\{0, -1\} \subseteq \{\, R(\mathcal{S}_i, \overline{\boldsymbol{p}_{p_\triangleleft}\boldsymbol{p}_{p_\triangleleft+1}}) \mid i \in \{q_\triangleright, \ldots, q_\triangleleft\} \subset \mathbb{N}_0 \,\} \tag{4.59}$$

applies, we know that $p^k_{\max} = N_p - 2$ and, thus, $r(I^k_{p,q}, p) = \infty$. It is worth noting that the existence checks are crucial here because there are always extrema of a given set but that may not be the correct halt or release respectively. Only if we know that it actually exists allows us to safely take the extrema of the underlying sets as halt and release points. Correctness is then implied by how the segment-hull intersection problem is solved, see Section 4.3.1.

Given the indices $((p_\triangleright, p_\triangleleft), (q_\triangleright, q_\triangleleft))$ of segments of a pairwise intersection $I^k_{p,q}$, the halt is computed by (cf. ComputeHalt())

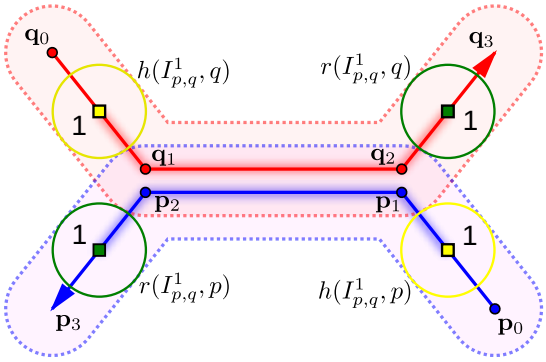
$$h(I^k_{p,q}, p) = \min \bigcup_{i=q_\triangleright}^{q_\triangleleft} \mathcal{B}^\star(\mathcal{S}_{p_\triangleright}, \mathcal{S}_i) \tag{4.60}$$

if it exists. Likewise, the release is computed by (cf. ComputeRelease())

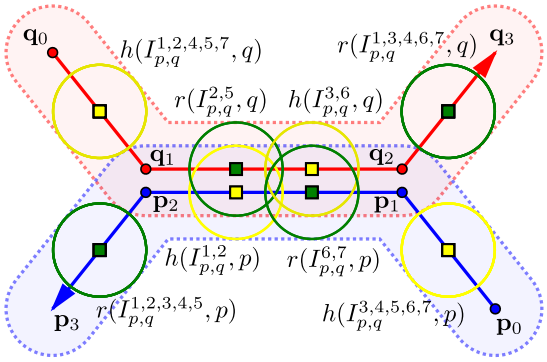
$$r(I^k_{p,q}, p) = \max \bigcup_{i=q_\triangleright}^{q_\triangleleft} \mathcal{B}^\star(\mathcal{S}_{p_\triangleleft}, \mathcal{S}_i) \tag{4.61}$$

if it exists. In the previous two equations, $\mathcal{B}^\star() \subset \mathbb{R}_{\geq 0}$ denotes the solution of the segment-hull intersection problem for the two given segments (i. e., the ordered set of halts and/or releases, cf. Equation (4.51)) whereby the r_{pq}-hull of $\mathcal{S}_i$ is intersected with the segment provided as first parameter. Like in the SGS algorithm, path modifications due to starting progresses are being reverted in Line 30 and remappings are applied if any of the computed halts or releases are located on the first (modified) segment (refer to Section 4.3.2 for more details).

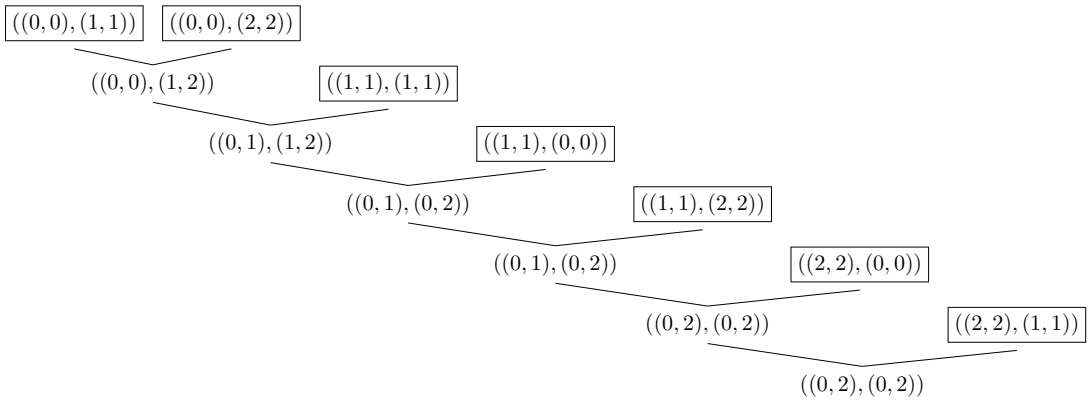
Finally, Figure 4.12 shows an example of the algorithm and the sequence of merge operations. Figure 4.12(a) shows the resulting merged intersection $I^1_{p,q}$ which originates



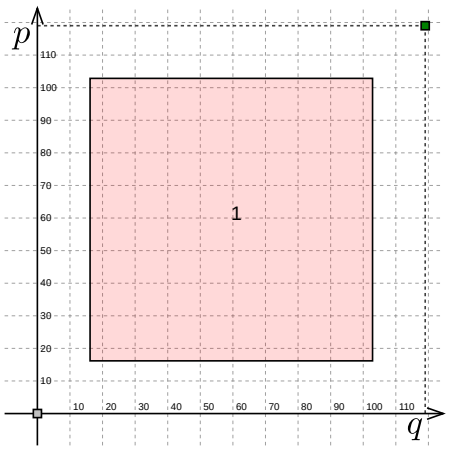
(a) Resulting merged pairwise intersection, i. e., $\mathcal{I} = \{((0,2),(0,2))\}$ (highlighted), computed by the MGS algorithm, see Algorithm 4.2



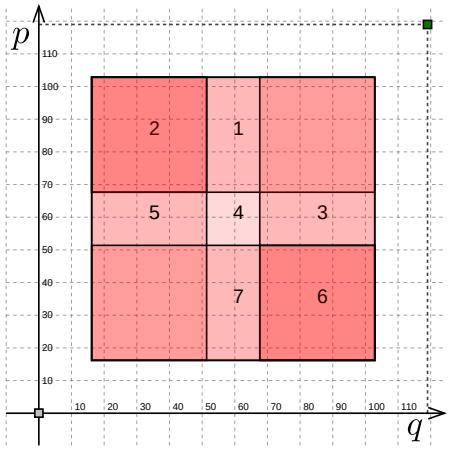
(b) Result of the SGS algorithm (7 pairwise intersections); note that multiple circles overlay at the same location



(c) Sequence of MERGE() operations (6 in total, see Algorithm 4.3) for the example in (b) with $|\mathcal{I}| = 7$ initial pairwise intersections. Numbers $((p_\triangleright, p_\triangleleft), (q_\triangleright, q_\triangleleft))$ correspond to the indices of the points in the path (see (a) and (b)) and therefore refer to subpaths.



(d) Coordination space for (a)



(e) Coordination space for (b)

Figure 4.12: Example for the (a) MGS algorithm in comparison with (b) the SGS algorithm. The number of intersections is reduced from 7 to 1 as indicated by (c).

from the pairwise intersections depicted in Figure 4.12(b) after applying the sequence of MERGE() operations displayed as a tree in Figure 4.12(c). All boxed subpath pairs in (c) are the results of pairwise segment-level intersection tests (cf. Lines 7-13 in Algorithm 4.2). The CSs for Figure 4.12(a) and (b) are visualized in Figure 4.12(d) and (e) and illustrate how intersections are combined. For example, the first merge operation in Figure 4.12(c) takes $I^m = ((0,0),(1,1))$ and $I^n = ((0,0),(2,2))$ as inputs and merges them to $((0,0),(1,2))$ with $\Delta p = 2, \Delta q = 1$ and $d_q^+ = d_q^- = 2$.

4.3.4 Evaluation

This section deals with the comparative evaluation of the SGS and the MGS algorithms. The benchmarks have been executed on a system with an AMD Ryzen 9 3900X 12-core processor, 32 GB RAM running Ubuntu 20.04.

Figure 4.13 shows the computation time for the SGS and MGS algorithm w. r. t. two different scenarios. Each point in the diagram is the result of averaging five measurements to account for varying system load. The input paths for p and q have been generated procedurally to analyze how both algorithms scale with regard to varying paths lengths. Two examples for a total of two path segments each are depicted in Figure 4.14 for reference. Note that the two scenarios are chosen in order to reflect the extreme cases, that is, no merge being possible in scenario 1 and all intersections getting merged in scenario 2. SGS creates up to $\mathcal{O}(N_p \cdot N_q)$ intersections whereby N_p and N_q are the number of points on the paths of $\mathcal{P}$ and $\mathcal{Q}$ respectively. This is also what MGS starts with initially. With respect to the MERGE() operation, two cases are possible: none of the intersections can be merged (equaling scenario 1) or all intersections can be merged to a single one (equaling scenario 2). For example (scenario 2), with 1000 segments as the input paths lengths, SGS found 4994 pairwise intersection in 0.119 s. MGS merged these to a single intersection taking 0.164 s in total, i. e., only 45 ms longer. The difference for scenario 1 (no merge possible) is even negligible (118.179 vs. 121.508 ms). However, notice that the number of pairwise intersections does not necessarily scale linearly in the number of path segments.

The data from Figure 4.13 suggests quadratic run-time complexity of both algorithms with a small additional amount of time required by MGS for the merging. A least squares polynomial fit for the data in Figure 4.13 underpins this assumption. For instance, the polynomial for MGS in scenario 2 (all merged) is approximately

$$0.00000016x^2 - 0.00000268x + 0.00034172 \tag{4.62}$$

with a fitting error of less than 0.0002002. It should be noted that MGS is not asymptotically optimal because there are algorithms for the *Rectangle Intersection Problem* running in $\mathcal{O}(n \log(n) + K)$ time whereby n is the number of input rectangles and K is the number of intersecting pairs of rectangles [35]. However, for smaller number of inputs considered here (< 200), MGS is assumed to perform faster in practice.

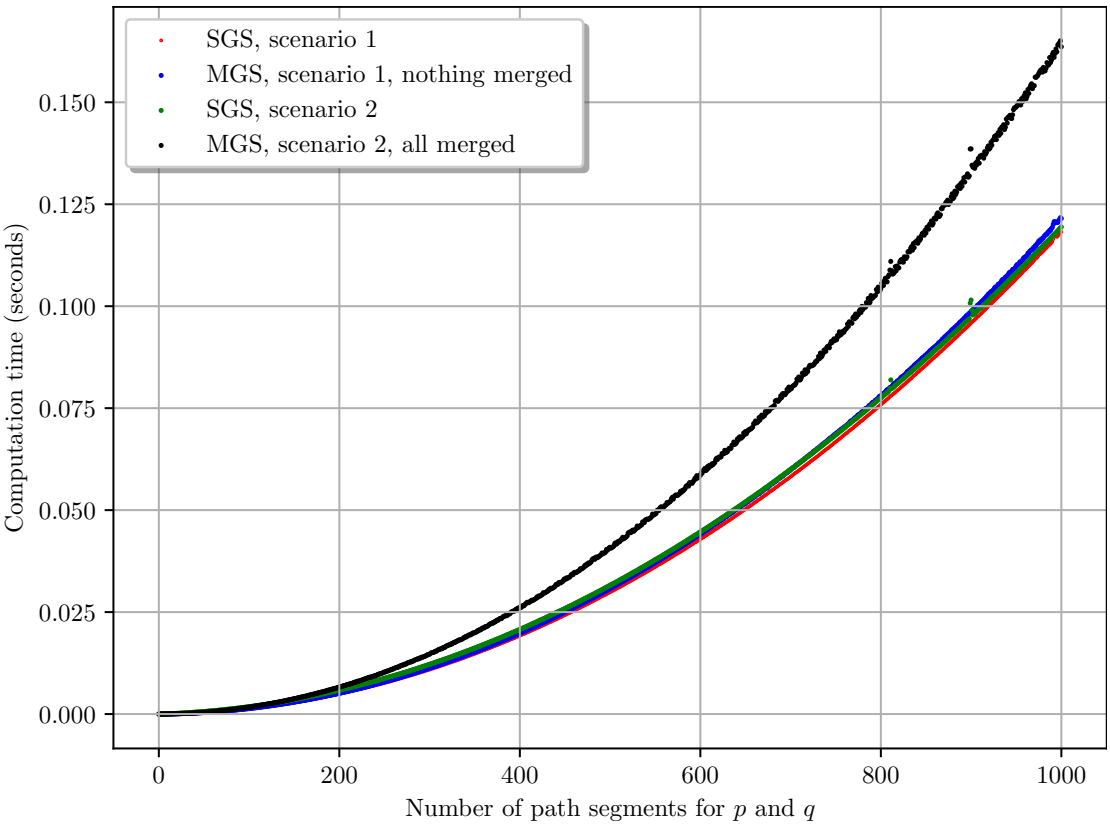
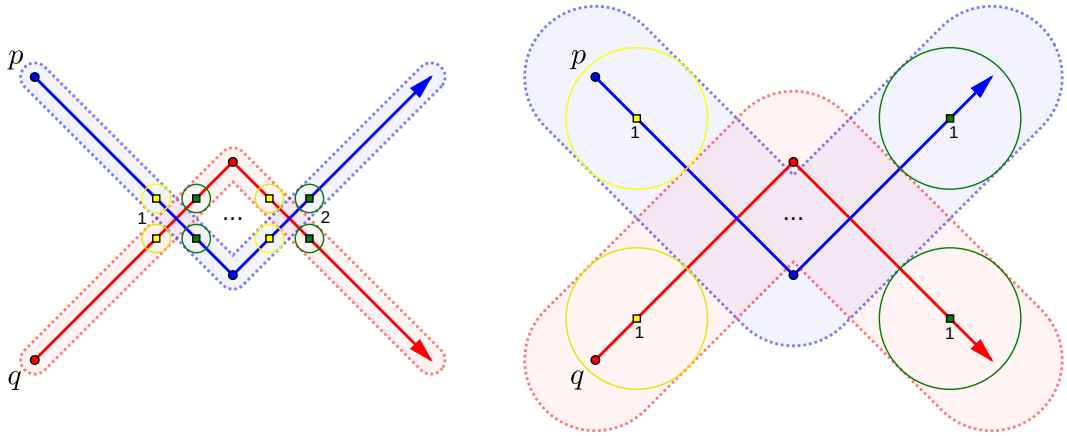


Figure 4.13: Computation time in seconds (ordinate) for the SGS and MGS algorithms and two scenarios based on varying path lengths (abscissa). In scenario 1, all intersections were too far apart for being merged ($r_{pq} = 10$). Scenario 2 was similar to scenario 1 but due to $r_{pq} = 50$ all intersections can be merged by MGS (black dots). Paths have been generated procedurally in a zick-zack fashion (see Figure 4.14) and displayed values are the average over five measurements to account for varying system load.

Given a set of N robots as input, a total of $\binom{N}{2} = \mathcal{O}(N^2)$ calls to these algorithms is necessary to inspect the intersections of all (unordered) pairs ($k = 2$) of paths which serves as the input for the solvers described next.

4.4 Incremental Coordination-Space Path Scheduler

This section deals with the explanation of the novel *Incremental Coordination-Space Path Scheduler* (ICSPS), an algorithm used to solve the RoW assignment for all pairwise intersections of a given set of robots with associated precomputed paths. More information about the formal problem has already been presented in Section 4.2.2. The framework introduced in Chapter 5 uses this solver algorithm in one of its states to negotiate a set of robots dynamically.



(a) Sample for scenario 1 (2 intersections) (b) Sample for scenario 2 (1 intersection)

Figure 4.14: Exemplary visualization of the two procedurally generated scenarios used for performance analysis of the SGS and MGS algorithms for two path segments, cf. abscissa in Figure 4.13. By increasing the number of path segments, more such "diamond shaped structures" are being created (see the three dots in the center of both figures).

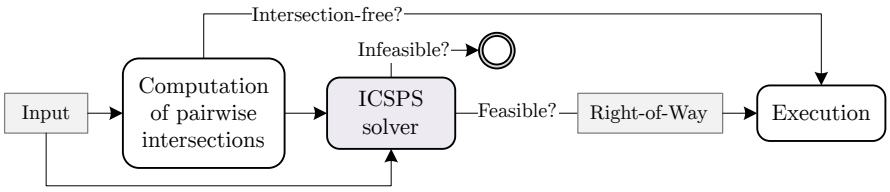


Figure 4.15: Overview of how ICSPS is applied after applying conflict detection (pairwise intersection computation), e. g., via SGS or MGS. Rectangular boxes represent in- and output while rounded boxes are processing steps. If ICSPS considers a scenario to be infeasible, it is discarded.

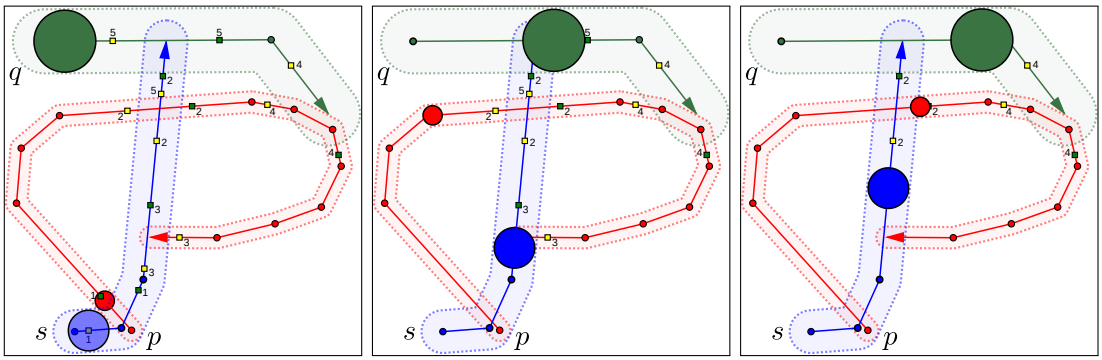
Section 4.4.1 begins with the explanation of in- and outputs of the algorithm along with a brief overview of how the algorithm works. Section 4.4.2 then continues with a more detailed discussion of the temporal CS which forms the base of the algorithm. Recall that the spatial CS, the CS with robot locations on the axes, has already been introduced briefly in Section 4.2.2. Based on the representation in a 2D CS, Section 4.4.3 explains how an input is solved for only two robots as input. Because the input is comprised of N robots, $N > 1$, Section 4.4.4 extends this to the N-dimensional case. Finally, Section 4.4.5 details how the resulting RoW assignment is deduced from the solution computed in CS and Section 4.4.7 completes this section with an evaluation of the solver algorithm.

4.4.1 Overview

With a few simplifications, Figure 4.15 illustrates how this solver is applied for a set of N robots in the input. After computing all pairwise intersections $\mathcal{I}_{i,j}$, either with the SGS or MGS algorithm (see Sections 4.3.2 and 4.3.3), it takes $\mathcal{I}_{i,j}, \forall i,j \in \{1,...,N\}, i \neq j$, the paths $\mathcal{P}_i$, the start progresses σ_i and the maximum velocities $v^i_{\max}$ for every robot as the input and computes the RoW assignment $\delta()$ for all intersections, cf. Section 4.2.2. Finally and once *all* RoWs have been determined, the solution (if any) is executed by the robots while respecting the computed RoW. Note that paths can degenerate to a point if a robot is already at its goal (indicated by its start progress, i. e., $\sigma_i = |\mathcal{P}_i| - 1$).

The algorithm operates in the temporal CS which is a velocity-scaled variant of the spatial CS and is therefore based on time and durations instead of locations and distances. Basically, it proceeds as follows. As a preprocessing step, all *initial timings* are computed for all involved robots and all halts and releases of associated intersections. An initial timing for a robot p and an intersection $I_{p,q}$ is given by the distance from the robot's start σ_p to the halt $h(I_{p,q}, p)$ divided by its maximum velocity $v^i_{\max}$, likewise for the release $r(I_{p,q}, p)$. The initial timings allow efficient access to the approximate time required by every robot to move to its halt and release for all intersections respectively. For a given order π of robots in the input, the algorithm starts to solve for the first two robots $\pi(0)$ and $\pi(1)$. This constructs the 2D CS with all intersections between robots $\pi(0)$ and $\pi(1)$. The CS has already been introduced briefly in Section 4.2.2 and will be explained in more detail in Section 4.4.2. Within the CS, it then tries to find a path from $(0,0)$, representing the starts, to the upper right, representing the goals of both robots. This path may not intersect with any CR in CS, each representing an intersection between $\pi(0)$ and $\pi(1)$. Additionally, the bounding boxes of all segments of such a solution path may not intersect with any of the CRs. Assuming feasibility w. r. t. $\pi(0)$ and $\pi(1)$ for now, the algorithm continues to solve for the remaining robots $\pi(j)$, $j = 2,...,N-1$, by iteratively constructing the next CS with the solution path of the previous iteration. That is, the previous solution path serves as the foundation for the new abscissa in the next CS. In iteration j, the RoW of intersections between robots $\pi(0),...,\pi(j-1)$ (placed on the abscissa) vs. $\pi(j)$ (placed on the ordinate) are solved which remains a two-dimensional input. In every iteration step, this requires the reprojection of halts and releases between all previously solved robots $\pi(k)$, $k = 0,...,j-1$, and all upcoming robots $\pi(l)$, $l = j,...,N-1$, onto the new abscissa, subsequently termed the *combined axis*. The 2D solution paths found in every CS (iteration) are used to deduce the RoW for every pairwise intersection (detailed in Section 4.4.3).

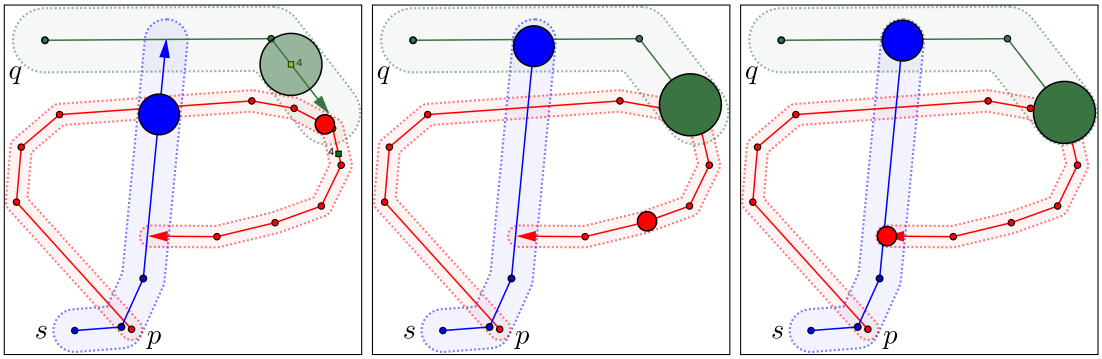
Figure 4.16 shows an introductory example for an input scenario solved by ICSPS. In Figure 4.16(a) to (f), six steps of how the solution is executed by the robots are shown. The corresponding CSs are depicted in Figure 4.16(g) and (h). Robots are presented as semi-transparent circles if they are waiting for the release of an intersection ahead (cf. (a) and (d)). Halts and releases have been removed once they become obsolete to improve readability. Notably, because robots are assumed to move with maximum velocities, the black dashed paths in both CSs represent the animated motions from Figure 4.16(a)



(a) Processing at time t_1; robot s (transparent) waits for p

(b) Processing at time t_2; all robots are in motion

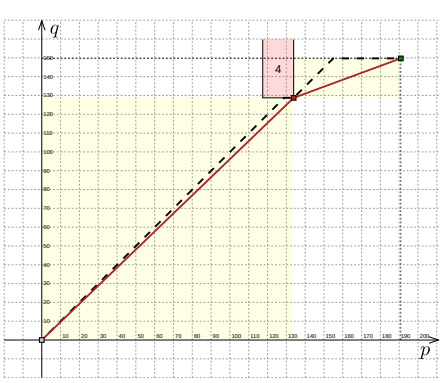
(c) Processing at time t_3; all robots are still in motion



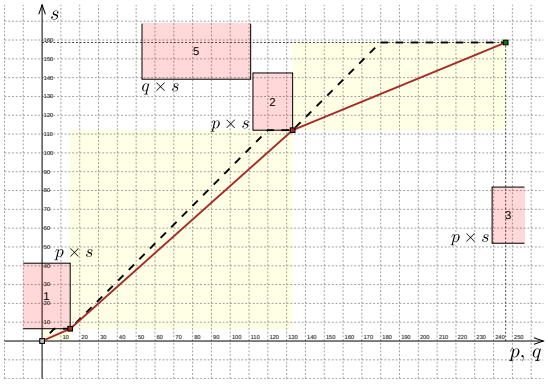
(d) Processing at time t_4; robot q (transparent) waits for p

(e) Processing at time t_5; all robots are in motion again

(f) Processing at time t_6; all robots have reached their goals



(g) First coordination space: p vs. q



(h) Second coordination space: p, q vs. s

Figure 4.16: The animation steps (a) to (f) illustrate how an input scenario with three robots p (red), q (green), s (blue) is executed after being solved by ICSPS as an introductory example while (g) and (h) visualize the corresponding CS with the solution paths (brown) and bounding boxes (yellow) for every segment. Black dashed paths represent the actual executed motions in the animation (with maximum velocity). The numbers at every intersection correspond to the red-shaded collision rectangles (CRs).

to (f). They reside in the segment-induced bounding boxes (shaded yellow). A closer look on robots p (red) and q (green, transparent) in Figure 4.16(d) reveals how q waits for the release of intersection 4 by p whose CR is shown in (g). Because q would block the path of p, the CR strives to $+\infty$ on the q-axis. (h) shows the combined axis for robots p and q. The waiting of q corresponds to the second (horizontal) segment (dashed black) in Figure 4.16(g) where no time for q passes ("waiting") while robot p still moves (through the intersection area). After the release, q will eventually reach its goal which corresponds to the second-last point on the dashed black path in (g). Finally, q has to "wait" again at its goal (reaching it first) while p completes its remaining path (cf. last segment on blue path). More details on how these trajectories are being deduced from the solution paths (brown) will be presented in Section 4.4.5, especially how this applies in higher dimensions (e. g., in (h)).

It should be noted that, theoretically, an N-dimensional CS solver with exponential runtime complexity is possible by constructing the N-dimensional CS in order to find an N-dimensional path from the origin to the goal of all robots with the same properties outlined above. However, this is impossible to compute for practically sized instances which justifies the choices made in the algorithm design.

4.4.2 Representations in Coordination Space

This section presents the CS with its properties which serves as the foundation of the proposed ICSPS algorithm. As already briefly introduced in Section 4.2.2, a CS for just two robots p, q is spanned by the two involved paths defining the horizontal and vertical axis respectively. Which robot is mapped to which axis is a matter of definition: we typically assign the "first robot" p to the horizontal and the "second robot" q to the vertical axis. More precisely, all segment endpoints of a robot's path (polyline) are projected onto its axis so that distances are being preserved and any location on that axis represents a position on that robot's path w. r. t. its starting point. Any 2D coordinate in such a CS then represents a specific position of *both* robots on their paths. The resulting space is termed *spatial coordination space*. Because a pairwise intersection $\mathcal{I}_{p,q}$ is (normally) comprised of a halt and release for both involved robots, we can identify intervals $[h(I_{p,q}, p), r(I_{p,q}, p)]$ and $[h(I_{p,q}, q), r(I_{p,q}, q)]$ on both axis in CS that describe a so-called *collision rectangle* (CR). If one of the limits is missing (due to a non-existing halt or release), its associated CR strives towards $\pm\infty$ according to Equations (4.11) and (4.12). This model is beneficial because if, for instance, p has no release ($r(I_{p,q}, p) = \infty$), the associated CR "blocks" reachability of the common goal location in the upper right of the CS for p, forcing q to get RoW (otherwise, p would block the intersection area forever). It is important to emphasize the special meaning of the borders of such a CR: because halts and releases are defined in such a way that collisions are still impossible when robots are located on their associated intersection guards, only the interior of a CR is representing the critical intersection area. That is why the solution paths in Figure 4.16(g) and (h) are allowed to use edges and corners of the CRs.

There is no modeling of time yet. We therefore propose a simple scaling approach: each axis is scaled by the associated robot's maximum velocity by dividing distances on an axis by the maximum velocity. This yields the *temporal coordination space*. Within the following context of ICSPS, we will always refer to time-scaled "locations" (becoming points in time w. r. t. a robot's start of a motion) and distances (becoming durations) in CS without mentioning that explicitly. Note that, however, the simplification by means of scaling axes via the robots' maximum velocities does not void the safety guarantees as explained in the next Section 4.4.3.

Notice that the size of a robot is implicitly modeled by the CRs as well and that no discretization is used at all.

4.4.3 Solving the 2-dimensional Case

Given the CS for two robots p and q as described in the previous Section 4.4.2, we will now describe how this input is used to find the RoW for all pairwise intersections $I_{p,q} \in \mathcal{I}$ between p and q. The underlying idea can be outlined as follows. Because the CS for p and q represents all intersections via CRs, we need to find a monotonic path inside the CS that connects the start of both robots at $(0, 0)$ with the goal in the upper right while being intersection-free with the interior of all CRs. Every point on this path represents the position of both robots in time since they have started moving at the same time. By ensuring no intersections with any of the CRs, it is guaranteed that the robots will never enter an intersection area simultaneously, given the underlying assumptions are met. It is important to note that this path must be monotonically increasing because robots are not allowed to move backwards. They are only allowed to move forward in the velocity range $[0, v_{\max}]$.

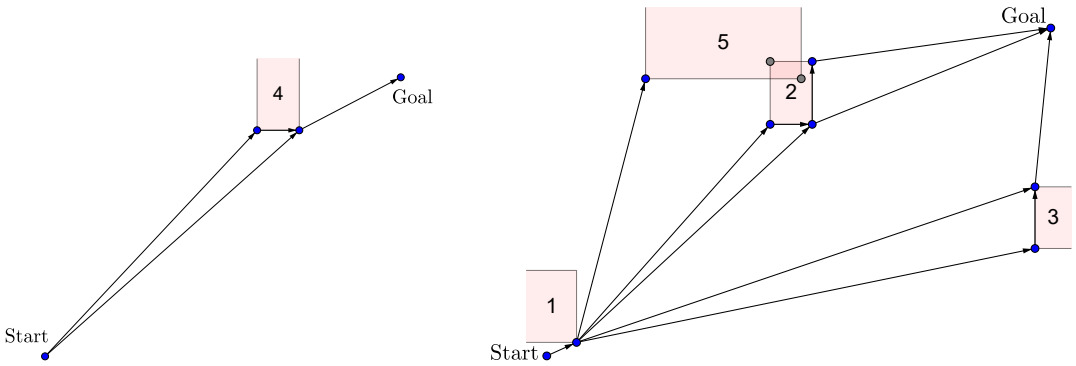
In order to encode all required properties of such a path, we model the underlying problem with a directed *Constrained Visibility Graph* (CVG) $G = (\mathcal{V}, \mathcal{E})$. The set $\mathcal{V}$ contains a vertex for every corner of a CR, that is, all combinations of halt and releases for both involved robots if they exist. Additionally, the start and goal locations are added as vertices. Given two vertices $u, v \in \mathcal{V}, u \neq v$, a directed edge $(u, v) \in \mathcal{E}$ is created if the following *monotonicity property*

$$v_x \geq u_x \wedge v_y \geq u_y \tag{4.63}$$

as well as the *intersection-free property*

$$\forall I_{p,q}^k \in \mathcal{I} : B(u_x, u_y, v_x, v_y) \cap B\left(\frac{h(I_{p,q}^k, p)}{v_{\max}^p}, \frac{r(I_{p,q}^k, p)}{v_{\max}^p}, \frac{h(I_{p,q}^k, q)}{v_{\max}^q}, \frac{r(I_{p,q}^k, q)}{v_{\max}^q}\right) = \emptyset \tag{4.64}$$

hold. Notice that every vertex $u \in \mathcal{V}$ has an associated position $(u_x, u_y) \in \mathbb{R}^2$ in CS. $B(x_1, y_1, x_2, y_2)$ denotes the interior of the bounding box of the two 2D points (x_1, y_1) and (x_2, y_2). CRs are just *axis-aligned bounding boxes* so that intersections can efficiently be tested by four comparisons only. The monotonicity property enforces edges to be



(a) Visibility graph for CS in Figure 4.16(g) (b) Visibility graph for CS in Figure 4.16(h)

Figure 4.17: CVGs based on the introductory example in Figure 4.16 with overlayed CRs (red) to indicate the correspondences between vertices $v \in \mathcal{V}$ and intersections. Blue vertices are relevant for the computation of the solution path while gray vertices (see (b)) are unconnected and can be ignored. Also note that vertices are omitted if their associated corners do not exit (strive to $\pm\infty$).

monotonically increasing w. r. t. both axes. The intersection-free property ensures that bounding boxes of edges in the graph remain empty with regard to all CRs. This has the advantage that robots do not need to adhere to a precomputed velocity profile as common in many related approaches (cf. Chapter 2) and are free to execute the determined RoW assignments (velocity tolerance). Moreover, unexpected delays, obstacles or model inaccuracies will not void the provided safety guarantees, making this approach very robust while sacrificing some efficiency. Figure 4.17 shows two examples for a CVG.

We suggest the A* algorithm [28] to find the shortest path in G. Alternative approaches are possible because the graph already encodes restrictions so that any path is valid in that graph. Paths that run on the 45°-diagonal of the CS[3] are preferred due to increased parallelism because, conceptually, both robots are then allowed to make progress on their paths simultaneously (with maximum velocity). An empty CS without any intersection is a simple example for this where both robots can obviously move independently and with the shortest path being the single diagonal segment (connecting the origin in the lower left with the goals in the upper right). This is automatically handled by A* because the shortest path favors diagonals. However, because one robot might reach its goal earlier than the other and as already explained for Figure 4.16(g), the aforementioned diagonal segment might not represent the actual path in CS being executed. Note that in some cases with SGS, "dead ends" can occur in the graph G but since paths are only valid if they are connecting the origin with the goal, this is automatically avoided by design. Figure 4.17(b) also illustrates this (cf. lower left corner of intersection 5).

[3]Note the difference to the diagonal in the CS from $(0, 0)$ to the goal (upper right) because both axes do not necessarily have the same length.

4.4.4 Transfer to the N-dimensional Case

ICSPS reduces the entire problem to a series of $N-1$ 2-dimensional problems which are solved incrementally. Given a specific order π of the robots in the input, the RoW for all pairwise intersections between the first two robots $\pi(0)$ and $\pi(1)$ is first computed as explained previously. This yields a solution path $\mathcal{S}_0$ in the CS of $\pi(0)$ and $\pi(1)$. The path's start represents the beginning of both motions and its end represents the time the longer moving robot requires to reach its goal. It fully specifies the RoW between $\pi(0)$ and $\pi(1)$ which will be explained in Section 4.4.5. The solution $\mathcal{S}_0$ defines the new abscissa for the next iteration when incrementally adding the next robot $\pi(2)$, thus solving all intersections between $\pi(0)$ and $\pi(2)$ as well as $\pi(1)$ and $\pi(2)$. More formally, given the solution path up to robot $\pi(j-1)$, the next iteration solves all pairwise intersections between all robots $\pi(0), \ldots, \pi(j-1)$ (all placed on the horizontal axis) and the newly added robot $\pi(j)$ (always placed on the vertical axis). That is, given N robots in the input, $N-1$ coordination spaces are being constructed incrementally.

Except for the first two robots, the abscissa always represents the set of robots between which pairwise intersections have already been solved. We denote it as the *combined axis* in the CS. In iteration j, its length is given by the Euclidean length $|\mathcal{S}_{j-1}|$ based on the previous solution path $\mathcal{S}_{j-1}$. This way, the relevant part of the abscissa (indicated by the two dotted lines in Figure 4.16(h) and (g)) can only grow and will eventually represent the time the longest moving robot in the input requires to reach its goal.

Conceptually, all halts and releases of robots being part of the CS in $j-1$ are mapped onto $\mathcal{S}_{j-1}$, that is, guards[4] of robots $\pi(0), \ldots, \pi(j-1)$ on the combined axis are vertically projected on $\mathcal{S}_{j-1}$ while guards of the robot $\pi(j)$ on the ordinate are horizontally projected on $\mathcal{S}_{j-1}$. More specifically, let j, $2 \leq j < N$, be the current iteration, i. e., the RoW of all pairwise intersections of (yet unconsidered) robot $\pi(j)$ with (already considered) robots $\pi(k)$, $k = 0, \ldots, j-1$ has to be determined. Because we will subsequently only encounter and process intersections between pairs $(\pi(k), \pi(l))$ whereby $l = j, \ldots, N-1$ indexes the yet unprocessed robots, all guards between k and l must be projected onto the combined axis based on the previous solution path $\mathcal{S}_{j-1}$; the result is termed *projected timings* and updated incrementally. Three cases are possible. First, there are already projected timings stored for $\pi(k)$ and $\pi(l)$. In this case, such timings are simply reprojected onto $\mathcal{S}_{j-1}$ and the stored values are updated. Second, there are no projected timings, i. e., none of the halts or releases of $\pi(k)$ and $\pi(l)$ have been considered so far. In this case, we project the initial timings of $\pi(k)$ w. r. t. $\pi(l)$ onto $\mathcal{S}_{j-1}$ and store them as projected timings for subsequent iterations $j+1, ..., N-1$. Finally, in case there are no intersections between robots $\pi(k)$ and $\pi(l)$, nothing is projected.

We will now describe how the projection is carried out, cf. Figure 4.18. As described, a guard consists of a halt and a release (if existing) and initial timings have already been precomputed for all intersections. Note that non-existing guards are ignored in the projection, that is, the projection of a non-existing guard remains a non-existing

[4] Recap that "guard" is the catch-all phrase for halts and releases.

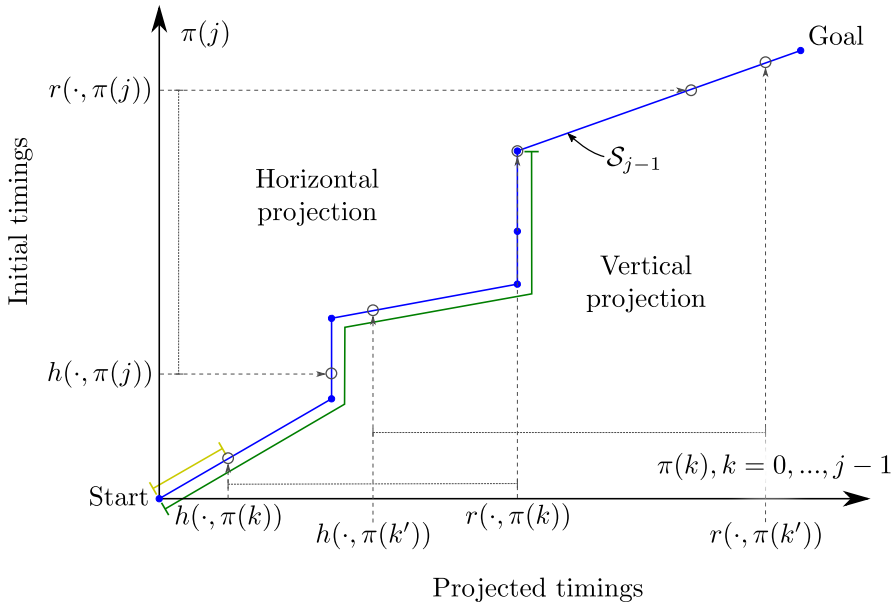


Figure 4.18: Scheme of how projections of timings are computed in coordination space in iteration j of the ICSPS algorithm for three exemplary intersection guards for robots $\pi(j)$, $\pi(k)$ and $\pi(k')$. The path $\mathcal{S}_{j-1}$ (blue) is the solution of the previous iteration $j-1$ and serves as the projection target. Timings for guards on the abscissa are taken from projected timings while timings on the ordinate are taken from the precomputed initial timings. Notice that the fourth and fifth (vertical) segments of $\mathcal{S}_{j-1}$ are collinear here. The projection result is exemplarly visualized for $h(\cdot, \pi(k))$ (yellow) and $r(\cdot, \pi(k))$ (green).

guard. Based on how the algorithm maps robots to axes, we project horizontally if $k \leq j-2$ and vertically if $k = j-1$. Without loss of generality, given such a timing t and the combined axis $\mathcal{S}_{j-1}$, we briefly describe how t is projected onto the abscissa (that is, for case $k \leq j-2$); the projection onto the ordinate is similar. At first, the segment $\mathcal{Z}_m = \overline{\boldsymbol{s}_m \boldsymbol{s}_{m+1}} \in \mathcal{S}_{j-1}$ must be identified such that $t \in [\boldsymbol{s}_m^{(x)}, \boldsymbol{s}_{m+1}^{(x)}]$. If there are subsequent segments $\mathcal{Z}_{m+1}, \mathcal{Z}_{m+2}, ...$ on $\mathcal{S}_{j-1}$ that are collinear to $\mathcal{Z}_m$, we set $\mathcal{Z}_m$ to the last collinear segment of that series. This is justified by the fact that the location on the abscissa is equal for all such collinear segments and they are all safe for the robot under consideration. By computing the intersection between $\mathcal{Z}_m$ and the vertical segment $\overline{(t, 0), (t, \boldsymbol{s}_{m+1}^{(y)})}$, we obtain the horizontal projection $\boldsymbol{p} \in \mathcal{S}_{j-1} \subset \mathbb{R}^2$ of t on the combined axis. The total distance along the path $\mathcal{S}_{j-1}$ from the beginning up to the location $\boldsymbol{p}$ yields the final projection result for t. With regard to Figure 4.18, it is important to note that the depicted path (blue) was computed based on CRs that are not visualized and are unrelated to the intervals shown on both axes.

Essentially, the combined axis determines how the timings are being "distorted" within the projection, thus influencing how CRs for subsequent CSs look like. In other words,

projected timings reflect how computed solutions paths affect and possibly influence the computation of solution paths in higher dimensions. This is intuitive because computing the solution for a subset of robots may affect the solution of others and the concept of projected timings is how the algorithm handles this. Notice that in the process of projection, the axes are already time-scaled so that no division by the maximum velocity is necessary. Also note that the path $\mathcal{S}_{j-1}$ serves as the input for the projection and for finding the next solution path $\mathcal{S}_j$ as explained next.

The projected and initial timings serve as the input for solving the 2D instance between robots $\pi(l)$, $l = j, \dots, N-1$ and $\pi(k)$. Since this is very similar to the basic 2D case described in Section 4.4.3, we briefly outline how the CS is constructed. By design, robot $\pi(k)$ is mapped to the ordinate so that the intervals $[h(I^m_{\pi(k),\pi(l)}, \pi(k)), r(I^m_{\pi(k),\pi(l)}, \pi(k))]$ for all pairwise intersections $I^m_{\pi(k),\pi(l)}$ are taken from the initial timings. Associated intervals $[h(I^m_{\pi(k),\pi(l)}, \pi(l)), r(I^m_{\pi(k),\pi(l)}, \pi(l))]$ on the combined axis are taken from the projected timings. This fully specifies all CRs within the current CS so that A* can find the shortest path in the underlying CVG (again, see Section 4.4.3), creating a new solution path $\mathcal{S}_j$. For the sake of completeness, if there are no intersections between $\pi(k)$ and all $\pi(l)$, $l = j, \dots, N-1$, within the current iteration j, the CS remains empty, and the diagonal would be the shortest solution path $\mathcal{S}_j$. This is because the CVG would consist of two vertices only: the start (lower left) and the goal (upper right). The path $\mathcal{S}_j$ serves as the input for the next iteration $j+1$ (if any).

Shorter solution paths close to the diagonal are more favorable compared to those that take a detour, although this is inevitable in some case where CRs are blocking areas in the CS. The path $\mathcal{S}_j$ is returned as solution. It may be empty ($\mathcal{S}_j = \emptyset$) if the area between the start (lower left) and goal (upper right) is completely blocked by CRs such that no path exists.

4.4.5 Right-of-Way Assignment

It has not been presented yet how ICSPS specifically deduces the RoW for a pairwise intersection. Recall from the previous section that in every iteration j, a solution path $\mathcal{S}_j$ is determined using A* and that every pairwise intersection is represented as a CR in the current CS j.

Let p (on the combined axis) and q (on the ordinate) be the two involved robots in an intersection $I_{p,q}$. The lower left corner of a CR represents the halt of p and q, and the upper right represents the release. The upper left corner represents the halt of p and the release of q (similarly for the lower right with roles swapped). That is, for instance, if the solution path runs to the lower left and then to the upper left corner, robot p is only allowed to move to $h(I_{p,q}, p)$ (lower left corner) while q can move to its release $r(I_{p,q}, q)$ (upper left corner). In other words, the positional relationship of $\mathcal{S}_j$ and a CR allows to infer the RoW for that intersection (binary decision). For this reason, we can deduce the RoW for an intersection based on how the path runs past the associated CR.

The RoW at $I_{p,q}$ is assigned to p if the CR is located *above* $\mathcal{S}_j$. Likewise, the RoW is assigned to q if the CR is located *below* $\mathcal{S}_j$.

Geometrically, this is robustly and efficiently tested by checking whether the center point of the CR is above or below the solution path. Although we know that the solution path does not intersect with any CR by construction, the boundary of a CR can be part of $\mathcal{S}_j$ so that the corners of a CR cannot be checked directly. This must be done in every dimensional step during the execution of ICSPS and the result is stored along with every intersection for later execution (cf. Figure 4.15).

In the spatial CS, any 2D point represents the location of both robots on their path simultaneously. Given an iteration $j \geq 2$ of ICSPS and assuming a spatial CS for now, that location may be "behind" a robot's goal because the combined axis represents the set of all paths of robots placed on that axis and they most likely do not have the same length. However, this is not an issue because we simply consider locations "behind a robot's goal" to be on the goal. That is, when sweeping along the solution path $\mathcal{S}_j$ from the origin to the goal, this specifies positions for all involved robots on their paths. If we now assume a temporal CS as foundation for the ICSPS again as explained in the previous sections, any 2D point represents the time (since motions have started simultaneously) involved robots have moved on their paths. This can be mapped to a unique location on their path since dividing by their maximum velocity just scaled the axes. On the combined axis, velocity scaling only changes the lengths of the paths (becoming durations). The justification for applying the scaling is to incorporate a notion of different velocities. For instance, if robot p has to move twice the distance of another robot q before reaching their first intersection but moves (approximately) twice as fast as q does, both will reach their first intersection roughly at the same time. A huge advantage of how ICSPS models velocities w. r. t. safety is that bounding boxes of segments of $\mathcal{S}_j$ are not allowed to intersect any (interior of a) CR. Although being a restriction, it allows robots to move at any velocity while executing the determined RoWs without being in danger of a collision.

Based on Figure 4.16(h), we will now justify why all bounding boxes (yellow) of the segments of a 2D solution path $\mathcal{S}_j$, $j \geq 2$ remain intersection-free regarding *both* all 2D CRs in iteration j *and* all CRs from dimensions j', $\forall j' < j$, i. e., why the solution is valid w. r. t. collision safety. Observe that iterations j are related to dimensions d of the overall search space. In other words, we justify that the d-dimensional bounding boxes of the segments of the d-dimensional solution path $\mathcal{S}_d$ remain intersection-free to all d-dimensional CRs. For instance, in Figure 4.16 there are three robots (p, q, s) so that a CR can be considered as a 3D cube. Intersection 4 in Figure 4.16(g) is a 2D CR but can be extended to $\pm\infty$ w. r. t. robot (dimension) s, making it a 3D CR. Even more apparent, the "2D CRs" in Figure 4.16(h) are assigned two coordinates on the combined axis, making them 3D CRs. Basically, this holds for all elements of the visualization; for the solution path $\mathcal{S}_2$ in particular.

That being said, we take a closer look at the transition from $j \to j+1$ in Figure 4.16(g) and (h) ($j = 1$). The first segment of the solution path in Figure 4.16(g) induces two

$$\overbrace{\underbrace{\pi_i(0), \pi_i(1), \pi_i(2), \cdots, \pi_i(j-1)}_{\pi_i(k)}, \quad \pi_i(j), \quad \underbrace{\pi_i(j+1), \cdots, \pi_i(N-1)}_{\pi_i(l)}}^{\text{All } N \text{ involved robots}}$$

Figure 4.19: Illustration on the use of indices within ICSPS: i refers to the current permutation π_i, j refers to the current iteration for a given permutation $\pi_i(j)$, k indexes all robots before and excluding robot $\pi_i(j)$ and l denotes all yet unconsidered robots after and excluding $\pi_i(j)$. An *involved* robot has at least one intersection to another robot.

intervals: $[a, b]$ on the p-axis and $[c, d]$ on the q-axis. The bounding box $B(a, c, b, d)$ is empty by construction. When transitioning from j to $j+1$ (see Figure 4.16(h)), the two intervals are represented on the combined p-q-axis and the segments of $\mathcal{S}_{j+1}$ are still intersection-free with the CR of intersection 4. That is, for example, the new p-q-interval $[e, f]$ of the second segment of $\mathcal{S}_{j+1}$ in Figure 4.16(h) is a subset of $[a, b]$ and $[c, d]$ which is known to be safe between p and q. This observation applies to all CRs and previous dimensions.

4.4.6 Algorithmic Description

Within the previous sections, the methodology of the ICSPS has been explained in detail. This section completes the explanation by summarizing it by means of the pseudocode in Algorithm 4.4. Additionally, the concept of analyzing different permutations of the robots in the input is presented.

Whether ICSPS will be able to find a solution also depends on the order of how robots are being processed, that is, the order of how the CSs are being constructed. To handle this issue, the algorithm is extended to loop over a limited set of permutations π_i. Figure 4.19 visualizes the used variables for indices whereby j, k and l have already been used previously with the same semantic.

