FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Multi-Agent Path Planning für Logistikroboter in ROS

Bacherlor-Thesis

vorgelegt von

Orchan Heupel

aus

Baku, Aserbaidschan

am

23. August 2023

Gutachter:
Prof. Dr. rer. nat. Alexander Ferrein
Dr. Stefan Schiffer

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Aachen, den 23. August 2023

Orchan Heupel

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Ziel der Arbeit 2

2 Stand der Forschung und Technik **3**

2.1 Multi-Agent Path Finding 3

2.2 Algorithmen für MAPF 6

2.3 Benchmarks 7

2.4 Robot Operating System 2 9

3 Simulationsprogramm **13**

3.1 Aufbau der Simulation 13

3.2 Ablauf der Simulation 15

3.3 Kommunikation unter ROS2 16

3.3.1 Kommunikation zwischen der Simulation und dem Controller . . 16

3.3.2 Kommunikation zwischen dem Controller und der Pfadplanung . 18

3.3.3 Kommunikation zwischen dem Controller und dem Auswerter . 20

3.4 Beschreibung der Module 21

3.4.1 Modul: Simulation 21

3.4.2 Modul: Controller 30

3.4.3 Modul: Pfadplanung 32

3.4.4 Modul: Auswerter 33

4 Integrierte Algorithmen **35**

4.1 Conflict Based Search 35

4.2 Priority Based Search 37

5 Experimentelle Ergebnisse **41**

5.1 Simulation mit 10 Robotern 43

5.2 Simulation mit 30 Robotern 45

5.3 Simulation mit 50 Robotern 47

5.4 Simulation mit 60-90 Robotern 48

5.5 Simulation mit 100 Robotern 51

5.6 Auswertung der Ergebnisse 53

6 Zusammenfassung **55**
6.1 Ausblick 56

Literatur **57**

Abbildungsverzeichnis

1.1 Beispielhafte Visualisierung der Simulation [5] 2

2.1 Illustration der Konflikte 4
2.2 Beispielhafte Aufführung der benötigten Informationen für Roboter [15] 8
2.3 Beispielhafte Darstellung von *Warehouse grid* für die Simulation MAPF [14] 8
2.4 Das Logo von ROS (l) und ein Bild eines PR2-Roboters (r) [11] 9
2.5 Das Ekosystem von ROS [10] 10
2.6 Das Logo der ROS2-Distribution *Foxy Fitzeroy* [9] 11

3.1 Aufbau des Frameworks 14
3.2 Ablauf eines vollständigen Simulationszyklus 15
3.3 Die Kommunikation zwischen der Simulation (mit zwei Robotern) und dem Controller unter ROS2 16
3.4 Die Definition der Nachricht *mapf_interfaces/MapInfo.msg* 17
3.5 Die Definition der Nachricht *mapf_interfaces/Position.msg* 17
3.6 Die Definition der Nachricht *mapf_interfaces/RobotInfo.msg* 17
3.7 Die Definition der Nachricht *mapf_interfaces/CheckpointInfo.msg* 18
3.8 Die Definition der Nachricht *mapf_interfaces/RobotAction.msg* 18
3.9 Die Kommunikation zwischen dem Controller und der Pfadplanung unter ROS2 18
3.10 Die Definition der Nachricht *mapf_interfaces/MAPFInstance.msg* 19
3.11 Die Definition der Nachricht *mapf_interfaces/MAPFSolution.msg* 19
3.12 Die Definition der Nachricht *mapf_interfaces/PathInfo.msg* 20
3.13 Die Definition der Nachricht *mapf_interfaces/StatisticInfo.msg* 20
3.14 Die Kommunikation zwischen dem Controller und dem Auswerter unter ROS2 20
3.15 Benutzeroberfläche der GUI 21
3.16 Der linke Bereich der GUI dient zum Interagieren mit dem Benutzer . . 22
3.17 Darstellung der Simulation in der Mitte der GUI 23
3.18 Der rechte Bereich der GUI gibt eine detaillierte Auskunft über die einzelnen Roboter wider 23
3.19 Das Verhalten der GUI modelliert 26

3.20 Übersicht über die einzelnen Komponenten und Klassen 27
3.21 Der Updateablauf des Moduls *Simulation* 29
3.22 Der Prozessablauf beim Controller 31
3.23 Darstellung der Schaubilder vom Auswerter 33

4.1 High Level Search beim CBS [12] 36
4.2 High Level Search beim PBS [7] 39

5.1 Die Darstellung der Umgebung für die Simulation eines Lagerhauses (*warehouse-10-20-10-2-1.map*) 41
5.2 Der Ausgangszustand der Simulation mit 10 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*) 43
5.3 Der Endzustand nach der Ausführung der Simulation mit 10 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) 43
5.4 Die Ergebnisse der Simulation mit 10 Robotern 44
5.5 Der Ausgangszustand der Simulation mit 30 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*) 45
5.6 Der Endzustand nach der Ausführung der Simulation mit 30 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) 45
5.7 Die Ergebnisse der Simulation mit 30 Robotern 46
5.8 Der Ausgangszustand der Simulation mit 50 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*) 47
5.9 Der Endzustand nach der Ausführung der Simulation mit 50 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) 47
5.10 Die Ergebnisse der Simulation mit 50 Robotern 48
5.11 Die Ergebnisse der Simulation mit 60 Robotern 48
5.12 Der Ausgangszustand der Simulation mit 70 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-4.scen*) 49
5.13 Der Endzustand nach der Ausführung der Simulation mit 70 Robotern (*warehouse-10-20-10-2-1-even-4.scen*) 49
5.14 Die Ergebnisse der Simulation mit 70 Robotern 50
5.15 Die Laufzeit von PBS und die Summen der Kosten für die ersten vier *scene*-Dateien bei einer Konfiguration mit 90 Robotern. 50
5.16 Der Ausgangszustand der Simulation mit 100 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*) 51
5.17 Der Endzustand nach der Ausführung der Simulation mit 100 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) 51
5.18 Die Ergebnisse der Simulation mit 100 Robotern aus zwei Versuchen . . 52
5.19 Die Ergebnisse der Simulation mit 100 Robotern aus drei Versuchen . . 52

Kapitel 1

Einleitung

In den letzten Jahren finden Logistikroboter immer breitere Anwendung in automatisierten Lagerhäusern, um den Warentransport zu beschleunigen. Eine typische Aufgabenstellung dabei ist: eine Gruppe aus mehreren Robotern soll sich ausgehend von ihren Startpunkten zu bestimmten Zielpositionen bewegen, Produkte an diesen Checkpoints aufsammeln und zur Abholstation transportieren. Die Pfade der einzelnen Agenten müssen dabei so geplant sein, dass sich zum einen keine Kollisionen zwischen den Robotern ereignen, zum anderen die Fahrtwege möglichst kurz sind. Die beschriebene Problemstellung ist allgemein bekannt als *Multi-Agent Path Finding* (MAPF). Es gibt verschiedene Varianten des beschriebenen Problems. Diese unterscheiden sich u.a. durch die festgelegten Annahmen bezüglich der erlaubten Bewegung der Roboter und die Definition der möglichen Konflikte, die zwischen den einzelnen Pfaden auftreten können. Automatisierte Logistikzentren sind nicht die einzige Anwendungsdomäne von MAPF. Andere Praxisbeispiele finden sich im Bereich des autonomen Fahrens, der Luftfahrtindustrie, der unbemannten Luftfahrzeuge und der Spielentwicklung [1, 13].

Der wachsende Einsatz von Robotern in der Industrie sowie in Logistikzentren zur Lösung komplexer Aufgaben hat dazu geführt, dass in den letzten Jahren umfangreiche Forschung auf dem Gebiet von MAPF betrieben wird. Standardalgorithmen zur Pfadplanung werden erweitert, um die kinematischen Einschränkungen des Roboters bei der Pfadermittlung zu berücksichtigen. Lösungen werden für verschiedene Anwendungsdomänen hinsichtlich bestimmter Parameter wie Optimalität, Berechnungsdauer spezifisch angepasst. [1, 13, 6]

Die Entwicklung von Robotern, Durchführung entsprechender Simulationen und Testen von Algorithmen erfordert bestimmte Werkzeuge. Dafür hat sich in der Forschung das Framework *Robot Operating System* (ROS) etabliert, welches in den letzten Jahren mehr Bedeutung in der Industrie gewinnt. ROS bietet Schnittstellen und Tools, um Software für die verschiedenen Komponenten eines Roboters getrennt voneinander zu entwickeln, zu testen, sowie die Kommunikation zwischen den einzelnen Roboterkomponenten zu regulieren. [8, 4]

Die bisher bekannten Simulationsprogramme aus dem Bereich von MAPF, bspw. *ASPRILO* [3, 14], setzten nicht auf ROS an. Das Vorhandensein einer Simulationsumgebung zur Ausführung und Visualisierung der Bewegung der Roboter mit unabhängiger Pfadplanung unter Nutzung der Schnittstellen von ROS zur Kommunikation zwischen den einzelnen Komponenten stellt eine weitere Möglichkeit zum Testen und Auswerten der Algorithmen als auch der Roboterbewegung für die Multi-Agent Pfadplanung dar.

1.1 Ziel der Arbeit

Das Ziel dieser Bachelorarbeit ist die Entwicklung einer Software zur Simulation der Pfadplanung von Multi-Agentensystemen für die klassische Problemstellung des MAPF, integrieren mehrerer Algorithmen in die erstellte Simulation und der Vergleich der Algorithmen unter verschiedenen Rahmenbedingungen und Konstellationen. Der Fokus der Applikation soll sich dabei insbesondere auf die Anwendung der Roboter in großen Logistikzentren richten. Entsprechende Benchmarks sollen zum Testen ausgewählt werden. Die Schwerpunkte der Arbeit bilden GUI-Programmierung in Kombination mit *Robot Operating System 2* (ROS2) und Pfadplanung für Multi-Agentensysteme (MAPF).

Der Aufbau des Programms soll modular gestaltet sein. Die Visualisierung der Simulation kann mit einer frei gewählten Grafikbibliothek erstellt werden. Das Problem der Pfadplanung kann mit Hilfe bekannter Algorithmen aus der Domäne des MAPF gelöst werden. Die Kommunikation zwischen den einzelnen Modulen und Robotern muss auf den Schnittstellen von ROS2 aufbauen.

Die einzelnen Aufgaben bei der Durchführung der Arbeit sind die Einarbeitung in das klassische Problem des *Multi-Agent Path Finding* sowie *Robot Operating System* (ROS2), Erstellen je eines Moduls zur Visualisierung der Simulation und zur Lösung des Problems der Pfadplanung und Aufbauen der Kommunikation zwischen den einzelnen Modulen unter ROS2. Am Ende können mehrere bekannte Algorithmen aus dem Gebiet von MAPF in die Simulation integriert, miteinander verglichen und die Ergebnisse dokumentiert werden.

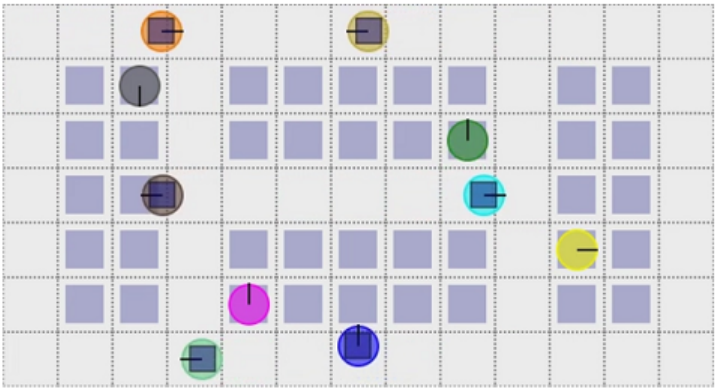

Abbildung 1.1: Beispielhafte Visualisierung der Simulation [5]

Kapitel 2

Stand der Forschung und Technik

Das Kapitel beginnt mit einer allgemeinen Einführung in das Thema *Multi-Agent Path Finding*. Zum besseren Verständnis der später beschrieben Zusammenhänge werden hier die wichtigen Definitionen und Grundbegriffe erklärt. Die wichtigsten Algorithmen bzw. deren Grundkonzepte zur Lösung der Pfadplanung sowie Vor- und Nachteile werden kurz erläutert. Zum Auswerten und Vergleich der Algorithmen aus dem Bereich von MAPF haben sich in der Forschung bestimmte Dateiformate für die Eingabe von Testdaten etabliert. Diese Dateiformate werden hier beschrieben. Als Hilfe bietet die Entwickler-Gemeinschaft den Nutzern Benchmarks für verschiedene Anwendungsdomänen an. Damit können Algorithmen mit zuvor getesteten Daten geprüft und mit bereits vorhandenen Ergebnissen aus anderen Studien verglichen werden. Das Kapitel wird mit einer kurzen Einführung in das ROS2 abgerundet.

2.1 Multi-Agent Path Finding

MAPF beschäftigt sich mit dem Problem der Suche nach gültigen Pfaden für mehrere Agenten so, dass jeder Agent seinen Zielpunkt möglichst schnell erreicht ohne dabei mit den anderen Agenten in Konflikte zu kommen [14, 13].

In der Literatur werden verschiedene Definitionen für das MAPF angegeben. Diese Arbeit beschränkt sich auf das klassische Problem des MAPF. Im Folgenden wird auf die Beschreibung des Problems eingegangen. Ein klassisches Problem für MAPF mit $k-$Agenten wird durch ein Tupel der Form $< G, s, t >$ beschrieben. Dabei steht $G = (V, E)$ für einen ungerichteten Graphen mit Knoten V und Kanten E. Die Knoten des Graphen beschreiben die möglichen Standorte der Agenten. Jede Kante $(n, n') \in E$ gibt an, dass der Agent sich vom Knoten n zum Knoten n' bewegen kann ohne dabei andere Knoten passieren zu müssen. Die Funktion $s : [1, ..., k] \rightarrow V$ weist jedem Agenten einen Startknoten zu. Entsprechend legt die Funktion $t : [1, ..., k] \rightarrow V$ für jeden Agenten einen Zielknoten fest. Die Zeit wird diskret betrachtet und zu jedem Zeitschritt kann der Agent nur eine einzelne Aktion ausführen. Die Aktion a wird hierbei durch die Funktion $a : V \rightarrow V$ definiert. Die Aktion $a(v) = v' \rightarrow V$ besagt,

dass der sich auf dem Knoten v befindende Agent durch das Ausführen der Aktion a zum Knoten v' bewegt wird. Agenten dürfen zwischen zwei Typen der Aktionen wählen: *wait* und *move*. Durch das Ausführen der Aktion *wait* verbleibt der Agent auf seinem aktuellen Ort bis zum nächsten Schritt. Mit der Aktion *move* wird der Agent zu einem Nachbarknoten bewegt. Ein *single agent plan* $\pi_i = (a_1, ..., a_n)$ setzt sich aus einer Sequenz von Aktionen zusammen, so dass der *Agent* i bei der Ausführung dieses Planes von seinem *Startknoten* $s(i)$ zu seinem *Zielknoten* $t(i)$ bewegt wird. Formal wird dies folgendermaßen ausgedrückt: $\pi_i[n] = a_n(....a1(s(i))) = t(i)$. Hierbei steht $\pi_i[n]$ für die Position des *Agent* i nach dem Ausführen der n Aktionen. Die Lösung des Problems ist ein Set aus $k-$Pfadplänen, jeweils ein Wegplan für jeden Agenten und wird allgemein als *solution* $\pi = \{\pi_0, \pi_1, ..., \pi_k\}$ bezeichnet. Die Lösung bekommt die Eigenschaft *valid*, wenn bei der Bewegung der Roboter durch die für sie jeweils berechneten Pfade keine Kollisionen zwischen den einzelnen Teilnehmern auftreten [14, 13].