Algorithm 4.4 shows a simplified pseudocode listing for ICSPS. In Line 2, the potential subset of *involved robots* is determined that actually have intersections with others. This way, robots without any intersections in the input are ignored; this is important for the correctness of the indexing in the loops of the algorithm. Note that this does only ignore a robot that has no intersection to all other robots. In Lines 7-12, the algorithm solves for the first two robots, stores the quality and solution path if feasible or continues with the next permutation $i \to i+1$ if the input between robots $\pi_i(0)$ and $\pi_i(1)$ was already infeasible. The function Solve2D() (cf. Lines 9 and 20) takes the length of the combined and new axes, the projected timings T_{project}, the initial timings T_{init}, the current permutation π_i and the previous iteration $j-1$ as input. It returns the solution path $\mathcal{S}_j$ and its associated quality d based on the explanation in Section 4.4.3. The main loop in Line 13 iterates over all remaining robots $\pi_i(j)$ in the current permutation π_i, projects the timings onto the previous combined axis $\mathcal{S}_{j-1}$ (Lines 14-19) in order to

Algorithm 4.4 Pseudocode of the Incremental Coordination-Space Path Scheduler: the input is given by the path $\mathcal{P}_m$, the start progress σ_m, the maximum velocity $v^m_{\max}$ for every robot of the input and the set of all pairwise intersections $\mathcal{I}$. The output is the RoW assignment for every $I_{p,q} \in \mathcal{I}$ and the final quality of the solution (Equation (**??**)).

1: **procedure** ICSPS($\forall m = 1, ..., N : (\mathcal{P}_m, \sigma_m, v^m_{\max}), \mathcal{I}$)
2: Determine involved robots
3: Precompute initial timings T_{init}; set $d_{\text{final}} \leftarrow \infty$
4: Compute starting permutation π_0 based on involved robots, let $N \leftarrow |\pi_0|$
5: Precompute remaining path time $\tau(p)$ for every involved robot p
6: **while** $\pi_i = \text{NextPermutation}(\pi_{i-1})$ **do**
7: $T_{\text{projected}} \leftarrow \emptyset$ ▷ Projected timings must be computed for every permutation
8: Add entries for $\pi_i(0)$ and $\pi_i(1)$ to $T_{\text{projected}}$ from T_{init}
9: $\mathcal{S}_1 \leftarrow \text{Solve2D}(\tau(\pi_i(0)), \tau(\pi_i(1)), T_{\text{projected}}, T_{\text{init}}, \pi_i, 0)$
10: **if** $\mathcal{S}_1 = \emptyset$ **then** ▷ Already infeasible between $\pi_i(0)$ and $\pi_i(1)$?
11: Continue with π_{i+1}
12: **end if**
13: **for all** $j \leftarrow 2, ..., N-1$ **do**
14: ▷ Project timings for the construction of the next CS:
15: **for all** $k \leftarrow 0, ..., j-1$ **do**
16: **for all** $l \leftarrow j, ..., N-1$ **do**
17: Update $T_{\text{projected}}$ for all intersections between $\pi_i(k)$ and $\pi_i(l)$
18: **end for**
19: **end for**
20: $\mathcal{S}_j \leftarrow \text{Solve2D}(|\mathcal{S}_{j-1}|, \tau(\pi_i(j)), T_{\text{projected}}, T_{\text{init}}, \pi_i, j-1)$
21: **if** $\mathcal{S}_j = \emptyset$ **then** ▷ Infeasible between $\pi_i(l), \forall l \in \{j, ..., N-1\}$ and $\pi_i(k)$?
22: Continue with π_{i+1} ▷ Leave `for`-loop, restart in outer `while`-loop
23: **end if** ▷ Else: feasible up to dimension j.
24: **end for**
25: ▷ Valid solution found:
26: $G \leftarrow \text{CreateDependencyGraph}(\pi_i)$ ▷ Assess the new valid solution.
27: $d_{\text{curr}} \leftarrow \text{ComputeTimings}(G)$ ▷ See Algorithm 4.6.
28: **if** $d_{\text{curr}} < d_{\text{final}}$ **then** ▷ Solution better than previous solution?
29: $d_{\text{final}} \leftarrow d_{\text{curr}}$
30: Store RoW assignments from π_i as best result (so far)
31: **end if**
32: **end while**
33: **return** d_{final}
34: **end procedure**

solve the constructed CS in Line 20. Note that a CS can be empty (no CRs) so that $\mathcal{S}_j$ is simply the diagonal as already explained in Section 4.4.4.

Once all robots of the permutation have been processed and a valid solution for π_i was found (see Line 25), the resulting timings are computed in Line 27 using Algorithm 4.6. It basically estimates the required time for the robots in the scenario to reach their goals given the provided RoW for π_i. The details will be presented along with the optimal solver in Section 4.5.1. ComputeTimings(G) stores the value in d_{curr} (Line 27) in order to compare it against the score d_{final} of the best solution yet known. As it can be seen in Line 28, the quality is minimized over different permutations because shorter timings correspond to faster execution times. Accordingly, the RoW assignment is kept for the permutation with the smallest value stored in d_{final}. If it remains ∞, there is no solution to the input.

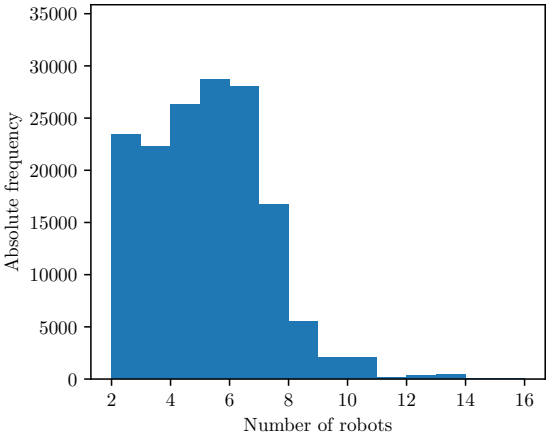
Iterating over all permutations requires $\mathcal{O}(N!)$ many processings and is therefore impractical. Possible termination criteria considered in NextPermutation() (cf. Line 6) include a fixed number of permutations, a time-bound for the execution of the algorithm, a threshold for relative changes of the quality, searching until a first feasible solution (if any) is found, or a combination of them. Alternatively, (pseudo-) random permutations could also be used to randomly sample the optimization space. Because the optimization space is large and unknown, this is difficult to judge and will be evaluated in the next section.

4.4.7 Evaluation

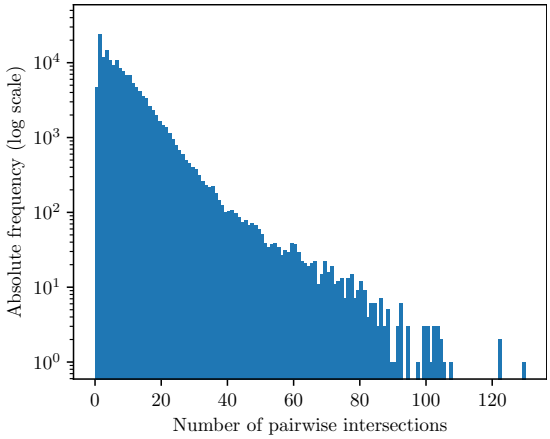
This section deals with the experimental evaluation of ICSPS. It is based on 1) the two scenarios already presented for the evaluation of SGS and MGS in Section 4.3.4 and 2) two large simulation generated datasets which will be presented in the following.

The Robotic Experimentation Framework (REF) from Chapter 3 has been used to generate input scenarios for the evaluation based on random goals in a shop-floor-like environment in such a way that intersections between the paths are more likely to stress-test the proposed algorithm. A single scenario is stored as a file containing the identifiers, paths, start progresses, radii and maximum velocities of all involved robots. Figure 4.20 shows the distribution of robots and pairwise intersections in datasets 1 (with 156 315 scenarios) and 2 (with 118 683 scenarios) respectively. As it can be seen when comparing Figure 4.20(a) with (c), dataset 2 tends to contain larger scenarios while dataset 1 exhibits a larger number of scenarios with up to six robots. Figure 4.20(b) and (d) depicts the absolute frequencies of pairwise intersections, indicating that they are very similar. Note that the ordinate is scaled logarithmically.

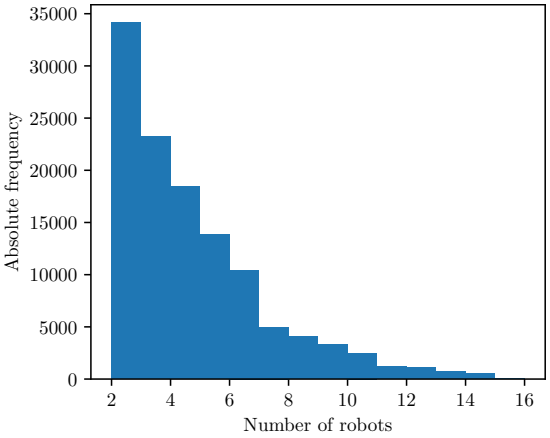
Two quality measures are mainly used throughout the experiments. The *total travel time* (TTT) T_{total} is given by the sum of travel times for all N robots in a scenario, including



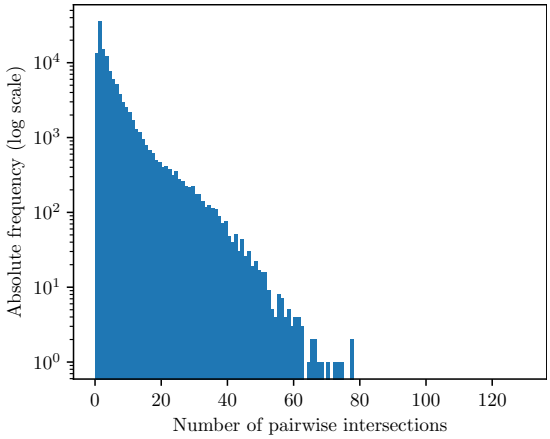
(a) Absolute frequencies of robots participating in the scenarios of dataset 1



(b) Absolute frequencies of pairwise intersections in the scenarios of dataset 1



(c) Absolute frequencies of robots participating in the scenarios of dataset 2



(d) Absolute frequencies of pairwise intersections in the scenarios of dataset 2

Figure 4.20: Distributions of the quantities of robots and intersections for the simulation-generated dataset 1 (156 315 scenarios) and dataset 2 (118 683 scenarios) used for the evaluation of ICSPS. Notice the logarithmic scaling of the ordinates in (b) and (d).

approximated motion times t_m (based on maximum velocities, cf. Section 4.4.2) and all waiting times t_w:

$$T_{\text{total}} \coloneqq \sum_{i=1}^{N} t_m^i + t_w^i. \tag{4.65}$$

The *critical path time* (CPT) T_{critical} is the time it takes for the longest moving robot to reach its goal among all robots in a given scenario:

$$T_{\text{critical}} \coloneqq \max\{\, t_m^i + t_w^i \mid \forall i \in \{1, \ldots, N\} \,\}. \tag{4.66}$$

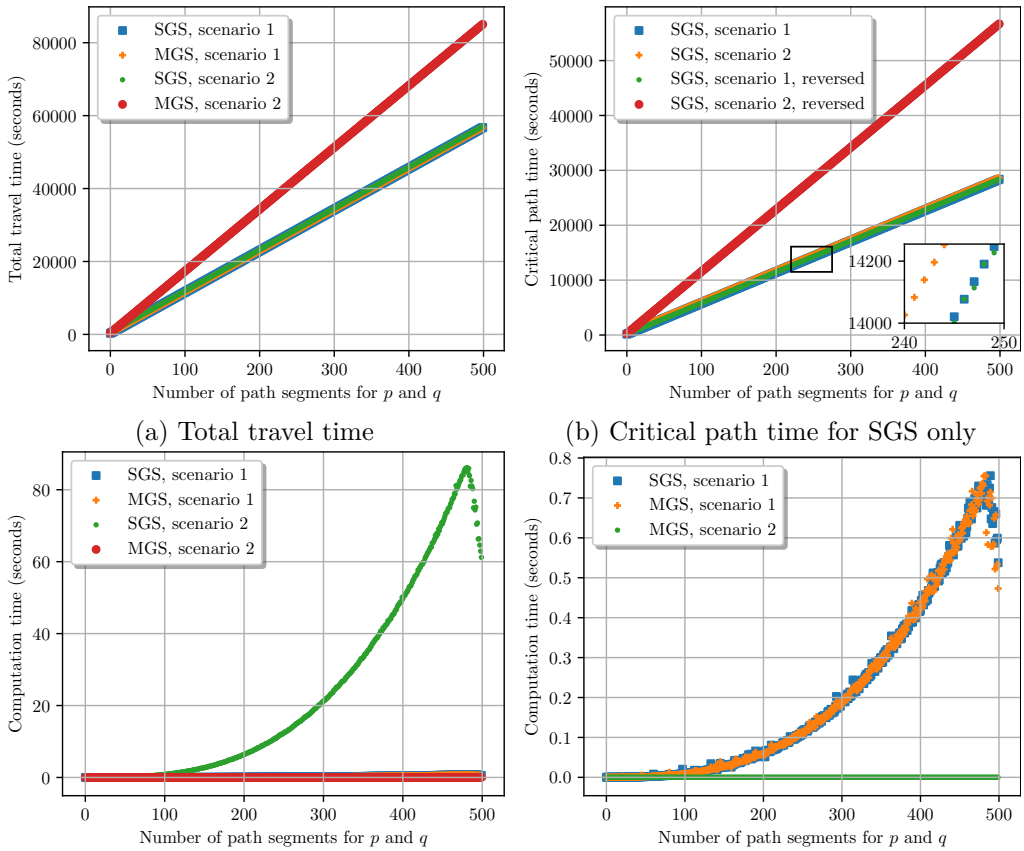
Note that $T_{\text{total}} \geq T_{\text{critical}}$ (equality is possible if there is just one robot). In operations research and scheduling, the critical path time is often also referred to as *makespan* [1], and it addresses the problem of minimizing T_{critical}.

4.4.7.1 Impact of Conflict Detection

We will first examine the impact of SGS and MGS (see Sections 4.3.2 and 4.3.3) on the algorithm based on the two scenarios 1 and 2 from Section 4.3.4 (see Figure 4.14) as their outputs are the most relevant inputs to ICSPS (cf. Figure 4.15). Figure 4.21 shows the results w. r. t. (a) the TTT, (b) the CPT and (c)–(d) the computation time of the solver itself. All results are computed with ICSPS using A* and the maximum metric with a permutation limit of 500.

Figure 4.21(a) shows the TTT for all robots in a scenario (ordinate) given the generated paths of different lengths (abscissa). Generally, the more path segments are being generated, the higher the TTT in a scenario. There is no difference for the solver between scenario 1 and 2 w. r. t. the TTT (blue squares and small green circle). In scenario 2, the TTT is considerably larger for MGS (all merged to one, red big circles) than for SGS. However, this depends on the input as it only applies if the input paths have the same directions (as shown here). Conceptually, the performance of SGS can also be achieved with MGS by allowed partially releases of (long-drawn-out) intersections. Additionally, it should be noted that this apperent advantage of SGS over MGS does also not come into play if very long segments are involved. Such cases would also benefit from a partial release strategy during execution. Note that the diagram looks about the same here if the CPT is used as criterion instead of TTT (ordinate): similarly, the CPT also increases with increasing path length (abscissa). When reversing one of the paths, the results are all equal to the result for MGS on scenario 2 (big red circles) because the robot not getting RoW must wait until the other robot has completely passed through the intersection area.

In Figure 4.21(b), the resulting CPT is visualized for both scenarios (and SGS only for readability), including their reversed variants. Notice how the scale of the ordinate has changed compared to (a). Similar to the TTT, the CPT increases with increasing segments in the inputs. Notably, SGS performs equal to MGS (not shown here) when



(a) Total travel time

(b) Critical path time for SGS only

(c) Computation time

(d) Like (c) but without SGS on scenario 2

Figure 4.21: Impact on ICSPS given the output of SGS and MGS on scenarios 1 and 2, i. e., procedurally generated paths up to length 500 (abscissas) with A* using the maximum metric, see Figure 4.14

one of the path's direction is reversed (cf. big red circles for SGS and scenario 2). This is justified by the fact that none of the segments can be processed until all adjacent intersecting segments have been passed. Interestingly, when looking at the zoomed view (cf. black rectangle) on the lower right in (b), there are subtle differences in scenario 1 (blue squares) compared to the same scenario with one path reversed (small green circles). Without reversing the direction, one robot (not getting RoW at the very first intersection) has to wait shortly for release until it can drive behind the robot getting RoW (at all intersections at once). Waiting for release just causes a small delay (making this robot the one with the critical path) which is added to the motion time, contributing to the CPT (no additional waits required). For the reversed scenario 1, two cases need to be distinguished. With an even number N of generated path segments, robots moving in opposite directions will meet in the middle where both paths move away from each other so that both robots can pass one another without waiting.[5] In contrast, given an

[5]This is also possible due to separated intersections returned by SGS.

odd number N of segments both robots meet at the intersection of their paths in the middle which requires waiting shortly as well. Thus, in the non-reversed scenario 1, a robot always has to wait (although alternating data points as blue squares and green small circles appear to be identical in the zoomed view). For instance, for a path length of $N = 300$, the CPT for the non-reversed scenario 1 is 17 075.42 s and 17 055.42 s for the reversed one. In case of an even N, the CPT drops a little more because in such cases, no waiting for release is necessary in the reversed scenario 1 (small green circles). This justifies the alternating visually perceptible differences in the zoomed view between the blue squares (scenario 1) and the small green circles (reversed scenario 1). Finally, the CPT in scenario 2 increases a bit more (cf. orange crosses) compared to scenario 1 (blue squares) due to the increased intersection areas ($r_{pq} = 50$ instead of $r_{pq} = 10$).

Figure 4.21(c) and (d) depict the required computation time (ordinate) of the solver given the differently sized inputs (abscissa) for both scenarios. Clearly, the larger the input the more time it takes to solve. A lot of time is needed by SGS for scenario 2 (small green circles), see Figure 4.21(c). This is caused by many segment pairs that are within a distance of r_{pq} to each other, causing SGS to encounter many of such pairs comprising a (small) intersection (cf. Figure 4.14(b)). Because MGS is able to merge this, its runtime appears to be constant (big red circles, orange crosses).

Because the scale of the ordinate is enlarged due to the huge amount of time required by solving the output of SGS for scenario 2, Figure 4.21(d) shows the same plot without the results for SGS on scenario 2. This reveals the huge improvement of MGS over SGS. For scenario 1, both are on par (upper two curves). For scenario 2 where MGS is able to merge all intersections to a single one, ICSPS requires only constant time (small green circles, even on a larger scale) because its input is constant regardless of the path length (abscissa). The polynomial

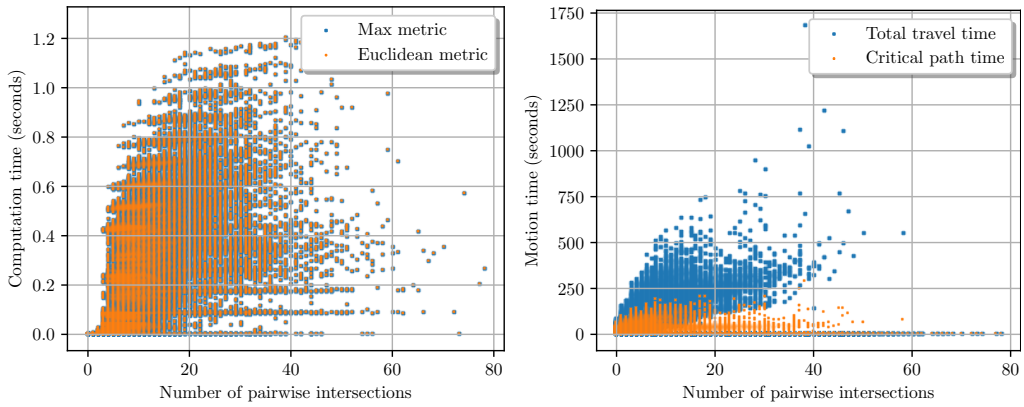
$$2.7011 \cdot 10^{-9} n^3 + 3.9987 \cdot 10^{-7} n^2 - 2.1604 \cdot 10^{-5} n + 0.0005$$

with error 0.007 1 was fitted to the points of MGS with scenario 1 (using data generated in single threaded mode) which suggests runtime in the order of $\mathcal{O}(n^3)$ for *this type of input* whereby n is the number of input segments for both paths (not the number of intersections). Finally, note that the drop on the right when approaching 500 segments is caused by multithreading artifacts and can be ignored. It does not happen in single threaded mode.

Finally, it should be noted that all this applies to the *specifically* generated scenarios only, that is, the quality and runtime highly depends on the type of input and it is impossible to generalize this. For that reason, we will now investigate the algorithm with the two generated dataset presented at the beginning.

4.4.7.2 Simulation-generated Inputs

As already indicated in Section 4.4.3, ICSPS with A* can be used with different metrics, typically Euclidean and maximum metric. Figure 4.22 therefore presents (a) the compu-



(a) Runtime of ICSPS with A* and max (blue) and Euclidean (orange) metric

(b) Total travel (blue) and critical path time (orange) for ICSPS with A* and max metric

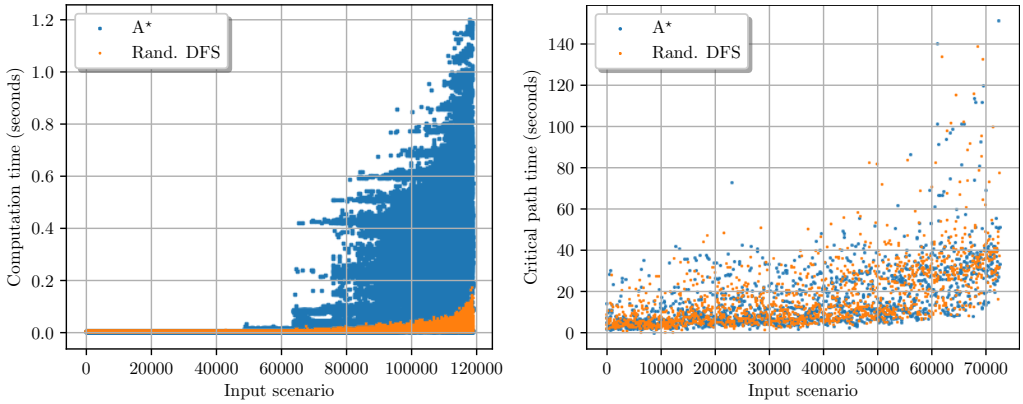
Figure 4.22: (a) computation time and (b) motion times for the 118 683 scenarios from dataset 2 (sorted in ascending order according to the intersection count) in relation to the number of pairwise intersections

tation time of both aforementioned metrics and (b) the two criteria TTT and CPT for A* with max metric only. Results have been generated using dataset 2 and sorted in ascending order according to the intersection count.

Computation times for both metrics are roughly the same as visualized in Figure 4.22(a). Statistically, the average solver runtime was 45.04 ms for the Euclidean (orange) and 45.16 ms for the max metric (blue). 19 out of 118 683 scenarios were solved differently while the Euclidean variant was able to solve one more scenario than the max variant (72435 vs. 72434, 61.03 % of the input dataset).[6] ICSPS's runtime roughly increases with the number of robots and intersections (cf. Figure 4.22). However, there is no obvious dependency as it highly depends on the characteristics of the input. For instance, there can be many robots in the input that do not have many intersections although in terms of probabilities, the risk of intersections naturally increases with an increased number of robots. Similarly, the higher the number of intersections, the longer it takes solving them, cf. Figure 4.22(a). Notice that the scatter plot is largely spread out, underpinning the dependency on the input.

Negative values in Figure 4.22(b) indicate infeasible scenarios (or at last scenarios where the solver was not able to find a solution). Because the TTT (blue) contains the CPT (orange), the blue points are primarily located above the orange ones. Tendentially, the larger number of intersections (abscissa), the more time it takes for the involved robots to reach their goals. However, as already explained w. r. t. runtime (see (a)), a larger number of intersections does not necessarily imply higher motion times (dependency on the characteristics of the input).

[6] Note that this does not (yet) state anything about the amount of scenarios in the dataset that can be solved at all because a permutation limit of 500 (termination criterion) was used.



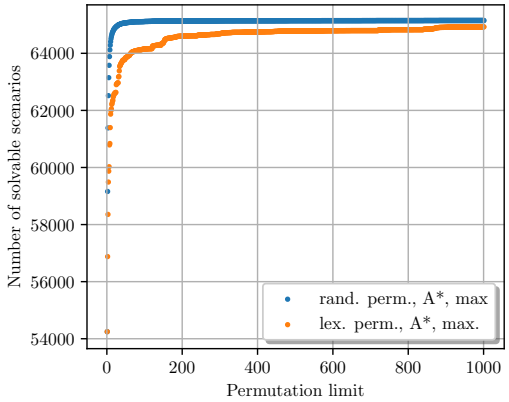
(a) Solver runtime (in seconds) of randomized Depth-First Search (DFS) (orange) compared to A* (blue) with max metric used in ICSPS; the results have been sorted according to the intersection count in ascending order

(b) Critical path time of ICSPS with randomized DFS (orange) vs. A* (blue) filtered for successfully solved scenarios only and downsampled to every 50-th scenario for improved readability

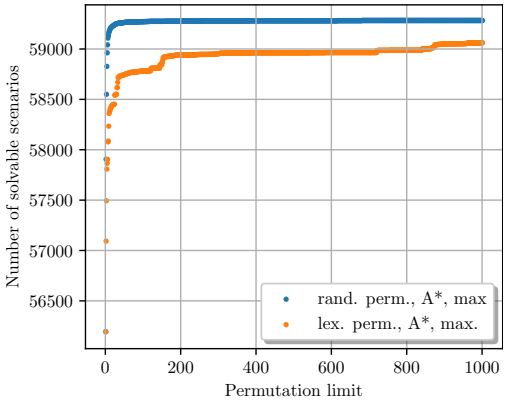
Figure 4.23: (a) runtime and (b) critical path time for the 118 683 scenarios in dataset 2; the abscissas show the indexed scenarios, sorted according to the intersection count in ascending order

We will now compare randomized DFS with A* (based on the max metric). In randomized DFS, the path from the lower left (start) to the goal in the incrementally constructed coordination spaces is computed by randomly selecting a neighboring vertex in the CVG (see Section 4.4.3). Figure 4.23(a) presents the runtime of ICSPS with A* (blue) and randomized DFS (orange). On average, randomized DFS is one order of magnitude faster than A* due to its simplicity (2.46 ms vs. 45.16 ms per solver execution). According to Figure 4.23(b), there is no notable difference regarding the achieved CPT between both variants (the plot for the TTT looks very similar). However, statistically, the average CPT of A* (17.94 s) is marginally better than for randomized DFS (18.06 s) among all 118 683 input scenarios.

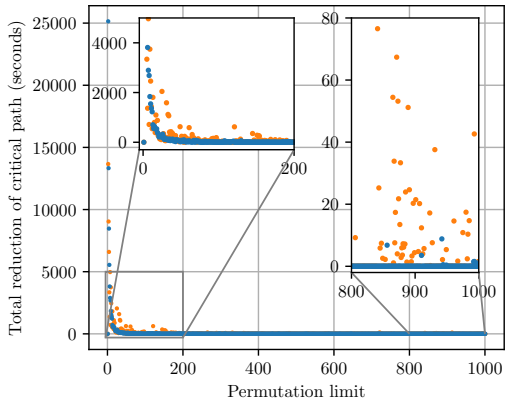
ICSPS uses different permutations of the robots in an input scenario for constructing coordination spaces (see Line 6 in Algorithm 4.4). This is favorable because the order in which coordination space are constructed has an impact on the resulting solution quality (and even solvability at all). Because the number of permutations of N robots is $N!$, a limit is required to bound the computation time in practice. Figure 4.24 compares the impact of the permutation limit (abscissa) on randomly selected permutations (blue) with lexicographically sorted permutations (orange) w. r. t. solvability and CPT (ordinate) for both datasets. As it turns out based on Figure 4.24(a) and (b), using random permutations has the advantage of being able to already solve the majority of inputs after a limit of just 100 permutations. In case of lexicographically sorted permutations, a higher limit of at least 350 is advised because some inputs require a large



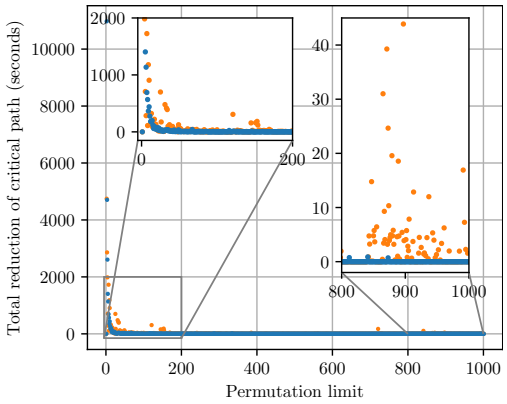
(a) Impact of the permutation limit on the number of solvable input scenarios in dataset 1



(b) Impact of the permutation limit on the number of solvable input scenarios in dataset 2



(c) Impact of the permutation limit on the critical path time in dataset 1

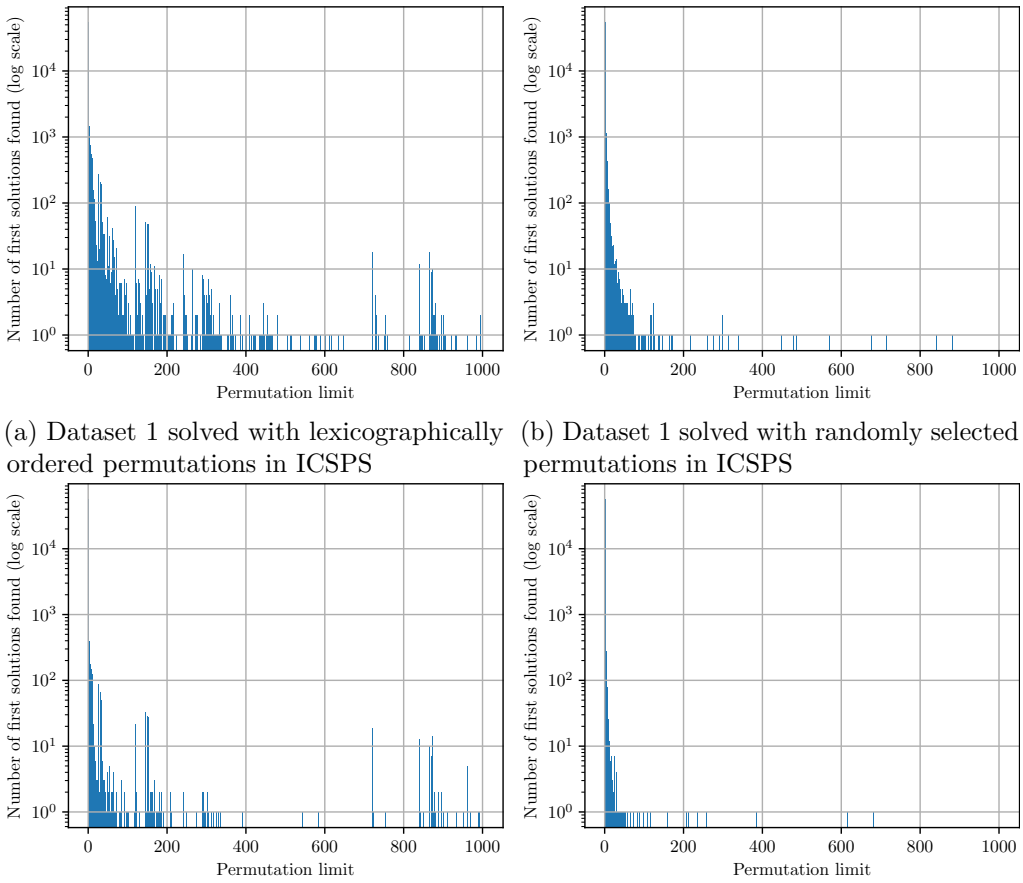


(d) Impact of the permutation limit on the critical path time in dataset 2

Figure 4.24: Impact of the permutation limit (order of how ICSPS solves an input) on the solvability (see (a) and (b)) and total sum of improvements of the critical path time (see (c) and (d)) for randomly sampled permutations (blue) and the first lexicographically ordered permutations (orange). All results are shown for ICSPS with A* and maximum metric.

number of permutations to be solvable at all. Correspondingly, the overall solvability is slightly higher in both datasets when using random permutations, i. e., 69 536 (44.48 %) vs. 69 759 (44.63 %) for dataset 1 and 72 533 (61.11 %) vs. 72 754 (61.30 %) for dataset 2. Note that, unlike Figure 4.24(a) and (b), these numbers *include* scenarios with a pairwise intersection count of zero (4 609 for dataset 1 and 13 471 for dataset 2), i. e., they are simply considered as solved successfully here.

Figure 4.24(c) and (d) visualize the impact of the permutation limit on improvements of CPT for all scenarios. More specifically, given the results for all scenarios and all permutations up to limit 1000 (abscissa), the points in the plot are computed by



(a) Dataset 1 solved with lexicographically ordered permutations in ICSPS

(b) Dataset 1 solved with randomly selected permutations in ICSPS

(c) Dataset 2 solved with lexicographically ordered permutations in ICSPS

(d) Dataset 2 solved with randomly selected permutations in ICSPS

Figure 4.25: Total number of first solutions found (depicted on the ordinates with logarithmic scale) for all input scenarios from datasets 1 (a, b) and 2 (c, d) depending on the permutation limit (abscissas). All results are shown for ICSPS with A* and max metric.

summing up all improvements in the CPT for a particular limit. A point at some limit i in the plot remains zero ("no improvement") if none of the scenarios was improved in the i-th permutation or if it was the first valid solution for that input. Interestingly, lexicographically sorted permutations (orange) have a larger spread when compared to the randomly sampled ones (blue). Nonetheless, only lexicographically sorted permutations guarantee exhaustive search if the limit is greater or equal to $N!$ (i. e., for small inputs). This is because random permutations are simply drawn with replacement until the limit is reached. The scatter plots for both types of generating permutations tend to drop quite quickly. Randomly sampled permutations, however, can cause improvements in higher permutations limits for small inputs as well due to sampling with replacement as indicated in the zoomed parts of the plots. Also notice that improvements in higher

limits (e. g., $> 6! = 720$) for lexicographically sorted permutations must originate from inputs with a larger number (> 6) of robots. It is important to note that the absolute values of both ordinates in Figure 4.24(c) and (d) do *not* have direct expressiveness w. r. t. the scenarios themselves because it shows the sum of all improvements for all scenarios of a dataset. One can conclude that random permutations result in a higher solvability while lexicographically sorted permutations seem to provide a slightly better CPT (56.89 s vs. 57.07 s).

Note that in an experiment with random permutations and randomized DFS, the results got worse compared to both random permutations only and without any randomization at all.

Finally, Figure 4.25 depicts the relevant permutations where a first solution was found by the solver, both for randomly selected ((a) and (c)) and lexicographically sorted permutations ((b) and (d)). Notice that the ordinates are scaled logarithmically, that is, the majority of solutions is already found in the first iteration (also cf. Figure 4.20). The plots also reveal that a larger number of solutions is found with a smaller number of permutations when using random selection, as already explained along with Figure 4.24.

As a conclusion, it is advised to use lexicographically sorted permutations for small scenarios where an exhaustive search is practically feasible and random permutations for complex inputs and those where no solution was found in the first place. Employing a threshold for relative changes of the quality does not guarantee convergence. However, it could be used in combination with another termination criterion (e. g., a permutation limit).

4.5 Optimal Multi-Robot Scheduling

In this section, the methodology and implementation of the *Optimal Multi-Robot Path Scheduler* (OMRPS) is presented (Section 4.5.1), that is, it aims to solve the problem described in Section 4.2.2. Due to its exponential run time complexity, it is intended to be used for small inputs only as well as to evaluate how close ICSPS (see Section 4.4) gets towards the optimum in terms of solvability and solution quality. After proofing the correctness of OMRPS in Section 4.5.2, applied concepts of parallelization will be described in Section 4.5.3. As it turns out in the experimental evaluation in Section 4.5.4, the suggested algorithm performs even faster than ICSPS for very small inputs and is therefore also applicable in practice.

The suggested algorithm produces *optimal* results in a sense that it either minimizes the TTT (see Equation (4.65)) or the CPT (see Equation (4.66)), given the underlying model and assumptions (constant velocity) hold.

As a sidenote, it should be remarked how infeasible robots are handled in an input. Due to the way the proposed algorithm proceeds to find the optimal schedule (cf. Section 4.5.1), it would always return infeasible even if only one robot in the input is infeasible (partial

infeasibility). For that reason, (statically) infeasible robots are detected upfront and excluded from the input. This is based on the existence of halt and release points. For example, if two robots have an intersecting goal area, there are no release points for both of them (because the intersection at their goals is never released). In such a situation, both robots are known to be infeasible. Nonetheless, this does not handle all cases of (dynamic) infeasibility, i. e., there can be inputs where the RoW at intersections is not predetermined by their associated halts and releases, although a subset of intersections is infeasible. Such a subset comprises a *deadlock* and is discussed in the following too.

4.5.1 Algorithmic Description

Algorithm 4.5 depicts the stripped-down pseudocode of OMRPS. Based on the input paths $\mathcal{P}_m$, the start progress σ_m and the maximum velocity $v_{\max}^m$ for every robot m as well as the set of pairwise intersections $\mathcal{I}$ along with associated halts and releases, it determines the optimal RoW assignment $\delta(I_{p,q})$ for all intersections $I_{p,q} \in \mathcal{I}$ (induced by the vector $\boldsymbol{\Delta}_{\text{opt}}$, see Line 32).

We are now going to explain Algorithm 4.5 in more detail. At the beginning in Line 3, all predetermined pairwise intersections are collected from the input. An intersection is *predetermined* if and only if one of its associated event points does not exist, that is, if and only if either a halt or release (or both) do not exist.

The underlying idea of the optimal solver is to define a decision vector

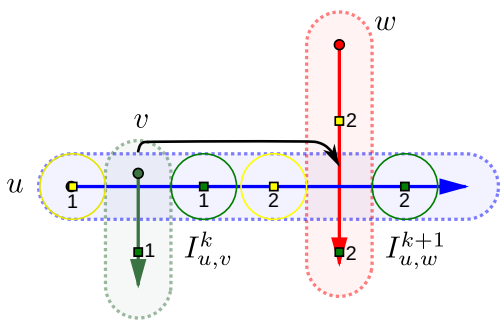
$$\boldsymbol{\Delta} \coloneqq (\delta_0, \dots, \delta_k, \dots, \delta_K) \tag{4.67}$$

with an entry δ_k for every pairwise intersection $I_{p,q}^k \in \mathcal{I}$ storing which robot gets RoW at that intersection. In principle, every entry is binary-valued but since we need to store predetermined decisions as well (being constant within the solver), additional values are required during the execution of the algorithm. In Line 5, the decision vector is initialized based on the collected predetermined intersections $\mathcal{I}_{\text{pd}}$ with values 0_{pd}, 1_{pd} and $\emptyset$ (if the decision must be enumerated and optimized, refer to Lines 21-30). The RoW at such intersections is already known a priori because a robot without a halt must already be located inside the intersections (requiring RoW to prevent a crash) and a robot without a release will never release the intersection (not getting RoW to prevent blocking it for the other robot).

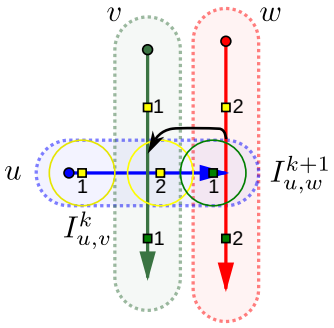
If, for instance, both releases are not existing, the intersection is known to be infeasible. All possible combinations of the existence of halts and releases for an intersection are tested in Lines 3 and 5 in order to correctly classifiy all predetermined intersections and infeasible robots known a priori. The Lines 8-18 are responsible for finding all directly and transitively infeasible robots in the input. Such robots are then excluded (see Line 13) to allow finding a solution for the remaining intersections (partial solution). Note that if they would be kept in the input, none of the enumerated solutions would be valid in Line 23. Infeasible robots are assumed to stay on their starting position. In the

Algorithm 4.5 Pseudocode of the Optimal Multi-Robot Path Scheduler algorithm: the input is given by the path $\mathcal{P}_m$, the start progress σ_m, the maximum velocity $v^m_{\max}$ for every robot of the input and the set of all pairwise intersections $\mathcal{I}$. The output is the optimal RoW assignment for every $I_{p,q} \in \mathcal{I}$ and the final quality of the solution according to the TTT or CPT.

```
1: procedure OptSchedule(∀m = 1, ..., N : (P_m, σ_m, v^m_max), I)
2:     ▷ Find infeasible and predetermined intersections in the input:
3:     I_pd ← FindPredeterminedIntersections( )
4:     ▷ Setup decision vector Δ_0 based on predetermined intersections:
5:     Δ_0 ← InitializeDecisionVector(I, I_pd)
6:     ▷ Handle directly and transitively infeasible robots that are a priori
7:     ▷ known based on the existence of halts and releases:
8:     for all infeasible robots r do
9:         Push(r, Q)                                  ▷ Add robot r to queue Q.
10:    end for
11:    while Q ≠ ∅ do                                  ▷ Propagate infeasibility.
12:        r ← Pop(Q)
13:        Exclude(r, Δ_0)                  ▷ Remove infeasible robots and intersections.
14:        R_dep ← FindDependentRobots(r, I)
15:        for all r ∈ R_dep do
16:            Push(r, Q)                ▷ Add dependent infeasible robots to queue, too.
17:        end for
18:    end while
19:    c_opt ← ∞
20:    i ← 1
21:    while Δ_i = Enumerate(Δ_{i−1}) do               ▷ Perform exhausive search.
22:        G_i ← CreateDependencyGraph(Δ_i)
23:        if G_i is acyclic then                      ▷ Is Δ_i a valid solution?
24:            v_i ← ComputeTimings(G_i)                       ▷ See Algorithm 4.6.
25:            if v_i < c_opt then                     ▷ Minimize waiting times.
26:                c_opt ← v_i
27:                Δ_opt ← Δ_i                  ▷ Remember improved RoW assignment.
28:            end if
29:        end if
30:    end while
31:    if c_opt ≠ ∞ then                              ▷ Solved optimally.
32:        return Δ_opt
33:    else                                           ▷ No solution found (infeasible).
34:        return ∅
35:    end if
36: end procedure
```



(a) Schematic visualization of the Forward Dependency Condition (FDC): the dependency is given alongside u's path towards the goal if u does not get RoW at $I^k_{u,v}$ (all subsequent intersections are affected)



(b) Schematic visualization of the Backward Dependency Condition (BDC): the dependency is given alongside u's path towards the start if u gets RoW at $I^k_{u,v}$ only (but not at $I^{k+1}_{u,w}$)

Figure 4.26: Schematic visualizations of the Forward and Backward Dependency Conditions for three robots u (blue), v (green) and w (red) as well as two dependent pairwise intersections $I^k_{u,v}$ and $I^{k+1}_{u,w}$ (with $k = 1$). According to Equations (4.68) and (4.69), edges $(I^k, I^{k+1}) \in G_{\mathbf{\Delta}}$ (black arrows) for the FDC and BDC in an underlying dependency graph $G_{\mathbf{\Delta}}$ model the possible effects on the waiting time at the intersection I^{k+1} pointed to by the edge. Notice the common robot u in both figures whose guards are of interest only here.

loop in Lines 8-10, all directly infeasible robots are stored in a queue $\mathcal{Q}$ based on $\mathcal{I}_{\text{pd}}$. Other robots passing through the start of an infeasible robot are considered transitively infeasible and will be detected in Lines 11-18. This effectively propagates infeasibility by following dependent robots.

All intersections marked with $\emptyset$ in the decision vector need to be decided optimally by the solver in the subsequent steps (Lines 21-30) while taking into account all predetermined intersections (0_{pd}, 1_{pd}). The solver exhaustively enumerates all possible combinations of the decision vector $\mathbf{\Delta}_i$ in order to test if the given candidate is a valid solution and to then evaluate its quality. The validity test and the quality evaluation is based on a dependency graph G_i created in Line 22 for a given enumerated candidate $\mathbf{\Delta}_i$. They are now explained in more detail as it is the essential part of the algorithm.

A given decision vector (candidate) $\mathbf{\Delta}$ specifies a complete potential solution for the input as it contains the RoW assignment for every pairwise intersection. For every candidate, a (directed) dependency graph $G_{\mathbf{\Delta}} = (\mathcal{V}, \mathcal{E})$ is constructed (Line 22) which reflects the dependencies between the intersections regarding the (computation of the) waiting times of the robots. The graph is based on the model in which the time required by the robots to reach their goals is given by the fixed travel time plus the waiting time at every intersection (if any) whereby waiting times solely depend on RoWs. The set $\mathcal{V}$ contains a vertex for every pairwise intersection $I_{u,v} \in \mathcal{I}$. Edges are being created based on two conditions which model the dependencies for the computation of the waiting

times, see Figure 4.26. Let $I^k_{u,v}$, $I^{k+1}_{u,w}$ be two pairwise intersections with associated robots u, v and w respectively. That is, both intersections share a common robot u or, in other words, u is involved in both intersections. The *Forward Dependency Condition* (FDC) is defined by (cf. Figure 4.26(a)):

$$h(I^k_{u,v}, u) < h(I^{k+1}_{u,w}, u) \wedge \delta_{\boldsymbol{\Delta}}(I^k_{u,v}) = v \tag{4.68}$$

and the *Backward Dependency Condition* (BDC) is defined by (cf. Figure 4.26(b)):

$$\begin{aligned} h(I^{k+1}_{u,w}, u) \geq h(I^k_{u,v}, u) \wedge \\ r(I^k_{u,v}, u) > h(I^{k+1}_{u,w}, u) \wedge \\ \delta_{\boldsymbol{\Delta}}(I^k_{u,v}) = u \wedge \delta_{\boldsymbol{\Delta}}(I^{k+1}_{u,w}) = w. \end{aligned} \tag{4.69}$$

A directed ("forward") edge $(I^k_{u,v}, I^{k+1}_{u,w}) \in \mathcal{E}$ is created between every two vertices $I^k_{u,v}$, $I^{k+1}_{u,w} \in \mathcal{V}$ if Equation (4.68) holds and the reversed ("backward") edge $(I^{k+1}_{u,w}, I^k_{u,v}) \in \mathcal{E}$ is created if Equation (4.69) holds. Notice that both conditions are expressed for the common robot u and that the conditions are mutually exclusive. Figure 4.26 visualizes the two conditions schematically. The FDC in Figure 4.26(a) can be explained as follows: if robot u (blue) has to wait ($\delta_{\boldsymbol{\Delta}}(I^k_{u,v}) = v$) at its first intersection $I^k_{u,v}$ (with v, green), it will arrive later at its next intersection $I^{k+1}_{u,w}$ (with w, red) such that a dependency is present, pointing forwards along u's path (indicated by the black arrow). This situation can be detected by the order of the halts (yellow circles) of u. Similarly, the BDC in Figure 4.26(b) models dependencies that are pointing backwards along a path (towards the start). If u (blue) gets RoW at its first intersection $I^k_{u,v}$ ($\delta_{\boldsymbol{\Delta}}(I^k_{u,v}) = u$) but not at $I^{k+1}_{u,w}$ ($\delta_{\boldsymbol{\Delta}}(I^{k+1}_{u,w}) = w$), being "very close" to the previous one, u will still block $I^k_{u,v}$ while waiting for the release of $I^{k+1}_{u,w}$. That is, there is a dependency from $I^{k+1}_{u,w}$ to $I^k_{u,v}$ pointing backwards along u's path (indicated by the black arrow). The proximity between the two intersections is expressed by $r(I^k_{u,v}, u) > h(I^{k+1}_{u,w}, u)$ and is essential. Note that the corner cases (w. r. t. the non-existence of halts and releases) depicted in Figure 4.26 are *not* necessary for the conditions but used to emphasize RoW requirements.[7] The application of these conditions onto all pairs of intersections with a common robot yields the set of edges in the graph.

A candidate $\boldsymbol{\Delta}$ inducing circular waiting dependencies causes a deadlock (cf. Section 4.2.2). Because all waiting dependencies have been represented as edges, testing $G_{\boldsymbol{\Delta}}$ for being acyclic suffices to ensure that $\boldsymbol{\Delta}$ is valid (see Line 23). For all remaining valid candidates, either the TTT or the CPT (parameterizable) is computed in Line 24. The resulting timings are minimized among all candidates because shorter timings are preferred.

For a given candidate $\boldsymbol{\Delta}$, both the TTT and the CPT can be computed based on $\mathcal{G}_{\boldsymbol{\Delta}}$ which is a directed acyclic graph, see Algorithm 4.6 (which has also been used to assess

[7] In fact, recall that due to the enumeration of all possible RoW decisions, every possible value is assigned to $\delta(\cdot)$, represented as $\boldsymbol{\Delta}$.