Die Überprüfung auf mögliche Kollisionen zwischen den berechneten Pfaden erfolgt durch vordefinierte Konflikte. Im folgenden werden die vier wichtigsten Konflikte erläutert: *vertex conflict*, *edge conflict*, *following conflict* und *swapping conflict*. Ein *vertex conflict* entsteht wenn zwei Agenten den gleichen Knoten zu einem bestimmten Zeitpunkt zu besetzen versuchen. Ein *edge conflict* tritt auf, wenn zwei Agenten zum gleichen Zeitpunkt eine Kante in gleicher Richtung zu überqueren versuchen. Beim *following conflict* nimmt ein Agent eine bestimmte Position ein, die genau ein Schritt vorher von einem anderen Agenten besetzt war. Beim *swapping conflict* tauschen zwei Agenten ihre Positionen aus. Für das bessere Verständnis sind die genannten Konflikte in der Abb. 5.2 verdeutlicht [14, 13].

Um ein MAPF Problem vollständig zu beschreiben, muss vorher festgelegt werden, welche Typen von Konflikten in der Lösung erlaubt sind. In den meisten bisher bekannten Studien zum Thema MAFP werden folgende Konflikte in der Lösung nicht zugelassen: *vertex conflict*, *edge conflict* und *swapping conflict* [14].

In der Regel erreichen die Agenten ihre Ziele zu voneinander unterschiedlichen Zeitpunkten. Daher muss zur vollständigen Beschreibung des Problems das Verhalten der

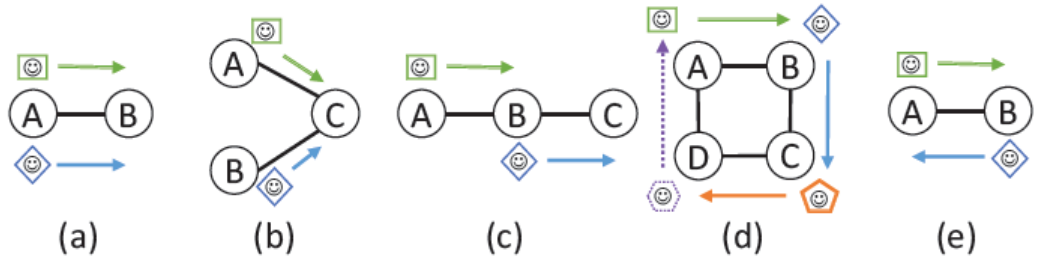

Abbildung 2.1: Illustration der Konflikte. Von links nach rechts: edge conflict, vertex conflict, following conflict, cycle conflict, swapping conclict [14]

Agenten vorgegeben werden, wenn diese an ihr Ziel ankommen und auf die anderen Agenten warten müssen. In den meisten Studien wird davon ausgegangen, dass der Agent auf seinem Zielknoten verbleibt. Dadurch kann dieser zu einem Hindernis für die anderen Roboter werden, die den Knoten passieren wollen. Anderseits kann die Annahme gewählt sein, dass der Agent nach dem Erreichen des Zielknotens verschwindet und aus dem Graphen entfernt wird [14].

Zu einem MAPF Problem können mehrere Lösungen existieren. Um die Lösungen zu evaluieren und zu vergleichen, werden Bewertungsfunktionen, in der Literatur als *objective function* bezeichnet, herangezogen. Die zwei bekannten Bewertungsfunktionen sind hierbei *makespan* und *sum of cost*. Die Funktion *makespan* (Gleichung 2.1) gibt die Anzahl der Zeitschritte an, die zum Erreichen der Zielpunkte für alle Agenten nötig sind. Die Summe der Zeitschritte, die alle Agenten zum Erreichen ihrer Zielpunkte ausgehend von deren Startpunkten brauchen, ergibt die Größe *sum of cost* (Gleichung 2.2) [14, 13].

$$makespan(\pi) = \max_{1 \leq i \leq k} |\pi_i| \tag{2.1}$$

$$SOC(\pi) = \sum_{1 \leq i \leq k} |\pi_i| \tag{2.2}$$

Single-agent shortest-path problem (SPP) beschreibt das Problem der Suche nach dem besten Pfad zwischen dem Startknoten $s \in V$ und dem Zielknoten $t \in V$ in einem Graph $G = (V, E)$. MAPF kann auf die Problembeschreibung von SPP reduziert werden. Dafür wird der Suchraum des Graphen G für $k-$Agenten erweitert. Daraus ergibt sich ein neuer Graph G_k. Die Knoten des Graphen G_k beschreiben die Vereinigungen der zulässigen Positionen für die $k-$Agenten. Jeder Agent wird im Knoten durch eine bestimmte Position, die dieser zu diesem Zeitpunkt belegen darf, wiedergegeben. Die Kanten des Graphen G_k beschreiben *joint actions*. Diese sind Sätze von Aktionen. Jedem Agenten wird eine bestimmte Aktion aus diesen Sätzen zugewiesen. Beim Durchqueren einer bestimmten Kante werden für jeden Roboter die ihm zugehörige Aktion simultan ausgeführt. Für die Sätze von Aktionen, die zu Konflikten zwischen den Agenten führen, werden keine Kanten im Graphen G_k erzeugt. Jeder Kante im Graphen erhält ein Gewicht. Dieser setzt sich aus der Summe der Kosten der zu dieser Kante zugehörigen Aktionen [13].

Neben der klassischen Definition des Problems bei MAPF gibt es noch andere Varianten der Problembeschreibung. Einige Beispiele werden der Vollständigkeit halber hier erwähnt. Agenten können bspw. neben nur einem mehrere Zielpunkte erhalten. Diese können je nach Anwendungsfall in vorgegebener oder beliebiger Reihenfolge erreicht werden. Beim *Anonymous MAPF* spielt es keine Rolle welcher Agent welchen Zielpunkt erreicht, solange alle Zielpunkte von Agenten besetzt werden. Beim *Lifelong MAPF* erhalten Agenten nach dem Ankommen am Zielpunkt eine neue zu erreichende

Position. Diese wird in der Regel aus einem Pool von Aufgaben ausgewählt und der aktuelle Ort des Roboters bei der Auswahl des nächsten Punktes berücksichtigt. Die Roboter befinden sich dadurch ständig in Bewegung. Anderseits kann für Agenten die Ausführung mehrere Aktionen pro Zeitschritt erlaubt werden. In manchen Problemstellungen wird die Zeit als kontinuierlich und nicht diskret betrachtet. Die Bewegung der Roboter kann durch kinematische Bedingungen, welche das reale Robotermodell abbilden sollen, beschränkt werden. Daneben existieren noch viele andere Varianten der Problembeschreibung beim MAPF. Hierbei sei auf weitere Literatur verwiesen [14].

2.2 Algorithmen für MAPF

Ein grundlegender Ansatz zum Lösen der Pfadplanung für eine Menge von $k-$Robotern besteht darin, die Aufgabe für jeden getrennt von und mit möglichst wenig Interaktion mit den anderen Robotern zu lösen. Dadurch versucht man das Problem der Pfadberechnung für $k-$Roboter auf die Pfadplanung für einen Roboter zu vereinfachen. Ein häufiges Vorgehen hierbei ist *Prioritized Planning*. Jedem Agenten wird im ersten Schritt eine eindeutige Priorität bzw. Nummer aus dem Pool von $\{1, \ldots .k\}$ zugewiesen. Im zweiten Schritt erfolgt die Pfadplanung für jeden Roboter der Reihe nach gemäß der festgelegten Priorität. Bei der Berechnung des Pfades wird darauf geachtet, dass der Agent nicht mit den zuvor berechneten Pfaden der anderen, der Priorität nach höher gesetzten Agenten kollidiert. Dieser Ansatz wird in der Praxis oft wegen seiner Einfachheit und geringer Laufzeit genutzt, jedoch ist er nicht vollständig und nicht optimal. Ein Algorithmus ist vollständig, wenn dieser immer eine Lösung findet falls diese existiert. Wenn die gefundene Lösung immer die günstigste ist, wird der Algorithmus als optimal betrachtet [13].

Viele MAPF Algorithmen wurden in letzter Zeit präsentiert. Die Beschreibung aller ist nicht das Ziel dieser Arbeit. Der Vollständigkeit halber werden hier nur wenige erwähnt. Bekannte Algorithmen sind *Conflict Based Search* (CBS), *Enhanced Conflict Based Search* (ECBS), *Priority Based Search* (PBS), *The Increasing Cost Tree Search*(ICTS) usw. [13, 6]

CBS ist ein häufig angewandter MAPF Algorithmus, der optimal und vollständig ist. Bei ihm erfolgt die Suche nach der Lösung auf zwei Ebenen. Auf der oberen Ebene (*high-level*) startet man mit dem *root-Knoten*, der die kürzesten Pfade für alle Agenten zu deren Zielpositionen enthält. Mögliche Kollisionen zwischen den einzelnen Pfaden werden bei der Berechnung der Wege hierbei nicht berücksichtigt. Die Pfade enthalten in der Regel Kollisionen. Der Algorithmus wählt eine der Kollisionen aus und löst sie auf. Dies erfolgt durch das Erzeugen von zwei Unterknoten für den ausgewählten Knoten, jeder mit einer weiteren Nebenbedingung, in der Literatur als *constraint* bezeichnet. Diese verbieten einem der beiden Agenten, die bei der ausgewählten Kollision beteiligt sind, zum Kollisionszeitpunkt am Kollisionsort zu sein. Anschließend erfolgt auf der unteren Ebene (*low-level*) eine neue Suche der Pfade für jeden Agenten unter

Berücksichtigung aller Nebenbedingungen des Knoten. CBS wiederholt diese Prozedur bis ein Knoten gefunden ist, der kollisionsfreie Pfade für alle Agenten enthält. Die Suche auf der oberen Ebene bzw. die Auswahl des nächstbesten Knotens sowie die Suche der Pfade auf der unteren Ebene erfolgt nach dem Prinzip von *best-first search* [13, 6].

ECBS ist eine Variante von CBS. Hier wird auf beiden Ebenen, zur Auswahl des nächsten Knotens auf der oberen Ebene sowie bei der Findung der Pfade auf der unteren Ebene, eine Modifikation des *A*-Algorithmus* angewandt. Charakteristisch für ECBS ist eine geringere Laufzeit gegenüber von CBS auf Kosten der nicht optimalen Lösung [6].

PBS kombiniert die Ansätze von CBS und *Prioritized Search*. Die Suche auf der oberen Ebene ähnelt dem von CBS. Im Unterschied zum CBS wird hierbei beim Auflösen eines Konfliktes bei den Unterknoten keine neue Nebenbedingung erzeugt, sondern die Prioritätenliste entsprechend aktualisiert. Einer der beiden am Konflikt beteiligten Agenten erhält eine höhere Priorität dem anderen gegenüber. Der Algorithmus beginnt beim *root-Knoten* mit einer leeren Prioritätenliste. Auf der unteren Ebene erfolgt die Suche der Pfade nach dem Prinzip von Prioritized Planning. PBS ist weder vollständig noch optimal, findet jedoch Lösungen viel schneller im Vergleich zu anderen Algorithmen [6].

2.3 Benchmarks

Um ein klassisches MAPF Problem vollständig zu beschreiben, müssen die Eingabedaten folgende Informationen enthalten: Beschreibung der Karte mit Hindernissen, Angaben über die Anfangspositionen der Roboter sowie deren Zielpunkte. In dieser Arbeit werden für die Eingabe von Daten die Dateiformate von Nathan R. Sturtevant benutzt. Deren vollständige Beschreibung findet man auf der folgenden Internetseite: `https://movingai.com/benchmarks/formats.html`

Die Daten werden über zwei verschiedene Dateien eingelesen: eine Datei beschreibt die Karte mit Hindernissen als Gitternetz, trägt die Endung .map und wird als *map file* bezeichnet, eine zweite Datei gibt u.a. die Informationen über die Anfangs- und Zielpositionen der Roboter jeweils als x- und y-Koordinaten an und trägt den Namen *scenario file*. Die *map*-Datei beginnt mit den folgenden Zeilen [15]:

```
type octile
height y
width x
map
```

In der zweiten und dritten Zeile werden mit x und y die Dimensionen der Umgebung aufgeführt. Danach folgen die Angaben über den Inhalt der Karte als ASCII Gitter [15].

Folgende Zeichen sind erlaubt:

```
. - passable terrain
G - passable terrain
@ - out of bounds
O - out of bounds
T - trees (unpassable)
S - swamp (passable from regular terrain)
W - water (traversable, but not passable from terrain)
```

Die erste Zeile des *scenario file* gibt mit dem folgenden Text die Version der Datei an: *version x.x.* Anschließend folgen zeilenweise die Informationen über die Roboter. Jede Zeile enthält neun Spalten. Der Inhalt der Spalten ist in der Abb. 5.3 beispielhaft [15].

Bucket	map	map width	map height	start x-coordinate	start y-coordinate	goal x-coordinate	goal y-coordinate	optimal length
0	maps/dao/arena.map	49	49	1	11	1	12	1
0	maps/dao/arena.map	49	49	1	13	4	12	3.41421

Abbildung 2.2: Beispielhafte Aufführung der benötigten Informationen für Roboter [15]

Roboter können zu Gruppen, hier auch *bucket* genannt, zusammengeführt werden. In der zweiten Spalte muss die zugehörige *map*-Datei angegeben sein. Zur Kontrolle werden in der dritten und vierten Spalte die Dimensionen der Umgebung beschrieben. Neben den Start- und Zielkoordinaten findet man in der letzten Spalte als Ergänzung die optimale Länge des Pfades vom Start zum Ziel dar. Für die Bewegung entlang der Diagonale werden dabei folgende Kosten angenommen: $sqrt(2) = 1.41421$ [15]

Auf der folgenden Seite findet man zahlreiche Benchmarks zum Thema MAPF: `https://movingai.com/benchmarks/mapf/index.html`. Die Benchmarks sind in verschiedene Gruppen entsprechend der Anwendungsdomäne, für die sie erstellt wurden, unterteilt. Von besonderem Interesse für diese Arbeit sind die Benchmarks aus dem Abschnitt *Warehouse grids*, siehe Abb. 5.4.