Algorithm 4.6 Pseudocode of the algorithm for computing the timings of a given valid solution $\boldsymbol{\Delta}$. The input is given by the dependency graph $G_{\boldsymbol{\Delta}}$, the path $\mathcal{P}_m$, the start progress σ_m, the maximum velocity $v^m_{\max}$ for every robot m and the set of all pairwise intersections $\mathcal{I}$ (implicitly referenced by the vertices of $G_{\boldsymbol{\Delta}}$). The output is the TTT or the CPT of the scenario (parameterizable) based on the underlying solution $\boldsymbol{\Delta}$.

procedure ComputeTimings($G_{\boldsymbol{\Delta}}$, $\mathcal{I}$, $\forall m = 1, ..., N : (\mathcal{P}_m, \sigma_m, v^m_{\max})$)
  for all $m = 1, ..., N$ **do** ▷ Initialize total waiting times for every robot.
    $\tau_R(m) \leftarrow 0$
  end for
  for all $I \in \mathcal{I}$ **do** ▷ Initialize waiting times for every intersection.
    $\tau_I(I) \leftarrow -1$
  end for
  for $I_{p,q} \in$ TopologicalSort($G_{\boldsymbol{\Delta}}$) **do** ▷ Iterate through topological ordering.
    ▷ Determine robot v that gets RoW and other robot u not getting it:
    if $(\boldsymbol{\Delta}[I_{p,q}] = 1 \wedge p < q) \vee (\boldsymbol{\Delta}[I_{p,q}] = 0 \wedge p > q)$ **then**
      $(u, v) \leftarrow (q, p)$
    else ▷ q gets RoW.
      $(u, v) \leftarrow (p, q)$
    end if
    $t_u = \tau_R(u) + \frac{dist(\sigma_u, h(I_{p,q}, u))}{v^u_{\max}}$ ▷ Time of u until reaching its halt.
    $t_v = \tau_R(v) + \frac{dist(\sigma_v, r(I_{p,q}, v))}{v^v_{\max}}$ ▷ Time of v until reaching its release.
    $\tau_I(I_{p,q}) \leftarrow \max(t_v - t_u, 0)$ ▷ Waiting time at $I_{p,q}$ only.
    if $|\mathcal{S}(\boldsymbol{\Delta}, u, I_{p,q})| > 1$ **then** ▷ See Equation (4.73).
      if $\forall I \in \mathcal{S}(\boldsymbol{\Delta}, u, I_{p,q}) : \tau_I(I) \geq 0$ **then** ▷ All equal halts processed?
        $\tau_R(u) \leftarrow \max_{\forall I \in \mathcal{S}(\boldsymbol{\Delta}, u, I_{p,q})}(\tau_I(I)) + \tau_R(u)$
      end if
    else
      $\tau_R(u) \leftarrow \tau_I(I_{p,q}) + \tau_R(u)$ ▷ Additional waiting time for u at $I_{p,q}$.
    end if
  end for
  $(t_{\text{TTT}}, t_{\text{CPT}}) \leftarrow (0, -\infty)$
  for all $m = 1, ..., N$ **do** ▷ Compute resulting timings given the waiting times.
    $t \leftarrow \frac{\text{dist}(\rho^{-1}(\sigma_r),\ \rho^{-1}(|\mathcal{P}_m| - 1))}{v^m_{\max}}$ ▷ Compute pure travel time for robot m.
    $T \leftarrow t + \tau_R(m)$ ▷ Total time: travel time + waiting time.
    $t_{\text{TTT}} \leftarrow T + t_{\text{TTT}}$
    $t_{\text{CPT}} \leftarrow \max(T, t_{\text{CPT}})$
  end for
  return t_{TTT} or t_{CPT}
end procedure

a solution within ICSPS). At first (see Lines 2-7), the total waiting time $\tau_R : \mathcal{R} \mapsto \mathbb{R}$ for every robot $u \in \mathcal{R}$ and the individual waiting time $\tau_I : \mathcal{I} \mapsto \mathbb{R}$ for an intersection is initialized to 0 and -1 respectively. The negative values for $\tau_I()$ indicate that it has not been processed because valid waiting times must always be ≥ 0. By iterating through the topological sorting of $G_{\boldsymbol{\Delta}}$ (cf. Line 8), all dependencies of visited intersections are considered in the correct order. Thus, at every visited intersection $I_{p,q}$, we are allowed to compute the waiting time for that intersection because all previously occurring waiting times have already been computed and therefore been taken into account. The computation (see Lines 10-25) is carried out as follows: let p and q be the associated robots of the intersection $I_{p,q}$ and, w. l. o. g., assume that q gets RoW. This is explicitly formulated in Lines 10-14 where variables u and v (representing robots) are assigned such that v always gets the RoW. Formally, we are interested in computing the waiting time for p and q at $I_{p,q}$. First, we have to compute the time (cf. Line 15)

$$t_p = \tau_R(p) + \frac{dist(\sigma_p, h(I_{p,q}, p))}{v^p_{\max}} \tag{4.70}$$

which p requires to reach its halt $h(I_{p,q}, p)$. It adds up the yet accumulated waiting time $\tau_R(p)$ for robot p and the travel time $\frac{dist(\sigma_p, h(I_{p,q}, p))}{v^p_{\max}}$ from the robot's start σ_p to the halt $h(I_{p,q}, p)$ of the intersection $I_{p,q}$ being currently visited. Similarly, we also require the time (cf. Line 16)

$$t_q = \tau_R(q) + \frac{dist(\sigma_q, r(I_{p,q}, q))}{v^q_{\max}} \tag{4.71}$$

which q requires to reach its release $r(I_{p,q}, q)$. The individual waiting time $\tau_I(I_{p,q})$ (for p) at $I_{p,q}$ (cf. Line 17) is then given by:

$$\tau_I(I_{p,q}) = \max(t_q - t_p, 0) \tag{4.72}$$

Because there may actually be no waiting required, which would yield a negative time, we ensure to have a lower bound of 0. The waiting time for q is zero because it gets RoW, thus, $\tau_R(q)$ does not need to be changed. Omitting the special case handling in Lines 19-21 for now, $\tau_I(I_{p,q})$ is simply added to the robot's total waiting time $\tau_R(p)$, not getting RoW here (p corresponds to u in Line 23).

The special case applies if p does not get RoW at more than one intersection whose halts w. r. t. p are all equal to each other. More formally, if there is a set

$$\begin{aligned}\mathcal{S}(\boldsymbol{\Delta}, p, I_{p,q}) \coloneqq \{\, I_{u,v} \mid I_{u,v} \in \mathcal{I} \wedge \delta_{\boldsymbol{\Delta}}(I_{u,v}) \neq p \wedge (u = p \vee v = p) \wedge \\ h(I_{u,v}, p) = h(I_{p,q}, p) \,\}\end{aligned} \tag{4.73}$$

with $|\mathcal{S}(\boldsymbol{\Delta}, p, I_{p,q})| > 1$ and all intersections $I \in \mathcal{S}(\boldsymbol{\Delta}, p, I_{p,q})$ have already been processed (see Line 19, that is, a value for $\tau_I(I)$ has been stored in Line 17), the additional waiting time for p needs to be computed by

$$\max\{\, \tau_I(I) \mid \forall I \in \mathcal{S}(\boldsymbol{\Delta}, p, I_{p,q}) \,\}. \tag{4.74}$$

This special case is justified as follows. Recall that p has to wait for all $I \in \mathcal{S}(\boldsymbol{\Delta}, p, I_{p,q})$. Because all halts are equal, p actually waits *simultaneously* for the release of *all* such intersections. Thus, the total waiting time for this set is given by the longest waiting time for all $I \in \mathcal{S}(\boldsymbol{\Delta}, p, I_{p,q})$. Notably, there are *no* dependencies among them, i. e., the individual waiting times for all $I \in \mathcal{S}(\boldsymbol{\Delta}, p, I_{p,q})$ can be computed independently. This is correctly reflected by the partial order returned by the topological sorting.

Once the total waiting times for all robots have been computed, the TTT (Equation (4.65)) and CPT (Equation (4.66)) are finally computed by summing up the fixed travel time from start to goal and the total waiting time (see Lines 31-28). They are then returned in Line 33 which completes the explanation of ComputeTimings() from Algorithm 4.6.

Finalizing the explanation of Algorithm 4.5, among all valid solutions $\boldsymbol{\Delta}_i$ passing the test in Line 23, the optimal solution $\boldsymbol{\Delta}_{\text{opt}}$ is given by the decision vector minimizing either the TTT or CPT v_i, stored as c_{opt}. If at least one solution was found, it is the optimum and returned in Line 32. An entry $\delta_k = 1$ in $\boldsymbol{\Delta}_{\text{opt}}$ means that the lexicographically smaller robot gets RoW at $I^k_{p,q}$ and similarly, if $\delta_k = 0$, the lexicographically larger robot gets RoW (robot names are assumed to be unique).

4.5.2 Correctness

We will now argue why the FDC (Equation (4.68)) and BDC (Equation (4.69)) are sufficient for representing dependencies and justify briefly why Algorithm 4.5 computes the correct optimal schedule. For simplicity, we assume that the total time required by a robot to reach its goal is composed of their *constant travel time* from start to goal and possibly *waiting time* $\tau_I(I)$ at intersections I whereby the latter solely depends on the RoW assignment.

By enumerating all possible decisions for the pairwise intersections (by exhaustively testing all permutations of the decision vector $\boldsymbol{\Delta}_i$), the algorithm is both *complete* and *optimal* if the methodology for the validity check and the scoring of solutions is correct.

The core of Algorithm 4.5 is the minimization of waiting times. Dependencies between them influence the order how they have to be computed, modeled with the two edge types in a dependency graph. We therefore have to justify the following:

Proposition 1. *All dependencies of the computation of waiting times are completely represented by forward (see Equation (4.68)) and backward edges (see Equation (4.69)).*

Proof. Generally, intersections are geometrically influenced by the size (radii) of the involved robots, the angle both paths are intersecting each other, the location of goals and starting points or, more generally, the location of the support points of both paths. The direction of the involved paths also plays an important role because it affects the order of a halt and its corresponding release point. A closer look reveals that all

influencing factors are already properly modeled with halt and release points, that is, such event points already encode how two paths intersect each other, obviating the explicit consideration of the influencing factors mentioned before. Nonetheless, the most important influencing factor are the RoW assignments themselves which directly affect the dependencies between intersections.

Let robots u, v, w and two arbitrary associated intersections $I^k_{u,v}$, $I^l_{u,w}$ be given. We are going to analyze all possible dependencies between $I^k_{u,v}$ and $I^l_{u,w}$ that could theoretically occur w. r. t. the common robot u (w. l. o. g.). If there is no common robot between $I^k_{u,v}$ and $I^l_{u,w}$, there are also no direct dependencies to be modeled. Indirect dependencies are modeled by a series of direct dependencies (path in the dependency graph), connecting the two intersections. Although the directions of v and w have an impact on the order of halt and releases along v and w respectively, they do not affect the events on u. The direction of u's path does not affect the location of the event but has an impact on their order. If we therefore analyze all possible arrangements of halts and releases on u's path, we implicitly handle the path's direction as well. Theoretically, (non-) existence of halts and releases accounts for 2^4 possibilities. However, since we can consider this to make the underlying intersections *predetermined*, it is already modeled with the RoW assignment.

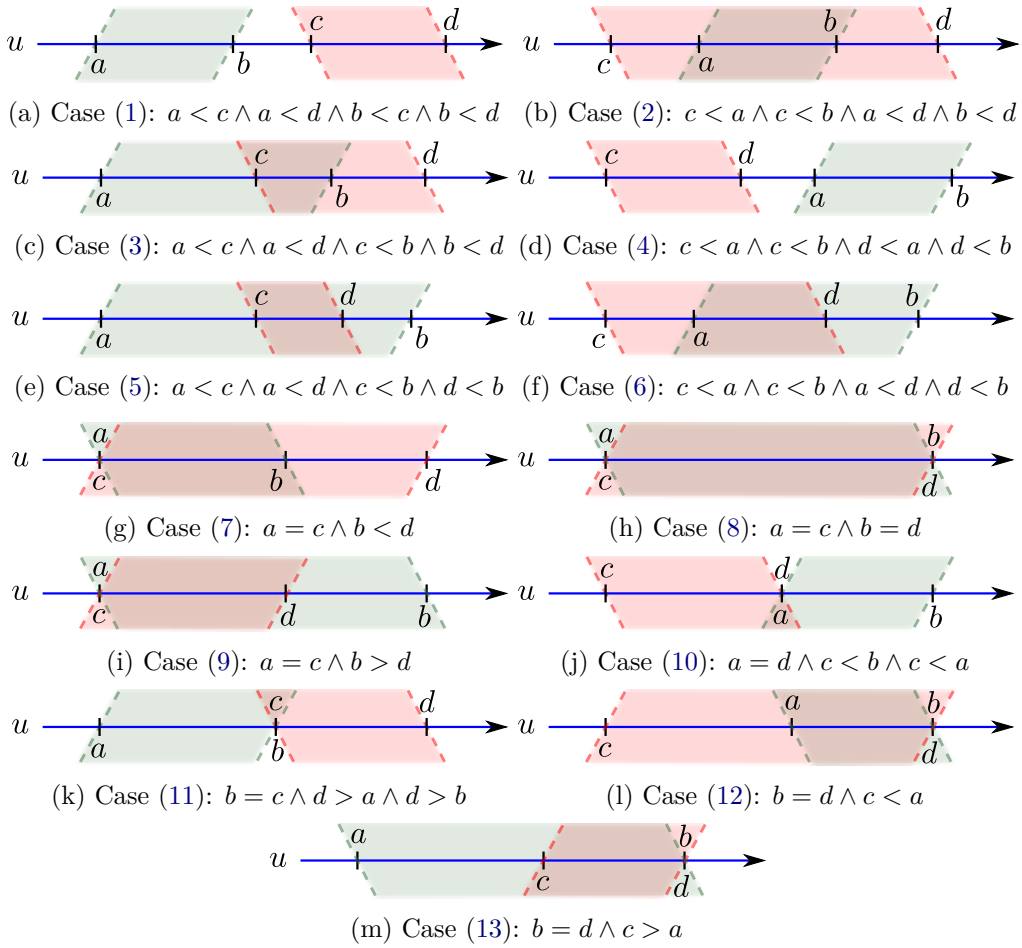
Define $a \coloneqq h(I^k_{u,v}, u), b \coloneqq r(I^k_{u,v}, u), c \coloneqq h(I^l_{u,w}, u)$ and $d \coloneqq r(I^l_{u,w}, u)$ to be the event locations of the two intersections on u's path with the two other robots v and w ($u = w$ is possible, $k \neq l$ counted along u's path).[8] It always holds that $a < b$ and $c < d$ because halts (a, c) are always located before their releases (b, d). Assuming $a, b, c, d \in \mathbb{R}$ exist, there are 13 possible arrangements in total with the previous constraints.[9] For every arrangement, there are 2^2 possible RoW assignments. The following is a complete analysis of all these cases w. r. t. occurring dependencies between $I^k_{u,v}$ and $I^l_{u,w}$, see Figure 4.27. For brevity, we will write $\delta(ab) = u$ to indicate that RoW at "intersection ab" (with halt a and release b) is given to robot u.

1. $a < c \wedge a < d \wedge b < c \wedge b < d$ ("abcd"): That is, both a and b are smaller than c and d which corresponds to the schematic visualization of the FDC in Figure 4.26(a) if we assume $\delta(ab) = v$. On the other hand, if $\delta(ab) = u$, no waiting is imposed at all. Similarly, because both intersection are sufficiently far apart from each other, the decision at cd does not affect the waiting at ab for u. This case is therefore correctly handled by Equation (4.68).

2. $c < a \wedge c < b \wedge a < d \wedge b < d$ ("cabd"): That is, ab is completely located in between cd, see Figure 4.27(b). For $\delta(ab) = \delta(cd) = u$, no waiting is required. This applies to all subsequent cases and is therefore omitted subsequently. In cases

[8] Note that $u = v = w$ is not possible because intersections are considered between two different robots. Self-intersections can be ignored.

[9] There are $4! = 24$ possible permutations of $abcd$ (to be read as $a < b < c < d$) but only 6 of these cases adhere to the constraints $a < b \wedge c < d$. Due to missing equality (e. g., $a = d \wedge c < b \wedge c < a$), there are 7 additional cases, thus, 13 cases in total.



(a) Case (1): $a < c \wedge a < d \wedge b < c \wedge b < d$ (b) Case (2): $c < a \wedge c < b \wedge a < d \wedge b < d$

(c) Case (3): $a < c \wedge a < d \wedge c < b \wedge b < d$ (d) Case (4): $c < a \wedge c < b \wedge d < a \wedge d < b$

(e) Case (5): $a < c \wedge a < d \wedge c < b \wedge d < b$ (f) Case (6): $c < a \wedge c < b \wedge a < d \wedge d < b$

(g) Case (7): $a = c \wedge b < d$ (h) Case (8): $a = c \wedge b = d$

(i) Case (9): $a = c \wedge b > d$ (j) Case (10): $a = d \wedge c < b \wedge c < a$

(k) Case (11): $b = c \wedge d > a \wedge d > b$ (l) Case (12): $b = d \wedge c < a$

(m) Case (13): $b = d \wedge c > a$

Figure 4.27: Schematic visualization of possible cases for dependencies between two intersections ab, cd along a common robot's path u (blue), identified by halts a, c and releases b, d. The subfigures illustrate different arrangements of halts and releases. Exemplary intersection areas are shaded in green w. r. t. ab (for u intersecting with v) and red w. r. t. cd (for u intersecting with w) for reference only. Also refer to Figure 4.26 for the cases (1), (3), (4), (5) and (6).

$\delta(ab) = u \wedge \delta(cd) \neq u$ and $\delta(ab) \neq u \wedge \delta(cd) \neq u$, there is a *forward* dependency from cd to ab ($l < k$) because the waiting time at ab depends on the waiting time at cd (FDC). However, if $\delta(ab) \neq u \wedge \delta(cd) = u$ holds, a *backwards* pointing dependency $ab \to cd$ (along u's path) is present, handled by the BDC.

3. $a < c \wedge a < d \wedge c < b \wedge b < d$ ("acbd"): That is, intersection ab comes first but is interleaved with cd (c in between ab). This is visualized in Figure 4.26(b) (although d is not present there because in Figure 4.26(b), cd is predetermined). The case $\delta(ab) = u \wedge \delta(cd) \neq u$ is covered by the BDC while $\delta(ab) \neq u \wedge \delta(cd) = u$ and $\delta(ab) \neq u \wedge \delta(cd) \neq u$ are covered by the FDC.

4. $c < a \wedge c < b \wedge d < a \wedge d < b$ ("cdab"): That is, both c and d are smaller than ab, This is already discussed with case (1) with roles of ab and cd swapped.

5. $a < c \wedge a < d \wedge c < b \wedge d < b$ ("acdb"): That is, cd is completely located in between ab, see case (2) (roles of ab and cd swapped).

6. $c < a \wedge c < b \wedge a < d \wedge d < b$ ("cadb"): That is, intersection cd comes first but is interleaved with ab (a in between cd), see case (3) (roles of ab and cd swapped).

7. $a = c \wedge b < d$: That is, both intersections start with the same halt on u's path and ab ends before cd, see Figure 4.27(g). In case of $\delta(ab) = u \wedge \delta(cd) \neq u$, ab is potentially blocked by u while waiting for the release of cd. A dependency $cd \rightarrow ab$ is therefore given, covered by the BDC ($c \geq a \wedge b > c$). The same applies for $\delta(ab) \neq u \wedge \delta(cd) = u$ with intersection vertices swapped ($ab \rightarrow cd$, BDC: $a \geq c \wedge d > a$). If u neither gets RoW at ab nor at cd, there are no dependencies to be considered between ab and cd because u waits *simultaneously* for the release of both.

   Generally, the additional amount of waiting time for u in case of $a = c$ (equal halts) is given by the maximum of the individual waiting times (computed independently) among all intersections with equal halts for u and where u does not get RoW. This is reflected by the special case in Line 18 of Algorithm 4.6.

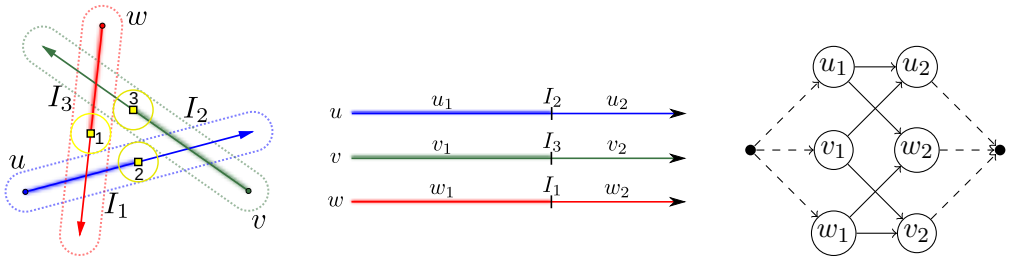
8. $a = c \wedge b = d$: That is, both halts and releases are equal, see Figure 4.27(h). Similarly to the previous case (7), all dependencies of this case are handled by the BDC because for $\delta(ab) = u \wedge \delta(cd) \neq u$, $c \geq a \wedge b > c$ and for $\delta(ab) \neq u \wedge \delta(cd) = u$ (roles swapped), $a \geq c \wedge d > a$ holds.

9. $a = c \wedge b > d$: That is, both intersections start with the same halt on u's path and ab ends before cd, see Figure 4.27(i). This case is equivalent to case (7) and (8) because they are independent of the relation between b and d.

10. $a = d \wedge c < b \wedge c < a$: That is, intersection ab immediately follows after cd without any free space in between ($a = d$), see Figure 4.27(j). For the cases $\delta(ab) = u \wedge \delta(cd) \neq u$ and $\delta(ab) \neq u \wedge \delta(cd) \neq u$, there is a forward dependency $cd \rightarrow ab$, covered by the FDC. No dependencies are present at all if u does not get RoW at its second intersection ab, i. e., $\delta(ab) \neq u \wedge \delta(cd) = u$.

11. $b = c \wedge d > a \wedge d > b$: That is, intersection cd immediately follows after ab, see Figure 4.27(k). This is already discussed in the previous case (10) with roles of ab and cd swapped.

12. $b = d \wedge c < a$: That is, intersection cd starts first and both end in the same release, see Figure 4.27(l). For cases $\delta(ab) = u \wedge \delta(cd) \neq u$ and $\delta(ab) \neq u \wedge \delta(cd) \neq u$, a dependency from cd to ab is given, covered by the FDC. However, for $\delta(ab) \neq u \wedge \delta(cd) = u$, the release of cd (RoW for u) depends on waiting at ab, handled by the BDC ($ab \rightarrow cd$).



(a) Exemplary input scenario (b) Partition of subpaths (c) Dependencies of subpaths

Figure 4.28: Example for (a) an input scenario with a given solution $\mathbf{\Delta} = (1, 0, 0)$, (b) the $\mathbf{\Delta}$-induced partition of the paths of u (blue), v (green) and w (red) into subpaths (aka sections u_i, v_i, and w_i respectively), and (c) the dependencies between the subpaths, directly yielding an execution order. $\mathbf{\Delta} = (1, 0, 0)$ means that u gets RoW at I_1, v at I_2 and w at I_3. Note that the subpaths u_1, v_1 and w_1 can be traveled in parallel because they are on the same layer. The same applies for u_2, v_2 and w_2 if u_1, v_1 and w_1 have been passed.

13. $b = d \wedge c > a$: That is, intersection ab starts first and both end in the same release, see Figure 4.27(m). This is already discussed in the previous case (12) with roles of ab and cd swapped. Note that full equality is already handled in case (8).

There is no need to consider other combinations of u, v and w because this would again be an isomorphism on the robot names, i. e., a swap of roles (names) only. Because these are all possible cases that affect the waiting time of a robot, the possible dependencies are completely reflected by the two proposed edges types, namely forward (FDC) and backward edges (BDC). □

In order to justify the correctness of Algorithm 4.5 (cf. Line 23), we also have to verify the following:

Proposition 2. *For an enumerated candidate* $\mathbf{\Delta}$ *from Line 21 of Algorithm 4.5, the dependency graph* $G_{\mathbf{\Delta}}$ *is acyclic if and only if* $\mathbf{\Delta}$ *is valid.*

First, a precise definition of "valid" is required. A candidate $\mathbf{\Delta}$ for a given input is *valid* if there exists an algorithm A which commands all robots of the input from their starting points to their goals while respecting the computed schedule $\mathbf{\Delta}$. In other words, the entries of $\mathbf{\Delta}$ imply a partial order of the pairwise intersections in the input[10] and A ensures that intersections are passed through according to that order which finally allows all robots to reach their goals without any collisions.

Proof. We start with "⇒", that is, given an acyclic dependency graph $G_{\mathbf{\Delta}}$ for a candidate $\mathbf{\Delta}$, we have to provide an algorithm that guides all robots to their goals which would imply

[10] If the input was partially solved, only the partial input and its solution is used here.

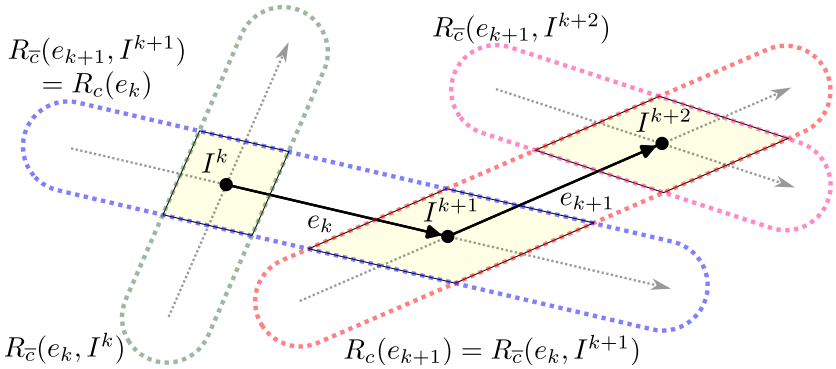


Figure 4.29: Illustration of the common R_c and non-common robot $R_{\overline{c}}$ along a path with edges e_k, e_{k+1} (black arrows) in a dependency graph $G_{\mathbf{\Delta}}$ being part of a cycle. The underlying robot paths (dashed gray arrows), areas (dashed red, green, blue and pink) and intersections (yellow) are exemplary shown in the background (semi-transparent) for reference. The main insights visualized by this figure are: (1) the common robot $R_c(e_{k+1})$ is equal to the non-common robot $R_{\overline{c}}(e_k, I^{k+1})$, and (2) the non-common robot $R_{\overline{c}}(e_k, I^{k+1})$ has to wait at I^{k+1} for release by the common robot $R_c(e_k)$ which itself is waiting at I^k (shown here if assuming forward edges, likewise for backward edges).

a valid $\mathbf{\Delta}$. Fortunately, this algorithm has already been presented: the topological sorting of the dependency graph (a directed acyclic graph (DAG)) as employed in Algorithm 4.6 reveals the partial order of the pairwise intersections and even allows us to compute an animation (assuming constant instant velocity) of processing the schedule using the travel and waiting times. An example has already been given in Figure 4.16. In other words, we can partition a robot's path into subpaths "from the start to the first relevant halt", "from the previous relevant halt to the next relevant halt", and "from the last relevant halt to the goal", see Section 4.2.2. The algorithm therefore allows every robot to move to the halt of the first intersection where it does not get RoW. Because we know there is a partial order of the intersections, there must always be a robot that releases another intersection by moving along its first subpath (because it got RoW). By releasing a robot's next intersection (if any), that robot is allowed to travel along its subsequent subpath, effectively releasing that intersection for the other involved robot. Note that there can be multiple robots being allowed to move in parallel, reflected by multiple pairwise intersections in $G_{\mathbf{\Delta}}$ being in the same "layer" (w. r. t. the topological sorting). However, this is not an issue because it even releases multiple intersections concurrently, thus, giving all associated robots clearance for traveling along their subsequent subpaths. The order in which subsequent subpaths are being selected is therefore implied by the topological sorting of the intersections. This continues until all subpaths of all involved robots have been traveled. Because the subpaths are a partition of the entire path, we can conclude that all robots have reached their goals, requiring $\mathbf{\Delta}$ to be valid. An example demonstrating how a valid solution is transformed into a set of moving robots is shown in Figure 4.28.

It remains to show "⇐", i. e., assume we have a valid candidate $\mathbf{\Delta}$, we have to show that its dependency graph $G_{\mathbf{\Delta}}$ is acyclic. Let us for now assume that $G_{\mathbf{\Delta}}$ has a cycle. In such a case, there must be a set $\mathcal{C}$ of vertices $I^1, \ldots, I^n \in G_{\mathbf{\Delta}}$ forming a cycle. When an algorithm tries to execute the schedule $\mathbf{\Delta}$, it will eventually encounter the pairwise intersections associated with $I^1, \ldots, I^n$, after executing all subpaths and passing through intersections located *before* the cycle. An edge of the cycle can either be a forward or a backward edge according to the Equations (4.68) and (4.69). Let $e_k = (I^k, I^{k+1}) \in G_{\mathbf{\Delta}}$, $I^k, I^{k+1} \in \mathcal{C}$ be two adjacent intersections of the cycle and let $R(I^k)$ denote the set of robots involved in I^k, with $|R(I)| = 2$ for all $I \in \mathcal{I}$. We define $R_c(e_k)$ to be the common robot of intersections I^k, I^{k+1} as follows, see Figure 4.29: If $|R(I^k) \cap R(I^{k+1})| = 1$, $R_c(e_k)$ maps to that single common robot $u \in R(I^k) \cap R(I^{k+1})$. However, if $|R(I^k) \cap R(I^{k+1})| = 2$ (because two robots have multiple intersections), $R_c(e_k)$ returns the robot whose intersection comes first according to the edge direction and halts. Note that $R(I^k) \cap R(I^{k+1}) = \emptyset$ is impossible because I^k and I^{k+1} are connected with an edge in $\mathcal{C}$ and must therefore have a common robot. Additionally, let $R_{\overline{c}}(e_k, I^{k+1}) \coloneqq R(I^{k+1}) \setminus \{R_c(e_k)\}$ be the non-common robot[11] at I^{k+1}, again see Figure 4.29. Starting with an arbitrary intersection $I^k \in \mathcal{C}$, we know that there must be an outgoing edge e_k connecting it to some other intersection $I^{k+1} \in \mathcal{C}$. Regardless of whether e_k is a forward or backward edge (cf. Equations (4.68) and (4.69)), we can conclude that the non-common robot $R_{\overline{c}}(e_k, I^{k+1})$ has to wait at I^{k+1} for release by the common robot $R_c(e_k)$ which itself is waiting at I^k. This directly follows by the RoW requirements in the definition of the Forward and Backward Dependency Conditions. As it turns out, this chains up until reaching I^k again, closing the series of waiting dependencies and making it a cycle.

In other words, if there is an edge $e_k = (I^k, I^{k+1})$, it can be read as: "the intersection I^k imposes a (waiting) dependency on intersection I^{k+1}", or just: I^k must precede I^{k+1}. Thus, if we replace e_k by the $<$-relation denoting that the common robot $R_c(e_k)$ is not getting RoW at I^k s. t. the non-common robot $R_{\overline{c}}(e_k, I^{k+1})$ at I^{k+1} has to wait for release, we could also write $R_c(e_k) < R_{\overline{c}}(e_k, I^{k+1})$. This is because "smaller robots" (w. r. t. to the $<$-relation) are allowed to move first. By substituting the entire series of vertices $I^1, ..., I^n \in \mathcal{C}$, we obtain the following inequalities:

$$R_c(e_1) < R_{\overline{c}}(e_1, I^2) \tag{4.75}$$
$$R_c(e_2) < R_{\overline{c}}(e_2, I^3) \tag{4.76}$$
$$\vdots$$
$$R_c(e_k) < R_{\overline{c}}(e_k, I^{k+1}) \tag{4.77}$$
$$\vdots$$
$$R_c(e_{n-1}) < R_{\overline{c}}(e_{n-1}, I^n) \tag{4.78}$$
$$R_c(e_n) < R_{\overline{c}}(e_n, I^1) \tag{4.79}$$

[11] For simplicity, we write $R_{\overline{c}}(e_k, I^{k+1})$ to refer to the single element in the set.

Because generally, the non-common robot $R_{\overline{c}}(e_k, I^{k+1})$ is equal to the common robot $R_c(e_{k+1})$ at the subsequent edge e_{k+1} for all $k \in \{1, \ldots, n\}$ in the cycle $\mathcal{C}$, whereby the successor of edge e_n is e_1 and the successor of intersection I_n is I_1, we obtain

$$R_c(e_1) < R_c(e_2) < \cdots < R_c(e_k) < \cdots < R_c(e_{n-1}) < R_c(e_n) \overset{!}{<} R_c(e_1) \tag{4.80}$$

which is a contradiction. The last inequality is justified by the common robot $R_c(e_1)$ of the first[12] edge e_1 being equal to the non-common robot $R_{\overline{c}}(e_n, I^1)$ of the last edge e_n.

We can conclude that there is no valid order (serialization) among the robots being part of the cycle. This means that an algorithm is unable to select one of the involved robots $\bigcup_{I \in \mathcal{C}} R(I)$ in order to allow it passing through its next intersection while adhering to $\boldsymbol{\Delta}$. This is a contradiction to the assumption that $\boldsymbol{\Delta}$ is valid and, thus, $G_{\boldsymbol{\Delta}}$ must have been acyclic. □

4.5.3 Parallelization

Given a decision vector $\boldsymbol{\Delta} = (\delta_1, \ldots, \delta_K)$ with entries $\delta_k \in \{0_{\text{pd}}, 1_{\text{pd}}, \emptyset\}$, all 2^{K-L} assignments of $\boldsymbol{\Delta}$ must be enumerated while retaining entries $\delta_k = 0_{\text{pd}}$ as 0, $\delta_k = 1_{\text{pd}}$ as 1 (constraints) and trying all combinations of $\{0, 1\}$ for entries $\delta_k = \emptyset$. L is the number of predetermined intersections in the input, effectively reducing the number of enumerations to be computed. Recall that for all enumerated decision vectors, the dependency graph construction, a validity test and, if considered valid, the computation of the timings need to be executed (processing steps), see Lines 21-30 in Algorithm 4.5. Due to the exponential number of assignments, this requires a lot of time. Parallelization is therefore desirable in order to exploit multi-core capabilities of modern computing hardware.

One way of doing this would be to parallelize the generation of assignments itself. However, because this is naturally done recursively, one would have to either parallelize the recursive implementation or transform it into an iterative algorithm to apply parallelization afterwards. Both are not trivial tasks which is why a different approach was chosen:

1. The generation of assignments is done recursively and sequentially. It generates a subset of assignments of the decision vector with a fixed chunk size $s \in \mathbb{N}$. The chunk size depends on the available memory but ideally, it should be as large as possible. This way, the set of all assignments of $\boldsymbol{\Delta}$ are partitioned into $\left\lceil \frac{2^{K-L}}{s} \right\rceil$ subsets (chunks).

2. For a computed chunk and with the maximum concurrency possible, threads process all pre-enumerated decision vectors in the given chunk in parallel. This just

[12] There is no unique "first" or "last" edge in a cycle but it is sufficient here to simply number them arbitrarily.

requires to synchronize the minimization of the timings (race condition) among the threads.

3. The previous two steps are repeated until all chunks have been computed (1.) and evaluated (2.), or a termination criterion becomes true.

The proposed concept allows for a memory limited parallel evaluation which is the computationally intensive part. Note that the depth of recursion is K, thus, not a limiting factor for practically sized instances.

4.5.4 Evaluation

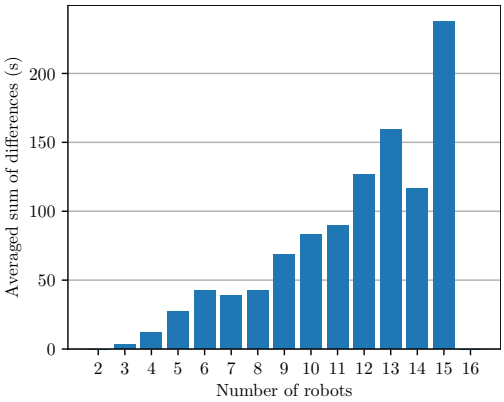
In this section, we are going to experimentally analyze OMRPS based on dataset 2 (see Figure 4.20) and compare it against ICSPS. All experiments have been executed on an Intel Xeon Gold 4246R with 16 physical and 32 logical cores if not stated otherwise. Two very complex out of 118 683 total scenarios have been excluded from the following results because their calculations ran for multiple months without completion due to the exponential run-time complexity.[13] ICSPS was always used with A*, the Maximum metric and a permutation limit of 500.

Figure 4.30 compares the performance of ICSPS against the optimal solver w. r. t. CPT and solvability. The averaged sum of differences of the CPT between OMRPS and ICSPS for all input scenarios is shown in Figure 4.30(a). More specifically, the CPT has been accumulated for all scenarios with a specific robot count N both for OMRPS and ICSPS, denoted as $S^{(N)}_{\text{OMRPS}}$ and $S^{(N)}_{\text{ICSPS}}$ respectively. The value of each bar in the plot is then given by

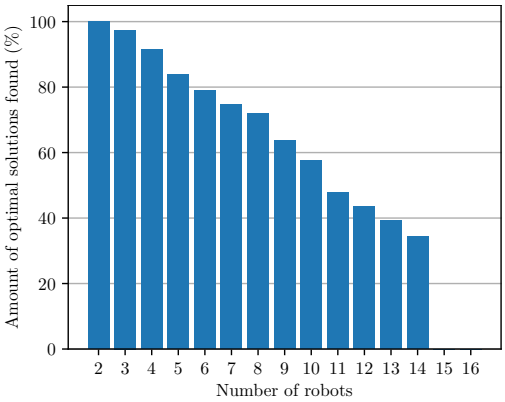
$$\frac{S^{(N)}_{\text{OMRPS}} - S^{(N)}_{\text{ICSPS}}}{M} \cdot 100 \tag{4.81}$$

whereby M is the number of successfully and optimally solved scenarios. The results are grouped by the number of robots within a scenario which serves as an indicator for the complexity because the more robots the more likely are intersections. In fact, a very similar figure is retrieved when grouping by intersection count (not shown here to avoid redundancy). For a robot count of two, ICSPS yields the same (optimal) results as OMRPS, indicated by the zero value. When the number of robots increases, the larger becomes the difference between the optimal result and ICSPS. Taking a closer look at Figure 4.30(b) reveals that ICSPS is able to solve up to 80 % of all inputs with a robot count up to five optimally. The figure shows the amount of scenarios which have been solved exactly like OMRPS, i. e., optimal. Similarly to (a), when the complexity increases, the amount of optimal solutions decreases. Figure 4.30(c) shows the amount of inputs that have been solved at all, i. e., where ICSPS was able to find a solution if one existed (solvability). Even for inputs with up to 10 robots, ICSPS was able to find

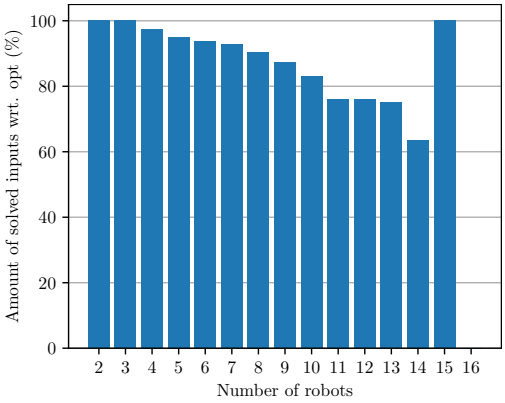
[13] For simplicity, however, we still refer to “all scenarios in the dataset”.



(a) Comparison of the optimal solver with ICSPS based on the averaged differences of the critical path time (in seconds, ordinate) for all scenarios grouped by the number of robots (abscissa). Zero means optimal.

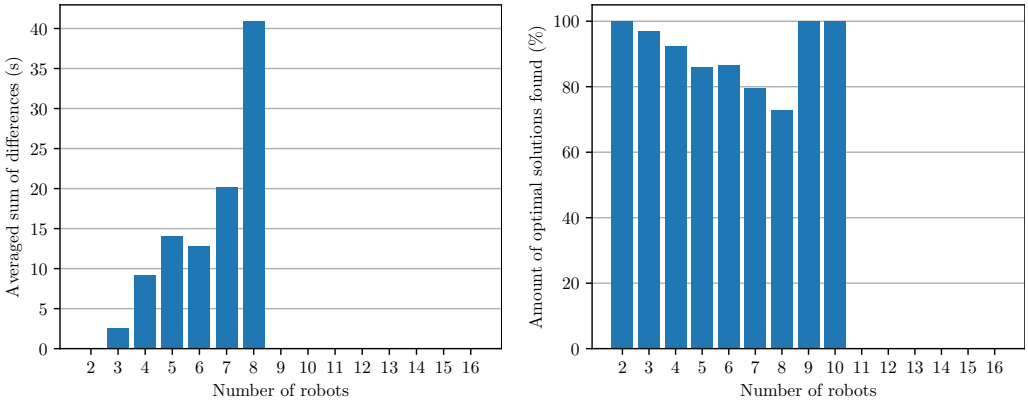


(b) Amount of optimal solutions computed by ICSPS for all scenarios grouped by the number of robots; solutions in this category must have a critical path time exactly equal to the one computed by the optimal solver.



(c) Amount of successfully solved scenarios by ICSPS grouped by the number of robots; solutions in this category must have a critical path time greater or equal to the one computed by the optimal solver.

Figure 4.30: Comparison of the optimal solver with ICSPS using A* and the Maximum metric based on dataset 2 (see Figure 4.20). Statistical significance decreases with an increasing number of robots in the input scenarios, e. g., only 151 solvable scenarios for a robot count of 14. The results for 14 and 15 robots are therefore considered as outliers.



(a) Comparison of the optimal solver with ICSPS based on the averaged differences of the critical path time (in seconds, ordinate) for all trivially solvable scenarios grouped by the number of robots (abscissa). Zero means optimal.

(b) Amount of optimal solutions computed by ICSPS for all trivially solvable scenarios grouped by the number of robots; solutions in this category must have a critical path time exactly equal to the one computed by the optimal solver.

Figure 4.31: Comparison of the optimal solver with ICSPS using A* and the Maximum metric based on dataset 2 (see Figure 4.20) for trivially solvable scenarios only. Statistical significance decreases with an increasing number of robots in the input scenarios, e. g., only 97 solvable scenarios for a robot count of six. The results for nine and ten robots are therefore considered as outliers. Note that ICSPS was able to solve *all* solvable inputs (similar to Figure 4.30(c) but for all trivially solvable inputs only, not shown here).

a solution in more than 80 % of the inputs. In contrast to Figure 4.30(b) (addressing optimality only), (c) indicates that ICSPS is able to provide high solvability even for quite complex scenarios (> 10 robots). However, it should be noted that the number of available (and solvable) scenarios drops for a larger number of robots, e. g. only 151 solvable inputs for a robot count of 14. Results for robot count 14 and 15 are therefore considered to be outliers.

We continue with the analysis and discussion of Figure 4.31 which compares OMRPS against ICSPS on trivially solvable scenarios of dataset 2 only (dataset 1 exhibits similar results). A scenario is defined to be *trivially solvable* if there are no predetermined intersections at all, thus allowing for a trivial serialization (as an obvious possible solution). Apart from solving all scenarios with two robots optimally (see Figure 4.31(a)), the differences to the optimal results in terms of CPT increase with the number of robots. As visualized in Figure 4.31(b), ICSPS is able to compute more than 80 % of the optimal solutions for up to six robots. Further experiments also revealed that ICSPS was able to solve *all* solvable inputs (not shown here). However, note that there are much fewer trivially solvable scenarios because many have at least one predetermined intersection.

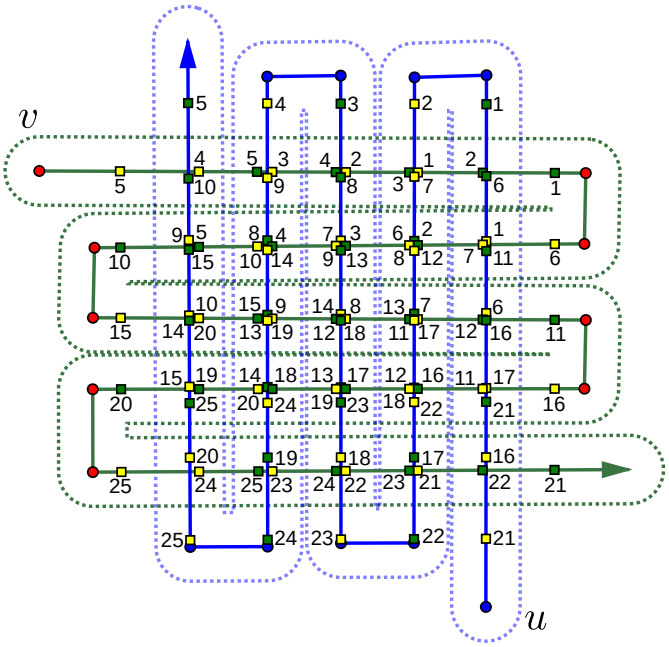


Figure 4.32: Example for a complex input between two robots u (blue) and v (green) with 25 intersections creating 33 554 432 possible solutions $\mathbf{\Delta}_i$ whereby only 252 are valid (acyclic). Note that the number of pairwise intersections is 25 in this example, regardless of whether SGS or MGS is used.

As already noted, this decreases statistical significance for an increasing number of robots.

As an example for the improvements made due to parallelization (see Section 4.5.3), for the input in Figure 4.32 with 25 pairwise intersections (33 554 432 possible solutions to be enumerated), the single-threaded version of Algorithm 4.5 required 11.4 min for solving. In contrast, the multi-threaded version just required 56.7 s for the same input with 24 threads on 12 physical and 24 logical cores (AMD Ryzen 3900X) which is more than 12x faster. Notably, ICSPS (Algorithm 4.4) returned a solution very close to the optimum for this input when minimizing the CPT (868.54 s $<$ 869.82 s) while being clearly non-optimal for the TTT (1 669.68 s $<$ 1 702.12 s). The run time for this example was only 0.54 ms. An important observation is that the run time complexity of OMRPS mainly depends on the number of pairwise intersections (due to enumerating assignments of the RoW) while ICSPS mainly depends on the number of robots (due to testing permutations of robots).

Finally, Figure 4.33 shows the run time (ordinate) of OMRPS for the scenarios (abscissa) of dataset 2. The color encodes the number of intersections within a given scenario. For scenarios with less than 10 intersections, the run time typically remains below 1 ms. Clearly, when the number of intersections increases, OMRPS exhibits its exponential run time complexity with up to months of calculation time (aborted after about 75 days).

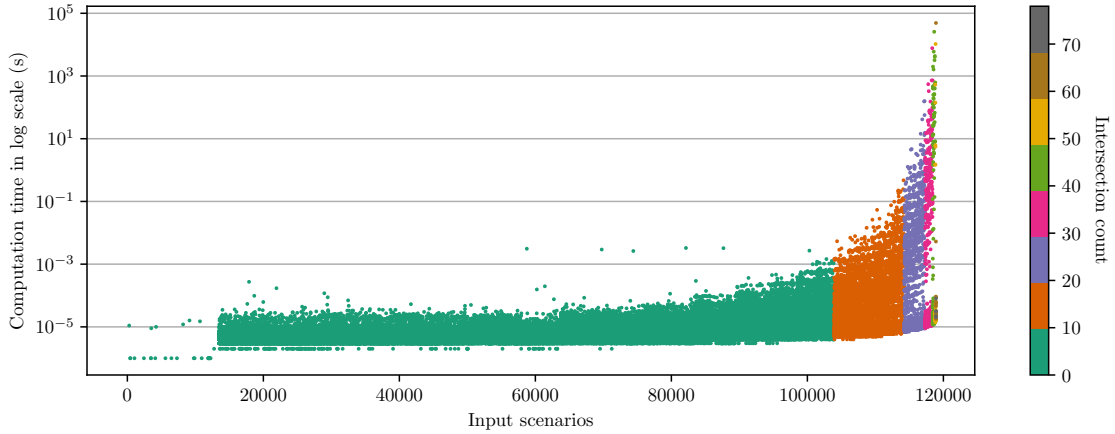


Figure 4.33: Run time of the optimal solver for the scenarios of dataset 2: scenarios are depicted on the abscissa while the computation times are shown on the ordinate in seconds in a logarithmic scale. Colors are used to encode the intersection count of a given scenario as visualized on the right. Solutions have been computed in a multithreaded way.

We can conclude that OMRPS is applicable for small inputs while for more complex inputs, ICSPS should be applied.

Chapter 5

Collaborative Collision Prevention

Path planning is an important concept in empowering robots to act efficiently and intelligently. It allows them to autonomously move themselves to a given goal while avoiding collisions with other objects in the environment. Within the framework described in Chapter 3, path planning along with localization are important ingredients to form the software stack of a mobile robot.

This chapter deals with a novel local path planning algorithm, termed the *Collaborative Local Planning Framework* (CLPF), that aims at preventing collisions by presciently sharing knowledge among all known robots in the system. For a given set of robots, CLPF defines the communication logic and state management for the involved robots to eventually execute one of the solvers presented in Chapter 4. An important distinction of the framework against the solvers from the previous chapter is its ability to handle *dynamic* inputs. A dynamic input refers to the situation where a subset of the robots may already be moving while others are standing still but requesting to move as well. In such a situation, CLPF provides the methodologies to integrate the yet non-moving robots into the set of moving robots whereby non-moving robots may request integration at any time. This happens frequently during operation. It may also already happen right at the beginning if goals are dispatched one after another with sufficient time in between.[1] In addition, CLPF's communication layer allows handling uncertainty in the underlying motion controller.

The framework not only avoids collisions but also completely prevents them provided the underlying assumptions are met. Thus, it offers higher safety guarantees compared to state-of-the-art local planning concepts, which are generally required in industrial contexts where (autonomous) long-time operation of mobile robots is striven. Addition-

[1] CLPF may also encounter static inputs e. g., at the very beginning if goals are dispatched in parallel.

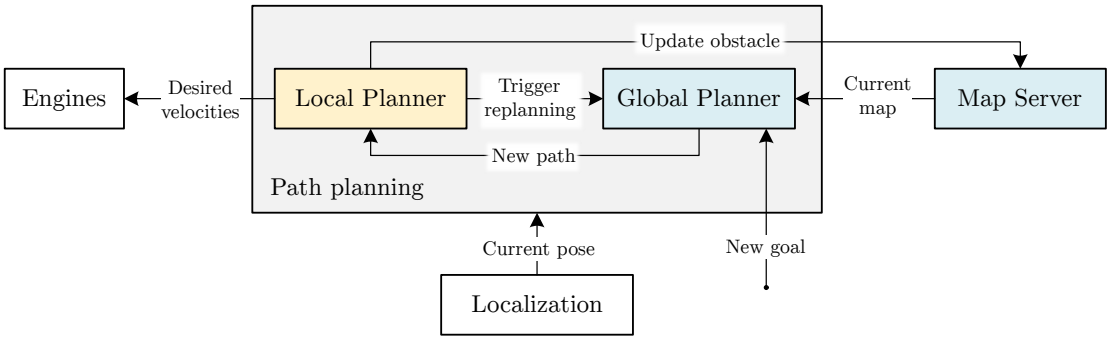


Figure 5.1: Software stack on a mobile robot platform for navigation; this chapter focuses on the local planner (yellow) and assumes that the other modules are mainly given. A new goal is assumed to be provided by some external entity.

ally, the suggested framework with its underlying algorithms allows to more precisely predict when a robot reaches its goal while still allowing planning in "free space" by a separate independent global planning algorithm (out of the scope of this thesis). This simplifies the transformation from centrally organized production lines to decentrally organized shopfloors.