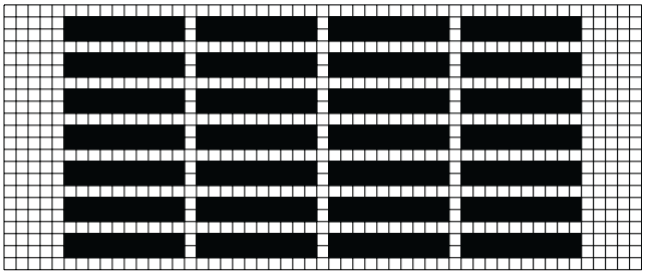

Abbildung 2.3: Beispielhafte Darstellung von *Warehouse grid* für die Simulation MAPF [14]

2.4 Robot Operating System 2

Robot Operating System (ROS) ist ein open-source Framework zur Entwicklung von Anwendungen und Software für den Bereich der Robotik. Es unterstützt die Entwickler durch die erforderliche Hardwareabstraktion, Gerätetreiber, Implementierung oft verwendeter Funktionalitäten und ein System zur Paketverwaltung. Der Datenaustausch und die Kommunikation zwischen den einzelnen Prozessen und Komponenten eines oder mehrerer Roboter wird durch das ROS-Framework ermöglicht. Es bietet Bibliotheken und Werkzeuge zum Erstellen, Programmieren und Starten von Programmen und Prozessen auf mehreren Computern. Die Hauptprogrammiersprachen sind C++ und Python. [2]

Die erste Version von ROS wurde im Jahr 2007 im Rahmen eines Projektes an der Stanford Universität entwickelt. Einer der Hauptmotive zur Entwicklung von ROS war die Programmierer bei der Implementierung von Robotik-Anwendungen zu unterstützen. Programme zum Steuern der Roboter setzten sich in der Regel aus ähnlichen Bausteinen zusammen. Ein wesentliches Hindernis für die Ingenieure zu diesem Zeitpunkt war, dass die erforderliche Grundinfrastruktur zum Entwickeln und Testen komplexer Algorithmen für die Roboter immer wieder neu implementiert werden musste. Diese Infrastruktur umfasste die Kommunikation zwischen den einzelnen Roboterbauteilen, Treiber für die Sensoren und Aktuatoren und weitere Aspekte. Um diese Hindernisse zu überbrücken, wurde ROS in der Version 1 (ROS1) ins Leben gerufen. Im Laufe der nachfolgenden Jahre wurde ROS1 ständig um weitere Funktionalitäten erweitert. Der Fokus bei der anfänglichen Entwicklung von ROS1 wurde auf die Anwendung des Systems für die *PR2-Roboter* (Abb. 5.5) gerichtet. Daraus ergaben sich bestimmte konzeptbedingte Einschränkungen. Zu diesen zählen bspw. der Fokus auf den Einsatz von ROS für die Ein-Roboter-Systeme, keine real-time Anforderungen an die Anwendungen, die Forderung nach guter Netzwerkverbindung, bordseitige Installation und Integration auf dem Roboter aller für die Berechnungen benötigten Ressourcen und Hardware usw. Des Weiteren wurde in den Anfängen der Entwicklung von ROS angenommen, dass dessen Nutzung vorwiegend für den akademischen und wissenschaftlichen Bereich gerichtet sei. Dadurch wurden bestimmte für die Industrie relevante Sicherheitsanforderungen bei der Entwicklung von ROS1 nicht ausreichend berücksichtigt [8, 4].

Abbildung 2.4: Das Logo von ROS (l) und ein Bild eines PR2-Roboters (r) [11]

Um die konzeptbedingten Nachteile von ROS1 zu kompensieren und den Einzug von ROS in die Industrie zu beschleunigen, wird seit 2014 ROS zusätzlich in der neuen Version 2 (ROS2) parallel zu ROS1 entwickelt. Die Vorteile von ROS2 gegenüber von ROS1 sind bessere Unterstützung für Multi-Roboter-Systeme und Echtzeitanwendungen, Multi-Plattform-Support, Beachtung bestimmter Sicherheitsaspekte beim Aufbau von Netzwerkverbindungen zwischen den einzelnen Prozessen usw. [8, 4]

Das Ekosystem von ROS stützt sich auf die vier folgenden Grundelemente ab: *plumbing*, *tools*, *capabilities* und *community* (Abb. 5.6) [10].

Der Aufbau der Kommunikation zwischen den verschiedenen Hardware- und Softwarekomponenten eines oder mehrerer Roboter gehört zu den grundlegenden Aufgaben bei der Lösung von Problemstellungen aus dem Bereich Robotik. Daher bildet den Kern von ROS ein Nachrichtenaustauschsystem zur Einrichtung der Kommunikation zwischen den einzelnen Prozessen und wird generell als *middleware* oder *plumbing* bezeichnet. Die zusammenhängenden Informationen werden zu Nachrichten (*messages*) gebündelt und über bestimmte Nachrichtenkanäle (*topics*) zwischen den einzelnen Prozessen *(nodes*) übertragen. Der Nachrichtenfluss wird als „anonymes Senden / Lauschen Verhaltensmuster *(publish/subscribe pattern*) umgesetzt. Prozesse erhalten Informationen über die von selbst abgesendeten und die ankommenden Nachrichten und keine Auskunft über die anderen am Datenaustausch beteiligten Prozesse. Dieser Ansatz führt zu klar definierter Aufgabentrennung zwischen den einzelnen Prozessen, eindeutigen Schnittstellen und letztendlich zu besseren Softwarelösungen. Darüber hinaus existieren bereits eine Menge vordefinierter Nachrichtentypen für die Schnittstellen verschiedener Anwendungsbereiche der Robotik, bspw. für den Datenaustausch der Lasersensoren und Kameras, diese können von Entwicklern genutzt werden. [10]

Um die Entwicklung von Anwendungen für die Robotik zu vereinfachen und zu beschleunigen, bietet das ROS-Framework verschiedene Werkzeuge (*tools*) an. Zu den Funktionen dieser Werkzeuge gehören bspw. das Visualisieren, Speichern und Abspielen von Daten (z.B. von Sensoren), Starten und Analysieren von Prozessen und viele andere Hilfestellungen. [10]

Einer der Vorteile von ROS ist die umfangreich zur Verfügung stehenden bereits implementierten Lösungen für bestimmte Problemstellungen aus dem Bereich der Robotik. Damit können sich Einsteiger oder erfahrene Entwickler auf ein spezifisches Problem konzentrieren, ohne alle Zusammenhänge und Abläufe der Anwendung verstehen zu müssen (*capabilities*). [10]

Abbildung 2.5: Das Ekosystem von ROS [10]

Die Entwickler-Gemeinschaft von ROS *(community)* umfasst internationale Unternehmen, Wissenschaftler, Studenten und Hobbyprogrammierer weltweit. [10]

ROS wird in verschiedenen Distributionen geliefert. In dieser Arbeit wird die ROS2 Distribution *Foxy Fitzeroy* in Kombination mit dem Betriebssystem Ubuntu Linux – *Focal Fossa* (20.04) eingesetzt.

Abbildung 2.6: Das Logo der ROS2-Distribution *Foxy Fitzeroy* [9]

Kapitel 3

Simulationsprogramm

Im Kapitel 3 wird die in dieser Arbeit entwickelte Simulation vorgestellt. Am Anfang wird der Aufbau und der Ablauf der Simulation erläutert. In den weiteren Unterkapiteln werden die einzelnen Module und die Kommunikation zwischen den einzelnen Komponenten detailliert dargestellt.

3.1 Aufbau der Simulation

Das Programm besteht aus den folgenden vier Komponenten: der Simulation, dem Controller, der Pfadplanung und dem Auswerter. Die Kommunikation zwischen den einzelnen Modulen erfolgt unter ROS2. Der Aufbau bzw. die Struktur der vollständigen Simulation sowie die Kommunikation zwischen den einzelnen Modulen ist in der Abb. 5.7 dargestellt.

Das Modul „Simulation“ führt die Visualisierung der Simulation aus und bietet dem Benutzer eine grafische Schnittstelle zur Interaktion mit dem Programm an. Zu den grundlegenden Funktionen gehören die Auswahl der Dateien zum Einlesen der Daten, Starten und Stoppen der Simulation sowie die Abbildung der Simulation zur visuellen Kontrolle aller Bewegungen der Roboter und der berechneten Pfade durch den Benutzer.

Das Modul „Controller“ kommuniziert mit den anderen Komponenten und dient als Schnittstelle zwischen der Simulation, der Pfadplanung und dem Auswerter. Nachdem die Simulation gestartet ist, sendet diese Nachrichten an den Controller. Diese Nachrichten enthalten Informationen über die Umgebung der Roboter sowie die Roboter selbst. Die Umgebung wird im Folgenden auch als die *map* und die Roboter als Agenten bezeichnet. Der Controller wartet bis alle benötigten Informationen vom Modul „Simulation“ übertragen sind, um daraus eine Instanz zu erzeugen. Die Instanz enthält alle Informationen, welche vom Modul „Pfadplanung“ bei der Lösung der Pfadberechnung erfordert werden. Nach dem Senden der erzeugten Instanz an die Pfadplanung wartet der Controller auf die entsprechende Antwort, welche die Lösung der Instanz enthält.

Das Modul „Pfadplanung“ löst die Instanz, indem es die gültigen Pfade für die einzelnen Agenten berechnet. Parallel werden Informationen über die beanspruchte Zeit des Algorithmus zur Berechnung und die Güte der Lösung erfasst. Nach dem Ermitteln der Lösung wird eine entsprechende Nachricht an den Controller zurückgesandt. Diese enthält neben den für die einzelnen Roboter berechneten Pfaden ergänzende Daten, bspw. die Dauer der Pfadermittlung und die Güte der Lösung.

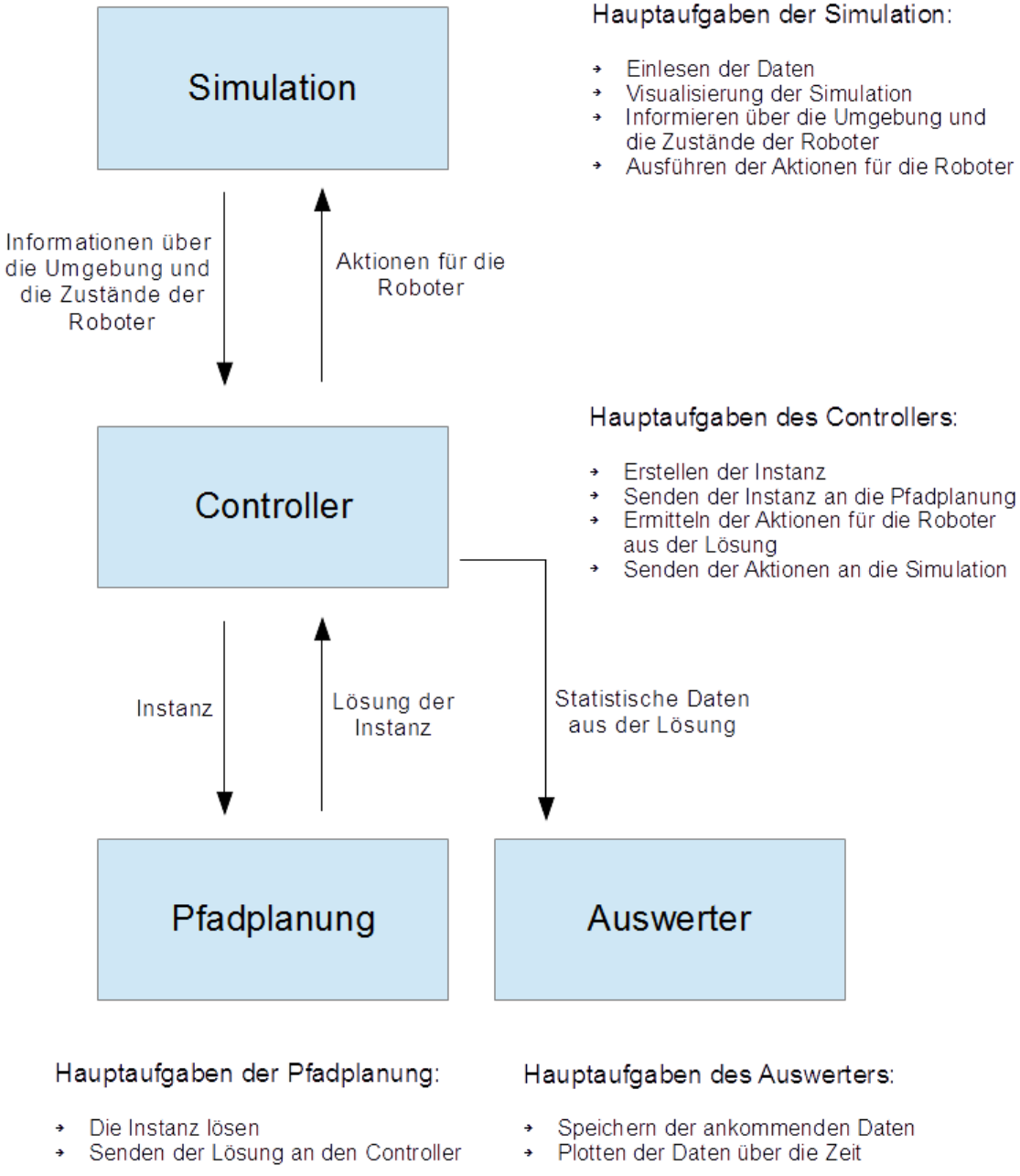

Abbildung 3.1: Aufbau des Frameworks

Der Controller erhält die Lösung der Pfadplanung und ermittelt aus den in der Lösung gespeicherten Pfaden die entsprechenden Aktionen für die einzelnen Roboter zum Folgen dieser Pfade für den nächsten Zeitschritt (step). Anschließend übermittelt der Controller die Aktionen an die Simulation. Die Simulation führt wiederum die Aktionen für die Roboter aus und informiert den Controller über die neuen Zustände der Roboter. Der Controller filtert beim Erhalt einer Lösung vom Pfadplaner die ergänzenden Informationen, bspw. die Laufzeit des Algorithmus und die Güte der Lösung aus der Nachricht heraus und sendet diese Daten unter Angabe des jeweiligen Zeitschrittes an das Modul „Auswerter“. Der Auswerter speichert die ankommenden Informationen und plottet diese über die Zeit. Dadurch können die verwendeten Algorithmen analysiert und miteinander verglichen werden.

3.2 Ablauf der Simulation

Im Folgenden wird der Ablauf der Simulation für einen kompletten Umlaufzyklus (step) erläutert. Zu Beginn jedes Zeitschrittes werden die Informationen über die Umgebung und die Zustände der Roboter vom Modul „Simulation“ an das Modul „Controller“ übermittelt. Wenn alle benötigten Daten an den Controller übertragen wurden, erstellt dieser im zweiten Schritt eine Instanz und sendet diese an den Pfadplaner. Im dritten Schritt wird die Instanz von dem Prozess der Pfadplanung gelöst und die Lösung an den Controller zurückgesandt. Im vierten Schritt erstellt der Controller aus der erhaltenen Lösung die passenden Aktionen für die Simulation, trennt die ergänzenden Informationen aus der Lösung für den Auswerter und sendet die jeweiligen Daten an die entsprechenden Module. Im letzten fünften Schritt führt die Simulation die vom Controller zuvor bestimmten Aktionen für die Roboter aus und der Auswerter speichert bzw. plottet die erhaltenen Daten in einen Graphen. Nach dem erfolgreichen Ausführen eines kompletten Umlaufes beginnt der Prozess erneut und der Zeitschritt (step) wird um eins erhöht. Der vollständige Ablauf ist in der Abb. 5.8 schematisch dargestellt.