The chapter is organized as follows. Section 5.1 gives a more detailed introduction and motivates the requirement for the proposed approach. Section 5.2 details the assumptions that need to be satisfied in order to ensure safety, as well as the limitations of the concept in general. After giving an overview of the proposed concepts in Section 5.3, the communication architecture between the robots is explained in Section 5.4, followed by the description of the state management in Section 5.5. Section 5.6 continues with the concept of intersection graphs being used to represent groups of robots being directly or indirectly in conflict with each other. How robots execute their negotiated motions is described in Section 5.7. Afterwards, Section 5.8 explains global planning and management of the environmental map including semi-static obstacles. Finally, an evaluation is presented in Section 5.9 based on an experimental analysis. Note that related work has already been reviewed in Chapter 2, especially in Section 2.3.4.

5.1 Introduction

In mobile robotics, *path planning* can be subdivided into local and global path planning [44]. *Global planning* aims at finding a path through a known (rather static) environment from the robot's current position to the desired goal given a map of (semi-) static obstacles provided by a *map server*, see Figure 5.1. The process of following the globally planned path by sending desired velocities to the engines is termed *local planning* and requires periodic updates of the robot's current pose by a *localization* algorithm. Divergences from the path are usually allowed to sidestep obstacles unknown to the global planner. Most local planners incorporate range sensor measurements from ultrasonic distance

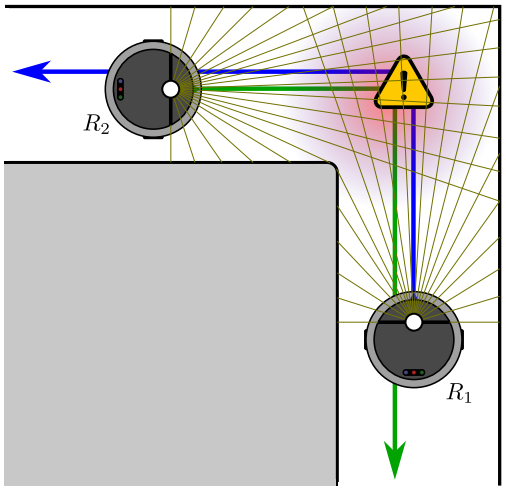


Figure 5.2: Exemplary situation where two robots R_1 (blue), R_2 (green) can collide at a corner because they can only suddenly see each other (indicated by the yellow laser scanner beams) once turning around that corner ("dangerous zone", red). Blocked areas are black and gray. The direction of movement is visualized by the arrows.

sensors or laser scanners in these velocity computations to sense and avoid nearby obstacles. Even though this works well in situations where obstacles can be detected early, cluttered environmental objects like walls can cause a missing line of sight from the robot to obstacles as exemplary depicted in Figure 5.2. If both robots are moving, the probability of a collision even increases because, given a constant velocity of v (in m/s) of both robots for simplicity and a remaining distance of d (in m), the reaction time (in s) until a collision occurs is given by $\frac{d}{2v}$. That is, the higher the velocities of the robots, the greater the probability of a collision if the paths overlap as shown in Figure 5.2. Even worse, smaller reaction times require faster obstacle processing algorithms. One possible solution would be the combination of behavioral approaches (similar to [53]) in narrow areas (e. g., at corners) and online motion planning in the remaining free space (like the Dynamic Window Approach (DWA) [22]). However, this requires the detection of such "dangerous zones" in a robust fashion which is a challenging issue.

The aforementioned considerations motivate the concept of the local planning algorithm (yellow in Figure 5.1) presented in this chapter. It provides safety w. r. t. to collisions between all robots that symmetrically execute the same algorithm given reasonable practical constraints, as detailed in Section 5.2. Several challenges have been addressed during the design and development. First, the underlying problem is highly parallel because robots communicate over network (parallelism of processes) that also causes non-determinism regarding the communication and message timings—and the way they are being processed. Notably, multiple executions of a simulation with the same parameters on a single machine already cause varying behavior and timings during the processing. Additionally, the algorithm itself needs to be executed in a multithreaded way to provide reactivity to other robots while still controlling the robot's engine simultaneously. Second,

the high dimensionality of the problem (time, goal poses, number and properties of robots, etc.) exhibits various corner cases that require additional attention. Third and finally, the algorithm's computation must be efficient and should restrict assumptions and limitations to retain practical applicability.

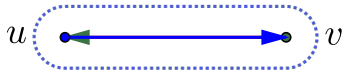
5.2 Assumptions, Limitations and Requirements

Within this section, the assumptions and limitations of the proposed concepts are presented which particularly relate to the situations and conditions it may be applied in. Additionally, requirements and design goals are presented and justified.

The software stack of a mobile robot is typically comprised of a localization algorithm, a local and global planning module and a motion controller that interacts with the robot's engines to control its speed and steering [51, 58]. All these components are somehow (tightly) coupled since one consumes and processes data of another. As an example, a local planner needs to know the current location of the robot (provided by the localization algorithm) to decide whether the current goal was reached. Likewise, the local planner typically takes the path from the global planner. For that reason, requirements of such components are presented in order to make the proposed algorithms applicable.

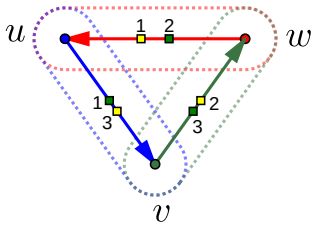
Within the entire scope of this chapter, global planning and localization are assumed to be given. The former provides a *non-self-intersecting path* to the algorithm described in this chapter that connects the robot's current pose with the goal pose through waypoints in free space with sufficient clearance. It incorporates limited knowledge of static known objects in the environment that are known a priori (e. g., walls). The global planner may not be extended to consider dynamic obstacles since they are changing too erratically. However, it is assumed that the underlying map of the environment can be extended by new (semi-) static objects which is then taken into account by the global planner. More formal details on the path itself are presented in Section 5.3. Thus, unknown dynamic obstacles (like humans) are not explicitly considered, neither by the global planner nor by the proposed algorithm. They must therefore be handled by analyzing distance data retrieved from sensors on the robot or excluded by design. If a robot detects an obstacle in its FoV, it would just stop and wait until the path is clear before continuing its previous motion *without* altering the initial path. This should be considered as a fail-safe mechanism, for instance, in case of localization inaccuracies, or unexpected obstacles. It may be relinquished if such problems cannot occur, e. g., due to a global (perfect) localization system or an encapsulated operation mode (in which dynamic obstacles cannot intrude).

In general, there are no restrictions on the *placing of the goals* in the environment or the timing when they are assigned to robots. If a goal is not reachable due to static (known) obstacles, it is assumed to be rejected by the global planner. However, the proposed local planning algorithm may also freely reject one or more goals if it considers a scenario

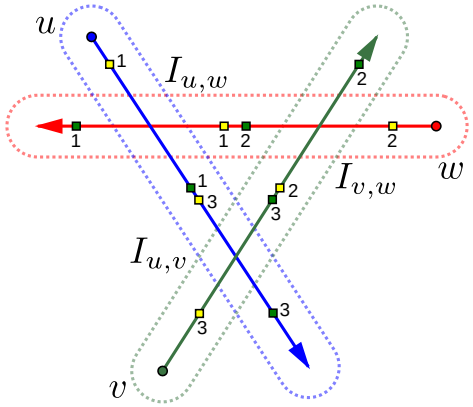


(a) Swap with full overlap (infeasible)

(b) Swap with partial overlap (dito)



(c) Cyclic swap (feasible, solution implicitly given by predetermined intersections)



(d) Feasible but bears the risk of a deadlock for $\delta(I_{u,v}) = u, \delta(I_{u,w}) = w, \delta(I_{v,w}) = v$. A possible solution (optimum) is given by $\delta(I_{u,v}) = v, \delta(I_{u,w}) = u, \delta(I_{v,w}) = w$ (no waiting required).

Figure 5.3: Examples of complex corner cases where (a) two robots u (blue), v (green) intend to swap their positions (considered infeasible without replanning), (b) is a similar scenario but with offset paths (infeasible as well), (c) three robots u (blue), v (green), w (red) want to swap their positions in a circular manner (feasible) and (d) bears the risk of a deadlock depending on the solution computed by a solver. Yellow and green rectangular markers along the paths indicate halt and release points (if any) respectively.

to be infeasible or goals unreachable. For instance, Figure 5.3 shows four rather complex scenarios that are briefly discussed in the following. In Figure 5.3(a), the two robots u (blue), v (green) want to swap their positions which is infeasible without replanning since one robot would block the path of the other. Note that this scenario already occurs if the goal areas just partially overlap with paths' of one another. The scenario visualized in Figure 5.3(b) is similar to the one in (a) and considered infeasible without replanning as well since both robots cannot bypass each other without blocking the other one's path. Figure 5.3(c) depicts a similar situation where three robots u (blue), v (green), w (red) want to swap their positions one after another in a cyclic manner. Assuming that there is sufficient space so that these robots can leave their starting area to make room for others (indicated by the existence of halts and releases), this scenario is feasible. Finally, Figure 5.3(d) shows an intersection scenario which is feasible although solving bears the risk of "creating" a deadlock if $\delta(I_{u,v}) = u, \delta(I_{u,w}) = w, \delta(I_{v,w}) = v$ (see Section 4.2.2), that is, if u gets RoW at intersection $I_{u,v}$, w gets RoW at $I_{u,w}$ and v gets RoW at $I_{v,w}$. Such situations need to be detected, and resolved if possible, *before* robots are allowed to move.

Localization inaccuracies may cause drifts from desired locations. Since this is a huge challenge in general [21, 65], it is assumed that such inaccuracies are within a reasonable amount and can thus be handled by, e. g., artificially increasing the robot's radius. It is

important to note that localization accuracy has a crucial impact on the property of actually preventing collisions since the proposed algorithm exclusively reserves dedicated areas in the environment to allow collision-free motions. However, if a robot deviates from its path and leaves the reserved area, this property may be violated.

For *communication*, it is assumed that all robots in the system are (wirelessly) connected to the same network. Ad-hoc networks between robots are not considered for simplicity (although they may be used under certain conditions) because they impede the robot detection logic as well as synchronized clocks (if the network is sparse and not all robots are connected pairwise). More details about the robot detection logic are presented in Section 5.4.1. Algorithms can assume that connectivity is mostly available but can fail for arbitrarily long periods of time so that random delays may occur. Packet loss is not considered explicitly since it is assumed to be handled by the network layer (through the Transmission Control Protocol (TCP)). Out-of-order packet reception within the same channel is ignored for the same reason. Again, more details on the underlying communication concepts (like channels) are explained in Section 5.4.

A *synchronized clock* between participating robots is *not* required by the suggested algorithm and all of its concepts and, in particular, by its implementation within ROS. However, ROS typically requires this for, e. g., providing coordinate transformations across the system [20]. Thus, a synchronized clock among all robots is just assumed to be given for the sake of a simplified implementation in ROS, meaning that all robots can request time stamps and compare them to time stamps received by other robots. Nonetheless, realize that this is not an assumption of the presented methodology.

Robots may be *inhomogeneous*, not only regarding their shape, size, and velocity but also w. r. t. their hardware equipment. For instance, calculations may have varying durations on different robot platforms although they are part of the same system. However, for simplicity, the CLPF approximates the robot's shape by the smallest enclosing circle and robots are assumed to be *differential wheeled robots*. From a methodological point of view, the latter restriction is not required but allows for a simpler unified implementation.

The major *design goal* of the algorithm is *safety* in terms of collision-free motion. This is justified by the requirement of (autonomous) long-time operation so that the algorithm should never reach a point where manual intervention is necessary (if the aforementioned assumptions are met). Basically, this is achieved by collaboratively negotiating desired paths before motions are actually allowed to start. Where applicable, the algorithm should prefer finding a solution for all robots affected by intersections over rejecting goals just to let one (or a subset) arrive earlier. Furthermore, the design should enable generalizability w. r. t. the number of robots, that is, there may be up to $N > 0, N \in \mathbb{N}$ robots in the same environment and collision-free motion must be ensured for all robots at all times. Additionally, it is possible to add and remove robots at run-time to adjust the size of the system to varying loads (*flexibility*) based on a best effort strategy. Best effort here refers to the assumption that new robots are only allowed to enter a running system if they have been detected by all other robots that are already part of it. This can be realized by a dedicated "merge zone" where robots need to wait until

they have been detected by others (to join the "wild zone"). Assuming that wireless connectivity is at least available on the long-run, sufficient waiting times or manual merging ensures safe operation. Once a robot has joined a running system, *robustness* regarding network delays, intersections with paths of other dynamic participants (robots) or varying calculation times is important. That means that even though communication is currently impaired or some robot is forced to stop along its path, collisions will not occur.

As already stated in Section 5.1, each robot executes the CLPF algorithm on its own *decentrally*. This not only prevents to have a single point of failure (at a central negotiating instance) but also facilitates the distribution of solving an intersection scenario among all affected robots of a group. Clearly, the higher the number of affected robots, the more computing capabilities for solving are available (scalability). The proposed algorithm is not implemented in a parallel fashion to exploit such capabilities yet but possible extensions will be discussed in Section 7.2. Alternatively and depending on a system's requirement, the solver (see Chapter 4) may also be executed centrally in order to distribute the computed result to all affected robots afterwards. This also reduces the computational load in simulations that are carried out on a single or limited machine.

5.3 Overview of Methodology

In this section, an overview of the core methodology of CLPF is given and a formalization of important parts are presented. Note that the algorithm consists of many components that can be seen as (smaller) algorithms themselves; their details are explained in the subsequent Sections 5.4-5.7. Supposing that the algorithm is executed symmetrically on all robots $\{R_1, \ldots, R_N\} = \mathcal{S}_t$ of the system $\mathcal{S}_t$ at time t, we will now focus on the view of executing CLPF on a specific robot $R \in \mathcal{S}_t$—termed the *current robot*. The time t is considered to be the global time of all robots and just used conceptually to refer to the order of certain events; it is *not required* by the algorithm and omitted hereafter if that information is not relevant for a particular explanation and readability is impeded.

Essentially, CLPF requires a few *inputs*, namely the paths, radii and maximum velocities of all robots S_t, optional progress and acknowledgment information (if available), and intersection information (if any). For every intersection between two paths, the RoW is determined based on maximizing efficiency and preventing deadlocks. It is achieved by deterministically reserving areas and delaying motions along the paths within these areas but not by altering the paths itself (replanning). Clearly, the path connects the starting position of each robot with its current goal if there is one. If not, the path just contains a single pose which denotes the robot's current pose; such a path is termed a *zero-length path* (ZLP), e. g., when idle or charging. The radii and maximum velocities are used throughout the algorithm to reserve sufficiently large areas and to estimate the travel time respectively. Acknowledgments are used to ensure that we have valid known feedback from other robots that do not have any intersection with the current robot. In

such a case, the current robot would receive an acknowledgment from each of the other robots whose paths do not intersect. In order to let a robot know that an intersection area has been passed by the robot that received the RoW, progress values are sent to communicate the passing of such areas. Progress values are also used to improve the overall reactivity since path segments can thereby be released partially. Note that the current and goal poses of all robots are implicitly given by the progress values and the paths' final poses respectively. Synchronized intersection information on all robots is needed to finally determine the RoW at intersecting paths and to yield a solution that is identical on all affected robots. A robot is said to be *affected* (by an intersection or another robot) if its path has at least one intersection with another robot. Note that being affected is transitive in a sense that if R_1 has an intersection with R_2 and R_2 has an intersection with R_3 then R_1 is also (indirectly) affected by (the path of) R_3. In contrast, if R_1 has an intersection with R_2 and R_3 has an intersection with R_4 then R_1 or R_2 are not affected by R_3 or R_4. The latter allows solving conflicts and executing motions independently.

The *output* of CLPF on a robot R is a sequence of velocities over time $\boldsymbol{v}_i^t$ (trajectory) that controls the engines to move to its goal without colliding with others. If the algorithm requires to stop R because another robot has the RoW at some intersection area, it emits zeroed velocities $\boldsymbol{v}_i^t = \mathbf{0}$. In contrast, if an intersection scenario is considered infeasible, R will stay at its current position and discard its goal and path. The outcomes of processing a goal may be categorized as follows: obviously, a goal may be *reachable* meaning the global planner finds a valid path and CLPF finds a deadlock-free solution (if there have been intersections at all, otherwise the goal is always feasible). A goal may also be *unreachable*, either due to blocking static obstacles (rejected by the global planner) or due to the entire input being infeasible (rejected by CLPF, cf. Figure 5.3(a)). The proposed algorithm will always end up in one of these cases. No replanning is currently applied since one cannot ensure to prevent oscillations (replanning requires replanning and so forth).

For the remainder of this chapter, the notion of a *path* is fundamental which has already been introduced in Section 4.2 along with the terms *line segment*, *progress*, and *intersection* (see also Figure 4.2). Conceptually, such a path is complemented with meta information termed the *path id* $(n_R, t_\mathcal{P})$ which contains the unique name n_R of the robot R (whose path is $\mathcal{P}$) as well as the strictly monotonically increasing (local) time stamp $t_\mathcal{P}$ when $\mathcal{P}$ was computed. The unique names are subject to a total order based on the lexicographical ordering of n_R and its uniqueness. Every robot always has *exactly one associated path* (which may be a ZLP) so that robots and paths can also be (totally) ordered. The time stamp allows to uniquely refer to the current path of a robot (which changes over time). More specifically, they allow us to detect whenever a robot was given a new path (having a larger time stamp) as well as to classify messages as outdated (cf. Section 5.4).

Figure 5.4 shows a simplified example of how CLPF's main building blocks interact with each other. All robots need to manage local knowledge that represent their current (local)

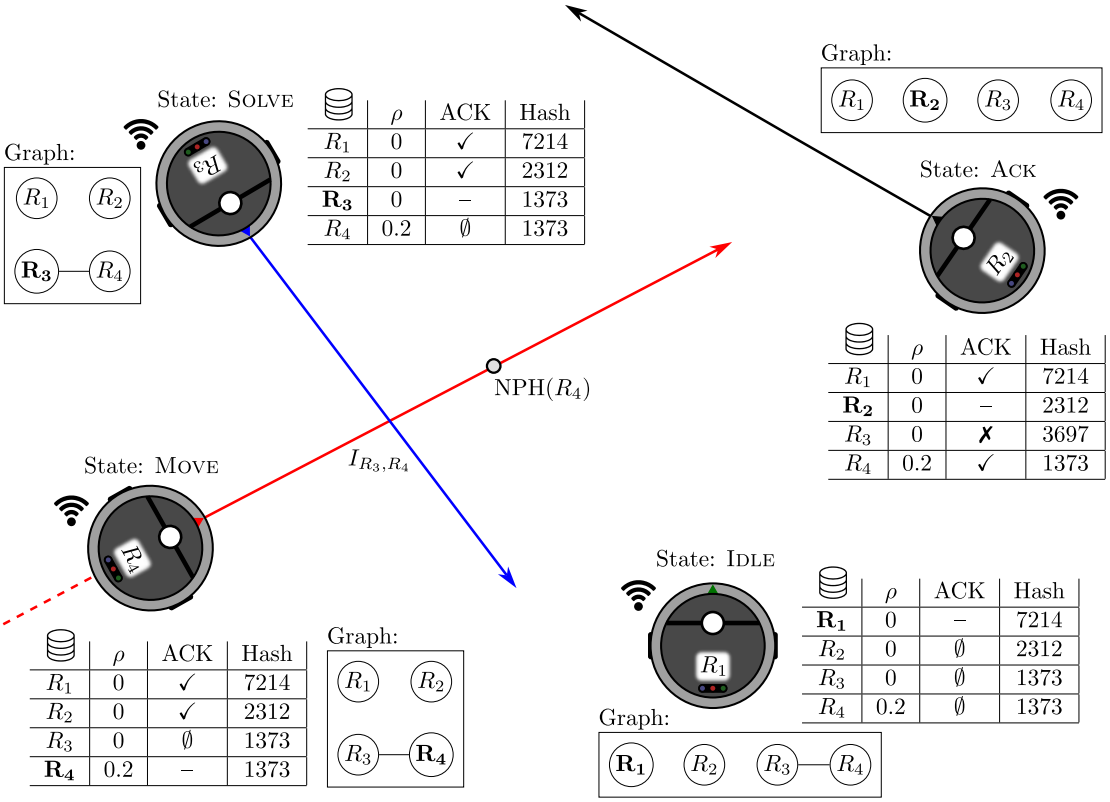


Figure 5.4: Conceptual overview of the local database, the intersection graph, communication channel and state management based on an example of a system $\mathcal{S}_t = \{R_1, R_2, R_3, R_4\}$ with four robots R_1 (green, ZLP, i. e., no goal given), R_2 (black, intersection-free), R_3 (blue) and R_4 (red) at some time t. At the current point in time within this particular example, robot R_1 does nothing (idle), R_2 is waiting for a yet missing acknowledgment (ACK) from R_3 before being allowed to move, R_3 is still solving the RoW at intersection I_{R_3,R_4} and R_4 has already started moving again after solving with R_3. Bidirectional communication is indicated by the wireless symbols next to each robot. Tables show the current local database (containing the local or received progress ρ, the received ACKs, and the local or received hash of the subgraph) on each robot. An ACK is either received (✓), still pending (✗), or not applicable aka ignored (∅). Similarly, the intersection graphs indicate the current local knowledge about detected intersections. Hashes in the table represent the local or received hash values indicating the subgraph, a given robot is part of. The current state of the finite state machine (FSM) is depicted above each robot. Because R_4 was asked to renegotiate by and with R_3 here, it emitted a next possible halt (NPH) on its path. Due to communication delays between R_2 and R_3 here, R_2 does not yet know about the intersection I_{R_3,R_4} as well as still waits for an ACK from R_3 (no motion allowed yet).

view on the system $\mathcal{S}_t$ at time t. This knowledge is stored as a robot-local database that uses n_R as a unique key to access the lastest known state of robot R. Basically, the aforementioned state refers to the input of CLPF as described in the beginning of this section, namely path, radius, velocity, progress, acknowledgments and intersection data. This information is communicated via *channels* over the network that all robots are connected to. Details of how messages are processed to update the database are explained in Section 5.4.

Generally, a robot can be in various states that reflect its current intention, modeled as a FSM. Normally, a robot is idle if it does not have a goal (e. g., R_1 in Figure 5.4). If it receives a goal, its current state determines how the robot reacts. If it is idle yet, the goal may be processed. In the simplest case, there may not be any intersections and the robot can start to move if acknowledgments have been received by all other robots (e. g., R_2 in Figure 5.4). However, if intersections have been detected which is possible due to publishing new paths right away, the type of intersection need to be considered: they may either be related to a robot that is already in motion (e. g., R_4 in Figure 5.4) or to a non-moving robot (e. g., R_3 in Figure 5.4). Depending on the current state and type, different actions need to be taken. It also determines how a robot participates and communicates with others. This observation motivated the use of a state machine to track changes in the local behavior of a robot and, thus, gave rise to employ a local (extended) FSM. It is described in detail in Section 5.5.

Another essential requirement of the algorithm is its ability to handle intersections. The published messages about detected intersections are condensed in a data structure, denoted the *intersection graph* whose details are presented in Section 5.6. Principally, the vertices of the graph represent the robots' paths and vertices are connected by an (undirected) edge if and only if their associated paths have at least one intersection, cf. Figure 5.4. Since robots and paths are in a 1:1 relation, it is valid to state that vertices represent robots and paths either way. Upon detecting an intersection, the graph is updated on a robot. This way, local intersection graphs will eventually converge to the same (global) graph on every robot in the system. For instance, this has already happened for robots R_3 and R_4 (although not on R_2) in Figure 5.4. Naturally, the initial graphs are decomposed into $|\mathcal{S}_t|$ connected components containing only a single vertex. New intersections then reduce the number of components since some of them are connected by new edges. Likewise, resolved (aka disappearing) intersections increase the number of connected components since edges are deleted. Every robot R is therefore always part of a particular connected component that contains all affected robots—such a component is termed an *(intersection) subgraph* (w. r. t. R). Once every robot detects that its subgraph is isomorphic to the subgraphs of all other affected robots, the current intersection scenario may be solved in terms of determining the RoW. Isomorphisms are efficiently detected by computing a *hash* that uniquely represents the structure of a subgraph. The details are presented in Section 5.6.2. Solving a set of intersection is also referred to as *negotiating the Right-of-Ways*. The solvers (described in Chapter 4) operate on the final synchronized subgraph and is executed on every robot decentrally, always yielding the same results. This is part of the SOLVE state (cf. R_3 in Figure 5.4).

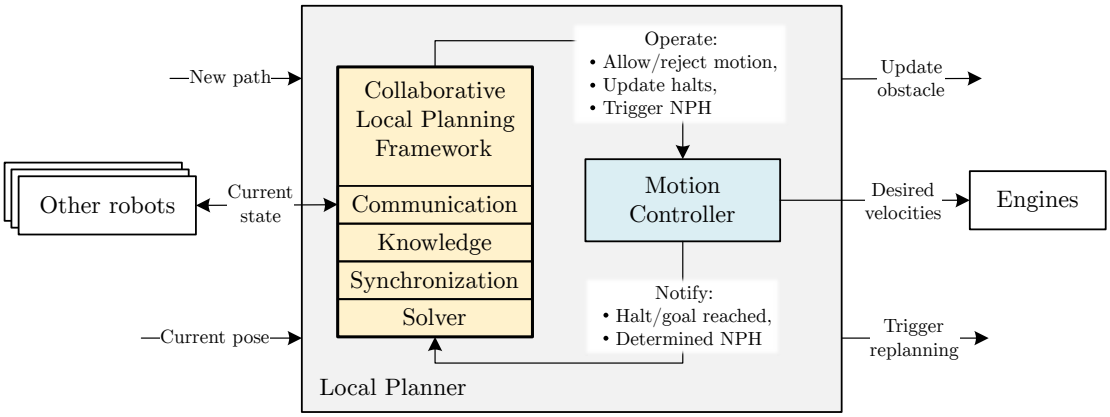


Figure 5.5: Overview of local planning with the Collaborative Local Planning Framework (CLPF) as part of the full software stack on a mobile robot (shown in Figure 5.1). The framework (yellow) is a local planner itself that provides communication, management of local knowledge and synchronization between other robots to negotiate intersections with a configurable solver module as described in Chapter 4. CLPF operates on top of and gets notified by the motion controller (blue) of a robot to abstract from the actual robot platform. The motion controller is responsible for precisely following the pre-planned path when allowed by CLPF.

As described in detail in Section 5.7, the underlying velocities are computed by a *motion controller* that is aware of the properties of the robot. Since (re-) negotiation may happen if robots are in motion, stopping them may be necessary (at least theoretically) to ensure safe operation. The motion controller not only allows to trigger the NPH along the robot's path if in motion (refer to robot R_4 in Figure 5.4 as an example) but also respects *halting positions* if it needs to give RoW to other robots at precomputed locations along its way. Such functionality is decoupled from CLPF since it depends on the motion model of the employed robots. Basically, the precomputed locations (halts) are the result of the solver, operating on the subgraphs.

Figure 5.5 shows how CLPF (yellow) is integrated into the software stack of a mobile robot (cf. Figure 5.1). The framework employs a motion controller as an abstraction to operate the robot (e. g., to allow or reject motion) and to get information about its current motion state (e. g., whether the goal was reached). While doing so, it exchanges the current state with other robots in the system to update locally stored knowledge about the current global system state.

The described concepts of a database, the intersection graph (local knowledge), and a state machine are combined and form the core methodology of the CLPF (yellow in Figure 5.5). However, note that updating the database is *independent* of the current state. The same observation applies to updating the intersection graph. The resulting algorithm is event-driven since the FSM performs state transitions based on events like receiving messages, obtaining and reaching goals, etc.

5.4 Communication Concepts and Local Knowledge

This section deals with the robot-local database and the communication concept that is used to update it. The latter employs messages of a fixed format to be sent over *topics* (aka channels) by means of the *publish-subscriber* pattern [3] so that every robot can receive all messages of others in a loosely coupled way.

Fundamentally, it is assumed that the underlying network layer prevents out-of-order reception and packet loss but possibly by causing arbitrary delays in the transmission—as provided by TCP [54]. Since two independent topics are used and all N robots are connected pairwise with each other using a single (TCP) connection for every topic, there are $2 \cdot N^2$ connections in total. The following message types are used:

- A `Path` update message is send whenever a robot has computed a valid path to a goal, if a robot has reached its goal and now has a ZLP, or if a robot is about to leave the system. Note that when a new robot enters the system, a path update message with a ZLP is sent as well to announce itself in the system. More details are explained in Section 5.4.1.
- An `Acknowledgement` message is sent to distribute acknowledgements once an intersection-free path has been received (see Section 5.4.2).
- A `Progress` message is sent when a robot makes progress along its path, especially when leaving an intersection. This notifies others that certain segments of the path have already been released as well as an intersection area is passable (again, see Section 5.4.2).
- A `Graph` message is sent to announce detected and disappeared intersections. An explanation of how data is processed and stored is given in Section 5.4.3. Basically, these messages (together with path messages) are used to update the intersection graph.
- A `Negotiation` message is published during the negotiation of a sycnhronized intersection scenario. They contain the current state of the sender participating at the negotiation. More details will be described in Section 5.4.3 as well.
- A `Hash` message is sent to inform other robots about the sender's own hash representing the current content of its local intersection graph. Because they are only relevant for negotiations, this will also be detailed in Section 5.4.3.

`Path`, `Progress`, `Graph`, `Negotiation` and `Hash` messages are published to one central global topic such that every robot receives all messages of all other robots. It especially enforces that a robot will receive the message (for all types except ACKs) from another robot in the same order in which they have been sent. However, this just applies to a pair of robots. There is no global ordering guarantee between more than two robots.

```
# Globally unique ID of the robot sending the message
PathId sender
# Sequence of poses defining the path (including a time stamp)
geometry_msgs/Pose[] path
# Inflation radius in m, approx. the smallest circle fitting the robot's shape
float32 radius
# Maximum velocity in m/s that this agent is capable of:
float32 max_velocity
```

Listing 5.1: Definition of the `Path` message in ROS; it is used to inform other robots about a newly planned path to a goal.

The second topic is used to publish ACKs. Because intersection-free paths are acknowledged individually, every robot has its own dedicated topic for receiving ACKs and maintains a connection to all other robots for publishing them.

As briefly stated in the previous section, every robot manages a database that stores information about other robots in the system, updated over time through the aforementioned messages. Note that next to the state machine, the *local* (own) *state* is managed but this is explained in Section 5.5; the database just stores the state of *remote* (other) *robots* (from the perspective of the *current robot*). For simplicity and improved readability, this has been combined in the tables next to each robot in Figure 5.4. Over time and when messages are being exchanged and processed, the database's content converges to the same content on all robots $R \in \mathcal{S}_t$ which enable them to have the same view on the system. This is used to solve intersection scenarios. This section mainly focuses on receiving messages and updating the database—the sending of such messages is described in Section 5.5. Therefore assume that the receiver is denoted by $R_r \in \mathcal{S}_t$ and the sender is $R_s \in \mathcal{S}_t$.

5.4.1 Updating Paths

From a theoretical point of view, it is assumed that the system is always in a *valid configuration*, meaning that at any time t, a robot $R \in \mathcal{S}_t$ always knows about any other robot $S_t \setminus \{R\}$ in the system (trivial if $\mathcal{S}_{t_0} = \emptyset$ at startup) and no collisions exist. When the first robot R_1 joins, it publishes its first `Path` message which is not yet received by any other (since there are none). If the second robot R_2 joins, it publishes its path as well and also receives the message from R_1 since a latch logic is employed to always deliver the latest (single) message to new robots. This is important to ensure consistency (not needed for the ACK topic). When more robots $R_3, R_4, ..., R_n$ join the system, operation is considered to be safe, if all of them have received all other `Path` messages. Depending on connectivity, this state may be reached quickly or requires more time until all messages have been delivered. To guarantee collision-free motion, merging new robots must be monitored (e. g., by means of a "merge zone").

Basically, a `Path` message contains the geometric path $\mathcal{P}_{R_r}$, its meta information $\mathcal{P}_{id}$, the radius of the robot and its maximum velocity, see Listing 5.1. When a robot enters

the system, it publishes its current pose with $|\mathcal{P}_{R_s}| = 1$. Likewise, when it leaves, it publishes $\mathcal{P}_{R_s} = \emptyset$ so that other robots R_r will remove R_s from their database. Again, it must be ensured (e. g., by entering the merge zone) that the leaving robot is not blocking the environment anymore since it gets somehow invisible to other robots after leaving.

A `Path` message from R_s is always stored (on R_r) except for the case where a received time stamp is less than the stored time stamp; such messages are discarded. Notably, messages of other types (`Acknowledgement`, `Progress`, `Negotiation`, `Graph` and `Hash`) are always discarded if the time stamp of the lastly received `Path` message is larger than the time stamp received along with those messages.

Due to parallel communication, it may happen that R_r receives a `Graph` message from R_s that refers to a path's time stamp t_j but the database entry for R_s still stores a path with the time stamp t_i with $t_j > t_i$. In other words, the intersection information refers to a more recent path that has not yet been received in terms of a `Path` message from R_s. Again, this is possible due to multiple and parallel connections with other robots. In such a case, R_r will invalidate the stored path for R_s which flags the entire database entry for R_s as so-called *incomplete*. Note that it is important that the database entry for R_s is still existing and, in this example, stores the just received `Graph` message (which would not get sent again). The entry for R_s would become *complete* again if an appropriate `Path` message is received. The same handling applies for all other types of messages.

5.4.2 Handling Acknowledgments and Progress

If a robot computes a path to its current goal that does not intersect with any other path, it still requires to receive ACKs from all other robots. This is required to guarantee safety: assume that a system $\mathcal{S} = \{R_1, R_2\}$ is given in a valid configuration, that is, R_1 knows about R_2 and vice versa. Assume further that R_1 moves along its current path and R_2 is idle yet. Let's suppose R_2 gets a path that does not intersect with the current path of R_1. Since there is no intersection, R_2 would start to move as well while R_1 may have reached its goal, already received a new path but does not have received the path of R_1 (arbitrary network delays). In this situation, R_2 would conclude that there is no intersection with R_1's (previous) path (which may even be a ZLP) and starts to move. Longer network delays would then even increase the probability of not receiving the most recent path of R_1 in due time on R_2. Thus, both would move while erroneously assuming that they have recent knowledge about each other. In contrast, if both would have required ACKs before starting their motion, they would have either detected the intersection or would not have started to move simultaneously. This justifies the need for ACKs.

An `Acknowledgment` message, as shown in Listing 5.2, consists of the sender's path ID as well as the target path ID that is being acknowledged. If the received time stamp of the sender is larger than the stored path's time stamp, the database entry for the receiver is marked as incomplete. This can happen due to parallel communication over the two

```
# Globally unique ID of the robot sending the message
PathId sender
# Id of path/robot that this message is targeting (aka acknowledging)
PathId target
```

Listing 5.2: Definition of the `Acknowledgment` message in ROS; it is used to acknowledge that a newly received path from another robot (`target`) is intersection-free with the path of the sending robot (`sender`).

```
# Globally unique ID of the robot sending the message
PathId sender
# Completion status of path in [0,N-1] whereby N denotes the number of poses
float64 progress
```

Listing 5.3: Definition of the `Progress` message in ROS; it is used to communicate the current completion status of a robot on its path and especially to release intersections.

channels. In such a case, the sender acknowledged the receiver's path but referred to a newer local path that the receiver has not received yet. Note that since ACKs are only sent in reaction to receiving a new path from another robot, it is not necessary to deliver latest ACK messages to new robots (that have just entered the system) as contrasted to the other employed message types, namely `Path`, `Hash`, `Negotiation` and `Progress`. `Graph` messages are re-published ("latched") in a specific way which will be explained in Section 5.4.3.

Sharing progress when moving along a path is another essential part of the algorithm. It serves two purposes. First, it allows releasing paths partially which increase the overall efficiency (as a trade-off regarding network use). Second, progress is used to inform robots at intersection areas that did not have RoW. To reduce network use, a more compact representation has been developed by means of *progress* values already presented in Section 4.2 (cf. Equations (4.3)-(4.5)).

Received `Progress` messages (see Listing 5.3) are simply stored and processed upon reception. If a robot detects that an intersection is released based on its own progress, it sends an ACK to that robot which is required if a renegotiation is needed during a motion. A renegotiation is very similar to a "normal negotiation" (and also requires to find a RoW assignment for all pairwise intersections) but involves robots that are already in motion. Such robots are handled by requesting them to commit to a halt position along their paths such that they can be considered as non-moving (located at their halt positions respectively); more details are explained in subsequent sections. The remaining steps (testing for valid ACKs, publishing progress, etc.) are described along with the FSM (Section 5.5) and the intersection graph (Section 5.6).

```
# Globally unique ID of robot/path sending the message
PathId sender
# Type of graph update to perform: true to add edges representing new conflicts,
# false to remove existing edges (i.e., all conflicts gone)
bool add
# - If add is true, sorted list of path IDs having a conflict with the sender.
# - If add is false, sorted list of path IDs not having a conflict with the
#   sender anymore.
PathId[] affected
# Must be set to the updated hash of the current local subgraph after applying
# the update operation locally on the sender's side.
uint64 hash
# true to signal a ZPI, false to publish a normal conflict/intersection.
bool zpi
```

Listing 5.4: Definition of the `Graph` message in ROS; it is used to inform other robots about newly detected and disappeared conflicts. Upon reception, a robot updates its local intersection graph. If a zero-length path intersection (ZPI) is signaled, path re-computation is triggered.

5.4.3 Managing Intersection States

The message type that is used to announce newly detected intersections is the `Graph` message as shown in Listing 5.4. The `sender` refers to the sending robot R_s and identifies its current path uniquely. The semantic of `affected` depends on the value of the `add` flag: if it is true, `affected` contains all paths having an intersection with R_s. If it is false, it contains all paths whose intersections have completely disappeared. Once a robot detects an intersection (see Equation (4.6)) in reaction to the reception of a `Path` message, a `Graph` message is published to all robots in the system. The details of how these messages are used to update the intersection graph will be presented in Section 5.6.

Note that ZPIs are handled in a special way: if a robot receives an intersecting ZLP, it sends a `Graph` message with `zpi` set to true and triggers a path re-computation locally (effectively dropping its current path). In contrast, if a robot receives an intersecting non-ZLP path but itself currently has a ZLP, it sends a `Graph` message with the `zpi` attribute set to true (without changing its local path). This triggers a path re-computation on the receiver which resolves the ZPI. This way, ZPIs are never represented in the intersection graph and therefore somehow ignored. As already described in Section 5.4.2, outdated messages are discarded and more recent messages will also make the database entry for the sender R_s (on the receiving robot R_r) incomplete. The `hash` attribute must always be set to the sender's current local subgraph hash. A new `Hash` message (see Listing 5.5) is published upon every local change of a robot's subgraph (more details will follow in Section 5.6.2.2).

A fundamental property of the CLPF regarding safety is that a robot R is only allowed to move if

(a) ACKs have been received by all robots not having an intersection with R and

(b) RoWs have been (at least for R) successfully negotiated at intersections.

Note that ACKs are also required (cf. property (a)) if an intersection occurs. (b) requires synchronized states beforehand and must also be true if robots are already moving. For example, if a set of robots has already negotiated their intersections and started moving, another robot may want to join with a path that also has an intersection with one or more of the moving robots. In such a case, that robot may only start to move if renegotiation has taken place successfully so that both properties still apply. Until then, moving robots will continue to move. Robots that are already moving have implicit precedence in the algorithm over non-moving robots.

These properties are enforced by communicating `Negotiation` messages (see Listing 5.6) and a robot runs through several *negotiation states* when intersections have been detected. These states may not be confused with the overall state of the robot as managed by the FSM (see Section 5.5). The negotiation states can be viewed as a nested state machine that is used if the specific intersection related states of the FSM are active. Notably and unlike the overall robot state of the FSM, negotiation states are transparent to other robots since transitions are communicated with the `state` attribute of the `Negotiation` message. Moreover, all affected robots will perform similar transitions during a synchronization and as a preparation for solving.

```
# Globally unique ID of robot/path sending the message
PathId sender
# Hash of the current local subgraph representing the content and connectivity
# (must always be > 0)
uint64 hash
```

Listing 5.5: Definition of the `Hash` message in ROS; it is used to communicate a robot's current subgraph hash (to eventually become synchronized).

```
# Globally unique ID of the robot sending the message
PathId sender
# The robot's current negotiation state; possible values are 'Unfrozen',
# 'Frozen', 'Solved':
uint8 state
# The next possible halt (NPH), if any. Only relevent for publishing
# (Unfrozen ->) Frozen. Set to zero if the robot is at its start point;
# must be set to -1 for all irrelevant cases.
float64 start
# Must be set to the (un)frozen/solved hash. If state is 'Unfrozen', hash must
# be set to the new hash of the graph that already contains the changes.
uint64 hash
```

Listing 5.6: Definition of the `Negotiation` message in ROS; it is used to communicate a robot's current negotation state as well as its NPH. The values of `state` correspond to the possible transitions shown in Figure 5.6.

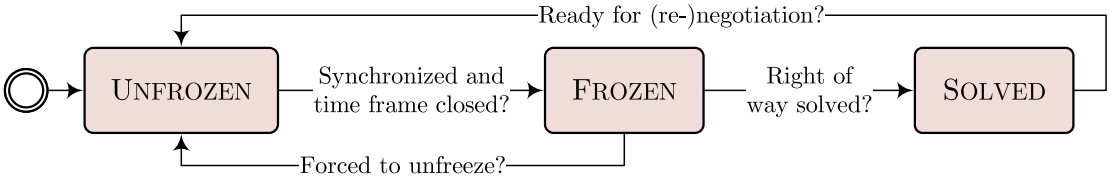


Figure 5.6: Negotiation states a robot may be in once an intersection has been detected. This can be considered as a "nested FSM" that is active when one of the red intersection related states of the "main FSM" (depicted in Figure 5.7) is active. Unlike the main FSM, state *changes* in this nested FSM are communicated via the `state` attribute in a `Negotiation` message (see Listing 5.6) and are therefore known to other robots. The shown state transitions (arrows) take place based on received messages (events). The circled state on the left denotes the starting state.

Figure 5.6 depicts the negotiation states and indicates when transitions take place. If an intersection has been detected, the Unfrozen state is active which indicates that further knowledge needs to be gathered in order to become synchronous. A precise definition of a "synchronized state" is given in Section 5.6.2; up to now it is sufficient to perceive it as a set of information that contains the intersection related data of all affected robots w. r. t. the current robot R. Basically, testing for synchronization is based on a hash representing the current subgraph that is stored in the `hash` attribute (see Listing 5.5). Once a robot detects that it is synchronous with other robots of its subgraph, a timer is started to expire within $T_d \geq 0$ seconds. The timer allows other robots to join a negotiation (and to prevent renegotiations over and over again) if goals are distributed in close succession—a regular case in practice. In fact, the timer is optional and can be disabled by setting $T_d = 0$. If it elapses, the current robot R transitions to the Frozen state which indicates that all required information is given, at least on the current robot. The state remains Frozen until either R is forced to go back to the Unfrozen state because the synchronized state was changed by an affected robot that was not synchronous yet or negotiating the RoWs has been completed. In that case, R will publish Solved in order to immediately go back to Unfrozen indicating its ability to renegotiate with other robots. If the negotiation scenario was considered feasible, transitioning to Solved also involves starting the movement. Renegotiations are implicitly delayed until an already pending negotiation has been completed because robots with pending negotiations will prevent synchronization (if they are part of the renegotiation).

Transitioning between negotiation states is independently handled by the current robot in contrast to other (remote) robots. More specifically, any robot $R_a \in \mathcal{S}_t$ may transition to, e. g., the Frozen state regardless of the state of the current robot $R \in \mathcal{S}_t \setminus \{R_a\}$. If R receives a `Negotiation` message indicating R_a's transition, it simply stores it and may trigger local transitions if state becomes synchronous. Thus, transitions from remote robots are always only "observed" and cannot be "directly triggered". Nonetheless it is important to note that the remotely frozen state, once received, must be stored

separately due to the highly parallel and asynchronous state transitions happening on all robots. More specifically, the current robot R may become synchronous at time t_i and any other affected robot may detect its synchronous state at t_j. Then both $t_i < t_j$ and $t_i > t_j$ (even $t_i = t_j$) is possible so that the received frozen state must be stored separately to avoid overwriting it with newer `Graph` messages as it is used by the solver.

A moving robot asks its motion controller to stop at the next possible position based on the idea that a robot on its own knows at best when it can stop along its path. It is converted to a progress (Equation (4.3)) and published in the `start` attribute of the `Negotiation` message. The solver considers that position as the beginning of the path. The moving robot may not actually come to a halt if the negotiation completes sufficiently fast. Moreover, the halt returned by the motion controller may also be parameterized so that it is far more ahead of the robot which reduces the probability of a complete stop. However, note that earlier halts may increase the probability of actually finding a solution (trade-off: feasibility vs. efficiency).

5.5 Finite State Machine: States and Semantics

This section focuses on what a robot stores for managing its own state and how the state is managed locally using a FSM. The latter also affects what and when messages are being sent.

First, all possible events are presented. Events may trigger state changes and can cause updates in local data structures.

Table 5.1: Overview of all possible states of the Finite State Machine along with their precondition and meaning. A robot is always in exactly one of these states at a time.

ID	Precondition	Semantic
Init	None	Initial state of the FSM.
Idle	• No goal given • Not allowed to move	Wait for a new goal.
Goal	• Goal given (possibly with intersecting ZLPs) • Not allowed to move	Wait for the final path.

ACK	• Final path given • No intersections known • Robot may be moving if it has re-entered ACK from a renegotiation	Represents that the robot has *no* intersections with other paths and waits for ACKs from all others.
ACKDELAY	• ACKs from all robots received • Robot may be moving or stopped	Delay motion for $T_d \geq 0$, increases efficiency if goals are issued in close succession.
INTERSECTION	• Intersection detected but not synchronized yet • Robot may be moving or stopped	Wait until all required information needed for solving are synchronous across all affected robots.
INTERSECTIONDELAY	• Local knowledge is synchronous with all affected robots • ACKs have been received from all non-affected robots • Robot may be moving or stopped	Delay synchronization for $T_d \geq 0$, increases efficiency if goals are issued in close succession.
LOCK	• See INTERSECTIONDELAY • State is locally frozen (but not synchronized yet)	If all affected robots are in this state, i. e., frozen, start the solver. If the synchronized state has changed (by another affected unfrozen robot), restart the synchronization. Do not change the frozen state itself.

SOLVE	• Synchronized and frozen state • ACKs have been received from all non-affected robots • Robot may be stopped or moving (if moving, a halt was triggered)	Solve the RoWs and set halting positions for the current robot if necessary. See, e. g., solvers presented in Chapter 4.
MOVE	• Goal considered feasible • All intersections have been negotiated (if any) • Halt positions have taken effect (if any) • Robot may be moving or stopped	Robot is allowed to move, possibly given constraints (halt points). Also check and handle upcoming or delayed negotiation requests (new intersections).

Second, given an understanding of the various events that can occur in the system, we now review the states. A summary of all states is given in Table 5.1. Figure 5.7 shows an excerpt of the FSM along with events that can cause state transitions.

5.5.1 The Init, Goal and Idle states

After initialization (INIT state), the IDLE state is entered and held if there is no current goal given to the robot. It transitions to GOAL once a goal is received and a global path could be computed successfully (event INTNEW). The GOAL state is used to request new paths from the global planner to the same current goal (event INTUPD) until there are no intersections with ZLPs anymore—we term the resulting path the *final path*. Note that a goal is only accepted if the robot is IDLE. In particular, if a new goal (INTNEW) is received while another goal is already being processed, the new goal will be rejected. Thus, the goal cannot be preempted which is generally not desirable in an industrial context where orders need to be processed until they are completed. Upon a termination request (event INTTERM), the robot will send an empty `Path` message indicating that it leaves the system. Note that INTTERM events are *ignored* in all other states to ensure safety, i. e., a robot may only leave the system when others perceive it as being idle (IDLE, GOAL). The FSM may veto the motion of the robot; this veto flag is always set when the IDLE state is entered (event INTENTER). Additionally, the ZLP is published because the robot will never be moving when IDLE (see precondition in Table 5.1). Section 5.7 will explain how the veto flag is processed. Note that ZPIs can still occur in subsequent

Table 5.2: Overview of all events that can affect the Finite State Machine of a robot. They are categorized in events that are triggered "locally" on the robot (internal events) and those events that other robots trigger via messages (external events).

	ID	Semantic
Internal	IntNew	A new path has been received from the overlying system level which may be rejected or accepted.
	IntUpd	A recomputed path has been received due to intersections with a ZLP.
	IntTerm	The current robot was requested to terminate from the overlying system level.
	IntAck	An acknowledgment delay timer has expired. Motion is about to start.
	IntIts	An intersection delay timer has expired. State is about to be frozen.
	IntEnter	A transition to a new state has taken place.
	IntLeave	An old state has been left.
External	ExtNew	New robot detected not present in the database yet.
	ExtUpd	An updated path has been received by an already known robot.
	ExtLeave	Another robot is about to leave the system and will no longer receive and process requests.
	ExtProgr	Progress has been received from another robot. Parts of the path have been released and an intersection area may be released.
	ExtIts	An intersection has been detected by a robot (synchronization and negotiation required), a negotiation has been completed or renegotiation is required.
	ExtAck	An acknowledgment has been received from another robot to indicate "no intersections" with the sender.

states (due to robots already being in motion) which causes a robot to go back to Goal until there is no ZPI anymore.