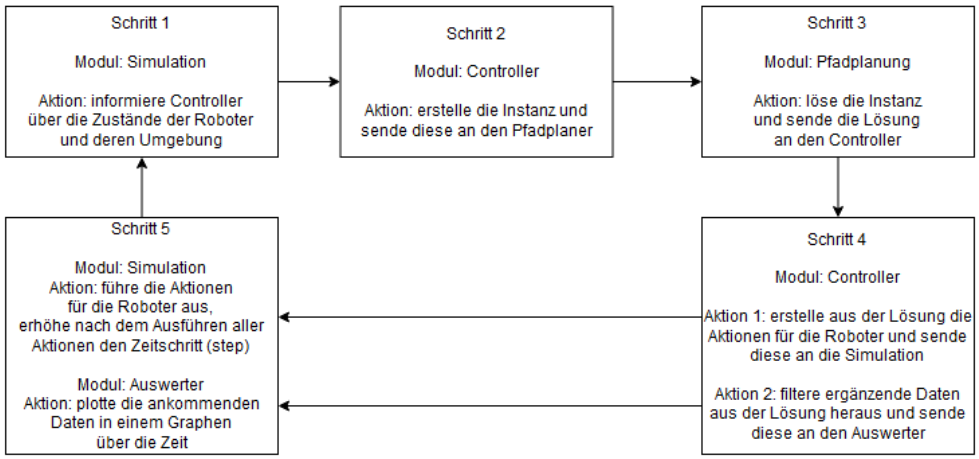

Abbildung 3.2: Ablauf eines vollständigen Simulationszyklus

3.3 Kommunikation unter ROS2

Der Datenaustausch unter den Komponenten der Simulation findet über die ROS2-Schnittstellen statt. In diesem Kapitel wird die Kommunikation zwischen den einzelnen Modulen separat betrachtet. Die einzelnen Prozesse (*nodes*), die Typen der vordefinierten Nachrichten *(message types*) sowie die eingestellten Nachrichtenkanäle (*topics*) werden vorgestellt. Alle die in dieser Arbeit eigenerstellten Nachrichtentypen sind im ROS2-package *mapf_interfaces* definiert.

3.3.1 Kommunikation zwischen der Simulation und dem Controller

Zuerst wird die Kommunikation zwischen der Simulation und dem Controller erläutert (Abb. 5.9). Die Prozesse werden in der Abbildung durch ovale Formen und die Nachrichtenkanäle durch Rechtecke dargestellt. Die Typen der Nachrichten zur Kommunikation auf jeweiligen Nachrichtenkanälen werden neben diesen, eingeschlossen in eckige Klammern, angegeben. Auf diese Darstellung des Datenaustausches zwischen den Prozessen unter ROS2 wird bei der Beschreibung der Kommunikation unter den anderen Modulen in den nächsten Kapiteln wieder zurückgegriffen.

Jeder Roboter wird in der Simulation durch einen eigenen Prozess (*node*) in ROS2 modelliert: *mapf_robot_ID_node*. Des Weiteren wird die Simulation selbst durch einen eigenen Prozess simuliert: *mapf_model_node*. Der Controller wird durch den Prozess *mapf_controller_node* ausgeführt. Die ID im Namen wird dabei durch die Roboter-ID ersetzt.

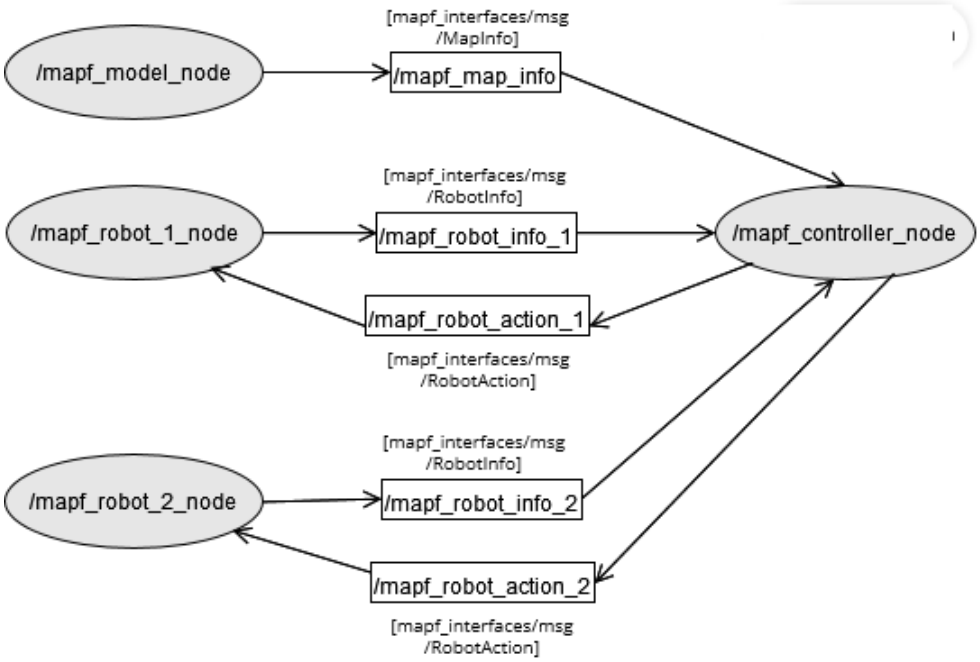

Abbildung 3.3: Die Kommunikation zwischen der Simulation (mit zwei Robotern) und dem Controller unter ROS2

Die Kommunikation zwischen *mapf_model_node* und *mapf_controller_node* erfolgt über den Kanal */mapf_map_info* [*mapf_interfaces/MapInfo.msg*]. Dabei werden folgende Informationen von der Simulation zum Controller übertragen: der momentane Zeitschritt, die aktuelle ID des Simulationszyklus und die Informationen über die *map*. Zu dem letzteren gehören die Größe der Umgebung, Positionen der statischen Hindernisse und die Anzahl der Roboter auf der *map*. Der Aufbau der Nachricht *mapf_interfaces/MapInfo.msg* ist in der Abb. 5.10 dargestellt.

```
int64 id
int64 step
int64 number_robots
int64 size_x
int64 size_y
mapf_interfaces/Position[] blocked
```

Abbildung 3.4: Die Definition der Nachricht *mapf_interfaces/MapInfo.msg*

Jede Position eines Objektes in der Umgebung wird durch die drei folgenden Parameter beschrieben: die x-Koordinate, die y-Kordintate und die Orientierung des Objektes. Zum Beschreiben der Position von Objekten wurde *mapf_interfaces/Position.msg* definiert und wird hier ergänzend aufgeführt (Abb. 5.11).

```
int64 pos_x
int64 pos_y
float64 theta
```

Abbildung 3.5: Die Definition der Nachricht *mapf_interfaces/Position.msg*

Jeder Roboter kommuniziert mit dem Controller selbst. Der Datenaustausch erfolgt über zwei, für jeden Roboter separate, Nachrichtenkanäle:

- */mapf_robot_info_*ID [*mapf_interfaces/RobotInfo.msg*]
- */mapf_robot_action_*ID [*mapf_interfaces/RobotAction.msg*]

Über den Nachrichtenkanal */mapf_robot_info_*ID informiert der Roboter den Controller über seinen aktuellen Zustand. Der Zustand eines Roboters wird durch die folgenden Informationen beschrieben: die aktuelle Position des Roboters sowie die Liste seiner Zielpunkte. Den vollständigen Aufbau der Nachricht sieht man in der Abb. 5.12.

```
int64 id
int64 step
int64 robot_id
int64 bucket
float64 optimal_length
mapf_interfaces/Position position
mapf_interfaces/CheckpointInfo[] checkpoints
```

Abbildung 3.6: Die Definition der Nachricht *mapf_interfaces/RobotInfo.msg*

Jeder Zielpunkt eines Roboters wird über eine eigene Nachricht des folgenden Typs beschrieben: *mapf_interfaces/CheckpointInfo.msg* (Abb. 5.13). Zu jedem Checkpoint

wird seine Position und die Information, ob dieser Zielpunkt schon besucht wurde, gespeichert .

```
bool visited
mapf_interfaces/Position position
```

Abbildung 3.7: Die Definition der Nachricht *mapf_interfaces/CheckpointInfo.msg*

Über den Nachrichtenkanal */mapf_robot_action_*ID übermittelt der Controller die nächste auszuführende Aktion an den Roboter in der Simulation. Jede Aktion umfasst folgende Angaben: den Weg, um den sich der Roboter in x- und y-Richtungen bewegt und den Winkel, um den der Roboter gedreht werden soll. Um den kompletten von der Pfadplanung geplanten Weg für den Roboter in der Simulation einzuzeichnen, wird zusätzlich dieser Pfad der Nachricht hinzugefügt. Der vollständige Aufbau der Nachricht *mapf_interfaces/RobotAction.msg* ist in der Abb. 5.14 dargestellt.

```
int64 id
int64 step
int64 robot_id
int16 move_x
int16 move_y
float64 rotate
mapf_interfaces/PathInfo path
```

Abbildung 3.8: Die Definition der Nachricht *mapf_interfaces/RobotAction.msg*

3.3.2 Kommunikation zwischen dem Controller und der Pfadplanung

Die Kommunikation zwischen dem Controller und der Pfadplanung ist in der Abb. 5.15 beschrieben.

Der Pfadplaner wird als ein eigener Prozess (*node*) ausgeführt. Der Name des Prozesses ist nicht festgelegt und kann verschiedene Namen tragen. Hier heißt der Prozess: *mapf_planner_node*.

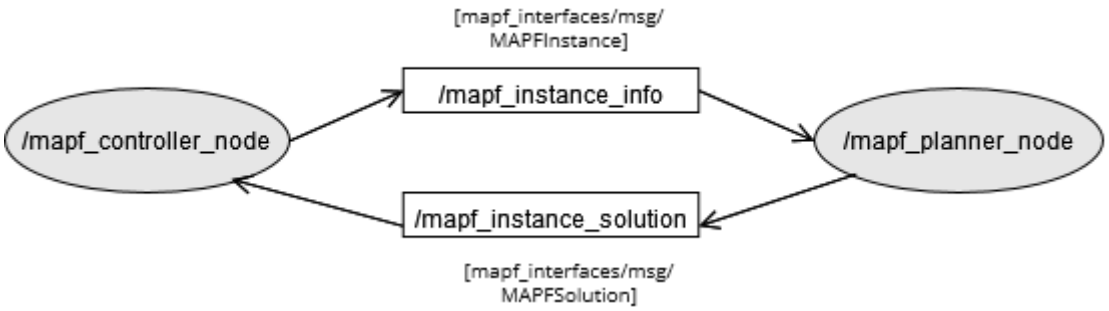

Abbildung 3.9: Die Kommunikation zwischen dem Controller und der Pfadplanung unter ROS2

Alternativ könnte der Prozess bspw. folgende Namen tragen:

- *mapf_planner_pbs_node*
- *mapf_planner_cbs_node*
- *mapf_planner_ecbs_node*
- *mapf_planner_icts_node*

Der Datenaustausch erfolgt hier über zwei Nachrichtenkanäle:

- */mapf_instance_info* [*mapf_interfaces/MAPFInstance.msg*]
- */mapf_instance_solution* [*mapf_interfaces/MAPFSolution.msg*]

Der Controller erzeugt aus den ankommenden Daten eine Instanz. Diese Instanz enthält die Informationen über die Umgebung sowie die Daten über die Zustände aller Roboter [*mapf_interfaces/MAPFInstance.msg*]. Anschließend sendet der Controller die erzeugte Instanz über den Nachrichtenkanal */mapf_instance_info* an den Solver (Abb. 5.16).

Der Prozess der Pfadplanung berechnet aus der erhaltenen Instanz die Pfade, speichert die Lösung in einer Nachricht [*mapf_interfaces/MAPFSolution.msg*] und sendet diese an den Controller über */mapf_instance_solution* zurück. Die Lösung enthält folgende Informationen: die einzelnen Pfade für die Roboter, den Namen des bei der Pfadermittlung verwendeten Algorithmus und die statistische Daten zur Auswertung des Algorithmus (Abb. 5.17).

```
int64 id
int64 step
mapf_interfaces/MapInfo map
mapf_interfaces/RobotInfo[] robots
```

Abbildung 3.10: Die Definition der Nachricht *mapf_interfaces/MAPFInstance.msg*

```
int64 id
int64 step
string name_algorithm
mapf_interfaces/PathInfo[] paths
mapf_interfaces/StatisticInfo statistics
```

Abbildung 3.11: Die Definition der Nachricht *mapf_interfaces/MAPFSolution.msg*

Jeder Pfad [*mapf_interfaces/PathInfo.msg*] wird durch ein oder mehrere Positionen *mapf_interfaces/Position.msg* definiert. Des Weiteren erhält die Nachricht zur Beschreibung eines bestimmten Pfades die ID des Roboters, für den dieser Pfad berechnet wurde (Abb. 5.18).

Die statistischen Daten können folgende Informationen umfassen: beanspruchte Zeit des Algorithmus zur Berechnung der Lösung, die Güte der Lösung und die Anzahl

```
int64 step
int64 robot_id
mapf_interfaces/Position[] path
```

Abbildung 3.12: Die Definition der Nachricht *mapf_interfaces/PathInfo.msg*

generierter und expandierter Knoten beim *highlevel-* und *lowlevel-search* (Abb. 5.19). Später können weitere Felder hinzuaddiert werden.

```
int64 step
int64 highlevel_generated_nodes
int64 highlevel_expanded_nodes
int64 lowlevel_generated_nodes
int64 lowlevel_expanded_nodes
string name_algorithm
float64 sum_of_cost
float64 runtime
```

Abbildung 3.13: Die Definition der Nachricht *mapf_interfaces/StatisticInfo.msg*

3.3.3 Kommunikation zwischen dem Controller und dem Auswerter

Die Kommunikation zwischen dem Controller und dem Auswerter ist in der Abb. 3.14 beschrieben.

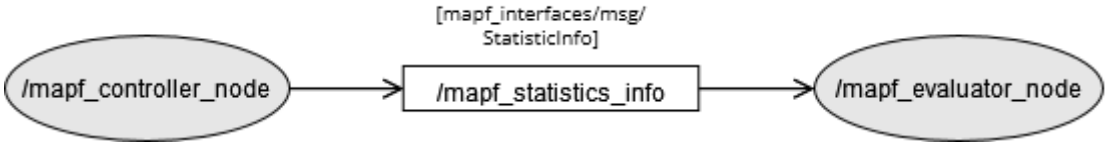

Abbildung 3.14: Die Kommunikation zwischen dem Controller und dem Auswerter unter ROS2

Der Auswerter läuft als ein eigener Prozess: *mapf_evaluator_node*. Jedes Mal wenn der Controller eine Lösung von der Pfadplanung erhält, werden die ergänzenden Informationen aus der Lösung herausgefiltert und an den Auswerter über den Nachrichtenkanal */mapf_statistics_info* als Nachricht des Typs *mapf_interfaces/StatisticInfo.msg* gesendet.

3.4 Beschreibung der Module

3.4.1 Modul: Simulation

Das Modul *Simulation* befindet sich im ROS2-package *mapf_simulation* und wird aus der Kommandozeile mit dem folgenden Befehl gestartet:

ros2 run mapf_simulation start_gui

Alternativ kann die Simulation durch das Ausführen der Datei *__main__.py* im ROS2-package *mapf_simulation* initiiert werden. Nach dem Ausführen des Befehls erscheint ein Fenster zum Interagieren mit der Simulation (Abb. 3.15). Die Benutzeroberfläche der GUI setzt sich aus den drei folgenden Bereichen zusammen: ein Widget zur Interaktion mit dem Benutzer (links), eine Fläche zur Visualisierung der Simulation (mittig) und ein Widget zum Anzeigen detaillierter Informationen über die einzelnen Roboter (rechts). Die einzelnen Bereiche lassen sich durch entsprechende Schiebeelemente vergrößern bzw. verkleinern oder durch Ziehen in ein separates Fenster verschieben. Im Folgenden werden die einzelnen Widgets ausführlicher beschrieben.