Recall from Section 5.4.3 that intersections with ZLPs are handled separately. In fact, negotiations will never ever deal with ZPIs. As explained for the Goal state, this is because paths are replanned until they are not intersecting with ZLPs. It is justified by the idea that robots may become immobilized, e. g., due to faulty engines so that requiring them to move away renders the entire system more error-prone. Additionally, moving them away from their position poses the question of an appropriate new location which is out of the scope of this algorithm and would even need to be addressed by an

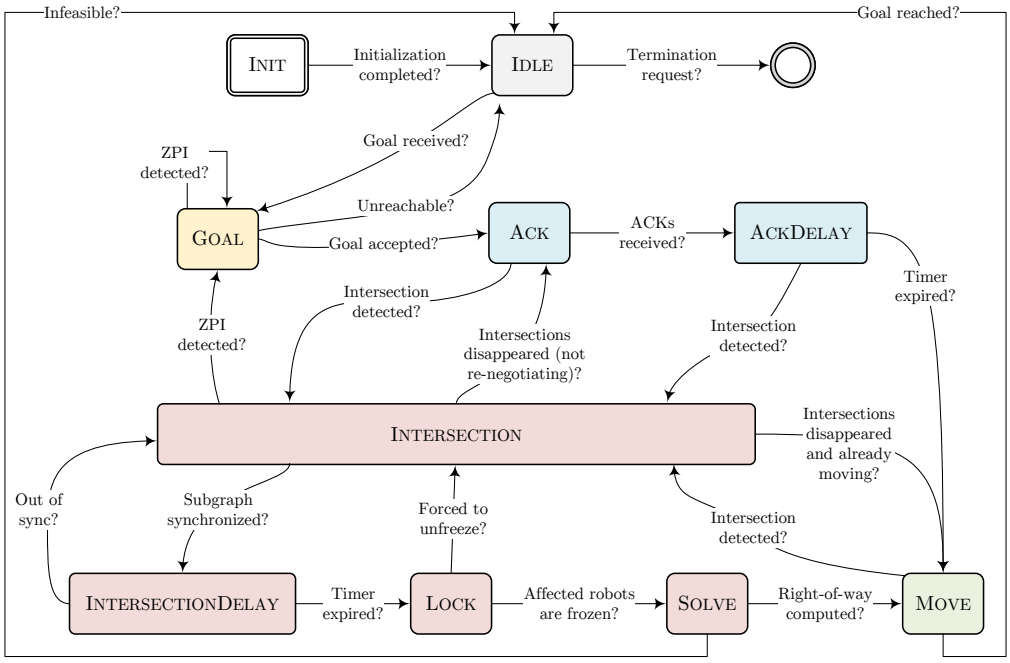
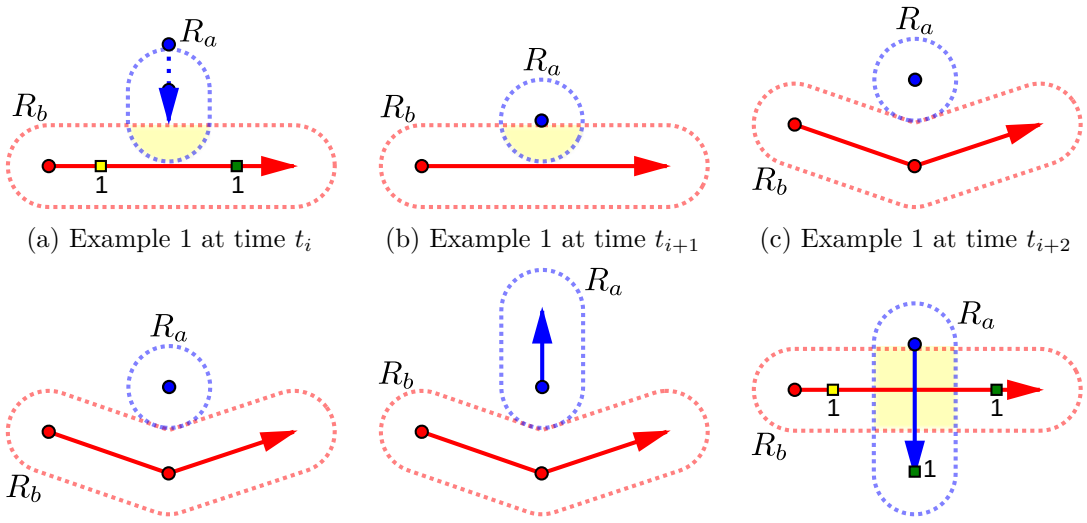


Figure 5.7: Excerpt of the states and transitions of the developed FSM executed on every robot. After initialization (starting state INIT), a robot enters the IDLE state (gray). After a goal has been received and no ZPI has been detected (GOAL state), it continues either in the intersection-free states (blue) or transitions to the intersection related states (red). Motion is allowed when MOVE (green) is reached.

overlying system level of the robot's software stack. For these reasons, a robot informs other robots about its current (final) position once it comes to a full stop (at a goal). Moreover, that position is removed from the known map of all robots when a final path is published. This way, other global planners can incorporate the induced ZLP polygon to avoid ZPIs, obviating the need to request other robots to move away. However, since this constitutes a race conditions, replanning is required if they occur nonetheless. Such ZLP polygons can be considered as semi-static obstacles because they are added to the global planner's map.

As an example, assume that at time t_i there are two robots R_a and R_b whereby R_a *is about to reach* the final goal of its current path $\mathcal{P}_a^i$ (but its not reached yet), see Figure 5.8(a)–(c). At the same time, R_b receives a new path $\mathcal{P}_b^i$ which intersects with $\mathcal{P}_a^i$ at the goal area of $\mathcal{P}_a^i$ (yellow). R_b therefore *starts* to exchange information with R_a to become synchronized and negotiate the RoW, see Figure 5.8(a). Moving ahead in time, at t_{i+1} robot R_a will reach its final goal and updates itself to a ZLP $\mathcal{P}_a^{i+1}$, see Figure 5.8(b). However, since transmissions takes time, R_b does not know about $\mathcal{P}_a^{i+1}$ yet and still



(a) Example 1 at time t_i (b) Example 1 at time t_{i+1} (c) Example 1 at time t_{i+2}

(d) Example 2 at time t_i (e) Example 2 at time t_{i+1} (f) Alternative to (e) at t_{i+1}

Figure 5.8: Two examples for ZPIs and possible evolutions over time due to parallelism and race conditions between two robots R_a (blue) and R_b (red). In the first example, to resolve the ZPI detected after (b), replanning on R_b takes place in (c). Depending on the actual path of R_a in the second example (d), there is either no ZPI at all as visualized in (e) causing a small detour for R_b when executed, or a normal intersection (yellow) has to be negotiated by the solver as shown in (f).

assumes that $\mathcal{P}_a^i$ is valid. For now we will assume that the time $t_{i+1} - t_i$ is too short for the synchronization and negotiation between R_a and R_b to complete, i. e., R_a will reach its goal before negotiation completes. At t_{i+2}, R_b will eventually receive $\mathcal{P}_a^{i+1}$ and will detect an intersection with the ZLP $\mathcal{P}_a^{i+1}$ which, in turn, will trigger replanning on R_b, see Figure 5.8(c). Notice that if $t_{i+1} - t_i$ is sufficiently large to get synchronized, the used solver (Chapter 4) will handle this situation. Apropos, if the initial situation is different in a sense that R_a is about to start moving and its ZLP $\mathcal{P}_a^i$ will disappear on R_b soon, R_b will possibly still plan its path taking $\mathcal{P}_a^i$ into account (again, since transmission takes a reasonable amount of time), see Figure 5.8(d). This can result in two cases. First, R_b may need to drive a small detour around R_a (without any intersection) which is accepted as an artifact of parallelism (see Figure 5.8(e)). Second, if R_b receives $\mathcal{P}_a^{i+1}$ (final non-ZLP of R_a) soon enough (see Figure 5.8(f)), an intersection (yellow) will be detected which is then again handled by the solver properly. Also note that R_b cannot have an intersection with the initial ZLP $\mathcal{P}_a^i$ of R_a. Thus, despite all possible race conditions, it is impossible that R_b will miss the existence of R_a due to the assumption of a valid (start) configuration: it will either (locally) detect an intersection with a ZLP which triggers replanning or an intersection with a final path is negotiated by the solver of the affected robots.[2]

[2]The actual decisions of the solver depend on the timings, distances, velocities, etc. of the scenario. Here, regarding the first example of an intersection between $\mathcal{P}_a^i$ and $\mathcal{P}_b^i$ it will (most likely) consider R_b's

5.5.2 The Ack state

Continuing with the explanation of the states, ACK represents the state in which a robot does not have any intersections at all and waits for acknowledgments from all robots in the system. It is entered from the GOAL state if there are no ZPIs either upon entering it or when an updated path has been received (event INTUPD). As already explained, ACK ensures that the absence of intersections is based on recent knowledge. Within the ACK state, three events are important. First, upon entering (event INTENTER), all ACKs may have already been received so that the current robot can immediately proceed to the next state, namely ACKDELAY. Likewise, when an EXTACK fires, the robot also transitions to ACKDELAY if all ACKs have been received. Third and finally, when a robot leaves the system (EXTLEAVE), the current robot was possibly refrained from transitioning to ACKDELAY just due to the missing ACK from the leaving robot. Since it left, that ACK is no longer required. Given the current robot $R^\star$, acknowledgments from all other robots $\mathcal{S} := \mathcal{S}_t \setminus \{R^\star\}$ are considered to be valid at some time t if the following condition holds

$$\forall R \in \mathcal{S} : c(R) \wedge \left(\neg\chi(R, R^\star) \Rightarrow t_R^{\text{ack}} = t_{\mathcal{P}_{R^\star}}\right), \tag{5.1}$$

whereby $c(R)$ indicates if R's database entry is complete on $R^\star$ and t_R^{ack} denotes the acknowledged time stamp received from R (i. e., `target` in Listing 5.2). In other words, all (other) robots R must have a complete database entry on $R^\star$ and having no intersections with another robot R (that is, $\chi(R, R^\star)$ is false, see Equation (4.6)) also requires $R^\star$ to have an ACK from R that targets $R^\star$'s current path (uniquely identified by its time stamp $t_{\mathcal{P}_{R^\star}}$).

5.5.3 The AckDelay state

In principle, when all ACKs have been received according to Equation (5.1), $R^\star$ may be allowed to move. However, since in practice goals are most probably assigned to all available robots in a bunch (very close succession), the ACKDELAY comes into play. It is entered from ACK when Equation (5.1) applies and a timer is started to expire within a configurable delay of $T_d \geq 0$ seconds. Within this amount of time, other robots can check for intersections and join the negotiation (if necessary) before actually moving. As already noted in Section 5.4.3, this step can be skipped if $T_d = 0$. The delay provides a parameter to tune the overall efficiency since a value of zero causes many transmissions of messages that may be obsolete soon. However, large values of T_d delay the start of motions of the robots which is not desired as well. When the timer has expired and there are still no intersections, the robot directly proceeds to the MOVE state. If the ACKDELAY is left prematurely (e. g., due to intersections, cf. events EXTNEW, EXTUPD, and EXTITS), the timer is stopped.

path to be infeasible since its goal is already permanently blocked by R_a. With respect to the second example at the beginning of R_a's path, it will (presumably) give R_a the RoW because it already blocks the intersection area.

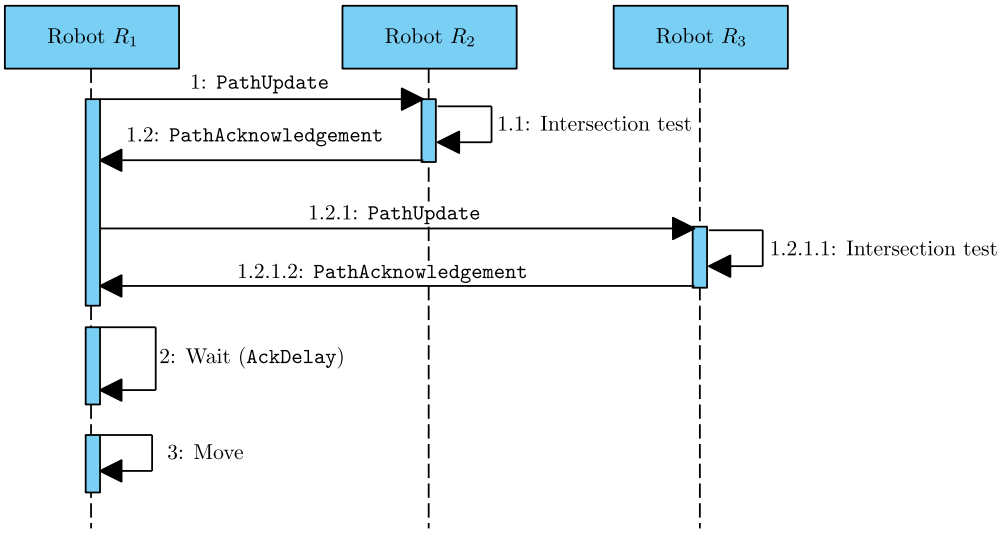


Figure 5.9: Sequence diagram illustrating how ACKs are exchanged after a new path has been computed on robot R_1. Upon reception of the new path, robots R_2 and R_3 check for intersections and publish an `Acknowledgement` message directly to R_1 indicating that the received path is intersection-free. After waiting for T_d seconds, R_1 starts to move.

The robot enters the MOVE state if and only if the goal is reachable and there are either no intersections (MOVE is then entered from ACKDELAY) or all detected intersections have been successfully negotiated (MOVE is then entered from SOLVE which is discussed later), see Figure 5.7. In the absence of intersections and upon entering MOVE, the motion-veto flag is unset which triggers the motion controller as described in Section 5.7 so that the robot starts moving. A brief example is shown in Figure 5.9 where robot R_1 just requires the ACKs from R_2 and R_3 to be allowed to move. In principle, these states would be sufficient if there would be no intersections at all.

5.5.4 The Intersection state

Four additional states are used to handle intersection related information, namely INTERSECTION, INTERSECTIONDELAY, LOCK, and SOLVE (red in Figure 5.7). INTERSECTION is entered from all states except for INIT, IDLE, GOAL and SOLVE if a new intersection has been detected. Such a detection is based on the EXTNEW, EXTUPD and EXTITS events. In case of the former two events, an explicit intersection test is required (see Equation (4.6)) while the latter only requires processing the received `Graph` message. If an intersection has been determined upon EXTNEW and EXTUPD, a `Graph` message is emitted to generate the EXTITS event. More specifically, intersection tests are only conducted upon receiving a `Path` message, i. e., a robot R_s with a new paths prompts other robots R_r to check their current path against R_s' newly published path for intersections by publishing a `Path` message. If one or more intersections have been

detected, a `Graph` message is sent. This distributes the work of testing intersections for a given path equally among all other robots in the system. Intersections are therefore detected either directly by an intersection test (after being prompted by a `Path` message) or by receiving a `Graph` message (in response to a prompt).

If a `Graph` message (with `add` set to false) indicates that all intersections have disappeared for the current robot, it transitions to Ack if motion is not yet allowed (veto for motion is set). Otherwise, the motion to the current goal is continued by entering Move.

In case an intersection with a ZLP has been detected (which implies that motion is currently not allowed), the robot transitions back to Goal to recompute a new path to the goal (respecting the blocked ZLP area). Observe that intersections can disappear because if some robots are already moving they may immediately stir out of an intersection area shortly after the intersection was detected.

Assuming there are still intersections, the current robot transitions to IntersectionDelay if all affected robots are synchronized and Equation (5.1) holds. Otherwise the current robot stays in the Intersection state until all affected robots are synchronized. The current robot $R^\star$ considers its state *synchronous*, if the following condition holds:

$$\forall R \in \mathcal{S} : c(R) \wedge (R \in G_{R^\star} \Rightarrow \text{hash}(R^\star) = \text{hash}(R)) \,. \tag{5.2}$$

In this equation, $G_{R^\star}$ denotes the current subgraph of $R^\star$ and $\text{hash}(R)$ returns the current subgraph hash for the given robot R on the current robot $R^\star$. In other words, all known robots must have a complete database entry and if a robot is affected (i. e., part of the current subgraph), its received subgraph hash must be equal to the current robot's hash—effectively requiring the subgraphs to be identical. Details of the hashing algorithm are presented in Section 5.6.2. Notice that the entire intersection graph may differ as it is not required to negotiate with robots that are not affected at all. However, we require complete database entries for all robots of the system in order to be able to test if a robot is affected at all (based on the path id $\mathcal{P}_{id}$).

Upon receiving ExtLeave or ExtProgr, the current robot transitions back to Ack if all intersections have disappeared (due to the left or progressed robot respectively). For every robot whose intersection has disappeared (due to received progress in ExtProgr), a separate ACK is sent to ensure that such intersection-free robots which have transitioned to Ack (after being in Intersection or IntersectionDelay) will eventually proceed to AckDelay. Observe the possibility that intersections may disappear completely even though a robot has joined a renegotiation. This is because the robot was moving while detecting the intersection and since it is still moving while trying to become synchronized, the intersection may disappear. Clearly, this can also happen in subsequent states, namely IntersectionDelay, Lock, and Solve. However, it is not a problem because the robot would either transition back to Ack (from Intersection, IntersectionDelay, and Lock) or the solver would detect that there is no intersection anymore (within Solve).

As explained in the previous paragraph, a robot transitions from Intersection to IntersectionDelay if

- all subgraphs are locally synchronized,
- database entries for all affected robots are complete, and
- ACKs have been received from all non-affected robots.

The idea of the IntersectionDelay state is similar to AckDelay: robots may be assigned to goals in close succession so that intersections will most likely occur very shortly, one after another. That is why solving can artificially be delayed to prevent reoccurring renegotiations. Thus, upon entering IntersectionDelay, a timer is started to expire within a configurable delay of $T_d \geq 0$ seconds. If it expires and the current state is still IntersectionDelay, the robot continues its processing in the Lock state. Elsewise, if all intersections have disappeared and the motion veto flag is unset, it transitions to Ack. If it is set (motion is allowed), the residual motion is continued by transitioning to Move. In case of a new or updated path with an intersection (events ExtNew, ExtUpd, and ExtIts), the robot goes back to the Intersection state to become synchronous again. In any case, when IntersectionDelay is left, the aforementioned timer is stopped.

Note that when a `Negotiation` message is received with state Unfrozen (cf. Figure 5.6) but the sender R_s was already marked as Frozen, this means that R_s was forced to go back to Intersection since the subgraph changed. The current robot only needs to unfreeze R_s' state and go back to Intersection, too (as already explained). Likewise, if the state of the received `Negotiation` message is Frozen, R_s' state will be saved and marked as being frozen. Obviously and like for the Intersection state, the state of a `Negotiation` message cannot be set to Solved at this point. A robot's negotiation state becomes Frozen when entering Lock which is explained next.

5.5.5 The Lock state

The Lock state ensures a consistent state between all affected robots in a negotiation. Because all robots execute the solver symmetrically, the same result (RoW assignment) is expected on all robots if the solver (see Chapter 4) itself works deterministically and the inputs are identical. The latter is ensured by freezing the state upon entering Lock so that all affected robots will eventually transition to Solve only if they have reached a synchronized and frozen state. Transitioning to Lock (i. e., freezing locally) also sends a `Negotiation` message with `state` set to Frozen. Conceptually, the current robot waits in Lock until it has received a Frozen state (cf. Listing 5.6) from all affected robots with the same subgraph hash. It then transitions to Solve in order to execute the solver given the collected data.

New robots that have detected an intersection with one of the affected robots in Lock are implicitly delayed joining the current subgraph until the current negotiation completes. However, there is one exception: it can happen that a subset of the affected robots already transitioned to Lock but the remaining robots are still waiting in IntersectionDelay

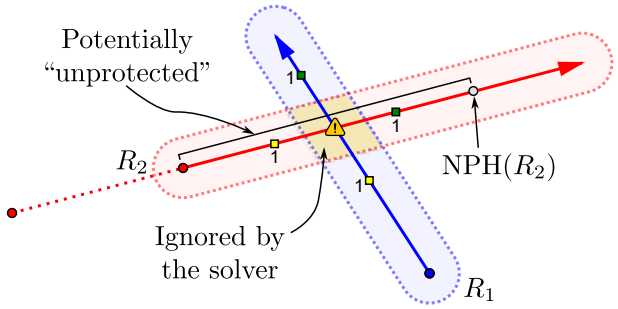


Figure 5.10: Example illustrating the potential risk of crashes in a renegotiation due to a NPH because the solver only considers intersections between a robot's starting position (here: the NPH) and its goal. Because the NPH is issued on R_2's path (red) while moving, there is a potentially unprotected section between its current position (red circle) and its NPH (gray circle). Without taking further care, R_1 (blue) would therefore start moving as well while both robots would ignore the yellow intersection between them.

for their timer to expire. Such robots can and will alter the subgraph which also forces frozen robots to go back to INTERSECTION. Note that this cannot cause a subset of the frozen robots (already in LOCK) to proceed to MOVE since they require to receive a `Graph` message with state FROZEN from *all* affected robots. In turn, this requires that a robot is not allowed to go back to INTERSECTION once it has already announced its FROZEN state. In other words, if a robot has reached the LOCK state, it may only be forced to transition back to INTERSECTION by means of another robot in its subgraph that it not in this state yet, effectively modifying the subgraph.

If a robot enters LOCK (INTENTER event), two different situations can occur: the robot may already be in motion (due to a previous negotiation) or it may be standing still. To ensure safety, the first case requires to define the henceforth-called *next possible halt* (NPH) along the path of the moving robot. It is defined as a position on the path, represented as a progress (cf. Equation (4.3)), with the following properties:

(i) Considering all known properties of the robot, the computed halt point (as described in Section 5.7) is the next possible position where the robot will come to a full stop along its current path. However, as a relaxation, robots can “artificially” return halt points that are farther afield to reduce the potential of actually coming to a full stop during a renegotiation.

(ii) An NPH can be the path's starting point (if the robot is about to move) or the final goal (e. g., if the robot cannot come to a halt earlier).

(iii) If there is already a halt point active, it will be kept active (and the robot must not continue to move over that point) and the progress of that point is used as the NPH. A new halt may never be before a halt (along the path) that was already issued.

The NPH is computed by the robot's motion controller autonomously and considered as the robot's current position in the next execution of the solver. This way, moving robots are perceived as if they are standing still at their NPH. Practically, a moving robot will continue to drive up to the NPH (if any) if the current negotiation has not been completed. However, if the negotiation completes, the robot can ignore the halt and may not even come to a (full stop) at its NPH. Theoretically, this creates the risk of collisions because robots are not actually at their halt points and still require time to reach such points. Figure 5.10 illustrates this problem. If solving is sufficiently fast, the robots are still in motion but the negotiated RoWs does not take into account the path segments right before the halt points (marked as "potentially unprotected" in Figure 5.10) which can cause collisions at such segments. In particular, Figure 5.10 shows such a situation at some time t_i where robot R_2 (red) is moving (completed 30 % of its path, indicated by the dotted path section at its beginning) and R_1 (blue) is standing still yet. Because both have detected the intersection (yellow), synchronized and solved, R_1 is about to start moving at $t_i + \epsilon$. Since R_2 was moving already, it issued a NPH (gray) which serves as its starting point for solving. However, the solver considers R_1 at its start (blue circle) and R_2 at its NPH (gray circle), thus, there is no intersection to be negotiated and both robots are allowed to move. This example illustrates that, without taking further care, intersections between a robot's current position and its NPH may be ignored because they are before a robot's starting position from the perspective of the solver.

To overcome this issue, different solutions are possible:

- Requesting the halt point must include waiting until the robot has actually reached that point, effectively delaying the FROZEN state. This is simple but not efficient since it prevents concurrency.
- The current negotiated solution (*right up to* the NPH) is processed according to the previous negotiation. New (non-moving) robots will never get the RoW in case of intersections with path segments right before the NPH. This is more complex but also increases the performance because waiting is avoided (or at least reduced to non-moving robots).
- *Artificial start halts* are added for every robot R_a having intersections to a robot R_b which in turn has activated a NPH. A start halt ensures that R_a must wait at its starting position (which might in turn be a NPH as well), until all robots R_b with a NPH have actually reached it. A start halt for R_a is therefore released when R_b reaches its NPH. This way, waiting for NPHs is moved to the execution of a negotiated solution but does not delay freezing or solving at all.

The latter approach can be considered as a compromise between performance and complexity, and was implemented for this thesis.

It must be noted that there are three different types of halts: the NPH is set by a robot *on its own* in order to ensure deterministic results when already in motion. It is removed

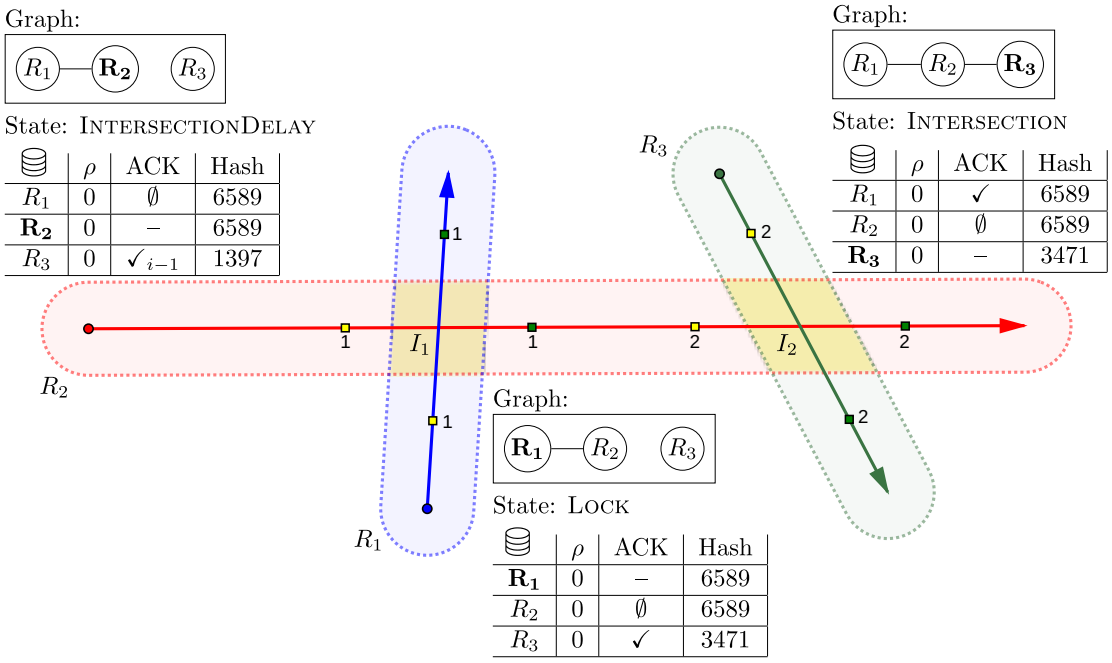


Figure 5.11: Example visualizing a situation at time t_i where a robot R_1 (blue) in the Lock state is forced to go back to Intersection because the affected robot R_2 (red) changes its subgraph due to a yet undetected intersection with R_3 (green). R_2 received an ACK from R_3 (which allowed it to become synchronized with R_1) based on R_3's previous ZLP at time t_{i-1} indicated by "$\checkmark_{i-1}$". Note that this visualizes the case right before making the transitions Lock → Intersection on R_1 and IntersectionDelay → Intersection on R_2 at time t_{i+1}.

when the solver completes. Start halts are somewhat similar although they are removed once corresponding NPHs are reached. The solver itself may compute halt points that are required for the current robot to respect the RoW of *others*, as already explained in Section 4.2.1. They are taken into account while the robot moves along its path. More detailed information are presented in the next section.

The most important event within this state is ExtIts, i. e., when `Graph`, `Hash` and `Negotiation` messages are received from another robot R_s. At first, for every received message, a robot R_r tests whether it is affected, that is, if R_s (`sender` attribute in all message types) is in the current frozen subgraph. If it is not affected, the message can be ignored w. r. t. Lock.

Let h_f be the currently frozen subgraph hash of R_r. If a relevant `Graph` message is received, the `hash` differs from h_f, the sender is neither marked as frozen nor *masked* w. r. t. h_f in its database entry (updated previously), then R_r unfreezes its local state, publishes Unfrozen and transitions back to Intersection. This is because R_s modified the subgraph and R_r is forced to follow. A database entry for a robot R_s is said to be *masked* w. r. t. a given hash if that robot has already sent the state Solved for that

given hash. It is an important property to keep track of because masked robots are not allowed to force other robots back to Intersection.

If a relevant `Negotiation` message is received, the `state` (cf. Listing 5.6) can either be Unfrozen, Frozen, or Solved. In case of state Unfrozen, R_s is in a renegotiation if and only if R_s is masked. In this case, the message in only stored in the (non-frozen part of the) database entry and the entire intersection graph is updated; the frozen state of R_s is not modified. Notice that the state of R_s is not unfrozen and, in particular, no state changes are performed yet because the negotiation of the current robot must be completed previously. Technically, in order to make this work, it hardly requires that messages arrive in the same order that they have been sent (cf. Section 5.4). Note that regardless of how many Unfrozen messages are received here (while frozen / in Lock), only the non-frozen database entry of R_s (and the intersection graph) is updated. Since the frozen state is not altered, the condition of when to transition to Solve remains verifiable.

In contrast, if the current robot did not receive a Solved state from R_s yet (i. e., it is not masked), we must assume that R_s changed its subgraph such that R_r also needs to transition back to Intersection. This operation is performed by all affected robots in the frozen subgraph, i. e., they will all end up being in Intersection again in order to become synchronous and frozen again. All these negotiation state changes are communicated via the `state` attribute of the `Negotiation` message (cf. Listing 5.6).

An example for this is shown in Figure 5.11 at time t_i. At time t_{i-1}, robots R_1 (blue) and R_2 (red) already have an intersection I_1 (yellow) as visualized while R_3 (green) has a ZLP without any intersection (not visualized). Because R_2 received an ACK from R_3 (indicated by "$\checkmark_{i-1}$" in its local database) w. r. t. its ZLP, both R_1 and R_2 were able to transition to IntersectionDelay, that is, both R_1 and R_2 got synchronized. Because the timer in IntersectionDelay already expired on R_1 here, R_1 already switched to Lock. However, R_3 published its new path and detected an intersection I_2 (yellow) with R_2 (as shown in Figure 5.11). Since the timer on R_2 has not expired yet, R_2 incorporates this information into the local graph (not shown anymore in Figure 5.11) causing it to transition to Intersection. This also includes publishing a `Graph` message containing the new subgraph hash. When R_1 receives that message, it detects that the synchronized and frozen subgraph with hash 6589 has changed from an affected robot, forcing it to go back to Intersection as well. Now all three robots will start to become synchronous again in order to negotiate the intersections in a group (the subgraph equals the entire intersection graph here which has just one connected component).

Most importantly, if the received state is Frozen, this indicates that the sending robot R_s locally got frozen so that the current robot will freeze R_s's state in its database as well. Internally, "freezing" means that a copy of all data of R_s, relevant to the solver, is created. This data is bound to the subgraph hash which means the if the subgraph changes, the hash changes as well which, in turn, renders the frozen data to be useless. For that reason, the hash received from R_s must be equal to the current robot's local subgraph hash. Additionally and according to Figure 5.6, it cannot happen that a robot

sends multiple FROZEN states one after another. Upon being informed about another affected robot that just became frozen, the current robot $R^\star$ must test if it can transition to SOLVE. This is true if all affected robots R are frozen and if their submitted subgraph hash $\text{hash}(R)$ equals the hash $\text{hash}(R^\star)$ of the local frozen subgraph $G_{R^\star}$:

$$\forall R \in G_{R^\star} : \text{frozen}(R) \wedge \text{hash}(R) = \text{hash}(R^\star). \tag{5.3}$$

For completeness, notice that if the received state is SOLVED and R_s was marked as frozen, the current robot R_r now knows that R_s has completed its solving. As already explained, R_s is marked as masked on R_r. However, since R_r is still in LOCK and has not even started its solver, the frozen state of R_s remains unchanged and only its non-frozen database entry is altered.

5.5.6 The Solve state

After a synchronized (INTERSECTIONDELAY) and frozen (LOCK) state has been reached, the current robot $R^\star$ enters the SOLVE state. This state represents the fact that all information is collected and synchronized to determine a RoW assignment at every (pairwise) intersection of the affected robots' paths. Within this state (upon INTENTER), the intersection graph is used to determine the RoWs locally, as detailed in Chapter 4. Solving itself may have two possible outcomes. First, the scenario may be feasible which requires to publish the SOLVED state (using a `Negotiation` message) and to preceed to the MOVE state in order to start the motion. Second, if the scenario is infeasible, the local state is unfrozen (discarded), the current goal is rejected and $R^\star$ transitions back to IDLE.

In order to be able to test if a renegotiation is required, the hash of the frozen (and just negotiated) subgraph is stored as a *negotiation ID*. It is set to zero, if a scenario is infeasible. Later on in the MOVE state, this ID is compared to the current subgraph hash. If it has changed, a renegotiation is necessary.

Solving has also been implemented centrally, i. e., instead of executing the solver on every robot, a central instance executes the solver algorithm once for a set of robots within a given scenario. This has a few advantages: it simplifies evaluation and testing, reduces the computational load on every robot, and it also reduces computational load of the simulation machine if everything is executed on a single machine. Solving centrally works as follows: if robots are configured to contact the central solver (server), they execute a remote procedure call (RPC) while providing the number of robots in the scenario and its hash. The robot with the lexicographically smallest name among all robots in the scenario–termed *input provider*–must also provide all paths, start progresses, radii, etc. as input to the server. This way, the input is transmitted only once. The server collects all requests and delays the response for the robots of a given scenario until (a) all robots have made the RPC and (b) all requests are referring to the same hash. This allows for verifying consistency among the requests. Solving starts right away in parallel when an

input provider has contacted the server for a given scenario. Clearly, multiple scenarios may be solved on the server concurrently (multi-threaded). Scenarios are referred to by the server using their unique hashes. If the central solver (server) has not been started or crashes (even while solving), robots fall back to solving locally without interrupting operation.

5.5.7 The Move state

This state represents the fact that the current robot $R^\star$ is allowed to move, possibly given a set of constraints (represented as halt points along its path). Additionally, other robots may receive new paths which can intersect with $R^\star$'s current path while moving. New intersections therefore need to be renegotiated which is handled in MOVE as well. When the goal is reached, $R^\star$ unconditionally transitions to IDLE, effectively becoming available again for new goals.

Upon entering this state (INTENTER), halt positions are updated according to the result of the solver which especially includes erasing the NPH, possibly issued during LOCK (cf. Section 5.5.5). Next, the motion veto flag is unset which triggers the motion controller (cf. Section 5.7) to actually start moving the robot's base through the environment. As already indicated in the explanation of the SOLVE state (cf. Section 5.5.6), the subgraph is checked for changes in order to immediately go to INTERSECTION in order to handle an accrued renegotiation. This also applies if a new intersection has been detected (events INTNEW and INTUPD). In such a case, the robot is moving although it is in the INTERSECTION state which can only happen due to renegotiations. However, note that this was "authorized" by entering MOVE previously to enforce collision-free motion.

When a robot receives an EXTPROGR event within the states ACK, ..., MOVE (cf. Table 5.1), it always uses the received progress value to test if intersections have disappeared and to remove associated halts along its current path. This is because any of these states may be active when the robot moves and, thus, may need to consider halts. Halts will also be removed if a robot with an associated halt leaves the system (EXTLEAVE event).

Finally, the reception of a `Graph` message within MOVE needs to be explained (EXTITS). Recall that the negotiation ID represents an unique identification of the previously negotiated intersection scenario (cf. Section 5.5.6). Basically, a renegotiation is required if the stored negotiation ID (see Section 5.5.6) is different to the current subgraph hash, the received message actually indicates a new intersection in the graph (`add` is set to true, see Listing 5.4) and $R^\star$ is really affected by the message, i. e., $R^\star$'s subgraph has changed due to the message. Apart from the fact that the negotiation ID is set to zero if the solver fails to find a solution (cf. Section 5.5.6), it is also set to zero if MOVE is entered from ACKDELAY because in such cases, there is no previous negotiation to consider (since there are no intersections at all). Once $R^\star$ enters MOVE (from SOLVE) again, the negotiation ID will be set to a new hash (during SOLVE) and the same procedure can happen again if necessary. In contrast, if intersections are gone, and $R^\star$ continues in

MOVE from ACKDELAY (see Figure 5.7), the ID is zero and the same procedure can happen again. Given this context, the MOVE state can be grasped as a "parent state".

Finally, Figure 5.12 shows a simplified sequence diagram that illustrates the entire process of negotiating a scenario with two conflicting robots. It visualizes the exchange of messages to become synchronized, the execution of the solvers and, eventually, the start of motions (the states INIT, IDLE and GOAL have been omitted for brevity). `Hash` messages have been omitted for readability and because the relevant hashes are exchanged here via `Graph` messages.

5.6 Intersection Graphs

The fundamental structure that is used to represent intersections in the system is the *intersection graph*; its definition is given in Section 5.6.1. During operation, the graph is updated continuously to represent the latest knowledge about intersections—this is explained in Section 5.6.2.

5.6.1 Overview and Definition

The intersection graph $G = (\mathcal{V}, \mathcal{E})$ serves as a representation for conflicts between robots in a system and is therefore time-dependent. Every robot manages its own local graph by incorporating received messages from the network. Thus, intersection graphs are a continueously updated data structure in the lifetime of a robot that is synchronized over time between all robots. Synchronization is done by sending and receiving `Graph` and `Path` messages which either contain an "add edges/vertices" or "remove edges/vertices" action. This way, the graphs on all robots will eventually converge to the same graph which will be explained in more detail in the next section.

Conceptually, each robot is represented as a vertex in the graph. Two vertices $u, v \in \mathcal{V}$ are connected by an undirected edge $\{u, v\} \in \mathcal{E}$ if $\chi(u, v)$ holds true, i. e., if u has at least one intersection with v (cf. Equation (4.6)). Figure 5.13(b) shows an example for an intersection graph with its associated scenario visualized in Figure 5.13(a). As it can be seen in Figure 5.13(b), the subset of vertices that are connected with a path to each other form a *connected component* $\mathcal{C}_i$ (gray) in the graph, also termed *subgraph*. Such subgraphs represent the groups of robots that require negotiation with each other and subdivide the entire set of robots into potentially smaller groups based on the current intersections. It is worth mentioning that transitive intersections are modeled as well because they can have an impact on the computed RoW.

An important property of the intersection graph is that all vertices are *unique* and there is a *lexicographical ordering* on the vertex labels. Given that ordering, the *root* of a subgraph (red bold vertices in Figure 5.13(b)) is defined as the vertex with the smallest label across all vertices in that subgraph.

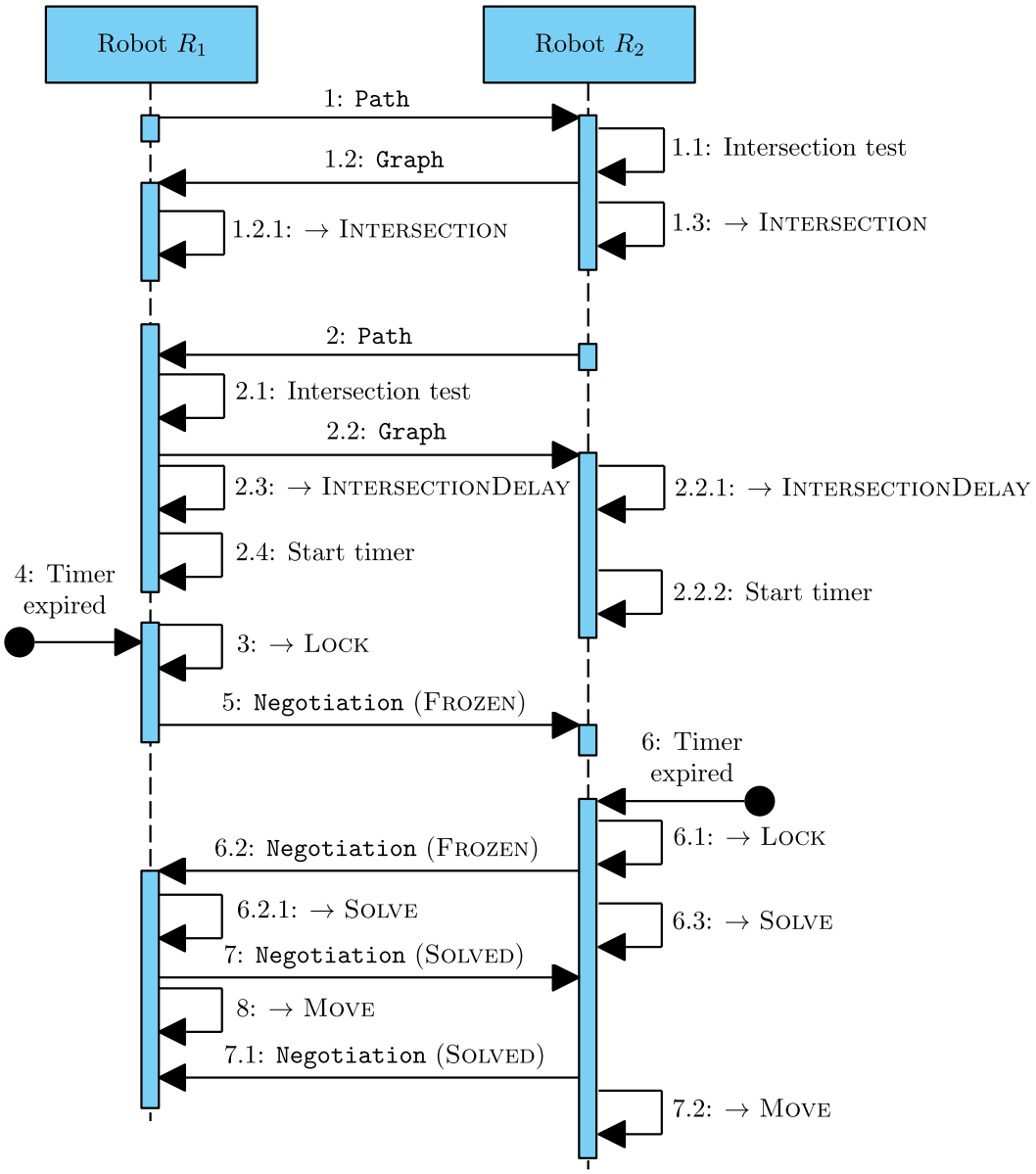
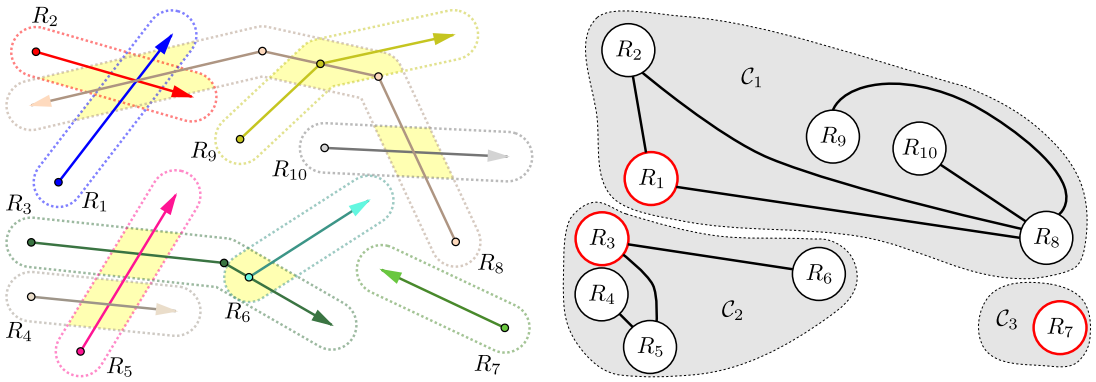


Figure 5.12: Example to illustrate the negotiation of intersections between two robots R_1 and R_2. After sending their paths via `Path` messages, they are tested for intersections and both robots react with publishing `Graph` messages. After transitioning from Goal to Intersection, both become synchronized causing the transition to IntersectionDelay which also starts a timer on each robot locally. After a timer expires, a robot switches to Lock individually, effectively freezing its state. This is communicated by a `Negotiation` message (with `state` set to Frozen). Because R_1 is already frozen locally, it immediately transitions to Solve and Move upon receiving the Frozen state from R_2. Note that based on timings, many variations of this diagram are possible.



(a) Input scenario with eight intersections (b) Intersection graph with three subgraphs

Figure 5.13: Exemplary input scenario in (a) with ten robots $R_1, ..., R_{10}$, eight intersections (yellow) and the associated intersection graph in (b) with three connected components $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ (aka subgraphs, gray). Vertices (black labeled circles) in (b) have been approximately placed at the beginning of each path for reference only. An undirected edge (black) indicates that the two connected vertices (robots) are in conflict, see Equation (4.6).

5.6.2 Synchronizing Graphs

We will now describe how an intersection graph is created and continuously updated on a robot by means of processing `Path` and `Graph` messages. Because the network it considered a spare resource, reducing the transmitted data is desirable. However, in order to ensure safety (collision-free motions), robots require global knowledge of each other. The proposed hashing of a robot's local subgraph—described in more detail in Section 5.6.2.2—is aimed at (a) reducing the required amount of data to be transmitted to a single (hash) value and (b) efficiently allowing to check if subgraphs are isomorphic (equality of hash values). If not stated otherwise, the described operations are independent of the state of the FSM (see Section 5.5).

Given a robot $R^\star \in \mathcal{S}_t$ and its current local subgraph $G_{R^\star} = (V_{R^\star}, E_{R^\star})$, the *synchronized state* contains

- a complete database entry for all $R \in \mathcal{S}_t$ (including radius, maximum velocity and the current path) and
- a robot's starting position (which might be the NPH if a robot was moving and asked to renegotiate).

Note that this is irrelevant if $R^\star$ has no intersections at all (since its subgraph just contains $R^\star$). The intersection graph is linked to the synchronized state by means of the `PathId` which is stored as the label in every vertex (Figure 5.13(b) just shows the robot's name for simplification). As already explained in Sections 5.5.4 and 5.5.5, once

a robot has collected all parts of the state, it becomes synchronous w. r. t. all robots $R \in V_{R^\star}$ once

1. hashes are equal, i. e., all subgraphs are equal to each others for all robots in the subgraph $G_{R^\star}$ and
2. ACKs have been received from all non-affected robots (those not being part of the subgraph $G_{R^\star}$).

The synchronized (and frozen) state is the input for the solver as described in Chapter 4.

5.6.2.1 Message Processing

Upon receiving a `Path` message (see Listing 5.1) from R_s (sender), the intersection graph is updated as well, depending on the message and graph content:

- If the graph already contains a vertex for R_s then the time stamp of the `PathId` is updated if it is larger than the stored time stamp. Additionally, all edges of the existing vertex (if any) are deleted because the updated vertex represents a new path of R_s so that old intersections (edges) are not valid anymore. The existing vertex is searched by the unique robot name only because its time stamp is already outdated. However, we require that the new time stamp is larger than the stored time stamp of the existing vertex. Otherwise, the message is discarded as outdated.
- If R_s is not yet known in the graph, a new unconnected vertex is created.
- If the message contains an empty path (i. e., R_s is about to leave the system), its vertex in the graph is deleted along with all associated edges.

Additionally, when a robot updates its own path (events IntNew and IntUpd, see Table 5.2), it also updates its own vertex with the new time stamp.

Similarly, upon receiving a `Graph` message (see Listing 5.4) from R_s, the graph update procedure is as follows:

- If R_s' intention is to add new intersections to the graph (`add` in the `Graph` message is true), a receiver first checks whether it is affected by the message based on the robot names only. That is, for all vertices v in the graph matching a stored name in the message fields `sender` or `affected`, the time stamp of v is updated to the matched one in the received messages if it is actually smaller. If it is larger and it matches the `sender`, the entire received message is considered outdated. However, if a time stamp of an element in the `affected` array is smaller than a matching vertex in the graph, only that element is marked as outdated. For all `PathId`'s in the received message that were not found in the graph, a new vertex is created.

Finally, for all such updated or created vertices, the edges $\{R_s, R\}$, for all R in `affected` are added if they have not been added already (e. g., in a previously received message of the same kind).

- If R_s' intention is to remove intersections from the graph (`add` equals false), all edges $\{R_s, R\}$ are deleted whereby R denotes all robots stored in the `affected` attribute of the `Graph` message. Note that the vertices are identified by fully matching the entire `PathId` (name and time stamp) to prevent deleting intersections that are still relevant. This is done by forcing R_s to link the information of what intersections (edges) to be deleted to the paths in conflict (`PathId`), or, in other words, edges must be specified as a tuple of `PathId`'s. Recap that an edge represents the existence of intersections between two robots and, thus, deleting an edge requires that *all* intersections have disappeared.

Additionally, when a robot detects intersections between its own *new* path and other known paths or if a received `Path` message is in conflict with a robot's current path, the graph is locally updated first in order to get the new hash of the resulting graph. The hash is then published to other robots in the system.

Aside from updating the graph based on `Path` and `Graph` messages, there are two additional cases where intersections (edges) need to be deleted locally first:

- Upon receiving a `Progress` message (see Listing 5.3) and while the robot is in Intersection or IntersectionDelay, issued halt points are deleted if the received progress values indicates clearance (i. e., the intersection was *released*). This allows a robot to actually pass a released intersection (where it previously did not get the RoW).
- When a robot moves on its path and the states Intersection, IntersectionDelay or Move are active, it periodically checks whether it went past a release point. If also *all* intersections to a particular robot have disappearched, a `Graph` message with `add` set to false is published to allow other robots updating their graph as well.

When a robot's state changes to Lock, it also freezes its current subgraph $G_{R^\star}$ because all robots $R \in G_{R^\star}$ form the input for the solver. The entire frozen state is unfrozen (discarded) 1) if a robot is forced to go back to Intersection, 2) if the solver considers the input infeasible, and 3) upon entering Move after solving.

5.6.2.2 Hashing

Instead of transmitting the entire intersection graph via network over and over again, just a hash of it is exchanged via `Hash` (and `Graph`) messages. The hash represents the structure including all vertex labels. If the graph changes, its new hash is published too

(cf. `hash` attribute in the `Hash` message in Listing 5.5). Robots store the lastly received hash for all other robots in their local database as already shown in Figure 5.11.

The hash is computed by iterating through a given subgraph $G = (\mathcal{V}, \mathcal{E})$ via deterministic DFS, starting at the root. This way, all vertices of the graph are visited in a deterministic order and their labels are concatenated to a large string and that string serves as the input of a hash function. In other words, this allows to check for isomorphisms between subgraphs in polynomial time due to the strict ordering of the vertices. In theory, the string representation could also be directly transmitted but hash values are much smaller and therefore reduce the network utilization. They also simplify the check for isomorphisms on the receiver's side because comparing two values is faster than comparing the full string representations.

The runtime complexity of computing a hash is $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}| \cdot D \cdot \log(D))$ whereby D is the maximum number of neighbors (vertex degree). Since such a graph can be fully connected (all robots have an intersection with all others), $D = \mathcal{O}(|\mathcal{V}|)$. The term $D \cdot \log(D)$ is justified by the need for sorting the children of a vertex inside the DFS traversal which itself requires $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$.

For two graphs G_1, G_2 with $G_1 \neq G_2$, the resulting string representations s_1, s_2 are always guaranteed to be different ($s_1 \neq s_2$). However, for two such strings s_1, s_2 with $s_1 \neq s_2$, the probability of $\text{hash}(s_1) = \text{hash}(s_2)$ approaches $\frac{1}{2^B - 1}$ whereby B is the number of bits of the resulting hash. Typically, $B = 64$ which results in a very low probability of $0.000\,000\,000\,000\,000\,005\,\%$ for a hash collision.

5.7 Motion Control and Right-of-Way

Recap from Section 5.5 that a robot is only allowed to move if its *motion veto flag* is not set. Virtually, this flag is unset when a robot transitions to Move from one of the following previous states (cf. Figure 5.7):

- If it is entered from AckDelay, there are no intersections and ACKs have been received from all robots already.
- If it is entered from Solve, it means that there are intersections which have been successfully negotiated in the previous solver execution.
- Entering it from Intersection means that a renegotiation was triggered (i. e., there are unnegotiated conflicts between moving robots with robots that have not yet moved) but during synchronization[3], all intersections have disappeared due to the motion executed in parallel.

[3]That is, while gathering the states of all robots in the subgraph.

The motion veto flag is set again when becoming idle. This procedure ensures that proper care was taken (acknowledgements and negotiation) before a motion takes effect.

The motion controller of the robot is responsible for considering and implementing computed halt points (if any) while the robot moves (cf. Figure 5.5). While doing so, it must ensure that halt points become active *before* the robot actually starts moving to not miss them and to respect the RoW. As already explained in Section 5.6.2.1, upon receiving `Progress` messages, halt points for released intersections are being removed such that a robot is eventually allowed to pass through them.

In terms of the ROS navigation stack [29], the motion controller is a local planner which is configured to precisely follow the path computed by the global planner if the motion veto flag is not set, i. e., it is responsible for managing the engines' velocities. Within a real setup, these velocities are consumed by, e. g., a Controller Area Network (CAN) bus. In simulation, the simulator moves the (simulated) robot based on the received velocities.

The implementation of CLPF is a local planner as well which is working on top of the motion controller and uses it to practically implement a solved input scenario by setting or removing halt points, allowing or preventing motions (veto flag) and by getting notified if a goal has been reached. In other words, the motion controller, as part of every robot, is owned and controlled by the herein proposed framework (see Figure 5.5). It is considered as an abstraction to interface with a robot's capability of moving its base.

5.8 Global Planning

Within this section, we describe the *global planning* approach employed throughout this thesis, see Figure 5.1. This also includes the concept of distributing the environmental map including (semi-) static obstacles using the developed *Vector Map Server (VMS)*.

The global planner is based on the work of M. Kallmann [30]. It uses a specific type of triangulated navigation meshes, the so-called *Local Clearance Triangulation* (LCT), to efficiently compute optimal paths of arbitrary clearance from a polygonal representation of the environment. The polygonal representation is given as a map which contains information about traversable and blocked areas. If globally optimal paths are desired, an extended search for the global optimum is possible while requiring significantly more computation time and only offering a small improvement on average.

The VMS is similar to the *map server* of the ROS navigation stack [29] aside from the fact that it uses vectorized instead of rasterized maps (bitmaps). It stores the map of the environment making it available to all global planners that are executed on a robot. In addition, it allows to dynamically add, update and remove obstacles in the current map which are then redistributed to all subscribed global planners. A vectorized representation of the environment was chosen because it allows for simpler updates

(including association to the updating entity) and floor plans of industrial buildings are most likely available as CAD-like drawings so that they can be converted more easily.

Within the methodology of CLPF, idle robots are modeled as semi-static obstacles as already briefly mentioned in Section 5.5.1. This has the benefit of automatically avoiding idle robots (thus, having a ZLP). Moreover, if a robot has some (mechanic) issues, the proposed methods do not depend on the broken robot's cooperation as it can simply be modeled as being idle at its current position. Besides, requiring idle robots to pull over would raise the challenging question of where to command them to, thus, would make the system more error-prone. The ZLP's contour is approximated by a sampled circle (polygon) and a robot, becoming idle, sends its ZLP to the VMS which redistributes the change to the global planners of all other robots. Similarly, when a robot is assigned a new goal and has computed its path through its global planner, it removes the contour from the VMS. Notice that updating the map (by means of adding and removing ZLP contours) and respecting such changes in the global planner of every robot is a race-condition, that is, it is possible that a global planner computes a new path to a given goal without yet knowing about a recent map update. It is therefore also handled in the FSM as a regular case: once a ZPI is detected with the ZLP of another robot, a robot (that was not allowed to move yet) returns to the Goal state to trigger a recomputation of its current path.

In a nutshell, the relation between the VMS and the global planner is as follows: every global planner is subscribed to the global environmental map, provided by the VMS. Robots can add and remove their ZLP contour to the map and such changes are then redistributed to all global planners. This way, global planners will eventually have the most recent map and automatically respect other idle robots, making the entire system more robust against faulty robots. Figure 5.14 shows an example for 15 robots in total from the perspective of robot p (located in the lower right) while all other robots (small black circles) are perceived as semi-static obstacles. All black polygons are provided by the VMS.

CLPF is not restricted to the employed approach [30]. The global planner is a configurable module of the software stack and just needs to adhere to the following assumptions:

- A path has no self-intersections according to the definition given in Equation (4.1).
- Path segments are piece-wise linear functions (i. e., line segments) whereby the start point of segment $i + 1$ is equal to the endpoint of segment i.
- A valid path consists of at least two support points.
- An environmental map is provided containing the (semi) static obstacles which must be considered during planning.

Note that CLPF allows for smoothing a path using the Douglas-Peucker algorithm [15]. If a global planner emits very densely sampled paths (e. g., for grid-based approaches),

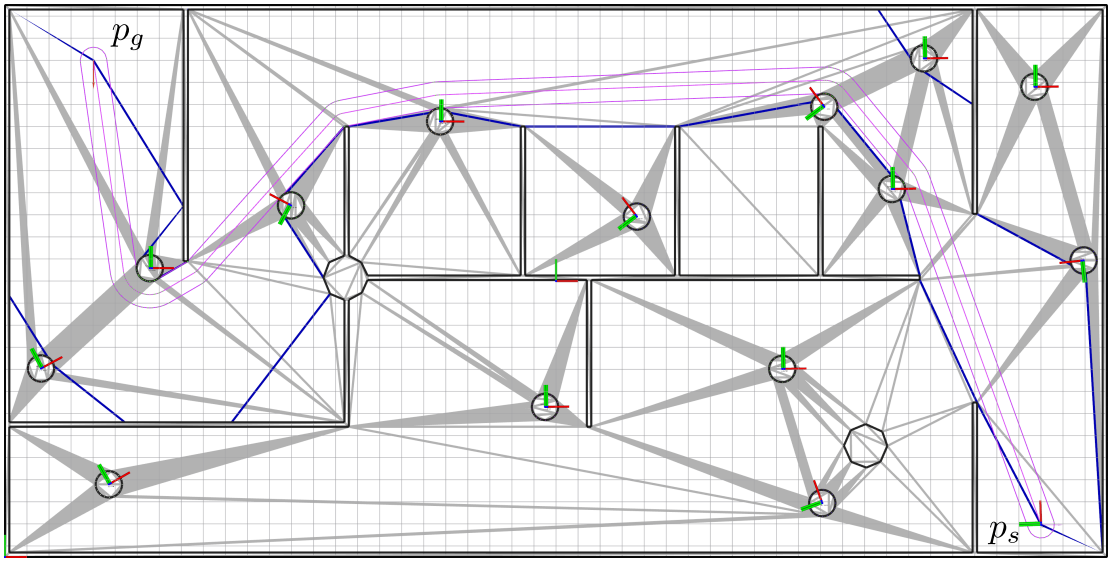


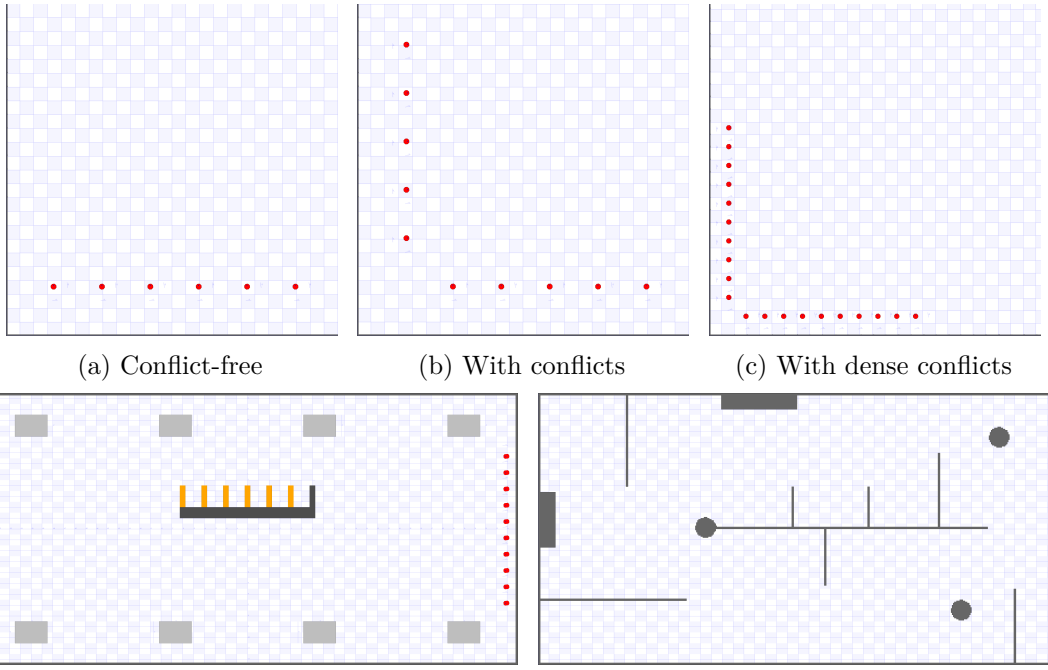
Figure 5.14: Example for the planning result of the employed global planner based on navigation meshes (gray lines showing the LCT). Black lines show borders and pilings of the building while white areas indicate free space. For illustration purposes, a planned robot path from p_s (lower right) to p_g (upper left) is shown along with 14 other robots standing still at their ZLP, effectively being semi-static obstacles (sampled circles with 36 corner points) for robot p. Dark blue lines in the mesh indicate the regions used by p's global planner for computing the resulting path (magenta). The inflated path polygon around the path based on the robot's size is shown in magenta as well. The front of every robot is visualized by the red axis starting at a robot's center point (e. g., p is facing upwards). The grid with a cell size of 1 m is shown for reference only.

the complexity of paths can be reduced by simplifying line segments represented by more than two support points using the parameter $\epsilon = 0$. However, note that massive smoothing ($\epsilon \gg 0$) is not advised as it changes the pre-planned paths too much and may cause collisions with (semi-) static obstacles. Finally, note that such a polygon simplification can especially be important if the SGS algorithm (see Section 4.3.2) is used.

5.9 Evaluation

It remains to evaluate the performance and throughput of the proposed framework (CLPF) with the two solvers presented in Chapter 4 and the global planner and VMS described in Section 5.8. All analyses are conducted using the *Stage* simulator [24] with the full software stack presented in the previous sections.

The setup and configuration for this evaluation is as follows. We choose ICSPS as the solver (see Section 4.4) with a limit of 500 iterations and lexicographically sorted



(a) Conflict-free (b) With conflicts (c) With dense conflicts

(d) Industrial hall with static obstacles (e) Narrow industrial hall with walls and pilings

Figure 5.15: Preview of different scenarios used for evaluation: (a) shows an example of six robots moving up and down in parallel without having any conflicts. (b) shows a setting with a total of ten robots whereby five of them (placed along the ordinate) move rightwards and leftwards, and the other five (placed along the abscissa) move up and down (like in (a)) creating a total of 25 complex conflicts (5-by-5). (c) shows a similar scenario except that robots are placed more closely to each other causing a higher probability of infeasible conflicts (as they pile up). (d) depicts an industrial hall-like environment with a few static known obstacles and 20 robots. (e) shows an equally sized environment that exhibits many narrow passages (walls, pillars) with up to 16 robots. The environments in (a), (b) and (c) have a total size of 200 m × 200 m (excerpt shown only) while (d) and (e) are 50 m by 24 m. A cell represents 1 m^2.

permutations (for increased reproducibility). OMRPS is not used at all because it can take too long for large scenarios which occur frequently, or it does not return a solution with a limited set of iterations. It has also already been extensively evaluated in Section 4.5.4. If not stated otherwise, the robots' type (differential drive) and size (approximated by a circle with a radius of 0.5 m) remains unchanged. Its maximum linear velocity is set to 1 m/s and its maximum linear acceleration is 2 m/s^2. The maximum angular velocity is 1 rad/s and a robot's maximum angular acceleration is 2 rad/s^2.

The radius of each robot was increased by a so-called *safety margin* d_s of 0.1 m. This value can be increased to compensate localization inaccuracies. However, to not affect the results by such issues, the simulator-provided (perfect) localization was used. This also reduces the load on the evaluating machine due to reduced computational load.

In addition to the safety margin parameter, there is a *conflict tolerance* parameter d_c, $d_c \ll d_s$ that allows to control the sensitivity regarding small intersections. More specifically, a conflict $\chi(p, q)$ between two robots p, q with associated paths $\mathcal{P}$, $\mathcal{Q}$ and radii r_p, r_q is ignored if

$$\text{dist}(\mathcal{P}, \mathcal{Q}) \geq r_p + r_q + 2d_s - d_c \tag{5.4}$$

holds. d_c is set to 0.05 m for all experiments to ignore very small conflicts. The desired speedup, also termed *real-time factor* (RTF), for the simulator is set to five, that is, the simulator tries to run five times faster than real-time (5 s of simulated time corresponds to 1 s of wall-time). However, if the simulator is not able to catch up with this rate because the simulated world is too complex, it is reduced automatically and can therefore vary over time. Using an RTF > 1 not only allows to execute longer simulations in shorter periods of times but also demonstrates the overall efficiency. More on this will be presented in Section 5.9.5.

Generally, there are multiple aspects that influence the performance of the presented framework which include but are not limited to the environment, the number of robots and the type of load (number and probability of conflicts, solvability and resulting complexity). For obvious reasons, it is virtually impossible to assess all combinations. A reasonable subset has therefore been selected for a deeper analysis.

Figure 5.15 shows examples of different scenarios used in this evaluation. They were chosen to have a wide variety of controlled conditions to evaluate different influencing factors. Having controlled conditions in which the experiments are being conducted allows one to reason more precisely about resulting observations. Figure 5.15(a) provides guaranteed conflict-free paths with a fixed length of 5 m. Robots simply move up and back down and this is repreated a configurable number of times. It serves to test the communication logic required to acknowledge conflict-free paths before robots are actually allowed to move. (b) arranges the robots in a vertical and horizontal lineup such that many feasible conflicts occur. (c) is very similar to the previous case except that robots are placed much closer to each other. This can cause goals to be reported as infeasible in the solver because conflicts can add up and robots only try to reach their current goal once. (d) contains various static obstacles which can cause paths to be more complex as well as conflicts to be more probable. It mimics an industrial hall where machines are placed in the environment as well. Lastly, (e) depicts a similar environment containing various blocking walls and pillars causing narrow passages. Such passages are expected to reduce the throughput considerably and will be used together with (d) to evaluate the environmental impact.

It remains to explain load generation. A simple way of creating new goals is random generation. This is used for the scenarios (d) and (e). A drawback is that while even considering the locations of robots and known static obstacles, many generated goals turn out to be unreachable, either because the global planner is not able to find a path or because the solver returns infeasible. In case of many conflicts and cluttered environments, this forces the framework to require a lot of communication, making it an ideal test for robustness of the communication protocol and to tackle worst-case inputs.

As already indicated, load generation for scenarios (a) to (c) uses procedurally generated (scripted) start and goal locations. Robots simply move back and forth between these locations. To also simulate some sort of batch processing where goals are dispensed in close succession, a synchronous mode was implemented. That is, all robots are given a goal and triggered to move simultaneously. Before assigning the next goal, all robots must have reached their previous goal. This makes the resulting scenarios very deterministic and similar and bundles the conflicts to very large instances.

Alternatively, goals may also be dispensed asynchronously, i. e., after a robot has processed its current goal, it immediately receives its next goal. After a short time of operation, a promiscuous mixture of robot poses, states and starting positions evolves. This has the advantage to test close-to-real behavior without entering uncontrolled random goal generation while still creating turmoil and many different inputs.

If a robot fails to reach its goals, it continues with the next one. In the scenarios (a) to (c), this results in simple back and forth motion. In the scenarios (d) and (e), a new random goal is generated and processed.

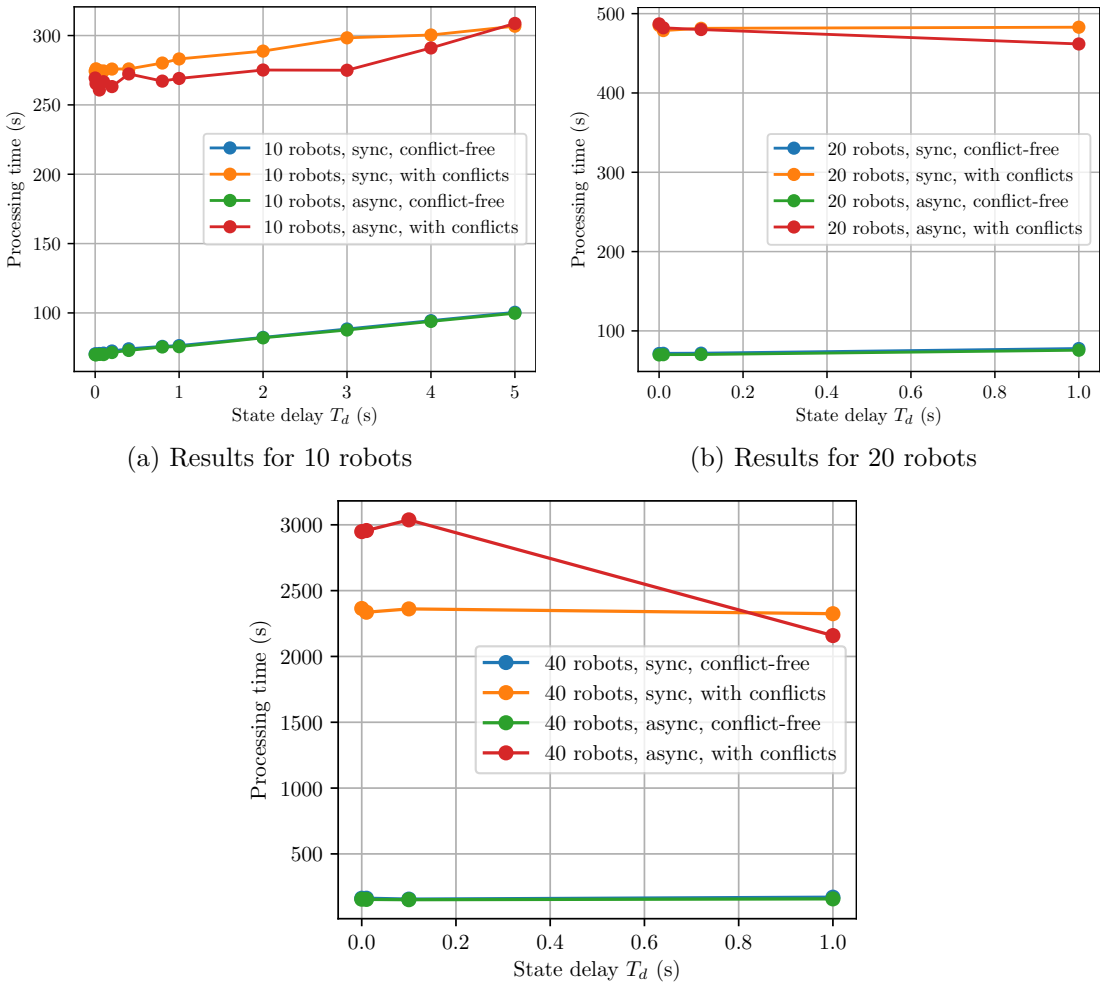
Finally, note that everything was solved centrally to reduce workload on the evaluating machine (see Section 5.5.6) and to increase the maximum limit of robots. It also simplifies verification and collections of statistics. The system executing the experiments was equipped with an AMD Ryzen 3900X (clocking up to about 4 GHz) with 12 cores (24 threads) and 64 GB of RAM.

The remainder of this section is organized as follows: Section 5.9.1 addresses the optimization of the state delay and NPH ahead distance parameters. Section 5.9.2 analyzes the runtime and reactivity including the amount of time spent in waiting for the release of halts. In Section 5.9.3, the throughput of the different scenarios and load generation modes is evaluated. This prepares for analyzing the scalability of the proposed framework w. r. t. varying numbers of robots in Section 5.9.4. Finally, Section 5.9.5 concludes the assessment with a brief review on safety aspects.

5.9.1 Optimization of Parameters

This section deals with the optimization of the parameters of CLPF. The presented results are averaged over at least three iterations to account for varying system load. Recall that the *state delay* parameter T_d represents the time to wait in the AckDelay and IntersectionDelay states of the main FSM (see Section 5.5). This is motivated to process goals sent to robots in close succession all together and to avoid renegotiations.

Figure 5.16 depicts the optimization results for different values (simulated seconds) of the state delay parameter (abscissa) and the resulting total processing time (ordinate) for a given scenario in seconds. Scenarios have been generated by varying the number of robots (10, 20 and 40), by selecting either the conflict-free (Figure 5.15(a)) or the conflicting environment (Figure 5.15(b)) and processing goals synchronously or asynchronously.



(a) Results for 10 robots

(b) Results for 20 robots

(c) Results for 40 robots

Figure 5.16: Optimization results for the state delay parameter T_d for varying numbers of robots within the conflict-free (Figure 5.15(a)) and conflicting scenarios (Figure 5.15(b)). For both scenarios, synchronous (blue, orange) and asynchronous goal assignment (green, red) have been analyzed. 100 goals were assigned to every robot.

This yields a total of 12 different setups (cf. legends in Figure 5.16) and a total of 100 goals has been assigned to each robot in every scenario. In contrast to (b) and (c), T_d has been analyzed for up to 5 s in (a). Generally, notice that the processing time for scenarios without any conflicts (blue and green) are much smaller and very close to each other compared to scenarios with conflicts (orange and red), regardless of whether goals are assigned synchronously or asynchronously. In fact, those scenarios turned out to be very similar because synchronization does not have much effect if no conflicts occur.

As it can be seen from Figure 5.16(a) for 10 robots, processing times increase for larger values of T_d. This would imply that the parameter adds no benefits and could be removed.

In fact, waiting is a trade-off because it will always add the waiting time to the total processing time measured here. However, if fewer renegotiations are needed afterwards, it might redeem the seconds waited in the first place. This being said, taking a closer look at Figure 5.16(b) already reveals that for a larger number of conflicting robots (20) and asynchronous goal processing, a delay of 1 s reduces the total processing time by 25.29 s compared to a delay of 0 s (red). For synchronous goal processing (orange), the time is reduced by 2.21 s. Not visible due to scaling, the processing time for the remaining 2 scenarios still increases slightly (blue: 6.29 s, green: 5.92 s).

A similar observation applies to Figure 5.16(c) with 40 robots: even though the processing time is still slightly increased for the conflict-free cases (blue: 8.09 s, green: 2.44 s), it is greatly reduced for the conflicting case with asynchronous goal processing (red) with a drop of 789.82 s from a zero to a one-second delay. Synchronous processing undergoes a decrease of 39.80 s (orange). This can be justified by the fact that smaller delays are sufficient because the goals are already assigned at the same time (synchronized). The asynchronous mode perturbs the processing, requiring more renegotiations and possible stops over time.

Another observation from random goal processing (not shown here) is that the total processing time is obviously dominated by travel time and therefore dependent on the number of successfully reached goals. Thus, depending on the robot density and environmental characteristics, even higher values of the state delay can yield lower total processing times (i. e., higher throughput) because many goals are being reached requiring time and dominating the overall processing time.

It can be concluded that rather small values ($\ll 1$ s) of T_d should be chosen for up to 15 robots and higher delays (≥ 1 s) are suitable for larger robot fleets. The current implementation does not yet allow to make this parameter dependent on the number of robots currently known in a robot's database but that is simple to implement. In addition, it would be advised to not use the parameter for the AckDelay state because all experiments have shown that this increases the total processing time. This makes sense since there is no need to wait for all ACKs anyway, thus, further delaying the beginning of a motion is unnecessary. The parameter T_d is currently used for both states AckDelay and IntersectionDelay.

Other criteria than the processing time could have been used as target value, too. For instance, the number of feasible goals would also be interesting to analyze. However, it is not a good fit for the proposed scenarios in Figure 5.16 because all inputs have been considered feasible. In addition, feasibility may be maximized while processing times increase which may not be desirable (shifting it to a more complex optimization problem with multiple criteria).

It remains to present the optimization results for the ahead distance values of the NPH parameter. Recall that the ahead distance parameter D specifies the desired distance for the selection of the NPH on a robot's remaining path. When a moving robot is asked to renegotiate with non-moving robots, the moving robot selects its NPH to be D meters ahead of its current position unless there is another halt in between which then becomes

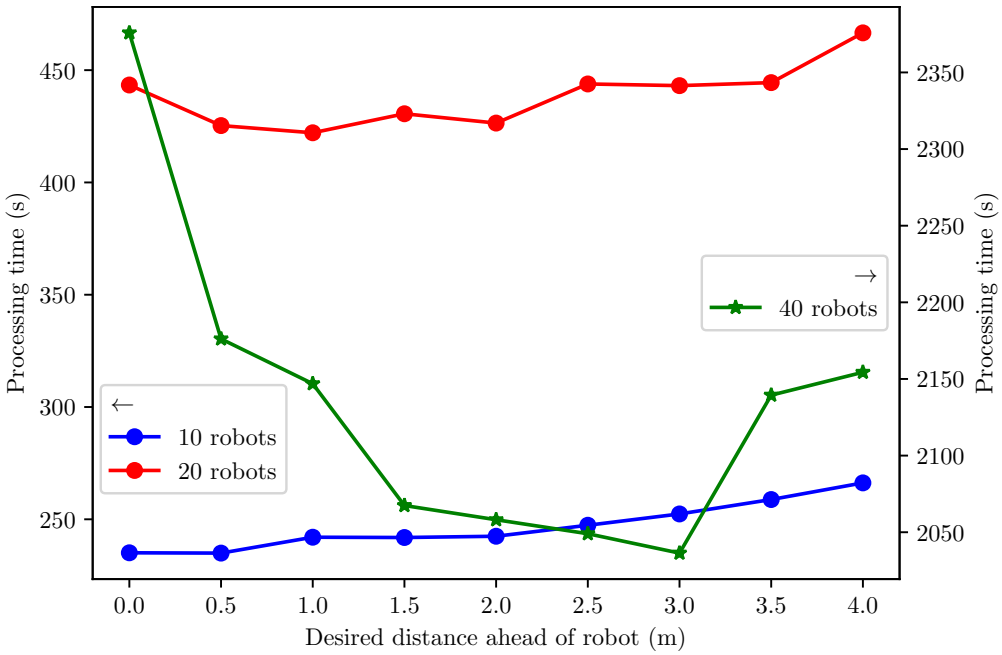


Figure 5.17: Optimization results for the NPH ahead distance parameter for a fixed input of 100 goals per robot. Arrows in the legends indicate the associated ordinate a plot belongs to.

the NPH (see Section 5.5.5). If the selected position would be on or past the goal, the NPH equals the goal (effectively causing no NPH to be set). Practically, this means that the moving robot will move to its goal without stopping. A renegotiation will take place nonetheless but the moving robot will be considered to be already at its goal position in the solver (diminishing the solution space).

The effects of varying the ahead distance values for the NPH are visualized in Figure 5.17 for the conflicting environment (Figure 5.15(b)), asynchronous goal assignment and 10 (blue), 20 (red) and 40 robots (green) respectively. Distance values D have been sampled from 0 to 4 m with steps of 0.5 m (abscissa) and the resulting processing times are plotted on the ordinate (seconds). Because the processing with 40 robots required considerably more time to complete, it is shown with reference to a dedicated ordinate on the right (as indicated by the arrows in the legends). Note that the use of conflict-free scenarios makes no sense because no NPH are used/emitted at all. The scenario with dense conflicts was omitted because similar results were expected. Synchronizing goal assignments is also not a good choice because robots would never be in motion (requiring to set an NPH) while asked to renegotiate.

For only ten robots (blue), the processing time continues to increase slightly when D is increased. From $D = 0$ to 2 m, the processing time increases by 7.37 s (3.14 %). For 20 robots, a decreased processing time of 16.95 s (3.82 %) can be observed when increasing D from 0 to 2 m. As expected, for a very large robot fleet with 40 robots (green), there

is a sweet spot at 3 m where the processing time is minimized. On the one hand, smaller values require much more communication and renegotiations such that distances ≤ 3 increase the processing time. On the other hand, larger distances than $D = 3$ allow moving robots to continue their motion (up to the given parameter D) without any stop. However, this forces non-moving robots to wait until moving robots have reached those locations on their paths, increasing the overall processing time.

The scenario has also been analyzed with an ahead distance set to ∞, i. e., not activating a NPH at all. That means that moving robots will always move to their current goals before non-moving robots are allowed to execute a (re-)negotiated scenario. In other words, moving robots are prioritized because non-moving robots will set an implicit start halt as explained in Section 5.5.5. For all three evaluated robot counts, the total processing time got worse with $D = \infty$ which underpins the previous explanation regarding the increased processing times for $D > 3$ of the green plot (40 robots) in Figure 5.17.

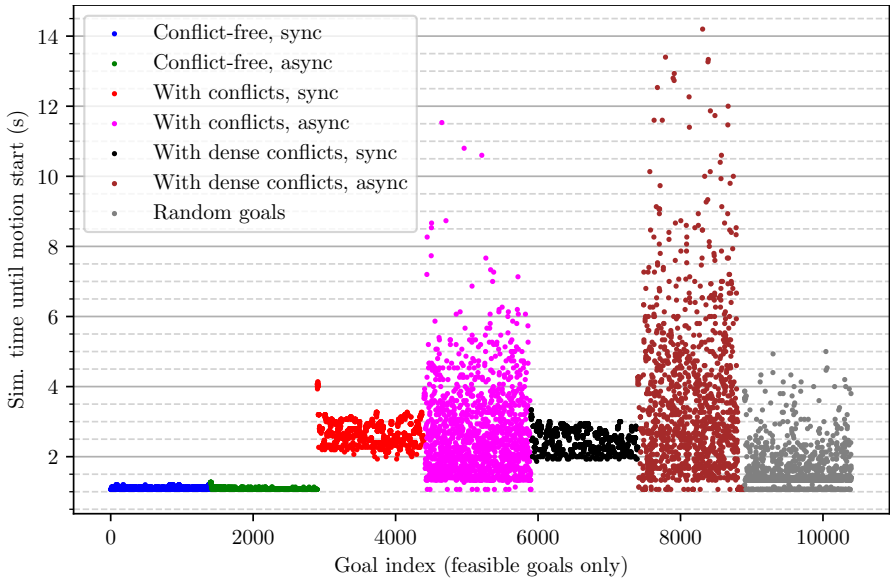
We conclude to set $T_d = 1$ s and the NPH ahead distance to 2 m for the remainder of this evaluation, both as a compromise between smaller and larger robot fleets. It should be noted that a major advantage of the presented framework is its tiny set of parameters, making it easier applicable in practice.

5.9.2 Runtime and Reactivity

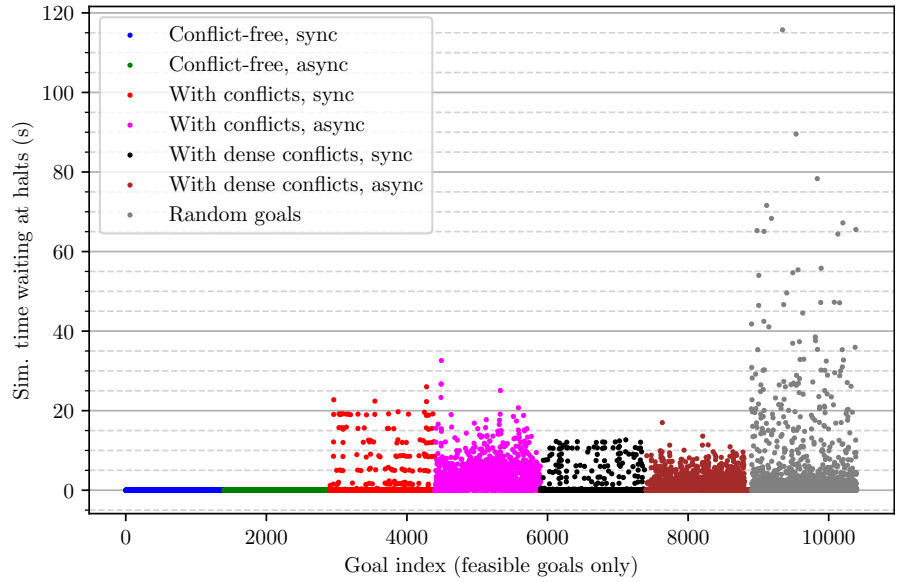
This section continues with the analysis of the runtime and reactivity of various parts of the framework. It should be noted that it is difficult to measure the "total runtime" because the framework processes goals when they are assigned to a robot and the resulting runtime depends on various different aspects like robot parameters (size, velocity, etc.), environmental characteristics, robot density, conflict probability, etc. This analysis therefore focuses on specific parts of the framework, namely the time it takes until a motion was started (start delay), the total time robots spend waiting for the release of halts, the average duration robots spend in various states of their FSM and the average number of messages sent and received during the processing of a goal.

Figure 5.18 visualizes the reactivity by means of (a) the start delay and (b) the total time a robot spent waiting for the release of halts while processing a goal. More specifically, a data point in both plots represent a single reached goal processed by one of the 15 robots. Time was measured in simulated seconds, i. e., wall-time was about 5x slower (smaller) than the shown duration due to a RTF of about five. Because the start delay also contains the state delay T_d measured in simulated seconds (i. e., depending on the simulator's speedup), it is reasonable to also measure simulated time here.

A "processed goal" is defined from leaving the IDLE state until entering it again, either because it was considered infeasible by the solver or because it was reached. The *start delay* is defined as the time a robot requires from starting the goal processing (leaving IDLE) until reaching MOVE for the first time. Note that MOVE may be entered multiple



(a) Time until motion started for every reached goal



(b) Time spent waiting in halts for every reached goal

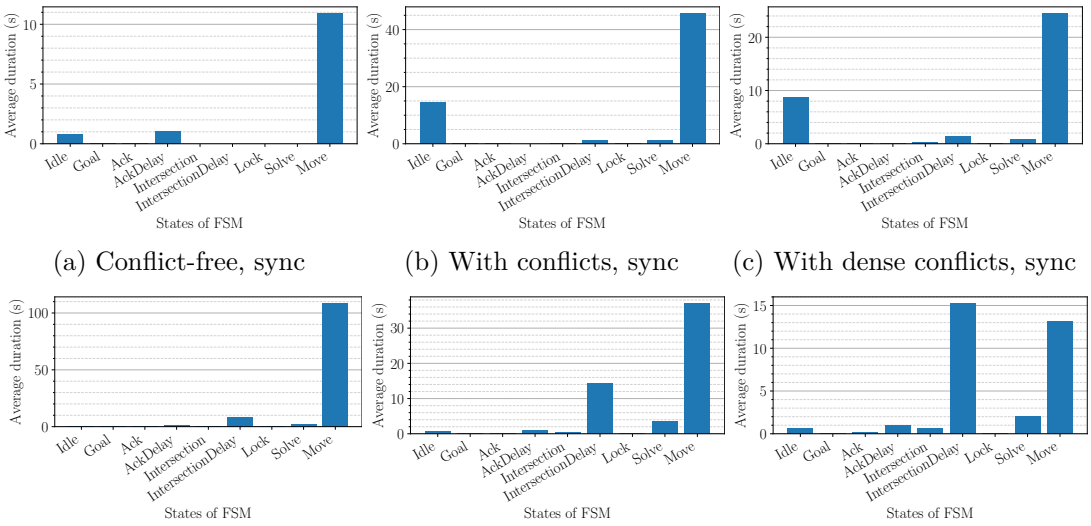
Figure 5.18: Reactivity analysis of CLPF w. r. t. (a) the start delay and (b) waiting time for halts for a total of 100 goals per robot and seven different scenarios. Every point in the diagrams represents a measurement on one of the involved robots for processing a single goal.

times if renegotiations are necessary and multiple halts may be added over time. The waiting time in Figure 5.18(b) is defined as the sum of all durations between coming to a full stop at a halt until continuing the motion afterwards. In both diagrams, the abscissa shows the somehow arbitrary index of the goals how they have been stored, grouped according to the different scenarios. The scenarios have already been presented in Figure 5.15. Random goals (gray) have been assigned to the 15 robots in the 200 m × 200 m environment (also partly shown in Figure 5.15(a)).

For the two conflict-free cases (1st in blue and 2nd in green) on the left side of Figure 5.18(a), the start delays are all within 1 and 1.5 s. This is justified by $T_d = 1\,\mathrm{s}$ (see Section 5.9.1) including some minor communication time for requesting and processing ACKs. The two conflicting scenarios with synchronous goal processing (3rd in red and 5th in black) are also very similar to each other with a low spread in start delays due to the synchronization. All conflicts are already negotiated right at the beginning when every robot gets its goal causing no disturbances. Because some robots will always detect conflicts a little later than others (effectively restarting the state delay), this will add up causing higher start delays compared to the conflict-free scenarios. Note that the number of robots was always 15 for all scenarios to make the results comparable. The start delay within AckDelay for the conflict-free scenarios can always start independently on all robots and there is no need for restarting it.

Taking a closer look at the conflicting scenarios with asynchronous goal processing (4th in magenta and 6th in brown) reveals the much higher spread compared to the previously discussed synchronous case. This becomes apparently clear in the dense conflicting case (brown) where start delays can be even higher than 14 s. This is justified by the disturbances caused by asynchronous goal processing requiring many renegotiations, although the majority of delays remains below 6 s. The random goal case (gray, rightmost dataset) shows a combination of the behavior of the conflict-free and conflicting scenarios because given the large environment of 40 000 m^2 where just 15 robots move to random positions, a few of them have no conflicts at all while others have a conflict here and there. All delays cannot be zero because the current implementation always imposes the state delay (cf. Section 5.9.1). This again reveals some further tuning potentials of CLPF.

Figure 5.18(b) shows the total sum of all waiting times in halts. For conflict-free scenarios (blue and green), waiting times are all always zero because halts are not necessary. Interestingly, comparing the synchronous (red) with the asynchronous case (magenta) of the (non-dense) conflicting scenarios, shows that asynchronous goal processing exhibit more gradually distributed waiting times while synchronous goal processing exhibits a rather fixed set of waiting times roughly sampled between 0 and 20 s. The latter is caused by synchronization forcing robots to move in "waves" and halting at similar locations over multiple runs. The disturbances in asynchronous processing distribute the waiting times more smoothly. The scenario with random goal processing (gray) exhibit the largest spread in waiting times which seems reasonable given that many different situations can occur due to randomness. It should be noted that the results for the



(a) Conflict-free, sync (b) With conflicts, sync (c) With dense conflicts, sync

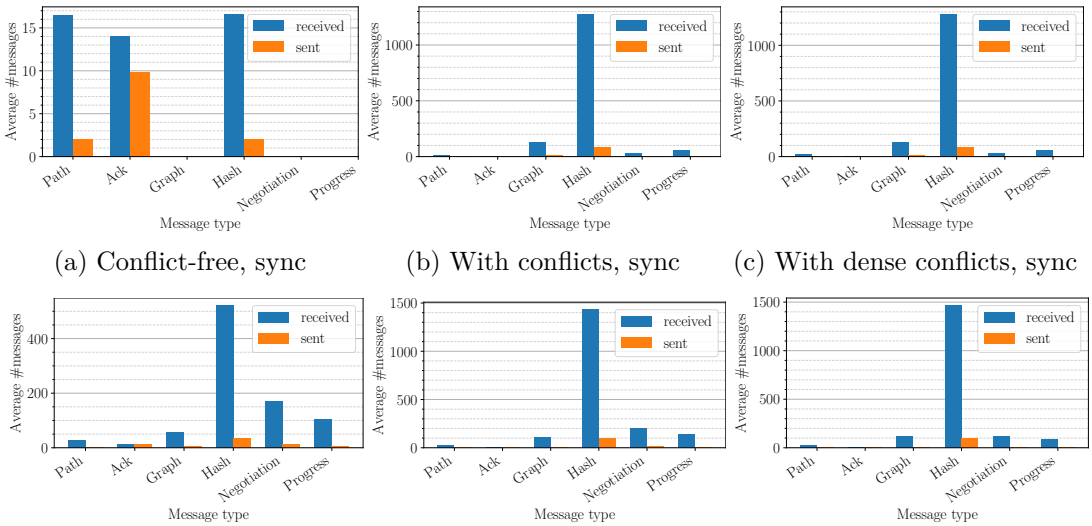
(d) With random goals (e) With conflicts, async (f) With dense conflicts, async

Figure 5.19: Average durations of states in the FSM while processing 100 goals per robot in different scenarios, load generation modes, and a total of 15 robots.

conflicting cases are all based on scenarios with not only a conflict probability of (or at least very close to) one but also a high number of conflicts. That is, the expected waiting time at halts is massively influenced by the characteristics of the scenario (conflict probability, density, etc.). The higher the number of conflicts, the more likely it is for robots to halt. In that sense, the analyzed cases are rather worst-case scenarios.

We continue with the analysis of how long a robot stays in the states of the FSM according to Figure 5.19. The states are given on the abscissa and the average duration in (simulated) seconds are shown on the ordinate. All synchronous goal processing based results ((a) to (c)) exhibit a much higher duration for IDLE because synchronization forces robots to wait until the entire set has reached its current goal. Meanwhile, they all wait in IDLE. The MOVE state is also the dominant state in nearly all scenarios which is desirable because robots should move most of the time. An exception is Figure 5.19(f) where INTERSECTIONDELAY dominates. This is justified by the huge amount of conflicts and resulting (re-)negotiations due to the dense positioning of the robots. This is also visible by means of the highest peak at INTERSECTION among all known results. However, it also underpins an observation already made: if the subgraph changes on a robot because another robot triggers a yet unknown conflict within the state delay of INTERSECTIONDELAY, it re-enters INTERSECTIONDELAY after incorporating that conflict, effectively restarting the timer for T_d. This leaves room for improvement.

In the conflict-free scenario (a), the ACKDELAY peaks at about $T_d = 1\,\mathrm{s}$ and all other states apart from IDLE and MOVE do not matter at all. The corresponding asynchronous case is not shown here as it looks about the same. The ACK related states also do not play an important role in all other scenarios considered because conflict management



(a) Conflict-free, sync (b) With conflicts, sync (c) With dense conflicts, sync

(d) With random goals (e) With conflicts, async (f) With dense conflicts, async

Figure 5.20: Average number of messages received (blue) and sent (orange) while processing 100 scripted goals per robot in different scenarios, load generation modes, and a total of 15 robots.

dominates. The SOLVE state also includes the time for actually executing the solver (see Chapter 4) and is higher in scenarios with more conflicts and/or disturbances.

The random goal case in (d) is very different to all other scenarios because the average duration in MOVE dominates all other states considerably. This is justified by the longer paths through the very large environment—98.77 m on average, e. g., compared to 35.04 m in (e) (2nd longest case).

Figure 5.20 visualizes the average number (ordinate) of messages received (blue) and sent (orange) during goal processing. For every processed goal, the number of messages was counted and averaged over all assigned goals and for every message type (abscissa). The presented diagrams also give an intuition of the required amount of communication based on the absolute number of messages exchanged (given that within all scenarios, the same total number of goals where assigned to every robot). For instance, in the conflict-free case (a) only Path, ACK and Hash messages where exchanged. The number of received messages is always higher than the number of sent messages because, except for ACKs, a robot receives messages of all other robots. In contrast, (e) and (f) peaking at nearly 1 500 received messages on average due to many conflicts and difficult conditions for negotiations due to disturbances caused by asynchronous processing. A possible optimization would be the waiver of publishing Hash messages if a robot has no conflicts at all to save bandwidth.

A similarity among all diagrams is the high number of Hash messages. It is justified by the fact that every modification in a local intersection graph must be communicated with an updated hash. Fortunately, Hash messages are also intentionally the smallest

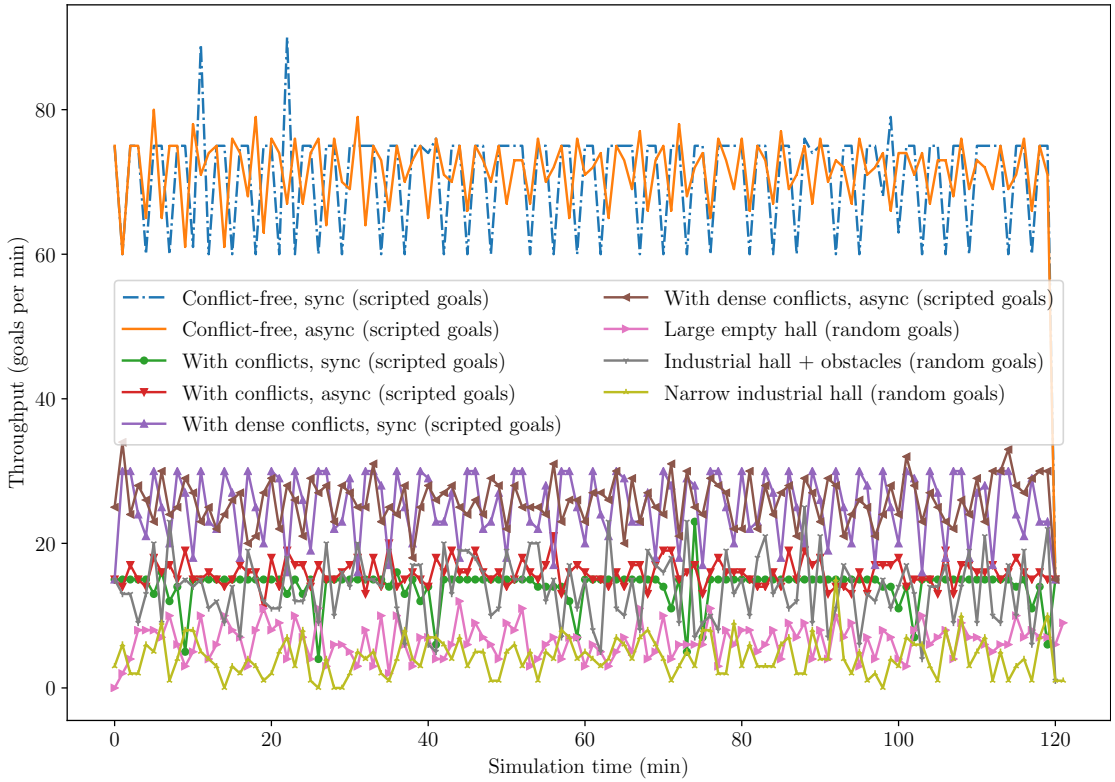


Figure 5.21: Throughput analysis for a fixed number of 15 robots and nine different scenarios over a total of 2 h of simulation time (abscissa). The ordinate shows the number of successfully reached goals per minute.

messages among all types. The number of Progress messages could be kept relatively low which offers potentials for partially releasing paths (not implemented yet).

The results indicate that CLPF is running as expected, apart from minor tuning potentials w. r. t., for example, the state delay handling.

5.9.3 Throughput

We are now going to analyze the throughput, that is, the number of goals successfully reached per (simulation) time with a fixed robot count of 15 and different scenarios. The timeframe for computing the throughput was set to one minute. Within the next section, the average throughput (across an entire pass) will be analyzed regarding different numbers of robots (scalability).

The analysis was conducted for nine different scenarios and is given in Figure 5.21. Compatible with the previous insights of the evaluated scenarios, the conflict-free cases exhibit the highest throughput because no conflicts need to be negotiated and, thus, no halts need to be considered. There are only marginally differences between

synchronized and non-synchronized goals with an average throughput of 70.89 goals/min vs. 71.59 goals/min. It is expected that the asynchronous mode is a bit more efficient due to the missing time for synchronization. The fluctuations in all cases are justified by the way the experiments have been conducted, i. e., assigning (possibly synchronized) goals, waiting for negotiation and solver execution, and executing the computed schedules while waiting for the release of halts (if any). By considering longer time slots than 1 min, the resulting plots would be smoother.

Surprisingly, the synchronized conflicting case (dotted) shows a relatively constant throughput of 14.21 goals/min. The synchronization combined with the existence of conflicts seems to cause robots not to reach their goals simultaneously as contrasted to the conflict-free case which smoothens the throughput over time. The asynchronous case fluctuates around it with a marginally higher throughput of 15.0 goals/min on average. A similar observation applies to the dense conflicting case. One might wonder why the dense case, exhibiting the same number of conflicts but with more densely placed and moving robots, has a higher throughput. This is because the paths are shorter (14.47 m vs. 35.04 m)—an effect of placing them more closely to each other.

A similar observation applies to the large empty hall with random goal assignment (blue) and an average path length of 104.18 m. The throughput is reduced due to the long travel times. The two industrial hall-like scenarios exhibit shorter path lengths (17.25 m and 26.55 m respectively) but suffer from many conflicts due to static obstacles and narrow passages. The narrow one even has the lowest throughput among all cases.

As already indicated in the previous analyses, we can conclude that the results are heavily dependent on the context (environment, conflict probability, etc.). If the underlying conditions allow, the proposed framework provides constant throughput rates without any congestion of message processing or other kinds of delays.