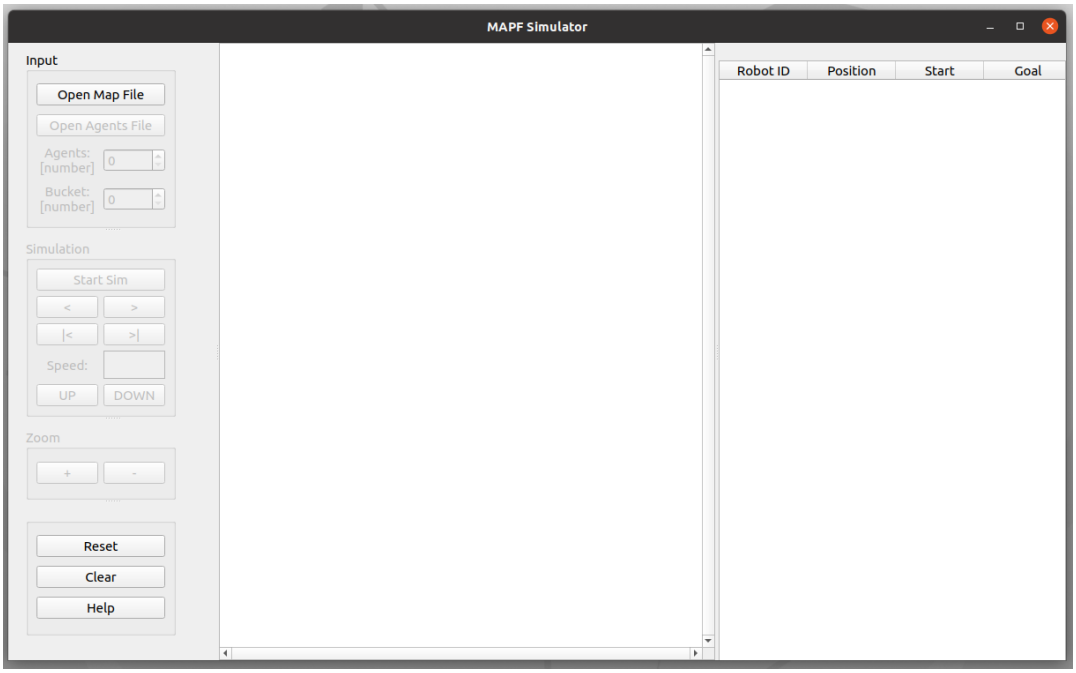

Abbildung 3.15: Benutzeroberfläche der GUI

Das linke Widget bietet dem Benutzer die Funktionen für das Einlesen von Daten, die Auswahl der Anzahl der Roboter zum Simulieren, das Starten und Stoppen der Simulation, die Navigation zwischen den einzelnen Zeitschritten, das Zurücksetzen der Daten auf die Anfangswerte, das Löschen der eingelesenen Daten, das Heran- und Herauszoomen der Ansicht der Simulation und die Hilfestellung bei Fragen über den Umgang mit dem Programm (Abb. 3.16).

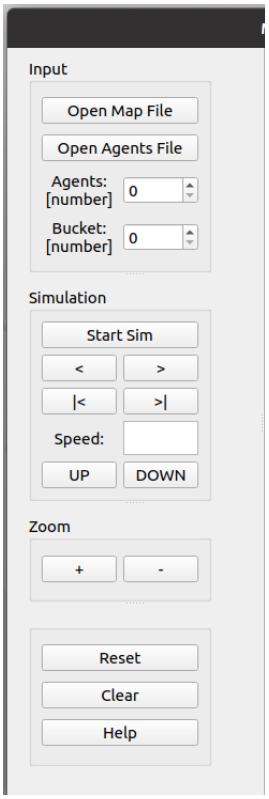

Abbildung 3.16: Der linke Bereich der GUI dient zum Interagieren mit dem Benutzer

In der Mitte des Fensters wird die eigentliche Simulation visualisiert. Die Umgebung wird durch ein rechteckiges diskretes Gitternetz (engl. *grid map*) mit gleich großen quadratischen Elementen beschrieben. Die freien Positionen der Umgebung werden in der Simulation hell und die unpassierbaren Positionen dunkel gezeichnet. Jeder Roboter wird mit einem Kreis und einem kurzen Strich dargestellt. Der Stricht zeigt die Ausrichtung des Roboters an. Wenn der Roboter sich nicht im Stehen dreht, gibt der Strich auch die Richtung, in die sich der Roboter zuletzt bewegt hat, an. Die Zielpunkte (engl. *checkpoints*) werden durch leere rote Rechtecke visualisiert. Beim Auswählen eines Roboters werden die ihm zugehörigen Zielpunkte in der Simulation hervorgehoben dargestellt. Nach dem Starten der Simulation können zusätzlich die berechneten Pfade für jeden Roboter visualisiert werden. Bei entsprechend vergrößerter Ansicht werden die Nummern der Roboter eingeblendet (Abb. 3.17).

Im rechten Widget werden die Informationen über die einzelnen Roboter in Tabellenform detailliert dargestellt. Beim Auswählen eines Roboters in der Simulation wird zusätzlich die entsprechende Zeile in der Tabelle markiert. Die angezeigten Informationen umfassen die Nummern der Roboter, deren Start- und Zielpunkte, die momentanen Positionen und den aktuellen Zeitschritt. Darüber hinaus wird in jeder Zeile die beim Einlesen der Daten gespeicherten ergänzenden Angaben zu jedem Roboter wie die Nummer der Gruppe (Bucket) und die optimale Länge vom Start- zum Zielpunkt angezeigt (Abb. 3.18).

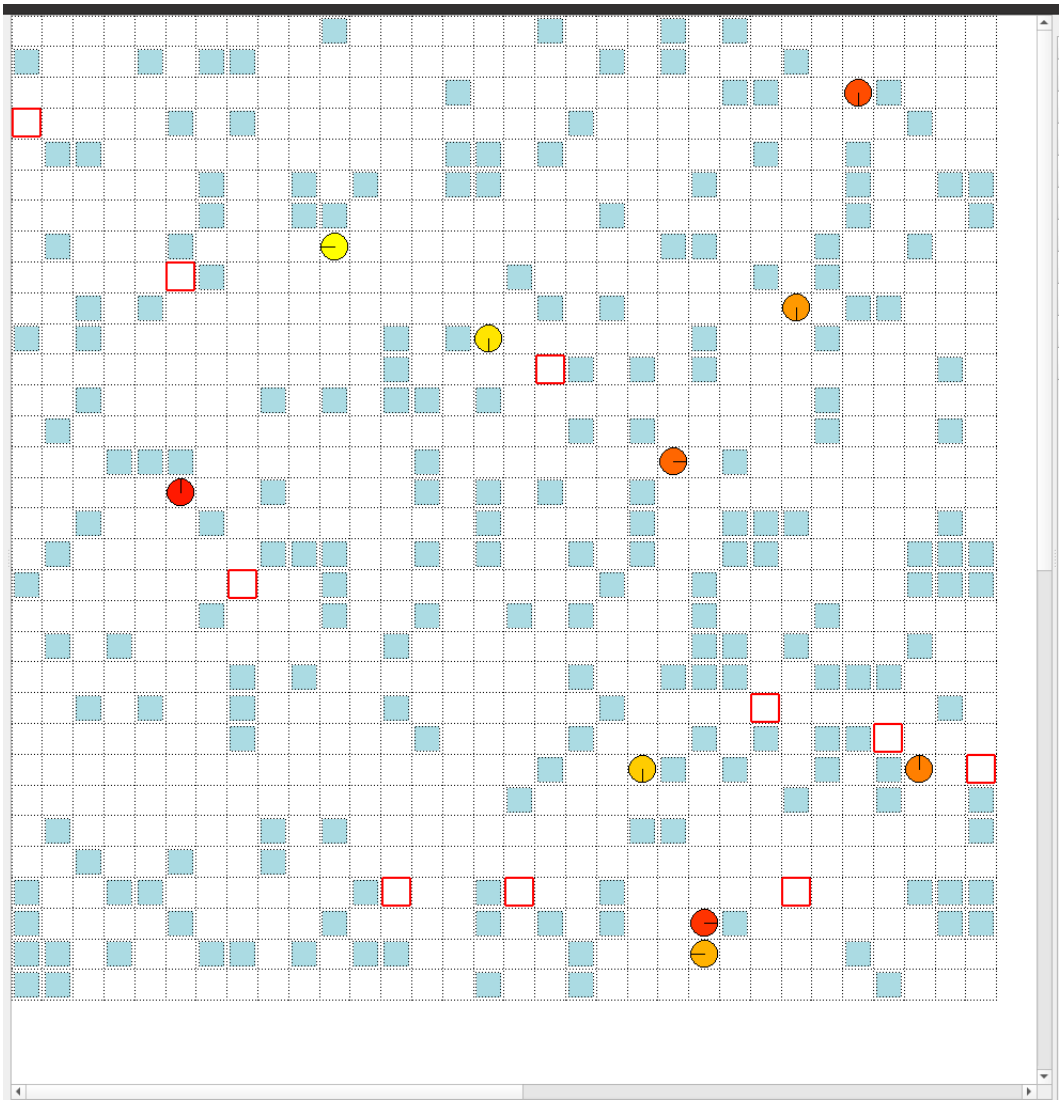

Abbildung 3.17: Darstellung der Simulation in der Mitte der GUI

	Robot ID	Position	Start	Goal	Step	Last action	Heuristics	Bucket
1	1	6, 17	6, 17	32, 25			31.3137085	1
2	2	22, 30	22, 30	25, 23			10.24264069	1
3	3	28, 2	28, 2	29, 24			27.48528137	1
4	4	21, 15	21, 15	17, 29			17.07106781	1
5	5	30, 26	30, 26	8, 19			27.48528137	1
6	6	26, 9	26, 9	6, 9			22.82842712	1
7	7	24, 31	24, 31	13, 29			13.24264069	1
8	8	21, 24	21, 24	26, 29			8.24264069	1
9	9	16, 10	16, 10	18, 12			2.82842712	1
10	10	12, 8	12, 8	1, 4			13.82842712	1

Abbildung 3.18: Der rechte Bereich der GUI gibt eine detaillierte Auskunft über die einzelnen Roboter wider

Hauptfunktionen der GUI

Im Folgenden werden die Hauptfunktionen der GUI erläutert. Beim Betätigen der Schaltfläche *Open Map File* öffnet sich ein Dialog-Fenster, über das die *map*-Datei zum Einlesen ausgewählt werden kann. Standardmäßig werden nur Dateien mit der Endung *.map* zur Auswahl aufgelistet. Das richtige Format dieser Dateien wurde im Kapitel 2.3 ausführlich beschrieben. Nach der Auswahl der Datei überprüft das Programm zuerst den Inhalt der Datei auf das richtige Format und lädt erst bei positiver Prüfung die jeweiligen Daten in die Simulation. Bei negativer Prüfung werden keine Daten in die Simulation geladen und die zuvor gespeicherten Informationen bleiben erhalten. Nach dem erfolgreichen Laden der Datei wird die neue Umgebung in der Mitte des Fensters angezeigt. Die alten Daten gehen dabei verloren.

Nachdem die Umgebung geladen wurde, können der Simulation Roboter hinzufügt werden. Dies erfolgt durch das Laden einer entsprechenden *scene*-Datei über das Drücken auf die Taste *Open Agents File*. Es öffnet sich ein Dialog-Fenster, um die Datei auszuwählen. Die *scene*-Dateien tragen die Endung *.scen* und wurden ebenfalls im Kapitel 2.3 detailliert besprochen. Das Dialog-Fenster bietet unten rechts die optionale Funktion, um den Typ der angezeigten Dateien nach *Scene files* zu filtern. Nach der Auswahl der Datei wird wiederum der Inhalt der ausgewählten Datei auf das richtige Format und auf die Konsistenz der Daten untersucht. Erst bei positiver Prüfung werden die Daten in die Simulation geladen. Nach dem erfolgreichen Laden der Datei werden die Roboter samt der Checkpoints in der Simulation visualisiert und der Inhalt der Robotertabelle im rechten Widget mit Inhalt gefüllt. Die zuvor gespeicherten Daten werden gelöscht.

Die Roboter können bei den *scene*-Dateien in Gruppen (engl. *bucket*) aufgeteilt werden. Somit kann beim Laden einer Datei zwischen verschiedenen Konfigurationen gewechselt werden. Um eine bestimmte Gruppe oder mehrere Gruppen der Roboter beim nächsten Öffnen einer *scene*-Datei zu laden, bietet die GUI ein entsprechendes Eingabefeld an. Das Eingabefeld wird mit einem links davon befindlichen Textfeld des Inhalts *Bucket:* hervorgehoben. Die Auswahl der Gruppe muss vor dem Laden der Datei erfolgen. Standardmäßig wird die Gruppe mit der Nummer 0 ausgewählt. Üblicherweise werden alle Agenten dieser Gruppe zugewiesen, falls keine Aufteilung der Roboter vorgesehen ist.

Die *scene files* können sehr viele Daten enthalten. Manchmal ist es erwünscht die Simulation mit einer bestimmten Menge an Robotern durchzuführen. Für diesen Fall bietet die GUI ein entsprechendes Eingabefeld für die Festlegung der Obergrenze der zu ladenden Roboter an. Das Eingabefeld wird mit einem links davon befindlichen Textfeld des Inhalts *Max. Number Agents* kennzeichnet. Beim Ändern des Limits wird die Menge der angezeigten Roboter im Simulationswidget und der Robotertabelle angepasst.

Nach dem Einlesen aller benötigten Daten kann die Simulation gestartet werden. Dies erfolgt durch das Betätigen der Schaltfläche *Start Sim.* Durch ein erneutes Betätigen

derselben Taste wird die Simulation gestoppt. Nach dem Starten der Simulation werden Daten über die dafür vorgesehenen Kanäle an den Controller gesendet und auf die entsprechenden Antworten gewartet. Nach dem Empfangen der zurückgesandten Daten werden die Roboter in der Simulation entsprechend den erhaltenen Aktionen bewegt und anschließend die neuen Zustände der Roboter über die gleichen zuvor definierten Kanäle an den Controller übertragen. Auf die detaillierte Beschreibung der Kommunikation zwischen der Simulation und dem Controller wurde im Kapitel 3.3.1 eingegangen.

Benutzung der GUI

In diesem Unterkapitel wird für den Benutzer die korrekte Anwendung des Programms beschrieben. Nach dem Erscheinen der GUI wird im ersten Schritt die *map*-Datei geladen (Taste *Open Map File*). Im zweiten Schritt wird die *scene*-Datei geladen (Taste *Open Agents File*). Optional kann vorher die Obergrenze zum Laden der Agenten festgelegt und die Gruppe der Roboter (*Bucket*) ausgewählt werden. Nach dem Laden der beiden Dateien kann im dritten Schritt die Simulation gestartet werden (Taste *Start Sim*). Zum Stoppen der Simulation betätigt man den gleichen Button.

In der Abb. 3.19 wird das Verhalten der GUI modelliert. Es werden nur die Hauptfunktionen dargestellt. Die optionalen Schritte werden aus Platzgründen zu Gunsten der Übersichtlichkeit nicht dargestellt.

Implementierung der GUI und der Simulation

Die Simulation ist vollständig mit *Python 3.8* implementiert. Die graphische Oberfläche wurde mit der Bibliothek *PyQt5* erzeugt. Bei der Implementierung wurden bestimmte Teile des Simulationsprogramms *ASPRILO* [3, 14] übernommen und weiter angepasst. Die einzelnen Komponenten des Moduls, deren Anordnung untereinander und die Verknüpfungen mit den anderen Klassen zur Modellierung der Simulation sind schematisch in der Abb. 3.20 dargestellt.