5.9.4 Scalability

This section evaluates the scalability of the proposed framework regarding different robot counts. Figure 5.22 shows the throughput (ordinate) for different numbers of robots from 1 to 20 (abscissa) in nine different scenarios for about 2 h of experiment time, that is, a time limit of 2 h was used while the number of goals was not limited. After expiry, goal dispatch has been stopped and the experiment continued to run until emitted goals have been processed completely. Within the given time limit, the average throughput (in goals per hour) has been computed and plotted w. r. t. every robot count. Only reached goals were considered.

Because the throughput was considerably higher for the conflict-free cases (blue and orange), a dedicated ordinate on the right of Figure 5.22 has been used. The diagram shows a clear linear correlation between the number of robots and the throughput for both synchronized (blue) and non-synchronized (orange) goal processing. Being on par with previous observations, synchronization introduces a small drop in throughput which

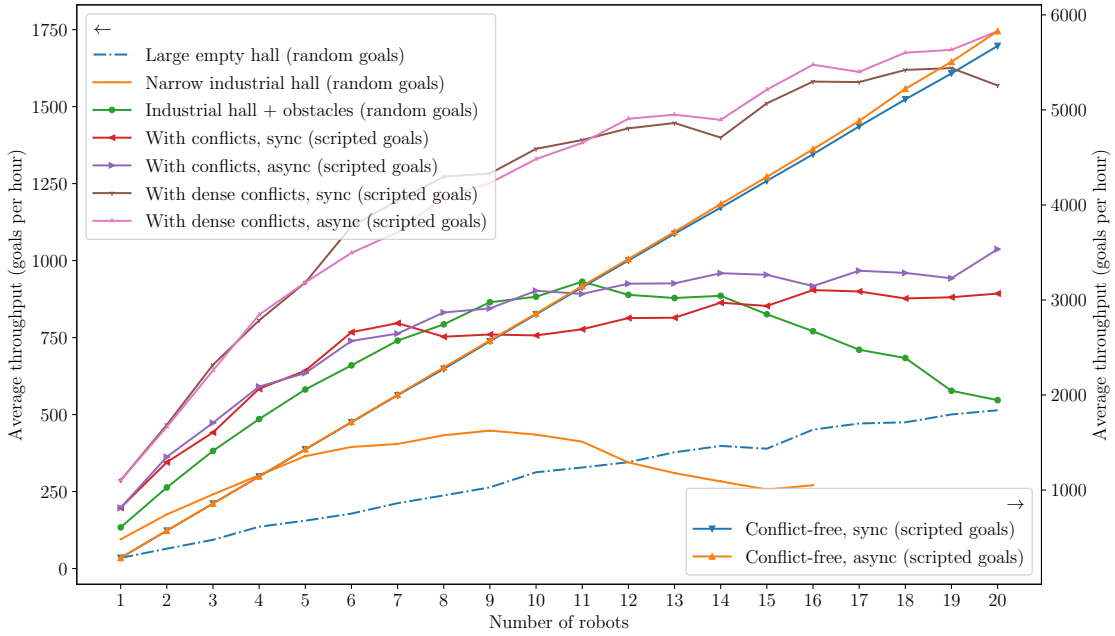


Figure 5.22: Throughput regarding varying numbers of robots (scalability) for nine different scenarios. Note that two plots for the conflict-free cases (blue and orange) have their own associated ordinate on the right due to a different scaling. The total runtime for every experiment was 2 h.

becomes especially visible when larger sets of robots (≥ 14) are used. This underpins the framework's efficiency for the non-conflicting case.

However, in practical situations, conflicts are normally present in the inputs. The remaining seven scenarios shown in the legend in the upper left of Figure 5.22 (all referring to the left ordinate) therefore deal with conflicting cases. Generally, it turns out that all observations are again highly dependent on the input characteristics. The scenario with the 200 m-by-200 m hall without any static obstacles (blue) has sufficient space for all 20 robots evaluated here such that its throughput continuously increases. That is, CLPF is able to handle the required communication and (re-)negotiations to keep up with the increasing robot count. However, as already stated previously, the paths for this input are relatively long (101.76 m on average) which explains that this scenario has a much slower ascending curve. It can therefore be seen as a combination of the conflict-free cases mixed with some conflicts "here and there".

The throughput of the industrial hall scenario with static obstacles (green, preview of scenario shown in Figure 5.15(d)) starts with a steep increase up to 11 robots and then continues to decrease (with a throughput at 20 robots equaling the throughput of five robots). This indicates that the scenario has a sweet-spot at approximately 11 robots and adding more robots causes congestion. This is expected behavior even though the limit is very specific to the characteristic of a scenario.

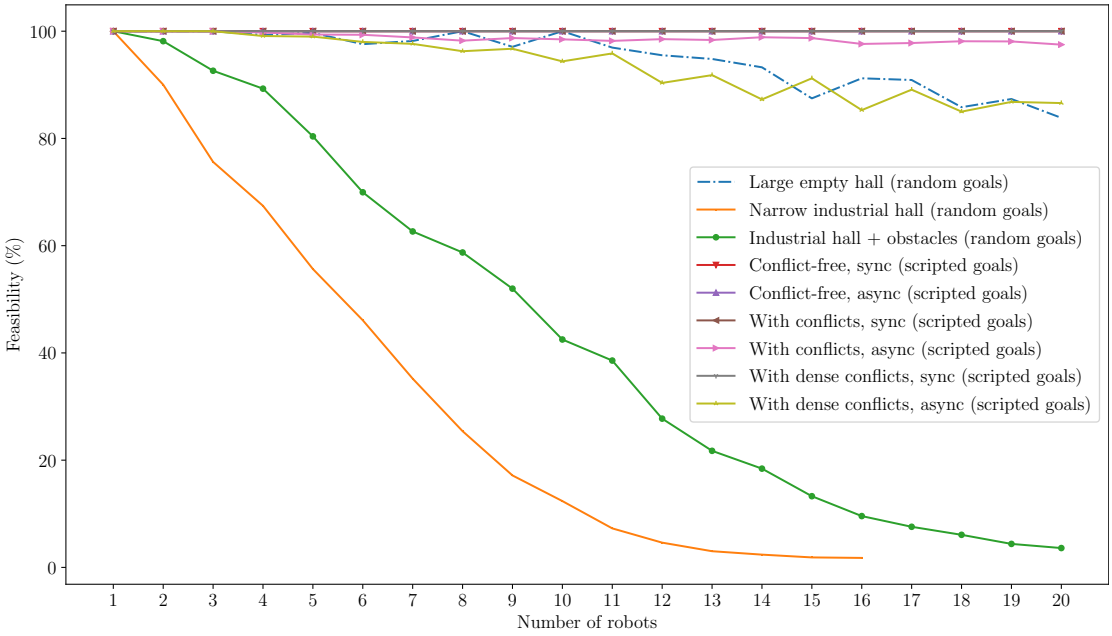


Figure 5.23: Feasibility regarding varying numbers of robots (scalability) for nine different scenarios. Also refer to Figure 5.22 for associated resulting average throughputs. The total runtime for every experiment was 2 h.

Comparing this with the narrow industrial hall scenario (orange, preview shown in Figure 5.15(e)) reveals a similar observation at already nine robots. It is justified by the narrow passages making it difficult for many robots to find a feasible way. Due to the small space and similar observations, evaluation has been capped at 16 robots for this case.

The remaining four curves are referring to the conflicting cases (see previews in Figure 5.15(b) and (c)). As already explained in Section 5.9.3, the higher throughput of the dense conflicting cases (compared to the non-dense cases) is justified by the shorter path (an effect of placing robots more closely to each other). For up to five robots, there is only a marginal difference between the synchronized and non-synchronized mode. Starting at 8 and 14 robots respectively, the asynchronous goal processing allows a slightly increased throughput. Fortunately, CLPF is able to benefit from increasing robot counts because the achieved throughputs increase when the number of robots is incremented.

To allow a closer look, Figure 5.23 visualizes the feasibility rates for the analysis shown in Figure 5.22, i. e., for all evaluated robot counts, the percentage of successfully reached goals is shown (ordinate). As expected, the non-conflicting cases always have a feasibility of 100 %. The narrow industrial hall scenario exhibits the highest and steepest decrease in feasibility, approaching 0 % after about 14 robots already. Note than all real scenarios use random goal generation to create input load and, thus, numerous goals are reported to be infeasible. This explains the steep decline (green) within the industrial hall scenario

containing static obstacles. Notice that random goal processing adds a lot of stress on the communication logic (see Section 5.4) because a high infeasibility rate forces robots to continuously start the processing and negotiation of new paths. It can therefore just be a flip-flopping in terms of communication: receiving a new goal, computing a new path, requesting ACKs, detecting conflicts, initiating negotiations, solving, detecting infeasibility, becoming idle again, requesting the next goal, and so on. This might even result in robots standing still or just moving gradually from time to time (in case of very congested and dense inputs). Nonetheless, it verifies the robustness of the presented communication concepts and also reveals inefficiencies in the underlying scenarios.

An interesting comparison is the large empty hall scenario (blue) against the dense conflicting case (async, green). Figure 5.23 illustrates that their feasibility rate is very similar while Figure 5.22 shows a much higher throughput for the dense conflicting case. Even though the dense conflicting case exhibits many challenging conflicts, especially for up to 20 robots (cf. Figure 5.15(c)), CLPF and its employed ICSPS are capable of utilizing the increasing number of robots by scaling up the resulting throughput. In other words, CLPF is able to perform well even in highly conflicting scenarios.

Note that a total of 746 139 goals has been processed for Figure 5.22 and 5.23 whereby 450 032 were feasible (60.31 %) and 296 107 were infeasible (39.69 %).

Finally, Figure 5.24 exemplarily depicts the distribution of scenario sizes, i. e., how many robots are typically part of an input to be forwarded to the solver in a negotiation. Recall that robots are organized in a dynamically updated data structure called the intersection graph (see Section 5.6). Negotiations only take place within a subgraph every robot is part of. The distribution in Figure 5.24 therefore shows how CLPF is able to reduce the set of robots to relevent subsets only while still retaining safety. In this particular example, 25 robots (blue) have processed a total 2 566 scenarios and 50 robots (orange) have processed 1 161 scenarios in the large empty hall scenario.

Fortunately, many scenarios exhibit much fewer robot counts than the total number of robots in the scenario. This increases the efficiency of CLPF because inputs for the solver become much smaller. However, and as already explained, this also highly depends on the input.

5.9.5 Safety

Safety refers to the property of ensuring collision-free motions which is fundamental for industrial automation. It is therefore worth noting that robots have never ever crashed into each other in all approx. 850 000 goals processed and more than 280 000 scenarios solved in total for this evaluation (excluding those processed for the evaluation of the solvers in Chapter 4).

However, the implementation requires that robots stop at their designated halt positions. In turn, this requires that processing loops are executed sufficiently fast to not miss halts or goal points while moving. When scaling up the number of robots or the (simulated)

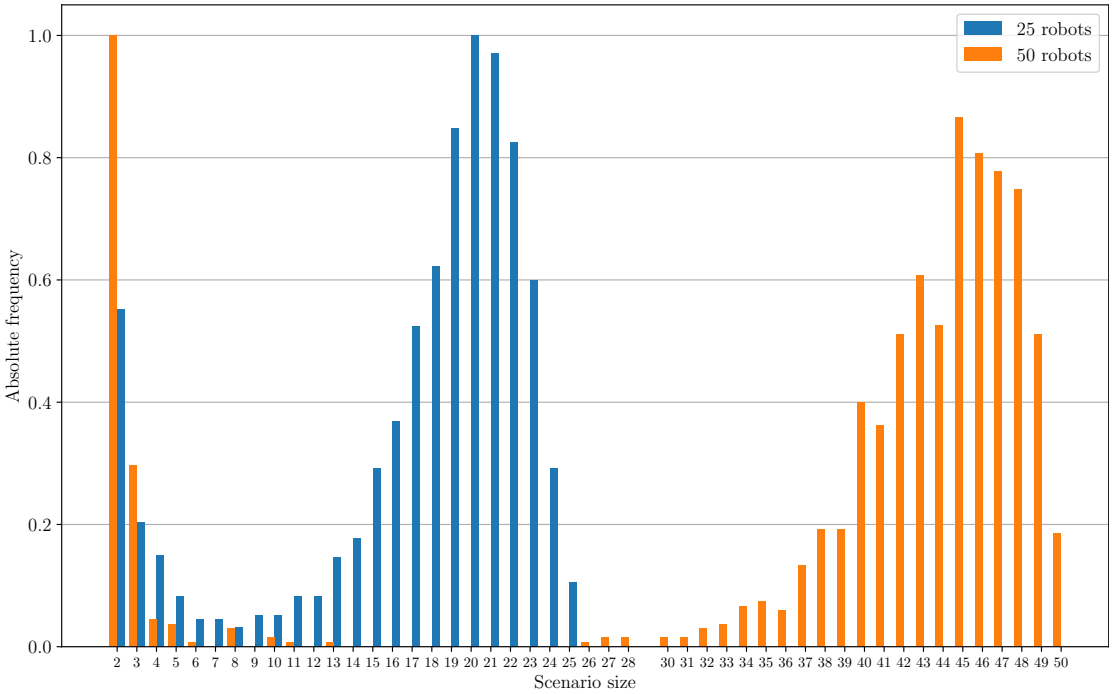


Figure 5.24: Distribution of scenario sizes in an empty squared environment (200 m × 200 m) with random goal assignment. 25 robots processed 2 566 scenarios while 50 robots processed 1 161 scenarios in total. All robots have been placed one after another along the x-axis initially.

environmental complexity, system load increases which could theoretically increase the probability for collisions due to halts being missed. It is important to note that this equals the behavior of real robots because their computing hardware and sensor data processing algorithms must also be designed in a way to deterministically handle all events in due time. This also has the consequence that the presented simulations cannot simply be executed on server CPUs (even with numerous cores) because they most typically lack higher boost frequencies, causing execution performances of single nodes to be worse compared to higher clocking desktop CPUs.

The upper limits, e. g., on the number of robots in the conducted experiments and the RTF, have therefore been experimentally approached given the employed simulator and test system.

Due to lacking support for dynamically adding or removing robots in the used *Stage* simulator, this feature has been partially tested by "disabling" a robot and "killing" it after some time, making it invisible to others that also employ CLPF because the killed robot gets removed from the other robots' databases. Also restarting that killed robot afterwards worked as indented.

Chapter 6

Case Study: Decentralized Assembly

This chapter deals with a case study of a decentralized assembly system that includes but is not limited to the production of automobiles similar to Figure 6.1 (repeated here according to Figure 1.1(c) for improved readability). It consists of a fully simulated system of decentrally operating robots that allow a customer to order some (possibly very individualized) product such that this product is then assembled by the system. Conceptually, the presented concepts are also transferable to real robot hardware. Notably, it uses the entire software stack of a mobile robot, especially what is presented in Chapter 4 and 5, and therefore represents a fully functional system. Also note that the proposed concepts may also be universally applied to other use cases, especially those presented in Figure 1.1.

Because every entity in the system is represented as an *agent*, Section 6.1 introduces the developed agent-based modeling. Given the decentralized and decoupled nature of the system, Section 6.2 elaborates on how fault tolerance is achieved and implemented. Section 6.3 explains the representation of product line-ups and the management of dependencies of assembly steps. After briefly demonstrating some visualization and introspection possibilities in Section 6.4, the entire system is evaluated from a high-level perspective in Section 6.5.

6.1 Agent-Based Modeling

The assembly system is modeled as a composition of decentrally and autonomously acting agents based on [7]. Figure 6.2 depicts the inheritance diagram with all relevant

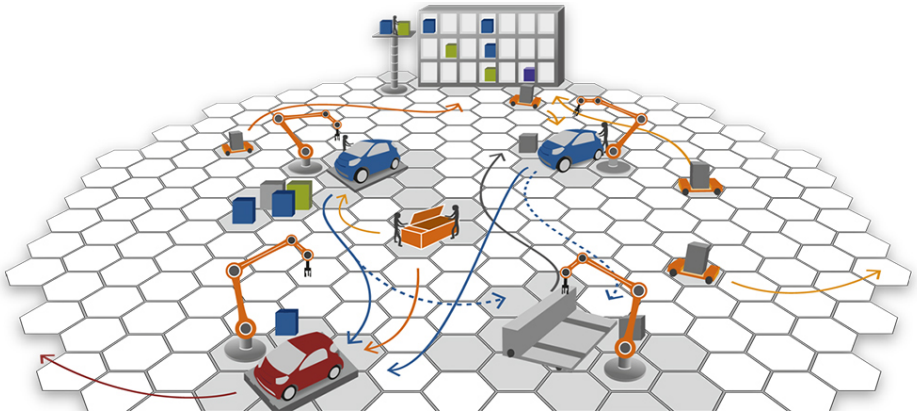


Figure 6.1: Sketch of the use case (already shown in Figure 1.1(c)) realized as a case study in this chapter: the drawing shows a fully decentralized assembly system also termed "Smart Factory" (from the SMART FACE project [25]). There are four workstations equipped with industrial robot arms and a warehouse (buffering parts and materials for assembly) in the background. Arrows indicate possible material flow. For simplicity, robot arms (manipulators) are not modeled in this case study.

```
1 uint8 type # Unique type of this agent (e.g. WORKSTATION or PARKING_LOT)
2 uint8 state # State (INIT, IDLE, BUSY, FAIL) of this agent
3 time stamp # Time stamp of this heartbeat
4 string name # Unique name of the sender
```

Listing 6.1: Compactified definition of the `Heartbeat` message in ROS; it is used to detect the agent's exitence or death and to get to know its current state and type.

classes representing various entities in the system. `AbstractAgent` is the base class for all agents and provides a periodically sent heartbeat whose content is given in Listing 6.1. It serves three main purposes. First, once an agent receives the heartbeat messages of another agent, it knows about its existence, effectively allowing for decoupled agent detection. Second, if heartbeats fail to appear within a configurable timeout, an agent is assumed to be dead. Since the agent's internal processing logic is coupled with sending the heartbeats, not only a crashed agent is detected this way but also agents being stuck in some computation can be detected. Third, the heartbeat also contains the current state (cf. Listing 6.1), so agents are aware of others being currently idle. Detected agents are stored in an agent-local database along with their current state managed by the `AbstractAgent` class (providing convenient access to it).

The `SmartAgent` class extends `AbstractAgent` by adding services for requesting an agent-specific status and for triggering fail states. For instance, the latter can be used to trigger maintenance in an agent which then invokes the `onMaintenance()` function. An agent can then implement maintenance behavior and even autonomously prolong the maintenance if necessary. Other agents will be able to detect the end of maintenance by a `state` transition (see Listing 6.1) from FAIL to IDLE. Additionally, `SmartAgent`

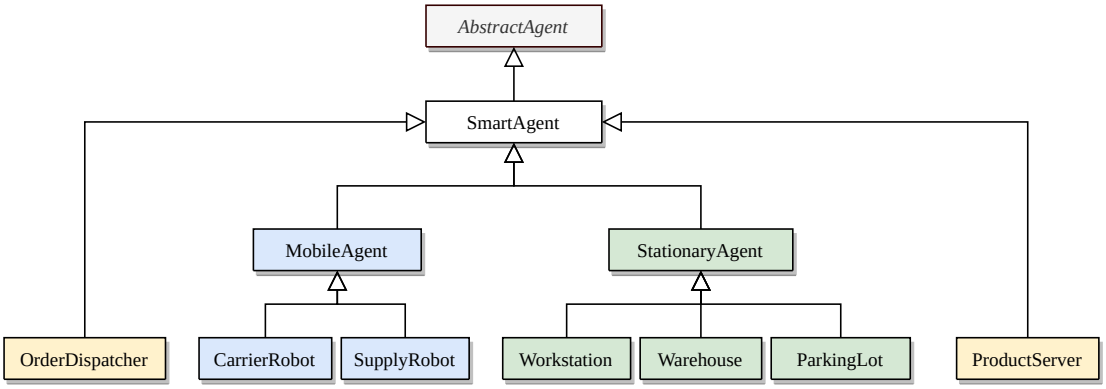


Figure 6.2: Inheritance diagram showing all classes of the agent-based modeling; `AbstractAgent` is the pure-virtual base class for all agents which defines an anonymous agent. A `SmartAgent` adds services for requesting status or maintenance as well as requires non-anonymity. `StationaryAgent`s and `MobileAgent`s both have a spatial extend but only `MobileAgent`s allow for changing their pose in the environment (motion). The `OrderDispatcher` and the `ProductServer` have no environmental representation; they remain virtual but can be detected by other agents to interact with them. The lowest row of classes is somehow specific to this case study.

requires derived classes to decide on a specific agent `type` (cf. Listing 6.1) because they are always detectable by other agents and send heartbeats themselves. In contrast, anonymous agents (having no own type specified) will never send heartbeats but are able to receive other heartbeats allowing them to be part of the system without actively participating (listener). The fail state triggering also allows for injecting failures in an agent which is useful for simulation and fault tolerance assessment. It will be used in Section 6.5.

The `StationaryAgent` and the `MobileAgent` both have a spatial extend and representation in the environment. However, only `MobileAgent`s are capable of changing (and reporting) their pose *dynamically*; they represent mobile robots. Targets in the environment are modeled as so-called *logical goals*. They are justified by the need for moving to logical targets like a workstation that may require a reservation before starting to move and a release after completely leaving them. Logical goals introduce the following features: in addition to specifying the pose (and an optional name), a robot should move to, a logical goal keeps track of state changes and invokes a callback function upon detecting any changes. That is, apart from error conditions, a logical goal may be in one of the states listed in Table 6.1.

If the motion to the pose (stored along with the logical goal) fails, a new attempt is automatically initiated until a configurable attempt limit is reached. Additionally, the percental progress of reaching a goal is reported. An important feature is the release callback: when a mobile robot has completely left the goal area of a logical goal (state LEFT, see Table 6.1), its release callback is invoked to allow releasing the logical target

Table 6.1: Overview of all possible states a logical goal may be in while being processed and tracked by a `MobileAgent`. States are always passed through from top to bottom. The state MOVING may be omitted if the robot has already ENTERED the goal area before actually starting to move.

State	Semantic
MOVING	The robot is moving to the goal but has not entered the goal area yet. This is the default initial state after issuing the logical goal.
ENTERED	The robot has entered the goal area while partially blocking it. This is a one-shot flag only: it is set once entered and immediately overwritten by APPROACHING after invoking the state-changed callback.
APPROACHING	After entering, the robot continues to move to the final pose while blocking it.
REACHED	The robot has reached the goal completely; it cannot have an error anymore. The transition to LEAVING is triggered once the next goal is issued.
LEAVING	After reaching, the robot now moves away to some other goal while still blocking it.
LEFT	The robot has finally left the goal area, effectively not blocking it anymore. This is the terminal state.

associated with the goal. On a higher level, this prevents multiple robots to try to move to the same goal. `MobileAgent`s therefore allow moving to such logical goals by means of a convenient API. In the background, they automatically track the release and occupation while handling possible corner cases (like the abortion of a motion or the partial occupation of a goal area). The `OrderDispatcher` and the `ProductServer` classes are concrete virtual agents directly derived from `SmartAgent`. `CarrierRobot` and `SupplyRobot` are two concrete `MobileAgent`s and `Workstation`, `Warehouse` and `ParkingLot` are concrete `StationaryAgent`s.

As already mentioned in the introduction of this chapter, the agent-based modeling may also be applied to the use cases from Chapter 1. The details of all already mentioned concrete classes in the lower part of Figure 6.2 will now be discussed along with Figure 6.3 showing the order processing flow and interaction between the concrete agents in the assembly system of this case study. Note that some entities (e. g. Order Load Generator) have been omitted in Figure 6.2 for brevity.

A *workload* for the system can be created by customers directly by ordering products via a *Webshop* and the assembly process can be monitored directly via the Webshop. Alternatively and especially useful for evaluation, workload can also be generated by the *Order Load Generator*. Both entities are derived from `AbstractAgent` allowing

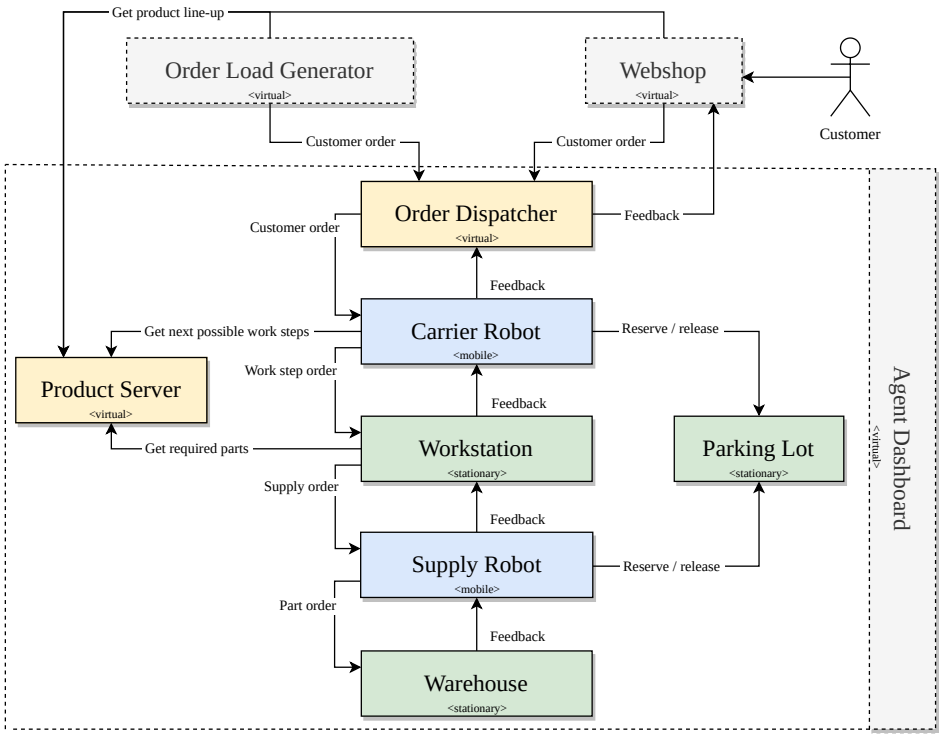


Figure 6.3: Block diagram showing the order processing flow and interaction between all virtual (gray and yellow), stationary (green) and mobile agents (blue) in the system. Virtual agents shaded in grey are invisible (undetectable) for other agents whereas yellow shaded agents do send heartbeats making them detectable for others. Unlike virtual agents, stationary and mobile agents have a topological manifestation in the environment (a pose and a footprint).

them to listen to heartbeats of *Order Dispatcher* agents. They collect the workload, represented as so-called *customer orders*, and try to dispatch them to *Carrier Robots*. They are called carriers because they piggyback the ordered products to be produced. After a customer order has been dispatched to a specific Carrier Robot, it provides feedback about the state of assembly to the issuing Order Dispatcher which, in turn, provides (condensed) feedback to the customer (if any). Once a Carrier Robot receives a new customer order to be produced, it requests the possible set of work steps for the product from the *Product Server*. More details of the Product Server will be presented in Section 6.3. Given the set of possible work steps, the Carrier Robot tries to find an appropriate *Workstation* agent for a given work step. Recall that this can simply be done by filtering the agent-local database for idle workstations. Once successful, it reserves the *carrier dock* of the selected Workstation. In turn, the Workstation will request the required parts for the reserved work step from the Product Server in order to emit *supply orders* to idle *Supply Robots*. An instructed Supply Robot will try to dispatch a so-called *part order* (containing the part needed at the Workstation) to a *Warehouse* to

transport the part to the *supply dock* of the requesting Workstation. More specifically, this is done by first reserving a dock at an appropriate Warehouse, trigger the part order after arrival at the dock to start the (simulated) loading, and then transport the loaded part to the Workstation. Once the Carrier Robot arrives at the Workstation, it emits a *work step order* to the Workstation to announce being ready for assembly. When also all required parts have arrived at the Workstation, the (simulated) assembly begins. As indicated, all agents in this chain send feedback to allow agents in the "hierarchy" above them to provide (possibly condensed) feedback themselves. This allows for a detailed status reporting although all agents act independently. This procedure iterates until all work steps have been completed. If there is no capable or idle agent available yet, an agent waits until it is (via incoming heartbeats). All types of order can be identified by a 128-bit *Universally Unique Identifier* (UUID).

It remains to describe the parking logic of a `MobileAgent` and how it is used by Carrier and Supply Robots in conjunction with the *Parking Lot* agent. `MobileAgent`s can be parked either manually (by calling `park()`) or automatically when becoming idle. This has the advantage of not blocking important locations in the environment. Parking involves reserving and then moving to an available *parking dock* at the Parking Lot agent. If parking upon becoming idle is activated, a `MobileAgent` tries to park until successful, i. e., if reaching the reserved parking dock fails for a configurable number of attempts, another (or random) parking dock is requested which is then tried to be reached. Some more details are presented in Section 6.2. This iterates until the robot has parked successfully.

Carrier Robots will try to find the next Workstation while still standing at the carrier dock of the current reserved Workstation for a configurable number of attempts. After that, they issue a park request to continue searching for an appropriate Workstation from there. Additionally, they will also park automatically when becoming idle, that is, once they completed the assembly of a product. Similarly, Supply Robots will also issue a park request upon becoming idle to not block the supply dock of a Workstation. Finally, note that many variations are possible because the described behavior can be influenced by parameters.

6.2 Fault Tolerance

Generally, if an agent has an issue, its state changes to Fail (see Listing 6.1) until recovered which forces others to ignore that faulty agent. All kinds of orders are monitored through a so-called *order client* provided by the developed *orderlib* (similar to ROS' *actionlib* but based on services instead of topics for reliability). Within a direct interaction of agents, errors will therefore be signaled directly via the order client. For instance, if a Carrier Robot arrives at a Workstation, it triggers a work step order which should finally be marked as completed by the Workstation (once the work step has been completed successfully). Upon errors, the work step is marked as aborted such that the Carrier Robot knows to re-trigger it at another Workstation.

For the sake of fault tolerance, the system puts some measures in place to avoid aborting a customer order which is considered last resort. It should only happen if the Carrier Robot itself has an issue. This is then detected by the issuing Order Dispatcher which can act accordingly (e. g., re-dispatching it to another Carrier Robot). The underlying concept for fault tolerance is quite simple and eventually based on the assumption of having redundant entities. If the motion to a logical goal fails, a limited (configurable) number of retries is attempted to reach the same goal again. Retries are delayed via *truncated exponential backoff* [54] because congested scenarios and narrow passages in the environment have been found to be quite similar to network related issues. If all attempts fail, another agent of the same type is selected as the logical target until one becomes available. These concepts especially aim for making the logical goal processing less prone to issues in the underlying path planning.

As already explained in Section 6.1, parking tries to choose a different or, after a given limit, completely random parking dock if all previous attempts failed to reach a dock. Random docks are ultimately used to avoid moving or planning back and forth in narrow corridors because they increase the probability to select a dock *not* requiring to pass one another. To summarize, if a Supply Robot fails, a Workstation tries to find another Supply Robot. Similarly, if a Workstation fails, a Carrier Robot tries to find another Workstation. If a Carrier Robot aborts its motion to a reserved Workstation (e. g. because path planning is unable to reach it), the Workstation tries to abort already emitted supply orders as soon as possible. Note that awaiting the Carrier Robot and requesting supplies is done in parallel at a Workstation for performance reasons. If a part is not available in a Warehouse, a Supply Robot waits until it is (which, in turn, forces Workstations and Carrier Robots to wait as well).

Finally note that fail state triggering described in Section 6.1 even allows to crash an agent abruptly which stops heartbeats from that agent instantly. They will eventually be removed from an agent's local database due to a timeout.

6.3 Dependencies of Production Stages

Within this section, we are going to describe the modeling of products, the management of dependencies of work steps, and how products, work steps and parts are related to the *capabilities* of an agent.

The product line-up is stored as a set of files loaded by the Product Server at startup. Every product is specified as a DAG $G = (\mathcal{V}, \mathcal{E})$ in a *GraphML file* also containing the product's name, its unique ID and the required capabilities for the Carrier Robot. GraphML was chosen because it is easy to understand, extendable with application-specific attributes, and widely supported by graph editors (e. g., yEd) and software libraries (e. g., Boost for C++ and NetworkX for Python). In such a graph, vertices $\mathcal{V}$ represent work steps and edges $\mathcal{E}$ represent the required order in which these steps must be processed. An example for such a graph is given in Figure 6.4 for a product named

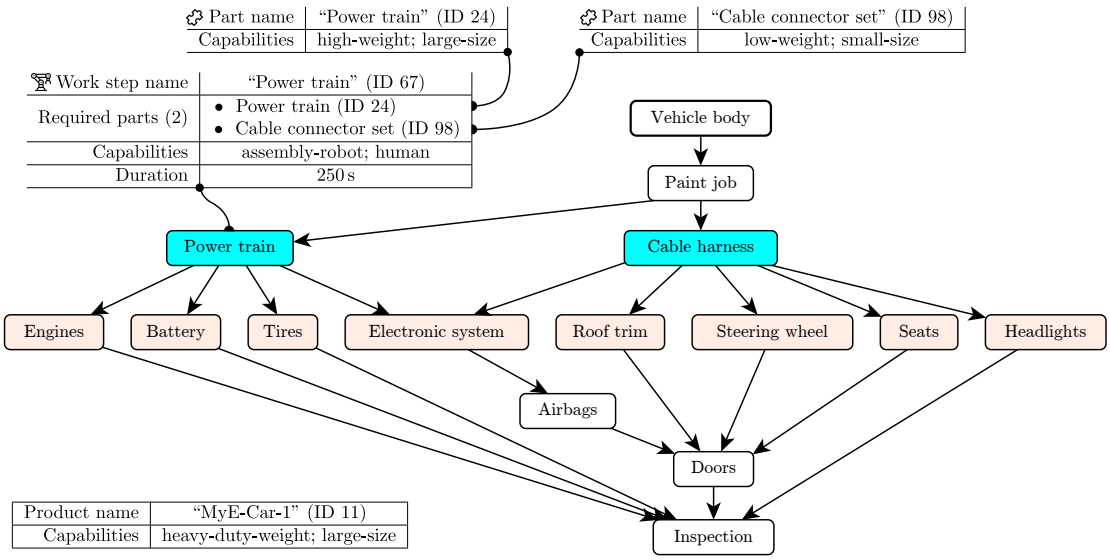


Figure 6.4: Simplified dependency graph of an electric car "MyE-Car-1" as an example for a product to be produced via decentralized assembly. Every vertex represents a required work step and edges indicate the precedence of processing them; the graph must therefore be fully traversed in topological order to complete the assembly process. For simplicity, the name of the most important part required by each work step is given as vertex label for reference. Color-shaded vertices indicate flexibility (multiple subsequent work steps possible). The details of the work step "power train" along with its required parts and capabilities is also illustrated as an example.

"MyE-Car-1" with ID 11 and capabilities "heavy-duty-weight" and "large-size" for the Carrier Robot. A vertex stores the ID of the associated work step (e. g., 67 for the step "power train", blue). This ID is used as a key for lookup in a *work step definition file* which specifies the work step name, the list of required parts, the required capabilities of the Workstation and the (simulated) duration it takes until completed. Every required part is again referenced by a unique ID which serves as a key for lookup in a *material definition file*. Such a file maps a part's key (ID) to a human-readable description of the part as well as a list of capabilities a Supply Robot needs to have in order to be able to transport that part. Two examples (IDs 24 and 98) are shown at the top of Figure 6.4 for the work step "power train".

Capabilities are specified as a list of strings predefined in a *capability definition file*. Capabilities can be grouped to combine multiple capabilities in a single string and there is a catch-`all` capability for convenience. The agent configuration for Workstations, Carrier and Supply Robots must also contain what specific capabilities such agents are capable of. Referring to Figure 6.1 again, for instance, this models that some Workstations may have a robot arm while others are operated by human workers only. Given a set $\mathcal{C}_O$ of required capabilities for some order O and a set $\mathcal{C}_A$ of specified capabilities of an agent A, A is said to be *capable* for O if $\mathcal{C}_O \subseteq \mathcal{C}_A$.

An important observation is that multiple choices of subsequent work steps may be possible depending on which work steps have already been completed. For example, if the "cable harness" from Figure 6.4 has been installed, at least four subsequent work steps are possible. If also the "power train" has been installed, eight work steps (green) are selectable as subsequent work step. This so-called *flexibility potential* allows for optimizing the production flow by compensating the temporary unavailability of parts or agents through switching to alternative work step. Unavailability may be caused by yet missing parts, faulty or congested agents [4]. It is automatically utilized by traversing the graph topologically. Such flexibility combined with the fault tolerance elaborated in Section 6.2 makes the proposed approach superior to existing fixed linear systems.

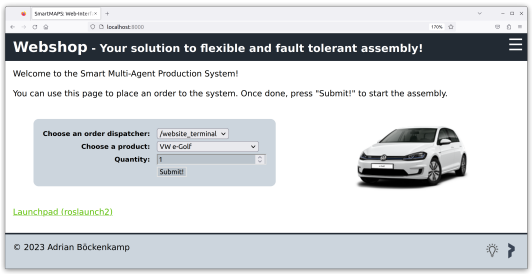
It remains to describe how the Product Server interacts with the other agents in the system (cf. Figure 6.3). After startup, the Product Server offers a set of services to query information about the loaded products, work steps and parts. The two most important services are:

- Given an ID of a product and the specific order ID (UUID), a Carrier Robot requires the current set of possible work steps. The order ID is used to check which steps have already been completed. These steps are stored in the Product Server to prevent submitting them over and over again. Each request of this type may be supplemented with a work step ID indicating the previously completed step. That step is then stored in the Product Server for further reference w. r. t. to the given order ID.
- Given an ID of a work step, a Workstation requires the set of parts required to process that work step. To reduce communication overhead, for each ID of a part in that set, it also returns the set of capabilities required to transport it. This is used to find appropriate Supply Robots.

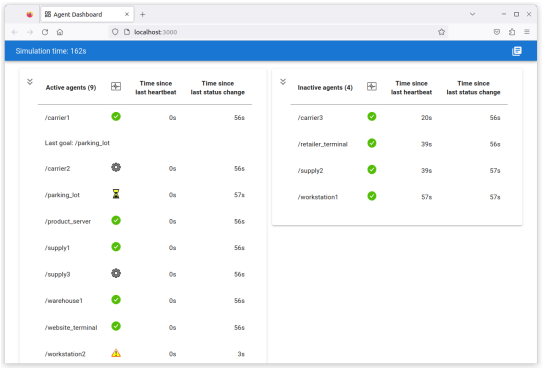
Notice that this design choice makes the Product Server a potential single point of failure unless running redundantly because it keeps track of all completed work steps of products being currently in production. Upon first request by a Carrier Robot asking for the set of possible work steps, the Product Server instantiates the graph associated with the provided product ID and executes a "single iteration" of topological sorting for every request of that Carrier Robot, associated with the provided order ID. If a work step was provided by the Carrier Robot to be marked as completed, that associated vertex and all of its outgoing edges are removed from the instantiated graph. To get the set of currently possible work steps, all source vertices need to be collected, i. e., vertices with an in-degree of zero.

6.4 Visualization and Introspection

Decentralized and decoupled systems have the drawback of being difficult to visualize and inspect while running because every agent is managing its own state. Within this



(a) Screenshot of the Webshop to create customer orders by choosing from the currently offered product line-up



(b) Website of the Agent Dashboard to inspect the existence of agents and their states (indicated by symbols) in the assembly system

Figure 6.5: Screenshots of developed tools for visualization and interaction: both websites show content that was gathered live from the assembly system running in the background.

section, we will briefly show how the presented system can be visualized and inspected. Note that this section only deals with visualizing the state of logical agents (as introduced in Section 6.1) and is not concerned with visualization of planned paths, obstacles or anything else that is related to navigation or localization because that is another topic on its own.

The heartbeat logic (described in Section 6.1) provides a convenient way to detect agents in the system. This serves as the foundation for all UI tools used for visualization and interaction. Introspection (if not already built into the UI tools) is realized by manually invoking ROS services to query information about agents at runtime (refer to `SmartAgent` in Figure 6.2).

Figure 6.5(a) shows a screenshot of the Webshop already depicted in Figure 6.3. Once started, it detects all running Order Dispatchers and lists them in the first drop-down list box. They serve as the entry point to dispatch the customer orders to. Similarly, once a Product Server has been detected, the current product line-up is requested and all offered products are added to the second drop-down list box. When the user clicks the “Submit” button, a customer order is generated and the system starts to build it.

Finally, Figure 6.5(b) shows the Agent Dashboard as also already referenced in Figure 6.3. The dashboard subscribes to all heartbeats in the system and lists detected agents in the center table. Once an agent times out, it is moved to the table on the right. The last known state of every agent is symbol coded for improved readability. For instance, because all mobile robots are currently parking (state IDLE, green checkmark), there is no free parking dock such that the `/parking_lot` agent is BUSY (hourglass). Since permanent failures have been injected in the Order Dispatcher named `/retailer_terminal` as well as `/workstation2`, both are in state FAIL (yellow exclamation mark). Finally, because `/supply2` has just been reset, it is still in the INIT state (gray gear wheel).

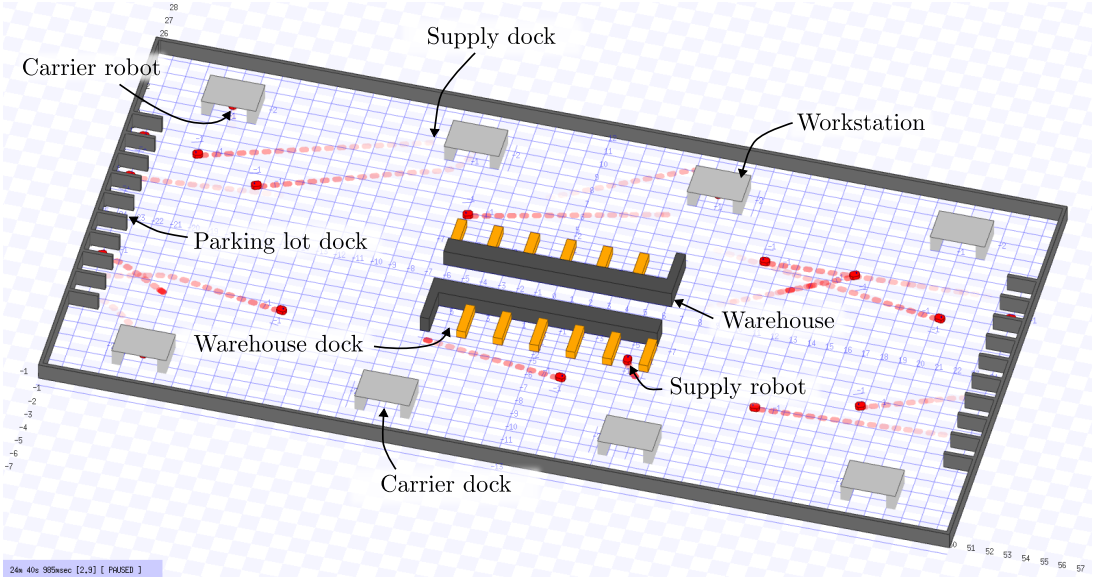


Figure 6.6: Simulated environment in Stage used for evaluation; the configuration shown here has eight workstations (light gray), two warehouses (dark gray with orange docks), a parking lot with 18 docks, nine carrier and nine supply robots (red). The grid size (shown for reference) is 1 m and the environment 50 m × 24 m.

6.5 Evaluation

It remains to assess the performance of the assembly system and to analyze its properties. CLPF is used with the same parameters (ICSPS, max. 500 lex. permutations, state delay $T_d = 1\,\text{s}$, NPH ahead distance $D = 2\,\text{m}$) and test system (AMD Ryzen 3900X 12 core CPU, 64 GB RAM) as already explained in its evaluation in Section 5.9. Note that those analyses (e. g., the parameter optimizations) and the ones being conducted here have all been implemented as experiments in the developed REF.

The *Stage* simulator [24] has been used as well and an example of the simulated environment is shown in Figure 6.6. The screenshot shows eight workstations (gray) whereby every workstation has a supply dock and a carrier dock. Carrier robots (red) reserve and move to the carrier dock for processing certain work steps while supply robots (red as well) move to the supply dock of a workstation to bring the required materials (cf. Section 6.1). Products are being manufactured on the carrier robots. The warehouse (gray) has a configurable number of docks (orange) where supply robots can collect the required materials. Both robot types can go parking at one of the parking docks. The latter are reservable locations on the shopfloor and represented as small bays with a wall on its left and right side. The number of robots, workstations and warehouses can be varied when generating the environment.

The product lineup, its assembly complexity and the distribution of dispatched customer orders mainly determine the resulting achievable throughput. Recall that carriers have

to process all work steps of a given product until it is completed. Clearly, the more work steps, the longer it takes to assemble the final product. Because this evaluation focuses on applying the proposed concepts of Chapter 4 and 5 in an industrial automation case study, we are not using a real, complex product lineup (e. g., as shown in Figure 6.5(a)). Instead, a random lineup has been generated with defined parameters. That is, a total of ten different products have been generated, each of them having a random number of work steps in the range $[1, 5]$ with dependencies modeled as a random DAG. Each work step has a constant processing duration of 1 s (simulated time), requires zero or one part from a generated set of parts with a probability of 40 %. The aforementioned constraints have intentionally been chosen to be rather restrictive/small to let the system generate observable output in terms of finished orders in a reasonable amount of time.

From this product lineup, random products (customer *orders*) are being selected to generate load—as contrasted with and in addition to random and scripted *goals* in Section 5.9. Note that the production of a customer order normally requires the processing of multiple goals from different robots.

6.5.1 Throughput

Within this section, we are going to analyze the throughput of customer orders while varying the number of robots, workstations and warehouses. The environment is shown in Figure 6.6 and the results are given in Figure 6.7 for three configurations of workstations and warehouses (blue, black, green). Carrier and supply counts have been varied from 1 to 9 in steps of 2, yielding a total of 25 combinations. The total number of robots (abscissa) in the diagram is the sum of carrier and supply robots, yet indistinguishable w. r. t. the robot type. At least 2 h of operation has been simulated, and the experiment has been terminated after all pending orders have been completed. The ordinate shows the resulting total throughput for the entire experiment time.

As it can be seen in the trend of the number of robots, a higher robot count increases the resulting throughput as expected. Similarly, the higher number of workstations and warehouses, the higher the throughput in general. The blue data series shows the maximum throughput (116 orders) with the largest number of agents. However, especially up to a total of ten robots, some data points in the diagram do not exhibit an increased throughput, even within the same configuration of workstation and warehouse count (cf., e. g., black crosses in Figure 6.7). This is justified by the allocation of the total robot count per type. For instance, many supply robots will not increase the throughput if there are only a few carrier robots (responsible for transporting the products). Another observation is that in case of only four workstations and one warehouse (black crosses), a total of six robots (3 carrier and 3 supply) is already sufficient to make 87 % of the achievable throughput because more robots only add a subtle increase.

To further differentiate between the two types of robots, Figure 6.8 shows a 3D diagram with the number of robots on the x- and y-axis and the throughput on the z-axis. It underlines the same trend already observed in its 2D simplified version: a larger number of

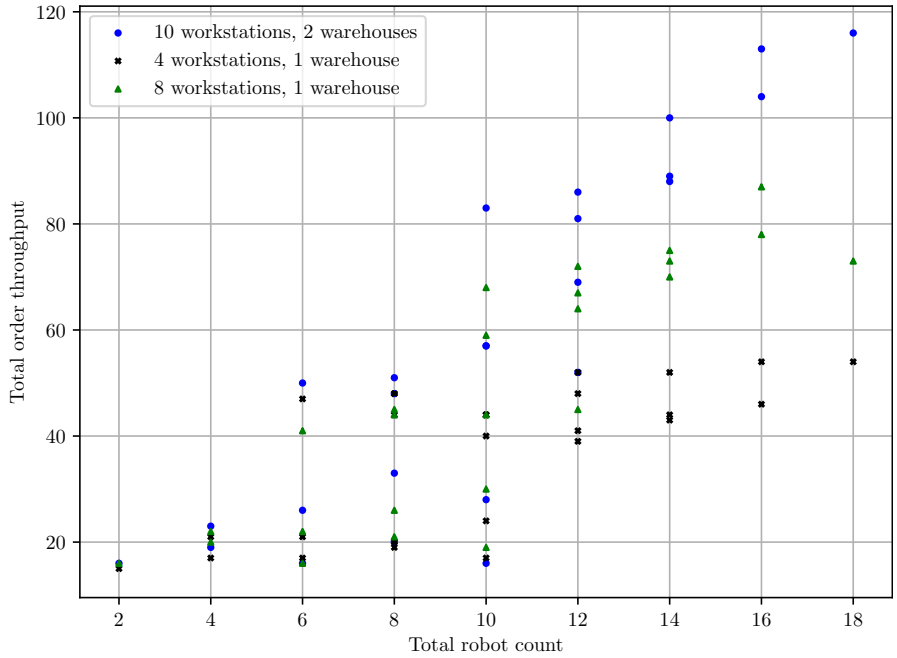


Figure 6.7: Scalability measured by throughput of customer orders (ordinate) while varying the total number of robots (abscissa) which is composed of a given number of carrier and supply robots (indistinguishable here). Three different experiments have been evaluated in which the number of workstation (4, 8, 10) and warehouses (1, 2) have been varied (blue, black, green), each of them simulating a total of at least 2 h of operation.

robots increases the throughput without showing any indication of congestion even in case of ten workstations and two warehouses. Given this analysis, it is still indistinguishable whether the increased number of workstations or the number of warehouses enhances the throughput. However, it is assumed that it is justified by the number of workstations because warehouses always had free docks.

A closer look at Figure 6.8 also clearly reveals that increasing the number of carrier robots outperforms an increased number of supply robots regarding throughput up to a certain level where the number of supplies becomes the bottleneck. This kind of analysis helps to find bottlenecks in general and allows to optimize the number of agents based on requirements and expected load. A plot very similar to Figure 6.8 results if the throughput would be represented per time (e. g., order count per minute). The resulting numbers would be much smaller than the goal throughput measured in Section 5.9.3 due to the complexity of the underlying assembly process.