Die graphische Oberfläche der GUI besteht aus einem Hauptelement (*CentralWidget*), in das die restlichen Komponenten (*Widgets*) eingefügt werden. Die Anordnung der Elemente erfolgt hier über die Klasse *QSplitter*. Dies erlaubt dem Anwender die Größe der einzelnen Komponenten durch gegenseitiges Verschieben zu verändern.

Zum Abspeichern der Simulationsdaten wird eine Instanz der Klasse *Model* benutzt. Zum Simulieren der Roboter dient die Klasse *RobotROS*. Im Model wird für jeden einzelnen Roboter eine eigene Instanz der Klasse *RobotROS* erzeugt und gespeichert. Jeder Roboter bekommt eine oder mehrere Aufgaben (*TaskROS*) zugewiesen. Jede Aufgabe setzt sich aus einem oder mehreren Zielpunkten (*CheckpointROS*) zusammen. Um die Roboter und Checkpoints in der Simulation graphisch darzustellen, werden die entsprechenden Klassen von der übergeordneten Klasse *VisualizerGraphicItem* abgeleitet.

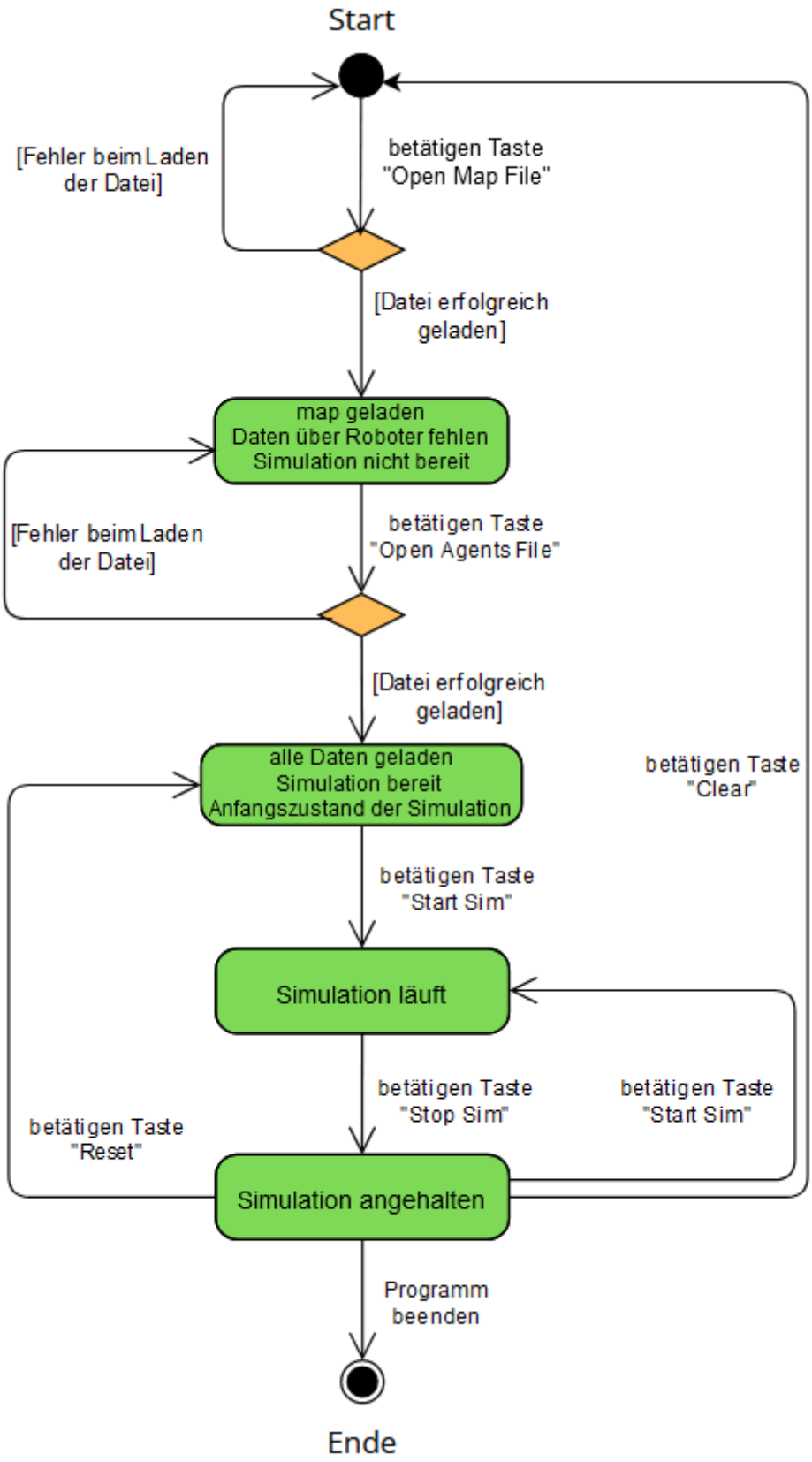

Abbildung 3.19: Das Verhalten der GUI modelliert

Um die Daten zwischen dem Model und der GUI auszutauschen, wird das Model mit den einzelnen Komponenten der GUI verknüpft.

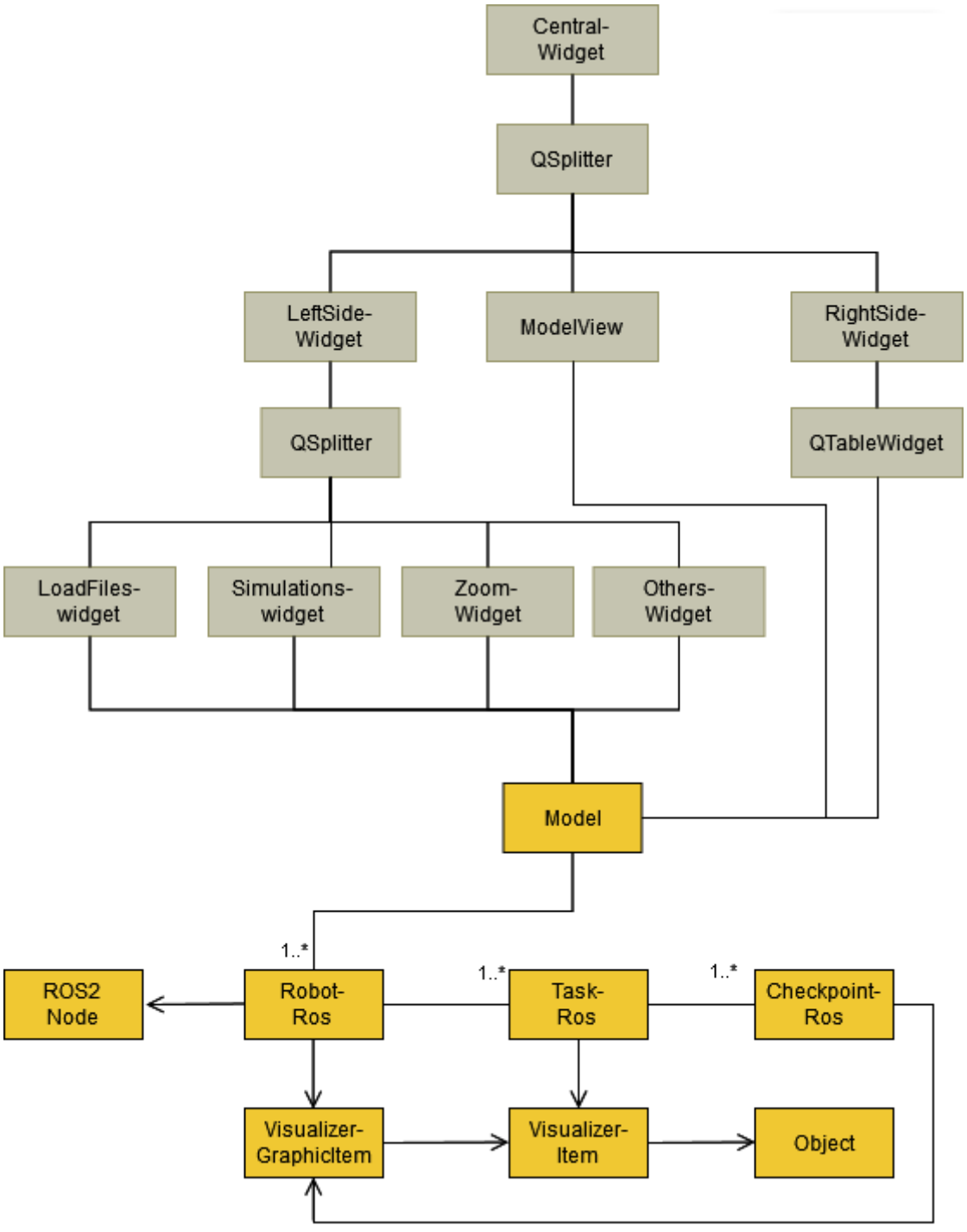

Abbildung 3.20: Übersicht über die einzelnen Komponenten und Klassen. Die graphischen Komponenten der GUI sind mit grau kennzeichnet. Die Klassen zur Modellierung der Simulation sind gelb dargestellt.

Updateablauf des Moduls Simulation

Im Folgenden wird der Updateablauf des Moduls *Simulation* für einen Zyklus beschrieben. Zu Anfang jedes neuen Zeitschrittes (*step*) werden die Informationen über die Simulation als ROS2-Nachrichten an die vordefinierten Kanäle übertragen. Für die Umgebung wird eine einzelne [*MapInfo.msg*] und für jeden Roboter eine separate Nachricht [*RobotInfo.msg*] erzeugt. Dieser Modus wird im Programm mit PUBLISH bezeichnet. Um die Zugehörigkeit der Nachrichten zu bestimmen, erhalten diese eine gleiche ID. Beim nächsten Ausführen des Modus PUBLISH wird die ID um eins erhöht.

Nach dem Modus PUBLISH wird in den Modus WAIT gewechselt. Hier wartet das Programm auf die entsprechenden Antworten von außerhalb. Das Programm befindet sich so lange im Modus WAIT bis für jeden Roboter eine entsprechende Nachricht *[RobotAction.msg*] mit der richtigen ID ankommt. Nachrichten mit einer älteren ID werden ignoriert. Falls innerhalb einer bestimmten Zeitdauer nicht alle erforderten Nachrichten ankommen, wechselt das Programm in den Modus PUBLISH zurück und übermittelt die gleichen Daten mit einer inkrementierten ID erneut. Das Time-out ist im Programm auf 30 Sekunden voreingestellt. Nach dem Erhalten aller benötigten Informationen wechselt das Programm in den Modus EXEC.

Im Modus EXEC werden die einzelnen Aktionen für die Roboter ausgeführt. Die Ausführung der Aktionen erfolgt gemäß den in den erhaltenen Nachrichten gespeicherten Informationen [*RobotAction.msg*]. Diese Informationen beschreiben, wie der Roboter bewegt oder gedreht werden soll. Die alten Positionen der Roboter werden schrittweise verändert und die Abbildung der Simulation aktualisiert. Ein vollständiger Schritt setzt sich durch Voreinstellung aus zehn einzelnen Zwischenschritten zusammen. Nach dem erfolgreichen Ausführen der Aktionen speichert das Programm die neuen Positionen der Roboter im Modell und wechselt in den Modus PUBLISH zurück.

Der Updateablauf des Moduls *Simulation* ist in der Abb. 3.21 modelliert.

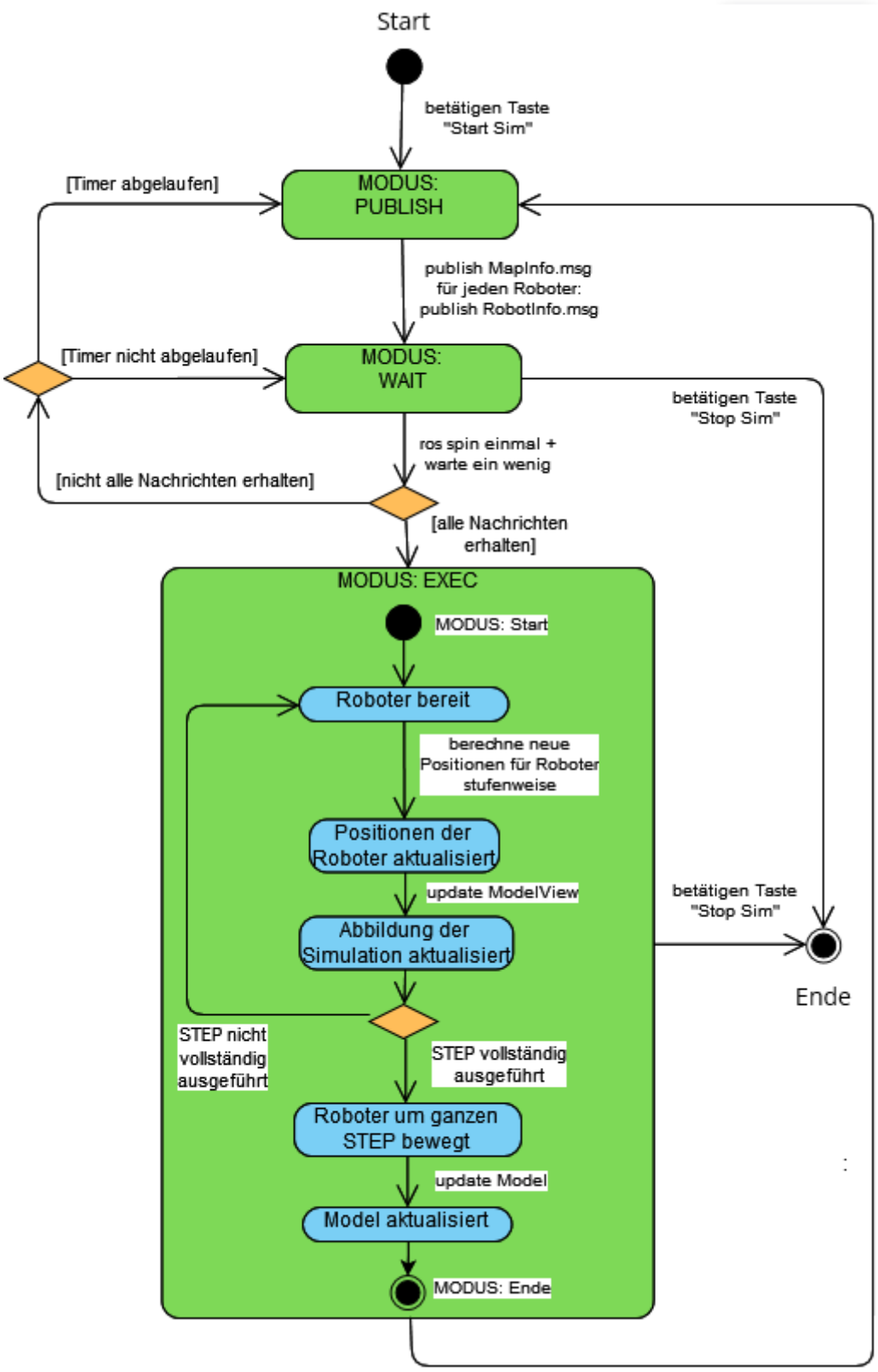

Abbildung 3.21: Der Updateablauf des Moduls *Simulation*

3.4.2 Modul: Controller

Das Modul *Controller* befindet sich im ROS2-package *mapf_controller* und ist in *Python 3.8* implementiert. Der Controller wird aus der Kommandozeile mit dem folgenden Befehl gestartet:

ros2 run mapf_controller controller

Alternativ kann der Controller durch das Ausführen der Datei *__main__.py* im ROS2-package *mapf_controller* initiiert werden. Es wird ein ROS2-Prozess (*node*) mit dem Namen */mapf_controller_node* erzeugt. Der Ablauf des Prozesses ist in der Abb. 3.22 dargestellt.