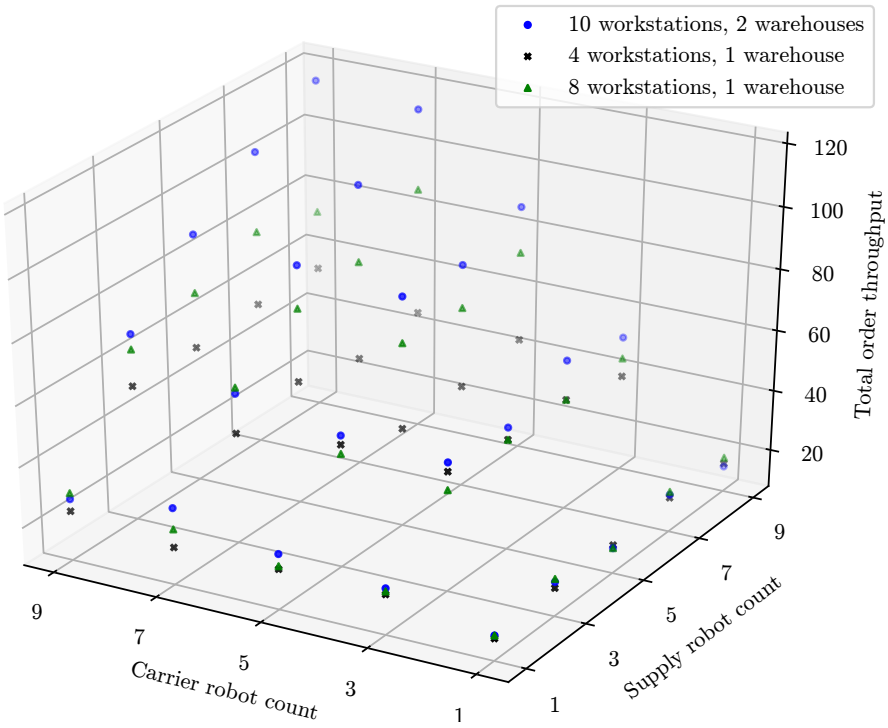


Figure 6.8: Scalability measured by throughput of customer orders (ordinate) while varying the number of carrier and supply robots (abscissas). Unlike Figure 6.7, this diagram distinguishes between the number of carrier and supply robots. At least 2 h of operation has been simulated for each of the three scenarios.

6.5.2 Fault Tolerance

Section 6.2 has described how the system handles various error cases. As already explained in Chapter 1, an important advantage of decentralized assembly over linear (conveyor) systems is its fault tolerance. This and its robustness are analyzed in this section.

Figure 6.9 depicts the behavior of the system while injecting failures into workstations (forced maintenance). The experiment took about 14 (simulated) hours (abscissa), that is, more than one day of continuous operation. Both ordinates show the throughput of customer orders per minute while only the cumulative throughput curve (gray) refers to the right ordinate. The blue data series shows the raw throughput per minute while the black curve visualizes a moving average over a timeframe of 30 min. The five events of failure injection have been marked with red dashed vertical lines.

After every 2 h of operation, another random workstation has been selected for failure injection. Immediately after injecting a failure, the workstation becomes inactive and is automatically avoided by carrier robots based on the status (`FAIL`) sent via heartbeats (cf. Listing 6.1). Note that if a workstation would even be unable to communicate,

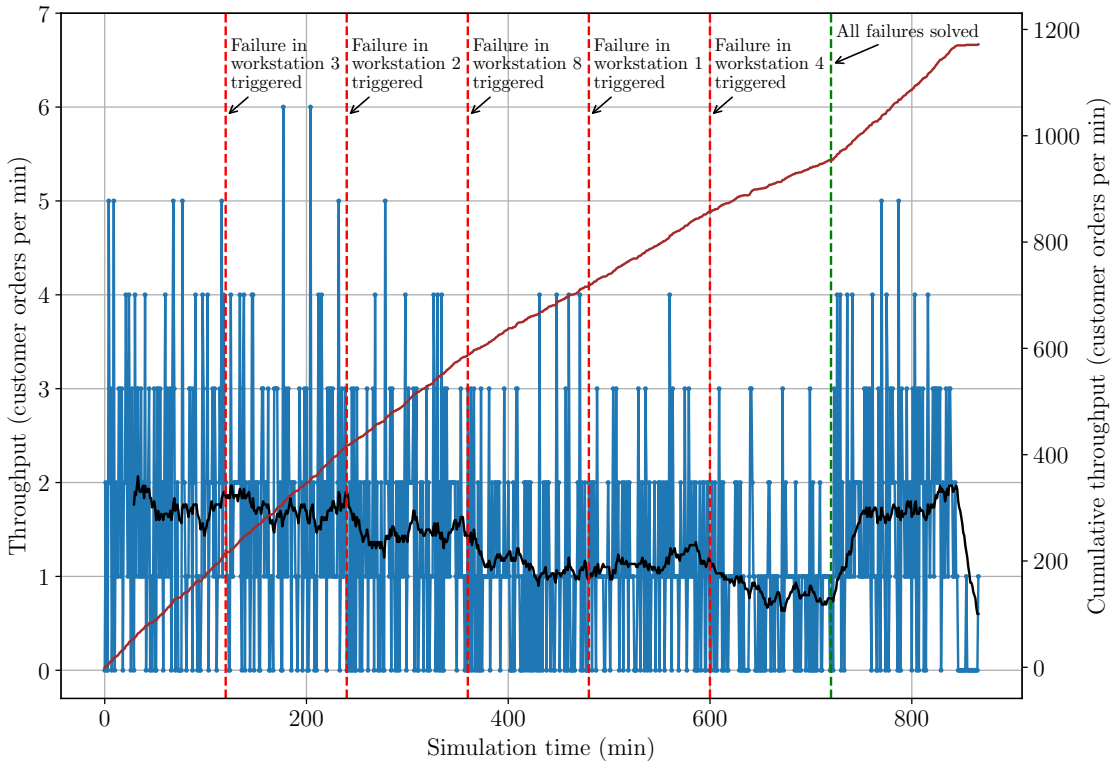


Figure 6.9: Analysis of the impact of artificially injected failures in workstations on the throughput of produced customer orders (left ordinate) for an experiment taking a total of 840 min (abscissa). The throughput is shown in blue and a moving average is shown in black. The gray curve shows the cumulative throughput over time with reference to the right ordinate. After every 2 h of operation, a failure has been injected in a random workstation (dashed red lines). After 12 h, all failures have been resolved (green dashed line).

heartbeats would timeout resulting in the same behavior. The system is operated for 2 h after every injection to let the failure take effect. The last failure is injected at 600 min (10 h) with 50 % of all ten workstations being faulty. After 12 h, all failures have been removed (green dashed vertical line) to let the system recover for 2 h. A total of 18 466 goals has been processed by both robot types whereby the majority (57.84 %) fall on supply robots. 12 843 out of 18 466 goals (69.55 %) have been reached successfully.

As it can be seen from the moving average (black) in Figure 6.9, every injected failure degrades performance of the system by means of a drop in throughput. However, even after halving the number of available workstations, the assembly system is still fully operational and emits completed orders. Similarly, the cumulative throughput (gray) slowly flattens out while more workstations are taken out of service until rising again when all workstations are being recovered. As expected, the system reaches its previous

production capacity once all workstations are running again. A linear conveyor system would be affected by downtime already after a single failure [64].

It is worth noting that all orders have been successfully produced without any issues despite the injected failures in the workstations. That is, all carrier robots correctly dealt with the disturbances caused by failing workstations. Note that such failures were injected at any time, meaning the carrier robots also needed to handle the case where a work step has already been started at a workstation that is about to become faulty. In such cases, carrier robots restarted the work step at another workstation.

6.5.3 Comparison to Other Approaches

We have also tried to evaluate the scenario used in Figure 6.8 (blue) with well known path planners (online motion planning approaches) like Vector Field Histogram+ (VFH+) [59], Timed Elastic Bands (TEB) [45] and Dynamic Window Approach (DWA) [22] as local planners (the latter being the default planner in ROS). However, all planners already had massive problems in performing the initial parking operations, that is, all robots are requested to park before the actual order processing can start. To do so, each robot requests a free parking area at the parking lot agent in order to exclusively reserve and drive to it. The parking lot agent assigns the closest free parking lot to every requesting agent to increase efficiency. At startup of the system, this creates the situation that every robot is assigned its current location (or a position very close to it) as the motion target for parking. All local planners are therefore challenged to either just confirm their current position to be the reached final goal or to just make minor positional adjustments. However, all mentioned planners struggled to do so. VFH+ even started to compute detours *away* from the goal, presumably to move back afterwards while avoiding sharp curves due to its parameterization (turn radius). As a consequence, the scenario in Figure 6.6 was intractable for all planners on most of the robots (except for CLPF). Note that the walls around the parking lots were known to the planners.

To overcome the previously described limitation, the environment has been simplified by getting rid of the walls enclosing the parking lots (10 on the left and 10 on the right of Figure 6.6). However, Vector Field Histogram+ (VFH+) was still unable to park all robots properly upon startup. The simulation has been aborted after 650 s without completing (or event starting) any customer order and more than 200 collisions. Collisions are automatically detected by means of the presented REF (more precisely, the EIF, see Section 3.1). Every time a collision is detected, the collided robots are automatically repositioned at a random collision-free location nearby to let the experiment continue. This functionality is realized by a combination of the agent detection logic (see Section 6.1) and the EIF. VFH+' difficulties are most probably explained by the fact that the algorithm is not designed to handle this kind of complex reorganization at startup causing robots to either collide or to get stuck at the park boxes. It is rather developed to handle longer forward-directed motions while avoiding unknown (semi-) static obstacles (and no dynamic obstacles in particular). Timed Elastic Bands (TEB)

was also not able to park at all and while not causing any collisions, also no customer orders were completed (not even started as well).

After simplifying the environment, DWA was more or less working. However, after simulating more than 1 h of operation, only five customer orders have been completed while creating a total of four collisions. The same collision handling has been used as for VFH+. From Figure 6.7 it becomes clear that CLPF was able to complete more than 100 orders in 2 h which is more than 10 times higher than DWA for this particular scenario. For fairness, it must be noted that the generated static obstacles (workstations and warehouses) are not known a priori to DWA (albeit via laser scans) because ROS' map server does not allow dynamic updates of the (generated) map (unlike the VMS presented here).

Generally, note that by design, the large number of robots (18) is challenging here because all compared algorithms (VFH+, TEB, DWA) require a lot of computation time for evaluating laser scan data to compute velocities online. By considerably lowering the RTF to approximately 1.1 in case of DWA, simulation was possible. During the experiment it went even down to only 0.5x, i. e., 1 s corresponds to 2 s of simulated experiment time. Distributed computing would surely help in speeding up this experiment.

Finally, despite the results presented in this section, it must be noted that all the compared planners may be improved by tuning parameters. However, this also constitutes a major drawback of those approaches because they have many parameters such that tuning them is a time-consuming and tedious task. In contrast, only a few parameters are sufficient for CLPF. Note that while using CLPF in all experiments conducted in this evaluation, robots have never ever crashed into each other.

6.6 Application of Experiments

As already noted, the Robotic Experimentation Framework (REF) from Chapter 3 has been used throughout the evaluations in this chapter (see Section 6.5) as well as in Chapter 5. We will retrospectively outline here, how the REF has been applied in those evaluations to clarify its application.

Except for roslaunch2, Figure 3.1 depicted the main components of the REF. Because all experiments presented here have been conducted with the *Stage* simulator, the `StageEnvironment` backend class of the abstract `EnvironmentInterface` class has been used. It provides all simulator-related functionality, for instance, pausing/resuming, registering collision notifications, etc.

Notably, the "Robot 1", ..., "Robot N" boxes indicated on the left of Figure 3.1 represent the full software stacks required to operate the robots (cf. Figure 5.1). For the evaluations of Chapter 5, this primarily comprises the localization, the Vector Map Server and the path planner (instantiating both the global and local planner). All these components are running in their own ROS node (dedicated processes). The global planner and the

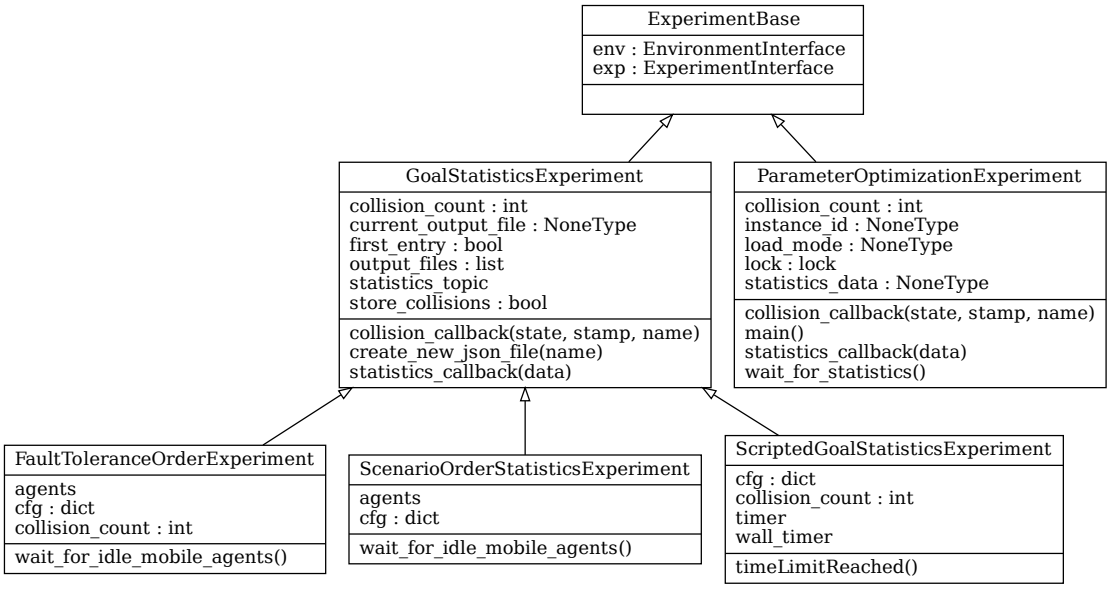


Figure 6.10: Excerpt of the experiment classes in Python used for evaluation, instantiated with the REF; for instance, the `FaultToleranceOrderExperiment` has been used for the fault tolerance assessment (see Section 6.5.2). The logic of storing the results as a JSON file on disk for subsequent analysis and visualization is encapsulated in `GoalStatisticsExperiment`. `ExperimentBase` contains the `env` object to interact with the simulator and the `exp` object to interact with the software stack of the robots.

VMS have been presented in Section 5.8 and the CLPF equals the local planner. Within this chapter, the agent logic (see Section 6.1) is started *additionally* and running as a ROS node in its own process. It is communicating with all aforementioned components also started for Chapter 5 via the ROS API.

The red block ("Experiment Supervisor" containing the "Experiment") in the middle of Figure 3.1 has been written specifically for all various evaluation cases, although based on a class hierarchy to reuse common functionality. An excerpt is shown in Figure 6.10 for reference. For instance, the `FaultToleranceOrderExperiment` has been used for the fault tolerance assessment within Section 6.5.2. This is similar to the exemplary experiment shown in Listing 3.1, also containing a (shortened) self-containing experiment.

Chapter 7

Conclusion

It remains to conclude this thesis by summarizing its main proposed concepts, results and findings in Section 7.1. Finally, Section 7.2 gives an outlook regarding further research in the presented topics.

7.1 Summary

After introducing and motivating the overall topic in Chapter 1, Chapter 2 dealt with the review of related work. The general Multi-Agent Path-Finding problem was presented and existing state-of-the-art approaches were sorted into different categories (taxonomy) to found a better understanding of the research field (Section 2.1). The contributions of this thesis were then classified and defined w. r. t. existing approaches (Section 2.2). A detailed review of related subjects was conducted (Section 2.3) whereby the proposed concepts of this thesis are closely related to fixed path coordination approaches.

Chapter 3 introduced the *Robotic Experimentation Framework* (REF) to simplify writing robotic experiments based on ROS using the developed Python API (Section 3.1). It is designed to analyze, optimize, validate and inspect a system and has also been used throughout this thesis. Experiments are being written using the APIs of the presented components, namely the *Environment Interface* (EIF), the *Experiment Interface* (EPI), the *Experiment Supervisor* (ESV) and *roslaunch2* (RL2). The framework's architecture has been explained, three supported simulators (Stage, Gazebo and MORSE) have been presented (Section 3.2), and the code of an exemplary experiment was discussed (Section 3.3). The REF has been verified along with the evaluations of both Chapters 5 and 6.

Chapter 4 addressed the problem of collision-free multi-robot scheduling. That is, given a set of robots at known start locations with pre-planned (fixed) paths, compute a schedule that coordinates the robots on their paths to their goals while preventing collisions (Section 4.2). Notably, it is assumed that robots are initially located at their start locations and that they are not already moving. The proposed solution was based on so-called intersection guards which are computed first. Every conflict is defined by a halt and a release parameter identifying the exact location of a conflict's boundaries—yet without the consideration of time. Detecting all conflicts and computing halt and release progresses was referred to as conflict detection (Section 4.3). Two algorithms have been presented and evaluated for conflict detection, namely *Smallest Guarded Segments* (SGS) and *Merged Guarded Subpaths* (MGS). The schedule is computed by a so-called solver and specifies the order in which robots of a given input scenario have to pass through the conflict areas. Two solver algorithms have been presented, evaluated and compared: the *Incremental Coordination-Space Path Scheduler* (ICSPS) and the *Optimal Multi-Robot Path Scheduler* (OMRPS). ICSPS is based on the coordination space that is spawned by the paths of conflicting robots (Section 4.4). Conflicts are being represented in the coordination spaces based on halts and releases for involved robots. Computing a shortest path inside the coordination space allows to deduce a *Right-of-Way* (RoW) assignment for all conflicts. For comparison, an optimal algorithm (OMRPS) has been developed and its correctness has been proven (Section 4.5). OMRPS is build upon full search space enumeration of all possible decision vectors and the assessment of every enumerated candidate regarding a given criterion (*total travel time* (TTT) or *critical path time* (CPT)).

The comparison between SGS and MGS for conflict detection has shown that SGS may create many very small intersection guards, especially if the input paths contain many support points in the conflicting sections. MGS turned out to be more efficient while exhibiting similar computation times because small adjacent intersection guards are merged (if possible) to form conflicts covering larger sections of a path. This reduces the runtime of subsequent solver algorithms considerably. MGS was therefore used throughout the remainder of this thesis. However, depending on the solver algorithm and input characteristics, not merging adjacent conflicts can have the advantage of allowing robots to already release sections of a conflicting path.

Evaluating the two solver algorithms revealed that ICSPS is able to find a solution quite quickly if one exists, even if a limited number (200) of permutations is used. In contrast, OMRPS with an enumeration limit may not be able to return a valid solution even if there are only a few robots. Surprisingly, OMRPS turned out to be faster than ICSPS for very small inputs. Together with its optimality, this suggests combining both algorithm while distinguishing the input complexity to avoid exponential runtime. Computation times for ICSPS were typically much smaller than 1 s and only peaked up to 1.2 s rarely for complex inputs. For up to six robots in the input of the analyzed test cases, ICSPS was able to achieve an optimal solution in more than 80 % of the test cases.

Chapter 5 dealt with the conception of the new *Collaborative Local Planning Framework* (CLPF) to coordinate a fleet of moving robots while guaranteeing collision-free motions. After explaining required assumptions, limitations and requirements (Section 5.2) an overview of the methodology has been given (Section 5.3). Basically, the framework defines the communication protocol for the robots to negotiate arising conflicts, if any, while requiring that all robots remain on their paths. In distinction to the solvers of Chapter 4, the framework allows for *dynamic* inputs, i. e., it provides the concepts for (re-) negotiating conflicts between already moving and yet non-moving robots. Arbitrary communication delays are tolerated, and message loss is considered to be handled by TCP.

The main components of CLPF are the *communication protocol* (message definitions and semantics), the *finite state machine* (FSM) (behavior model), the *intersection graph* (conflict representation) and the representation of *local knowledge* (robot-local database). Robots exchange messages of different types to share knowledge about assigned goals and computed paths (Section 5.4). This way, every robot will eventually know about other robots in the system and their state. If robots are not in conflict with each other, they still have to exchange ACKs in order to ensure recent knowledge about their paths. In contrast, if robots detect conflicts by means of received paths, they have to negotiate the RoW first before being allowed to move. If a robot gets the RoW at some intersection over another robot, it publishes progress after completely leaving the conflict area to notify the other robot of the release. The progress is a compact parameterization of a robot's position on its path. The different states require a robot to react differently to incoming messages. A robot's behavior is therefore modeled as an FSM (Section 5.5). Every state in the FSM represents the combination of a robot's own state and the currently known state of other robots. Receiving messages or getting a goal may cause state transitions. Actually starting to move is only allowed after either receiving all ACKs from other robots if there are no conflicts at all, or after negotiating all direct or indirect conflicts.

Conflicts are being represented in the intersection graph (Section 5.6) managed locally on every robot. That is, every robot maintains a graph data structure that is updated when conflicts are being detected or have disappeared. Updates are mainly triggered by received messages. A vertex in the graph represents a single robot and an edge indicates a conflict. This way, the intersection graph on every robot eventually converges to the same graph on all robots. For performance reasons, negotiations are limited to the subsets of robots given by the connected subgraphs of the whole, possibly unconnected, intersection graph. A subgraph w. r. t. a given robot R is defined by the set of vertices and edges that are directly or transitively reachable from R. A developed hashing algorithm allows a robot to efficiently identify whether its local subgraph is equal to all subgraphs from robots also being part of it. In such a case, the scenario is locked to ultimately trigger solving the synchronized scenario through solver invocation.

Once the RoW has been negotiated (solved), a robot's motion controller executes the computed schedule while adhering to the RoWs and releasing conflict areas when

appropriate (Section 5.7). This way, uncertainty in the motion controller and, more generally, in the Environmental and Planning Model (EPM) is handled.

Non-moving robots are added as semi-static obstacles to the so-called *Vector Map Server* (VMS) which stores a map of the current environment (Section 5.8). When a robot is assigned a goal, the global planner uses its current map (distributed by the VMS) to compute a path from the robot's position to the goal while avoiding obstacles from the map. Given that fixed path, the previously described procedure for coordinating the motion along that path is executed. Non-moving robots are allowed to request renegotiation from moving robots in order to safely move as well. This was realized by the concept of the so-called next possible halt (NPH) emitted by moving robots (and controlled by the *NPH ahead distance* parameter) when requested to renegotiate.

The *state delay* parameter was used to specify the delay in (simulated) seconds before transitioning from the AckDelay and IntersectionDelay states to a subsequent state in the FSM. The optimization of these two main parameters (NPH ahead distance, state delay) revealed that the state delay is only beneficial for larger scenarios (Section 5.9). As a rule of thumb, at least 20 robots should be part of the system, but the complexity also depends on the environment and the distribution of goals. The NPH ahead distance can be considered useful because it allows already moving robots to continue their motion while negotiating in parallel and avoiding stop-and-go behavior. All analyses have generally shown that the findings are highly dependent on the input characteristics (e. g., environmental complexity, distribution of goals, etc.). As expected, the runtime analysis has shown that a higher number of robots adds complexity which yields longer execution times. Up to 50 robots have been simulated in a single scenario on the same machine (including simulation) which illustrates the effectiveness of the developed approach. CLPF is able to retain a constant goal rate while fluctuations are possible due to communication and (re-)negotiations. This is a requirement for safety and unavoidable. The evaluation of scalability has demonstrated that CLPF scales linearly if there are no conflicts (analyzed for up to 20 robots). For some scenarios, a sweet spot between 9 to 11 robots could be observed which is justified by (complex or narrow) environments causing a high conflict probability. Thus, adding more robots in these environments will not increase the resulting goal throughput. For the other environments, an increasing throughput could be observed while adding more robots to the system, indicating its scalability. It is worth mentioning that in all the analyzes performed, none of the robots ever crashed into each other.

Chapter 6 concluded with a case study of a simulated, yet fully operational, assembly system (Section 6.1). The system is composed of autonomous agents acting decentrally with the major advantage of being more fault-tolerant than classic linear conveyor systems (Section 6.2). The modeled system allows the order and production of complex and highly individualized automobiles to demonstrate the flexibility of the approach. A customer order is dispatched to a carrier robot and manufactured piggyback on the robot. The product lineup has been modeled with a set of so-called product dependency graphs (Section 6.3). Every possible product is represented as a graph containing a

vertex for every work step, and edges indicate precedence between work steps. Work steps are being processed at workstations and may require parts for their processing. Workstations can request supply robots to bring the required parts. Supply robots move to a warehouse dock in order to collect the required parts and transport them to the requesting workstation afterwards. Customer orders, work steps and parts are labeled with so-called capabilities. This allows to differentiate between differently capable agents (carrier robots, workstations, warehouses, supply robots). Once robots become idle, they park themselves to not block the environment. Automobiles have been chosen as an example—concrete products can be substituted depending on the intended purpose. The entire implementation is based on ROS and can therefore be also applied on real robots. Tools for visualization and introspection have been showcased exemplarily (Section 6.4). Because CLPF is ideally suited for industrial automation due to its robustness, it has been used together with ICSPS as a solver algorithm on all mobile robots for the case study. This combined all concepts of this thesis in a single evaluation.

A repeated throughput analysis (Section 6.5), now based on customer orders instead of goals, showed that the assembly system was able to increase the throughput of completed customer orders for a generated product lineup while increasing the number of robots reasonably (up to 18). Some configurations of agent counts indicated stagnation caused by an inappropriate mixture of agent types. That is, the number of carrier robots should be at least equal or greater than the number of supply robots.

By artificially injecting failures in workstation while measuring the throughput (completed orders), the fault tolerance of the system was experimentally proven. Even after 50 % of all workstations had a failure, the system was still operational. Moreover, after recovery of all faulty workstations, the system's production capacity went up to its previous level. Notably, all customer orders were produced without failures. This shows the robustness of the proposed approaches.

A final comparison against other well known planners from the literature (VFH+, TEB, DWA) has shown that none of them was an adequate replacement for CLPF because they were not able to even perform the initial parking, caused collisions or just completed only a tenth of the completed customer orders compared to CLPF. Finally, note that the REF has implicitly been evaluated and thereby proven its effectiveness.

7.2 Future Prospects

Within this section, further research is discussed for each of the previous chapters.

A useful addition to the Robotic Experimentation Framework from Chapter 3 would be snapshots for experiments. This way, robotic experiments in simulations can be saved at some point in time and continued or repeated multiple times, possibly with different parameters or configurations—similar to snapshots known from virtual machines. However, this is a challenging feature because all components and nodes need to be aware of creating and restoring a snapshot to take measures accordingly.

Manually tuning the real-time factor (RTF) has also been tedious. If all ROS nodes would report missed deadlines to some diagnostics topic, the simulator may self-tune its speed (RTF) by subscribing to that diagnostics topic and adjusting the simulation speed in some controller feedback loop.

Additionally, adding support for more simulators, like NVIDIA Isaac Sim [38], a simulator providing photorealistic, physically accurate virtual environments, may further reduce the described reality gap.

During the analysis of both solvers in Chapter 4, it turned out that a combination of both would be beneficial. This way, small scenarios would be solved optimally by OMRPS and large scenarios are quickly handled by ICSPS. This just requires analyzing the input's complexity w. r. t. the number of conflicts in order to invoke the appropriate solver. ICSPS might be improved by taking more paths inside every coordination space into consideration. Currently, only the shortest path is used (as computed by A*). However, since the construction of coordination space $i+1$ is based on all solutions in the coordination spaces $1, ..., i$, other solution paths may allow for an overall improved solution. Nonetheless, this would increase computational complexity and further investigations would be required to restrict oneself to "more appropriate" solution paths.

There is no handling of partial solutions yet. For an input where a subset of robots has infeasible conflicts, ICSPS will eventually encounter a coordination space in all inspected permutations that will prevent A* from finding a path at all. Similarly, OMRPS will also always encounter a partially infeasible decision vector which is discarded as invalid completely. An experimental implementation for OMRPS has already indicated that the consideration of partial solution can improve the feasibility rate. However, because infeasible robots can transitively make other robots infeasible as well, a detailed analysis of corner cases is necessary.

CLPF from Chapter 5 does not yet support the modification of paths after they have been computed by the global planner. Theoretically, a robot may stop on certain sections on its path, anounce itself as being idle again in order to recompute and publish an updated path. However, actually making use of modified paths is another topic on its own. Similarly, aborting goals is currently not allowed as well. That is, after announcing a new path, a robot is required to move to its goals unless it detects infeasibility in a renegotiation along its way. This can also be used to implement abortion. A direct advantage for the assembly system would be the abortion of parking in favor of starting a new customer or supply order right away. As a further extension of CLPF, renegotiation may be triggered if robots are forced to wait for a time longer than expected due to unpexpected disturbances (e. g., a human blocking a robot's path). However, this would require the special handling of such blocked robots in the solvers.

The current design does not yet allow for partially releasing intersections, i. e., a robot always must have left the entire conflict area before the robot, not getting the RoW, is allowed to enter. This can quite simply be implemented during the execution of a negotiated input scenario for pairs of robots moving in similar directions based on their

published progress. However, sending periodic progress for partial release is a trade-off w. r. t. performance.

The benefit of the state delay parameter suggests adjusting it depending on the number of robots. That is, in smaller setups, a smaller value (or even zero) is more suitable. Also in case all robots have already joined a negotiation or all ACKs have already been received, delaying the further processing makes no sense and should be removed. In case of no conflicts, delaying the motion has also been considered useless and should be removed. The handling of moving and non-moving robots currently emits so-called start halts for non-moving robots having an intersection with moving robots. An improvement over the current design would be the consideration of a previous solution for unprotected path sections (see Section 5.5.5). While being more complex to implement, it is assumed to increase throughput.

The case study of Chapter 6 may be extended in various ways. For instance, other use cases of those already mentioned in Chapter 1 could be modeled. While being out of scope for this thesis, there is room for improvement w. r. t. the selection and optimization of agents in the assembly process (e. g., the selection of a subsequent workstation from a carrier robot's perspective). However and finally, the specific application and its requirements should guide the development because all analyses have revealed that results are highly dependent on the input characteristics (with special focus on the environmental complexity and load distribution).

It would also be interesting to analyze the behavior of the assembly system on real hardware as an extension to the conducted evaluation. For simplicity, all experiments in this thesis have been executed on a single machine. However, ROS also facilitates multi-machine setups and scaling to multiple machines may allow larger robot fleets with the presented concepts compared to the ones already evaluated.

Mathematical Nomenclature

Notation	Meaning
$\mathbb{N}$	Set of natural numbers $(1, 2, 3, \ldots)$
$\mathbb{N}_0$	Set of natural numbers including zero $(0, 1, 2, 3, \ldots)$
$\mathbb{Z}$	Set of all integers $(\ldots, -2, -1, 0, 1, 2, \ldots)$
$\mathbb{R}$	Set of real numbers (may be limited by a subscripted condition)
$\mathcal{S} = \{s_1, s_2, \ldots\}$	(Unordered) Set of elements s_i
$\mathcal{S} = \langle s_1, s_2, \ldots \rangle$	Ordered set of elements (a set $\mathcal{S}$ with a given order on s_i)
$\boldsymbol{p}$	N-dimensional vector $\left(p_i^{(1)}, \cdots, p_i^{(N)}\right)^\mathsf{T}$
$p_i^{(j)}$	j-th coordinate (scalar component) of the i-th vector $\boldsymbol{p}_i$
$\overline{\boldsymbol{pq}}$	Line segment between position vectors (points) $\boldsymbol{p}, \boldsymbol{q}$
$[a, b]$	Closed interval $\{\, x \in \mathcal{S} \mid a \leq x \leq b \,\}$ on some set $\mathcal{S}$
$\boldsymbol{p} * \boldsymbol{q}$	Scalar (dot) product $\sum_{i=1}^{N} p^{(i)} \cdot q^{(i)}$

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] M. M. Ahmadian, M. Khatami, A. Salehipour, and T. Cheng. Four decades of research on the open-shop scheduling problem to minimize the makespan. *European Journal of Operational Research*, 295(2):399–426, 2021.

[2] A. Babinec, F. Duchoň, M. Dekan, P. Pásztó, and M. Kelemen. VFH*TDT (VFH* with Time Dependent Tree): A new laser rangefinder based obstacle avoidance method designed for environment with non-static obstacles. *Robotics and Autonomous Systems*, 62(8):1098–1115, 2014.

[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, Nov. 1987.

[4] L. Bochmann, L. Gehrke, A. Böckenkamp, F. Weichert, R. Albersmann, C. Prasse, C. Mertens, M. Motta, and K. Wegener. Towards Decentralized Production: A Novel Method to Identify Flexibility Potentials in Production Sequences Based on Flexibility Graphs. *International Journal of Automation Technology*, 9(3):270–282, 2015.

[5] A. Böckenkamp. roslaunch2: Versatile, Flexible and Dynamic Launch Configurations for the Robot Operating System. In A. Koubaa, editor, *Robot Operating System (ROS): The Complete Reference*, volume 4, pages 165–181, Heidelberg, 2020. Springer International Publishing.

[6] A. Böckenkamp, C. Mertens, C. Prasse, J. Stenzel, and F. Weichert. A Versatile and Scalable Production Planning and Control System for Small Batch Series. In S. Jeschke, C. Brecher, H. Song, and D. B. Rawat, editors, *Industrial Internet of Things: Cybermanufacturing Systems*, pages 541–559, Cham, 2017. Springer International Publishing.

[7] A. Böckenkamp, F. Weichert, J. Stenzel, and D. Lünsch. Towards Autonomously Navigating and Cooperating Vehicles in Cyber-Physical Production Systems. In

O. Niggemann and J. Beyerer, editors, *Machine Learning for Cyber Physical Systems*, pages 111–121, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[8] A. Caprara, M. Fischetti, and P. Toth. Modeling and Solving the Train Timetabling Problem. *Operations Research*, 50(5):851–861, 2002.

[9] Y. F. Chen, M. Liu, M. Everett, and J. How. Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 285–292, 05 2017.

[10] E. Coumans. Bullet Physics Simulation. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. Association for Computing Machinery.

[11] R. Cui, B. Gao, and J. Guo. Pareto-optimal coordination of multiple robots with safety guarantees. *Autonomous Robots*, 32(3):189–205, 2012.

[12] B. De Wilde, A. W. Ter Mors, and C. Witteveen. Push and Rotate: A Complete Multi-Agent Pathfinding Algorithm. *Journal of Artificial Intelligence Research*, 51(1):443–492, sep 2014.

[13] V. R. Desaraju and J. P. How. Decentralized path planning for multi-agent teams with complex constraints. *Autonomous Robots*, 32:385–403, May 2012.

[14] V. Digani, M. A. Hsieh, L. Sabattini, and C. Secchi. Coordination of multiple AGVs: a quadratic optimization method. *Autonomous Robots*, 43:539–555, Mar. 2019.

[15] D. Douglas and T. Peucker. *Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line Or Its Caricature*, volume 10. Canadian Cartographer, 1973.

[16] DPD. "DPD testet Schwarmroboter für das *Depot der Zukunft*", Nov. 2022. https://www.dpd.com/de/de/news/loadrunner/, News, last checked: 11/03/2023.

[17] T. B. Dutra, R. Marques, J. Cavalcante-Neto, C. A. Vidal, and J. Pettré. Gradient-based Steering for Vision-based Crowd Simulation Algorithms. *Computer Graphics Forum*, 36(2):337–348, May 2017.

[18] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan. Modular Open Robots Simulation Engine: MORSE. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 46–51, June 2011.

[19] C. Eran, M. O. Keskin, F. Cantürk, and R. Aydoğan. A Decentralized Token-Based Negotiation Approach for Multi-Agent Path Finding. In A. Rosenfeld and N. Talmon, editors, *Multi-Agent Systems*, pages 264–280. Springer International Publishing, 2021.

[20] T. Foote. tf: The transform library. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, Open-Source Software workshop, pages 1–6, Apr. 2013.

[21] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99).*, July 1999.

[22] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine*, 4(1):23–33, Mar. 1997.

[23] K. Fransen, J. van Eekelen, A. Pogromsky, M. Boon, and I. Adan. A dynamic path planning approach for dense, large, grid-based automated guided vehicle systems. *Computers & Operations Research*, 123:1–10, 2020.

[24] B. Gerkey, R. T. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, volume 1, pages 317–323, 2003.

[25] German Federal Ministry for Economic Affairs and Climate Action (BMWK). "Smart Micro Factory for Electric Vehicles with Lean Production Planning (SMART FACE)", Oct. 2016. https://www.digitale-technologien.de/DT/Redaktion/DE/Standardartikel/AutonomikFuerIndustrieProjekte/autonomik_fuer_industrie_projekt-smartface.html, project profile within the "Autonomics for Industry 4.0" initiative, last checked: 11/03/2023.

[26] J. Guo, S. Zhang, J. Xu, and S. Zhou. Kalman prediction based VFH of dynamic obstacle avoidance for intelligent vehicles. In *International Conference on Computer Application and System Modeling (ICCASM)*, volume 3, pages V3–6–V3–10, Oct. 2010.

[27] Y. Guo and L. Parker. A distributed and optimal motion planning approach for multiple mobile robots. In *IEEE Proceedings of the International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2612–2619, 2002.

[28] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[29] L. Joseph. *Mastering ROS for Robotics Programming.* Packt Publishing, 2015.

[30] M. Kallmann. Shortest Paths with Arbitrary Clearance from Navigation Meshes. In *Proceedings of the Eurographics/SIGGRAPH Symposium on Computer Animation (SCA)*, 2010.

[31] C. Kapalschinski. "Arculus will Zeitalter des Fließbands beenden", May 2020. https://www.handelsblatt.com/unternehmen/mittelstand/familienunternehmer/start-up-arculus-dieser-gruender-will-das-

zeitalter-des-fliessbands-beenden/25840220.html, Handelsblatt, last checked: 11/03/2023.

[32] M. Kleinbort, K. Solovey, Z. Littlefield, K. E. Bekris, and D. Halperin. Probabilistic Completeness of RRT for Geometric and Kinodynamic Planning With Forward Propagation. *IEEE Robotics and Automation Letters*, 4(2):x–xvi, 2019.

[33] N. Koenig and A. Howard. Design and Use Paradigms for Gazebo, An Open-source Multi-Robot Simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154, 2004.

[34] S. LaValle and S. Hutchinson. Optimal motion planning for multiple robots having independent goals. *IEEE Transactions on Robotics and Automation*, 14(6):912–925, 1996.

[35] D. T. Lee. An optimal time and minimal space algorithm for rectangle intersection problems. *International Journal of Computer & Information Sciences*, 13(1):23–32, 1984.

[36] A. Liaqat, W. Hutabarat, D. Tiwari, L. Tinkler, D. Harra, B. Morgan, A. Taylor, T. Lu, and A. Tiwari. Autonomous mobile robots in manufacturing: Highway Code development, simulation, and testing. *The International Journal of Advanced Manufacturing Technology*, 104(9):4617–4628, Oct. 2019.

[37] R. Losch. "Audi will das Fließband abschaffen", Nov. 2016. https://www.heise.de/newsticker/meldung/Audi-will-das-Fliessband-abschaffen-3504451.html, heise online, last checked: 10/03/2023.

[38] NVIDIA Corporation. "Isaac Sim - Robotics Simulation and Synthetic Data Generation", June 2023. https://developer.nvidia.com/isaac-sim, last checked: 06/07/2023.

[39] R. Olmi, C. Secchi, and C. Fantuzzi. An efficient control strategy for the traffic coordination of AGVs. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4615–4620, 2011.

[40] L. E. Parker. Path Planning and Motion Coordination in Multiple Mobile Robot Teams. *Encyclopedia of Complexity and System Science*, pages 1–24, 2009. Robert A. Meyers (Editor-in-Chief).

[41] D. Poll. "Daimler Truck und BMW Motorrad: Das Ende von Takt und Band", Feb. 2023. https://www.produktion.de/technik/daimler-truck-und-bmw-motorrad-das-ende-von-takt-und-band-950.html, Produktion - Technik und Wirtschaft für die deutsche Industrie, last checked: 11/03/2023.

[42] O. Purwin, R. D'Andrea, and J.-W. Lee. Theory and implementation of path planning by negotiation for decentralized agents. *Robotics and Autonomous Systems*, 56(5):422–436, may 2008.

[43] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[44] E. Rimon, I. Kamon, and J. F. Canny. Local and Global Planning in Sensor Based Navigation of Mobile Robots. In Y. Shirai and S. Hirose, editors, *Robotics Research*, pages 112–123. Springer London, 1998.

[45] C. Rösmann, F. Hoffmann, and T. Bertram. Timed-Elastic-Bands for time-optimal point-to-point nonlinear model predictive control. In *2015 European Control Conference (ECC)*, pages 3352–3357, 2015.

[46] E. Salvato, G. Fenu, E. Medvet, and F. A. Pellegrino. Crossing the Reality Gap: A Survey on Sim-to-Real Transferability of Robot Controllers in Reinforcement Learning. *IEEE Access*, 9(1):153171–153187, 2021.

[47] M. Samà, A. D'Ariano, D. Pacciarelli, and F. Corman. Lower and upper bound algorithms for the real-time train scheduling and routing problem in a railway network. *IFAC-PapersOnLine*, 49(3):215–220, 2016. 14th International Federation of Automatic Control (IFAC) - Symposium on Control in Transportation Systems (CTS).

[48] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence*, 219(C):40–66, Feb. 2015.

[49] M. Shneier and R. Bostelman. Literature Review of Mobile Robots for Manufacturing. *NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD*, May 2015.

[50] T. Simeon, S. Leroy, and J.-P. Lauumond. Path coordination for multiple mobile robots: a resolution-complete algorithm. *IEEE Transactions on Robotics and Automation*, 18(1):42–49, 2002.

[51] C. Stachniss. *Robotic Mapping and Exploration*. Springer Tracts in Advanced Robotics. Springer Berlin Heidelberg, 2009.

[52] Süddeutsche Zeitung. "Audi testet Montage-Inseln als Ergänzung zum Fließband", July 2022. https://www.sueddeutsche.de/bayern/auto-ingolstadt-audi-testet-montage-inseln-als-ergaenzung-zum-fliessband-dpa.urn-newsml-dpa-com-20090101-220726-99-161346, dpa news channel, last checked: 11/03/2023.

[53] D. Sun, A. Kleiner, and B. Nebel. Behavior-based multi-robot collision avoidance. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1668–1673, May 2014.

[54] A. S. Tanenbaum and D. Wetherall. *Computer Networks*. Prentice Hall, Boston, 5 edition, 2011.

[55] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*, volume 3. Prentice Hall Englewood Cliffs, 2006.

[56] M. ten Hompel, H. Bayhan, J. Behling, L. Benkenstein, J. Emmerich, G. Follert, M. Grzenia, C. Hammermeister, H. Hasse, D. Hoening, et al. Technical Report: LoadRunner®, a new platform approach on collaborative logistics services. *Logistics Journal*, 2020(10), 2020.

[57] A. W. Ter Mors, J. Zutt, and C. Witteveen. Context-Aware Logistic Routing and Scheduling. In *17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 328–335, Jan. 2007.

[58] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics*. MIT Press, Cambridge, Mass., 2005.

[59] I. Ulrich and J. Borenstein. VFH+: reliable obstacle avoidance for fast mobile robots. In *IEEE Proceedings of the International Conference on Robotics and Automation (ICRA)*, volume 2, pages 1572–1577, 1998.

[60] T.-Y. Wang, H.-C. Lin, and K.-B. Wu. An improved simulated annealing for facility layout problems in cellular manufacturing systems. *Computers & Industrial Engineering*, 34(2):309–319, 1998.

[61] B. Wessling. "Mobile robot shipments increased by 53 % in 2022", Dec. 2022. https://www.therobotreport.com/mobile-robot-shipments-increased-by-53-in-2022/, The Robot Report, last checked: 13/03/2023.

[62] F. Xu, H. V. Brussel, M. Nuttin, and R. Moreas. Concepts for dynamic obstacle avoidance and their extended application in underground navigation. *Robotics and Autonomous Systems*, 42(1):1–15, 2003.

[63] J. Yu and S. M. LaValle. Planning optimal paths for multiple robots on graphs. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3612–3617, 2013.

[64] L. Zhao. Typical Failure Analysis and Processing of Belt Conveyor. *Procedia Engineering*, 26:942–946, 2011.

[65] X. Zhong, Y. Zhou, and H. Liu. Design and recognition of artificial landmarks for reliable indoor self-localization of mobile robots. *International Journal of Advanced Robotic Systems*, 14(1):1729881417693489, 2017.

[66] W. Zhou, J. Tian, L. Xue, M. Jiang, L. Deng, and J. Qin. Multi-periodic train timetabling using a period-type-based Lagrangian relaxation decomposition. *Transportation Research Part B: Methodological*, 105(C):144–173, 2017.

[67] W. Zhou, X. You, and W. Fan. A Mixed Integer Linear Programming Method for Simultaneous Multi-Periodic Train Timetabling and Routing on a High-Speed Rail Network. *Sustainability*, 12:1131, Feb. 2020.

Acronyms

- *ACK* acknowledgment 117, 120–123, 125, 128, 129, 133–136, 139, 140, 146, 148, 156, 160–162, 167, 189, 193, 198
- *AGV* Automated Guided Vehicle 1, 18, 19, 21
- *AMCL* Adaptive Monte Carlo Localization 29
- *AMR* Autonomous Mobile Robot 1
- *API* Application Programming Interface vi, viii, 8, 9, 186, 187
- *BDC* Backward Dependency Condition 91, 92, 95, 97–99, 101, 198
- *BGE* Blender Game Engine 31
- *CAD* computer-aided design 150
- *CADRL* Collision Avoidance with Deep RL 25
- *CAN* Controller Area Network 149
- *CBS* Conflict Based Search 18
- *CD* coordination diagram 20, 21
- *CL-RTT* Closed-Loop-RTT 24
- *CLPF* Collaborative Local Planning Framework v, vii, 6, 7, 9, 10, 15, 16, 109, 114–116, 118, 119, 124, 149–151, 154, 159, 160, 163, 165–168, 179, 184–186, 189–192, 197, 198
- *CPT* critical path time 81–88, 90, 92, 93, 95, 103, 105, 106, 188, 203

- *CPU* central processing unit 168, 179
- *CR* collision rectangle 68–72, 74–77, 79
- *CS* coordination space 7, 16, 20, 21, 45, 46, 65, 67–79, 197
- *CVG* Constrained Visibility Graph 71, 72, 75, 85, 197
- *DAG* directed acyclic graph 100, 175, 180
- *DFS* Depth-First Search 85, 88, 148, 198
- *DMA-RRT* Decentralized Multi-Agent Rapidly-exploring Random Tree 24
- *DOMPP* Distance Optimal Multi-robot Path Planning 17
- *DWA* Dynamic Window Approach 111, 184, 185, 191
- *EIF* Environment Interface 28, 29, 31–33, 184, 187
- *EPI* Experiment Interface 28, 29, 32, 33, 187
- *EPM* Environmental and Planning Model 15, 16, 190
- *ESV* Experiment Supervisor 28, 29, 187
- *FDC* Forward Dependency Condition 91, 92, 95–99
- *FoV* field of view 25, 112
- *FSM* finite state machine 117–119, 123, 125–127, 129, 131, 145, 150, 154, 158, 161, 189, 190, 198
- *ICSPS* Incremental Coordination-Space Path Scheduler v, viii, 7, 9, 15, 16, 46, 66–71, 73–88, 94, 103–107, 151, 167, 179, 188, 191, 192, 197, 198, 203
- *ILP* Integer Linear Programming 16, 17, 23
- *JSON* JavaScript Object Notation 186
- *LCT* Local Clearance Triangulation 149, 151
- *LP* Linear Programming 22, 23
- *MAPF* Multi-Agent Path-Finding 5, 13, 15–18, 22, 23, 25, 187
- *MGS* Merged Guarded Subpaths 47, 59–61, 64–68, 79, 81–83, 106, 188, 197, 203
- *MILP* Mixed Integer Linear Programming 16, 23

- *MORSE* Modular OpenRobots Simulation Engine 8, 28, 30, 31
- *NPH* next possible halt 117, 119, 125, 137–139, 142, 145, 154, 156–158, 190
- *OMRPS* Optimal Multi-Robot Path Scheduler v, viii, 7, 9, 15, 16, 88–90, 103, 105–107, 152, 188, 192, 198, 203
- *PRM* probabilistic roadmap 21
- *QLP* Quadratic Linear Programming 16, 18, 23
- *RAM* random access memory 65, 154, 179
- *REF* Robotic Experimentation Framework vi, viii, 8, 9, 28, 30, 33, 79, 179, 184–187, 191, 197
- *RL2* roslaunch2 9, 29, 32, 33, 185, 187
- *ROS* Robot Operating System vi, viii, 8–10, 27, 29, 114, 121, 123–125, 149, 170, 178, 184–187, 191–193
- *RoW* Right-of-Way v, vi, 7, 15, 16, 24, 36, 37, 42–45, 66–68, 70–73, 75, 76, 78, 79, 81, 82, 89–96, 98–101, 106, 113, 115–119, 123, 125, 126, 129, 131, 133, 136, 138, 139, 141, 143, 147, 149, 188, 189, 192
- *RPC* remote procedure call 141
- *RRT* Rapidly Exploring Random Trees 14, 24
- *RTF* real-time factor 153, 158, 168, 185, 192
- *SGS* Smallest Guarded Segments 47, 56–59, 63–68, 72, 79, 81–83, 106, 151, 188, 197, 203
- *TAOP* Token-Based Alternating Offers Protocol 25, 26
- *TCP* Transmission Control Protocol 5, 114, 120, 189
- *TEB* Timed Elastic Bands 184, 185, 191
- *TOMPP* Time Optimal Multi-robot Path Planning 17
- *TTP* Train Timetabling Problem 22, 23
- *TTT* total travel time 79, 81, 84, 85, 88, 90, 92, 93, 95, 106, 188, 203
- *UUID* Universally Unique Identifier 174, 177

- *VFH+* Vector Field Histogram+ 184, 185, 191
- *VMS* Vector Map Server 6, 149–151, 185, 186, 190
- *YAML* Yet Another Markup Language 31, 33
- *ZLP* zero-length path 115–117, 120, 122, 124, 127, 129–132, 135, 139, 140, 150, 151
- *ZPI* zero-length path intersection 124, 129–133, 150, 198