Nach dem Starten des Controllers befindet sich der Prozess zu Beginn im Modus INIT. Der Controller wartet auf eine Nachricht des Typs *mapf_interfaces/MapInfo.msg* auf dem Nachrichtenkanal */mapf_map_info*. Nach dem Erhalten einer passenden Nachricht erstellt der Controller die nötige Anzahl von ROS2-Subscribern und ROS2-Publishern für den Aufbau der Kommunikation mit einzelnen Robotern aus der Simulation.

Nach dem Aufbau der Verbindungen zu den einzelnen Robotern wechselt der Controller in den Modus WAIT_SIM. Hier wartet der Prozess, bis jeder Roboter die Information über seinen Zustand an den Controller übermittelt. Nach dem Erhalten der Informationen über die Zustände aller Roboter wechselt der Prozess in den Modus PUB_INSTANCE.

Im Modus PUB_INSTANCE erzeugt der Controller aus den von der Simulation erhaltenen Daten eine Instanz und sendet diese an die Pfadplanung. Anschließend wechselt der Prozess in den Modus WAIT_SOLVER.

Im Modus WAIT_SOLVER wartet der Controller auf die Lösung der Instanz von der Pfadplanung. Falls der Controller innerhalb einer bestimmten Zeit keine Antwort von der Pfadplanung erhält, dann wechselt der Prozess in den Modus PUB_INSTANCE zurück.

Nach dem Erhalten einer Lösung wechselt der Controller in den Modus PUB_SIM. Hier werden die berechneten Pfade analysiert und die einzelnen Aktionen für die Roboter zum Folgen der jeweiligen Pfade ermittelt. Anschließend werden die Aktionen an die Roboter in der Simulation übergeben. Ausserdem werden die statistischen Daten der Lösung an den Auswerter gesendet. Der Prozess wechselt anschließend in den Modus INIT zurück.

Beim Erhalten einer neuen Nachricht des Typs *mapf_instance/MapInfo.msg* geht der Controller automatisch in den Modus INIT über. In jeder neuen dieser Nachrichten wird die ID um eins erhöht. Die ankommenden Daten mit einer älteren ID werden vom Controller ignoriert. Anhand dem Inhalt der neuen Nachricht wird zudem geprüft, ob sich die Anzahl der Roboter in der Simulation verändert hat. Bei Bedarf werden neue ROS2-Subscriber und ROS2-Publisher erzeugt oder alte gelöscht.

Der Prozess kann mit Strg+C im Terminal beendet werden.

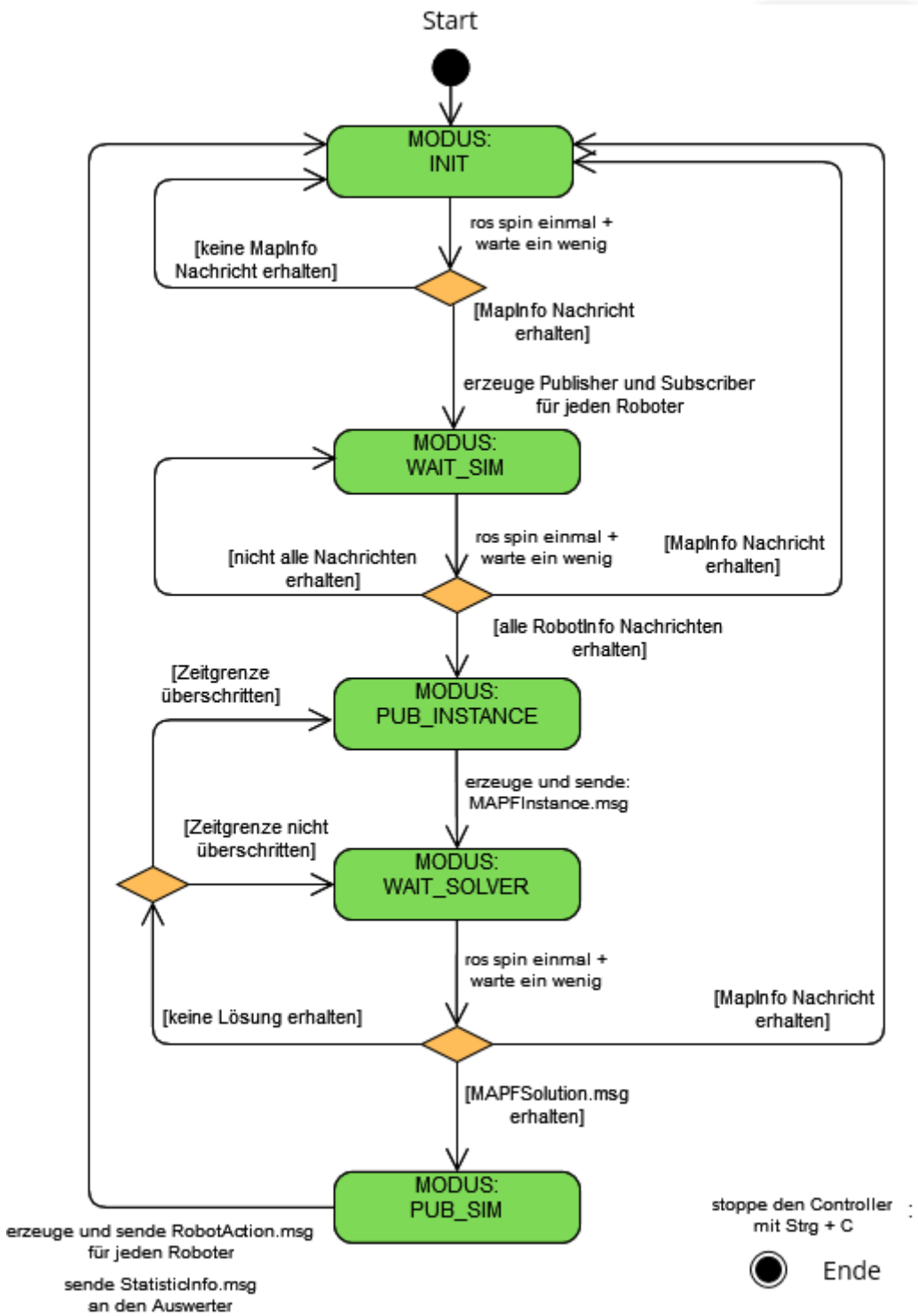

Abbildung 3.22: Der Prozessablauf beim Controller

3.4.3 Modul: Pfadplanung

Für die Berechnung der Pfade werden in die Simulation Algorithmen integriert. Jeder Algorithmus wird in einem separaten ROS2-package definiert. Für die Integration eines Algorithmus in die Simulation müssen neben der Definition des eigentlichen Algorithmus noch folgende Funktionen im ROS2-package implementiert sein:

- die Schnittstelle zur Kommunikation mit dem Controller über den Nachrichtenkanal */mapf_instance_info* [*mapf_interfaces/MAPFInstance.msg*]
- eine Funktion für die Vorverarbeitung von Inputdaten, bevor diese vom Algorithmus ausgewertet werden können
- eine Funktion zum Abfragen der berechneten Pfade für die einzelnen Agenten
- und eine Schnittstelle zum Senden der Lösung an den Controller über den Nachrichtenkanal */mapf_instance_solution* [*mapf_interfaces/MAPFSolution.msg*]

In dieser Arbeit wurden für die Berechnung der Pfade zwei grundlegende Algorithmen aus dem Bereich von MAPF integriert: *Priority-Based Search*[] und *Conflict Based Search*[]. Dafür wurden die folgenden ROS2-packages erzeugt:

- *mapf_pathplanner_pbs* für PBS
- *mapf_pathplanner_cbs* für CBS

Die Algorithmen werden im Kapitel 4 vorgestellt. Hier werden nur die ROS2-packages beschrieben. Der PBS-Solver wird aus der Kommandozeile mit dem folgenden Befehl gestartet:

ros2 run mapf_pathplanner_pbs pathplanner

Es wird ein ROS2-Prozess (*node*) mit dem Namen */mapf_planner_pbs_node* erzeugt. Der CBS-Solver wird aus der Kommandozeile mit dem folgenden Befehl gestartet:

ros2 run mapf_pathplanner_cbs pathplanner

Es wird ein ROS2-Prozess (*node*) mit dem Namen */mapf_planner_cbs_node* erzeugt.

Bei der Integration eines neuen Algorithmus in die Pfadplanung kann hilfsweise auf die beispielhafte Implementierung von CBS und PBS zurückgegriffen werden.

3.4.4 Modul: Auswerter

Das Modul *Auswerter* befindet sich im ROS2-package *mapf_evaluator*. Der Auswerter wird aus der Kommandozeile mit dem folgenden Befehl gestartet:

ros2 run mapf_evaluator evaluator

Alternativ kann der Auswerter durch das Ausführen der Datei *__main__.py* im ROS2-package *mapf_evaluator* initiiert werden. Es wird ein ROS2-Prozess (*node*) mit dem Namen */mapf_evaluator_node* erzeugt.

Das Modul ist in *Python 3.8* implementiert. Für die Darstellung und Anordnung der Graphen wird das *pyplot*-Interface der Bibliothek *matplotlib* benutzt.

Der Ablauf des Prozesses wird im Folgenden erläutert. Der Prozess speichert die auf dem Nachrichtenkanal */mapf_statistics_info*[*mapf_interfaces/StatisticInfo.msg*] ankommenden Nachrichten. Jede Nachricht enthält die Information über den verwendeten Algorithmus, zu dem die Daten in dieser Nachricht zugehören. Anhand dieser Information werden die gespeicherten Daten nach den einzelnen Algorithmen sortiert und in dafür vorgesehenen Schaubildern dargestellt. Die Schaubilder werden in bestimmten Zeitabständen mit den neuen Daten aktualisiert.

Die aktuelle Implementierung erlaubt das Plotten von bis zu vier verschiedenen Graphen, je einen pro Algorithmus, in einem gemeinsamen Schaubild. Folgende zwei Schaubilder werden vom Auswerter erzeugt: die Laufzeit der einzelnen Algorithmen über der Zeit und die Güte der Lösung der einzelnen Algorithmen über der Zeit. Die Güte der Lösung wird hierbei durch die Funktion *sum_of_cost* ausgedrückt (Abb. 3.23).

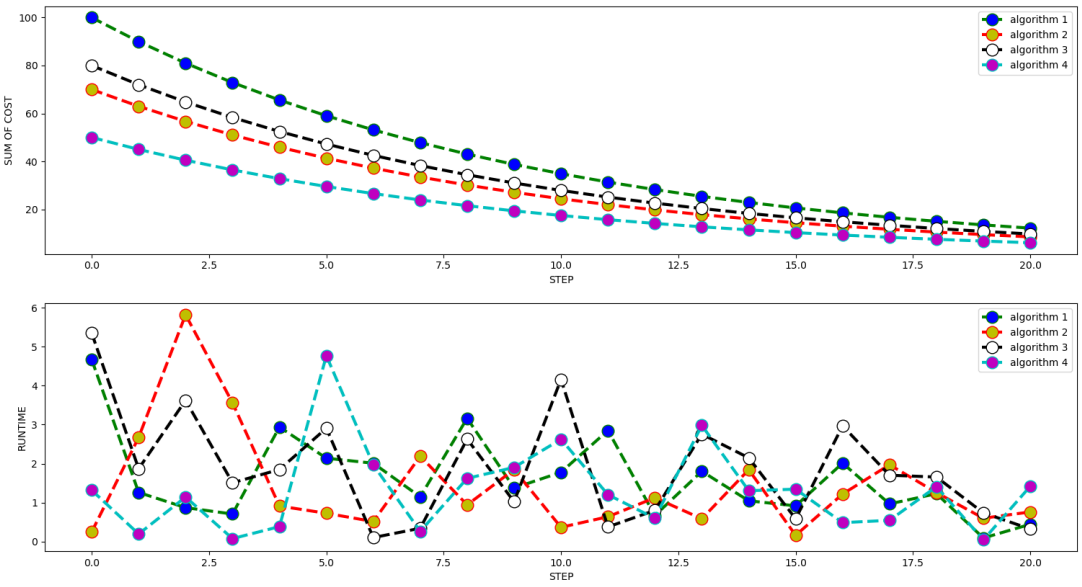

Abbildung 3.23: Eine beispielhaft erzeugte Darstellung der beiden Schaubilder vom Auswerter

Kapitel 4

Integrierte Algorithmen

In diesem Kapitel werden die zwei in die Simulation integrierten Algorithmen, *Priority Based Search* (PBS) und *Conflict Based Search* (CBS) beschrieben. Zuerst wird das CBS und danach PBS vorgestellt.

4.1 Conflict Based Search

CBS ist ein optimaler Algorithmus zur Lösung von MAPF-Problemstellungen. Die Suche nach der Lösung erfolgt dabei auf zwei Ebenen. Auf der höheren Ebene (*High Level Search*) werden die Konflikte zwischen den berechneten Pfaden der einzelnen Agenten ermittelt und daraus Nebenbedingungen (engl. *Constraints*) generiert. Auf der unteren Ebene (*Low Level Search*) werden die Pfade unter Berücksichtigung der zuvor für den jeweiligen Agenten definierten *Constraints* bestimmt. Die Berechnung der Pfade erfolgt für jeden Agenten getrennt. Dadurch wird das Problem des exponentiellen Wachstums des Zustandraumes mit der Anzahl der Agenten gelöst. Im Folgenden wird das *High-* und *Low Level Search* näher beschrieben [12].

Der Algorithmus für das *High-Level Search* ist in der Abb. 5.1 dargestellt. CBS durchsucht auf der höheren Ebene einen binären Baum (*constraint tree*, CT). Jeder Knoten N dieses Baumes enthält folgende Daten: eine bestimmte Anzahl an *Constraints* ($N.constraints$), die ermittelten Pfade für die Agenten unter Berücksichtigung der für jeden Agenten definierten *Constraints* ($N.solution$) und die Kosten des Knotens, die gleich der Summe der Pfadlängen der einzelnen Agenten sind ($N.cost$) [12].

Zu Beginn wird der Wurzelknoten (R) generiert. Dieser enthält keine Constraints [Zeile 1]. Die Pfade für die Agenten werden durch das *Low-Level Search* ermittelt und im Wurzelknoten gespeichert [Zeilen 2]. Die möglichen Konflikte zwischen den einzelnen Pfaden werden hierbei ignoriert. Die Kosten für den Knoten werden ermittelt und im Wurzelknoten gespeichert [Zeile 3]. Die einzelnen Knoten werden in einer Liste ($OPEN$) gespeichert. Dieser Liste wird zu Beginn der Suche der Wurzelknoten hinzugefügt. In einer *while*-Schleife wird anschließend nach einer Lösung gesucht [Zeilen 5-17]. Diese Suche erfolgt so lange, bis eine Lösung gefunden wurde oder die Liste keine weiteren

Knoten mehr enthält [12].

Algorithm 1: high-level of CBS

```
   Input: MAPF instance
1  R.constraints = ∅
2  R.solution = find individual paths using the
   low-level()
3  R.cost = SIC(R.solution)
4  insert R to OPEN
5  while OPEN not empty do
6      P ← best node from OPEN // lowest solution cost
7      Validate the paths in P until a conflict occurs.
8      if P has no conflict then
9          return P.solution // P is goal
10     C ← first conflict (a_i, a_j, v, t) in P
11     foreach agent a_i in C do
12         A ← new node
13         A.constriants ← P.constriants + (a_i, s, t)
14         A.solution ← P.solution.
15         Update A.solution by invoking low-level(a_i)
16         A.cost = SIC(A.solution)
17         Insert A to OPEN
```

Abbildung 4.1: High Level Search beim CBS [12]

Im Folgenden wird eine Iteration der *while*-Schleife detailliert betrachtet. Bei jeder neuen Iteration wird im ersten Schritt der Knoten mit den niedrigsten Kosten aus der Liste selektiert und herausgenommen (*best-first search*). Die im Knoten gespeicherten Pfade der Agenten werden analysiert. Sofern keine Konflikte zwischen den einzelnen Pfaden erkannt werden, ist die Lösung gefunden und die Suche kann abgebrochen werden [Zeilen 8-9]. Ein Konflikt zwischen den Pfaden von zwei Agenten kann folgendermaßen beschrieben werden: (a_i, a_j, v, t). Dabei stehen a_i und a_j für die am Konflikt beteiligte Agenten, v für den Knoten bzw. den Ort des Konflikts und t für den Zeitschritt des Konflikts. Der Konflikt wird durch das Erzeugen von zwei Unterknoten aufgelöst [Zeile 12]. Beide Unterknoten übernehmen die Pfade und die Constraints des *parent*-Knotens [Zeilen 13-14]. Für den linken Unterknoten wird die *Constraint*-Liste um die folgende zusätzliche *Constraint* erweitert: (a_i, v, t). Diese besagt, dass der Agent a_i zum Zeitpunkt t nicht den Knoten v belegen darf. Beim rechten Unterknoten wird entsprechend die folgende *Constraint* hinzuaddiert: (a_j, v, t) [Zeile 13]. Nach dem Erweitern der *Constraints* werden die Pfade für alle Agenten durch das *Low-Level-Search* neuberechnet [Zeile 15]. Anschließend werden die Kosten für den Unterknoten bestimmt [Zeile 16]. Am Ende der Iteration werden die beiden erzeugten Unterknoten der Liste OPEN hinzugefügt [12].

Beim *Low Level Search* wird eine Variation des $A*$-Algorithmus eingesetzt. Die Suche erfolgt dreimensional (x, y, t), da der Zeitfaktor zusätzlich berücksichtigt wird. Der Input wird durch einen Agenten a_i (Start- und Zielpunkt) und eine Liste der ihm zugehörigen *Constraints* definiert. Das *Low Level Search* berechnet für jeden Agenten einen gültigen Pfad, der die für den Agenten vorgegebenen *Constraints* berücksichtigt. Falls der Agent a_i bspw. eine Constraint (a_i, x, t) enthält, so wird jeder Pfad, der für den Agenten zum Zeitpunkt t den Zustand x ergibt, vom *Low Level Search* ignoriert [12].

4.2 Priority Based Search

PBS ist ein nicht optimaler, jedoch sehr effizienter Algorithmus zur Lösung von MAPF-Problemstellungen. Die Suche nach der Lösung erfolgt dabei auf zwei Ebenen. Auf der höheren Ebene (*High Level Search*) wird dynamisch eine Prioritätenliste konstruiert und daraus mit *depth-first search* schrittweise ein Prioritätenbaum (engl. *priority tree*, PT) generiert. Der Konflikt zwischen zwei Agenten wird durch das Erweitern der Prioritätenliste aufgelöst. Dabei wird der eine Agent in der Prioritätenlise dem anderen gegenüber höher gestellt. Um eine Lösung zu finden, werden verschiedene Modifikationen der Prioritätenliste ausprobiert. Auf der unteren Ebene (*Low Level Search*) wird versucht die Pfade für die einzelnen Agenten zu bestimmen. Die ermittelten Pfade dürfen jedoch nicht im Konflikt mit den zuvor berechneten Pfaden der, in der Prioritätenliste höher gestellten, Agenten stehen. Im Folgenden wird das *High-* und *Low Level Search* näher beschrieben [7].

Der Algorithmus für das *High-Level Search* ist in der Abb. 5.2 dargestellt. Zu Beginn wird der Wurzelknoten (*Root*) mit einer leeren Prioritätenliste generiert [Zeilen 1-2]. Im nächsten Schritt werden die Pfade für die Agenten (*Low-Level Search*) ermittelt und im Wurzelknoten gespeichert [Zeilen 3-6]. Die möglichen Konflikte zwischen den einzelnen Pfaden werden hierbei ignoriert. Falls für einen der Agenten kein Pfad gefunden werden konnte, existiert für das Problem keine Lösung und die Suche bricht ab [Zeilen 4-5]. Die Kosten für jeden Knoten berechnen sich aus der Summe der im Knoten gespeicherten Pfadlängen der einzelnen Agenten (*sum of cost*) [Zeile 7, 22]. Die einzelnen Knoten werden im STACK (*last in first out*) gespeichert. Dem STACK wird zu Beginn der Suche der Wurzelknoten hinzugefügt. In einer *while*-Schleife wird anschließend nach einer Lösung gesucht [Zeilen 9-23]. Diese Suche erfolgt so lange, bis eine Lösung gefunden wurde oder der STACK keine weiteren Knoten mehr enthält [7].

Im Folgenden wird eine Iteration der *while*-Schleife detailliert betrachtet. Bei jeder neuen Iteration wird im ersten Schritt der zuletzt hinzugefügte Knoten (*top node*) dem STACK entnommen [Zeile 10] und aus diesem gelöscht [Zeile 11]. Sofern der gespeicherte Plan im auserwählten Knoten keine Konflikte enthält, ist die Lösung gefunden und die Suche kann abgebrochen werden [Zeilen 12-13]. Anderenfalls wird der erste Konflikt aus dem Plan des Knotens ausgewählt und zwei Unterknoten erstellt. Beide Unterknoten

übernehmen den Plan und die Randbedingungen des *parent*-Knoten [Zeilen 17-18]. Die Prioritätenliste bei den Unterknoten wird jedoch um die folgende Bedingung gegenüber dem *parent*-Knoten erweitert: einer der beiden am Konflikt beteiligten Agenten wird in der Prioritätenliste des einen Unterknoten dem anderen Agenten gegenüber höher gestellt. Beim anderen Unterknoten wird die Priorität der beiden Agenten umgekehrt gesetzt. [Zeile 20]. Für jeden Unterknoten erfolgt anschließend die Suche nach dem möglichen Pfad für den am Konflikt beteiligten und der Priorität nach niedriger gesetzten Agenten a, so dass die Pfade der anderen, in der Prioritätenliste höher gesetzten, Agenten berücksichtigt werden. Es wird geprüft und sichergestellt, dass die Pfade der anderen, dem Agenten a der Priorität nach niedriger gesetzten Agenten keine Konflikte mit den anderen, dem Agenten a höher gesetzten Agenten aufweisen. Bei Bedarf werden die Pfade der dem Agenten a der Priorität nach tiefer gesetzten Agenten neuberechnet [Zeilen 25-32]. Nach dem, mit Erfolg abgeschlossenen, Neuberechnung der Pfade werden für den jeweiligen Unterknoten die Kosten bestimmt. Am Ende der Iteration werden die beiden erzeugten Unterknoten dem STACK hinzugefügt. Dabei wird der Unterknoten mit den niedrigeren Kosten zuletzt eingefügt [7].

Auf der unteren Ebene (*Low-Level Search*) wird eine Modifikation des A*-Algorithmus eingesetzt, um den optimalen Pfad für einen Agenten unter Berücksichtigung der Pfade der anderen, der Priorität nach höher gesetzten Agenten zu bestimmen [7].

Algorithm 2: High-Level Search of PBS (with initial priority ordering $\prec_0$)

Input: MAPF instance, $\prec_0 (= \emptyset$ by default)
$\prec_{Root} \leftarrow \prec_0$;
Root.plan $\leftarrow \emptyset$;
foreach $i \in [M]$ **do**
  success $\leftarrow$ *UpdatePlan*(*Root*, a_i); /* *success is always true if* $\prec_0 = \emptyset$ */
  if $\neg$*success* **then**
    return "No Solution";
Root.cost $\leftarrow$ sum of the arrival times in *Root.plan*;
STACK $\leftarrow \{$*Root*$\}$;
while STACK $\neq \emptyset$ **do**
  $N \leftarrow$ top node in STACK;
  STACK $\leftarrow$ STACK $\setminus \{N\}$;
  if N.*plan* has no collision **then**
    return N.*plan*;
  C $\leftarrow$ first vertex or edge collision $\langle a_i, a_j, \dots \rangle$ in N.*plan* ;
  foreach a_i involved in C **do**
    $N' \leftarrow$ new node;
    N'.*plan* $\leftarrow$ N.*plan*;
    N'.*constraints* $\leftarrow$ N.*constraints* $\cup \{\langle a_i, \dots \rangle\}$;
    $\prec_{N'} \leftarrow \prec_N \cup \{j \prec i\}$;
    success $\leftarrow$ *UpdatePlan*(N', a_i);
    if *success* **then**
      N'.*cost* $\leftarrow$ sum of the arrival times in N'.*plan*;
  Insert new nodes N' into STACK in non-increasing order of N'.*cost*;
return "No Solution";
Function *UpdatePlan*(N, a_i)
  LIST $\leftarrow$ topological sorting on partially ordered set $(\{i\} \cup \{j | i \prec_N j\}, \prec_N)$;
  foreach $j \in$ LIST **do**
    if $j = i$ or $\exists a_k : k \prec_N j$, a_j collides with a_k in N.plan **then**
      Update N.*plan* by invoking a low-level search for a_j that avoids colliding with all agents a_k with higher priorities ($k \prec_N j$);
      if no path is returned by the low-level search **then**
        return *false*;
  return *true*;

Abbildung 4.2: High Level Search beim PBS [7]

Kapitel 5

Experimentelle Ergebnisse

Die integrierten Algorithmen werden mit verschiedenen Eingabedaten simuliert und miteinander verglichen. Die Ergebnisse werden in diesem Kapitel dokumentiert. Folgende zwei Kriterien werden für den Vergleich der Algorithmen hinzugezogen: Die Laufzeit der Algorithmen bis zum Finden der Lösung (engl. *runtime*) und die Güte der Lösung (*sum of cost*).

Für die Eingabe von Daten werden die bereitgestellten Benchmarks von der folgenden Seite verwendet: `https://movingai.com/benchmarks/mapf/index.html`. Für die Simulation der Umgebung wird die folgende *map*-Datei benutzt:

warehouse-10-20-10-2-1.map

Die Dimensionen der Umgebung sind: *161 × 63 Einheiten* . Die Umgebung beschreibt ein typisches Lagerhaus, wie es in großen Logistikzentren oft vorzufinden ist und ist in der Abb. 5.1 dargestellt.

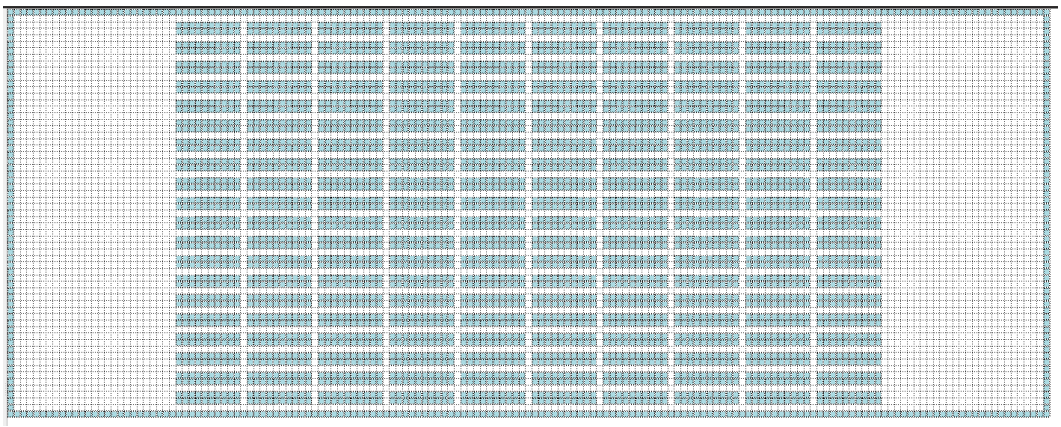

Abbildung 5.1: Die Darstellung der Umgebung für die Simulation eines Lagerhauses (*warehouse-10-20-10-2-1.map*)

Für die Simulation der Roboter werden die folgenden fünf *scene*-Dateien ausgewählt:

warehouse-10-20-10-2-1-even-1.scen

warehouse-10-20-10-2-1-even-2.scen

warehouse-10-20-10-2-1-even-3.scen

warehouse-10-20-10-2-1-even-4.scen

warehouse-10-20-10-2-1-even-5.scen

Alle *scene*-Dateien befinden sich im Ordner *warehouse-10-20-10-2-1-even.scen* und können ebenenfalls heruntergeladen werden. Jede Datei enthält 45 *Buckets* (nummeriert von 0 bis 44) mit je 10 Robotern. Somit speichert jede Datei Informationen über 450 Roboter, aufgeteilt in 45 gleich große Gruppen.

Im Weiteren werden die durchgeführten Experimente beschrieben. Es werden mindestens die folgenden Konfigurationen für jede *scene*-Datei in einer eigenen Simulation ausprobiert:

- 10 Roboter (*bucket*: 0)
- 30 Roboter (*buckets*: 0,1,2)
- 50 Roboter (*buckets*: 0,1,2,3,4)
- 80 Roboter (*buckets*: 0-7)
- 100 Roboter (*buckets*: 0-9)

Die statistischen Daten werden aufgenommen und vom Auswerter geplottet. Die Ergebnisse werden beschrieben und analysiert. Zur Veranschaulichung werden die selektierten Schaubilder für die Darstellung der Ergebnisse hier eingefügt. Die Graphen für PBS (CBS) werden in den Schaubildern mit grüner (roter) Linie und blauen (gelben) Punkten gezeichnet. Die Punkte stellen die Zwischenwerte aus den Berechnungen der Algorithmen dar und werden in die Schaubilder vom Auswerter eingetragen. Für eine bessere Darstellung werden die Punkte miteinander durch gerade Linien verbunden.

Für alle Berechnungen wurde als Rechenstation die folgende Konfiguration benutzt: *Notebook Lenovo Legion 5* mit *AMD Ryzen™ 5 4600H Mobil-Prozessor*, 16 GB Arbeitsspeicher, SSD-Festplatte und *Nvidia GeForce RTX 2060 Grafikkarte*.

5.1 Simulation mit 10 Robotern

In den Abb. 5.2-5.3 sind der Ausgangszustand und der Endzustand der Simulation bei einer Konfiguration mit 10 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) dargestellt. In der Abb. 5.4 sind die Ergebnisse der Simulation mit der Angabe der Laufzeit der Algorithmen und der Güte der Lösungen über die Zeit beschrieben. Beide Algorithmen lösen die einfache Aufgabe unter 0.003 Sekunden für jeden Zeitschritt. Die berechneten Pfade müssen nicht übereinstimmten, da es mehrere optimale Lösungen gibt. Jedoch sind die beiden Pfade gleich gut. Die beiden Algorithmen finden die Pfade bei einer Konfiguration mit 10 Robotern für alle scene-Dateien.

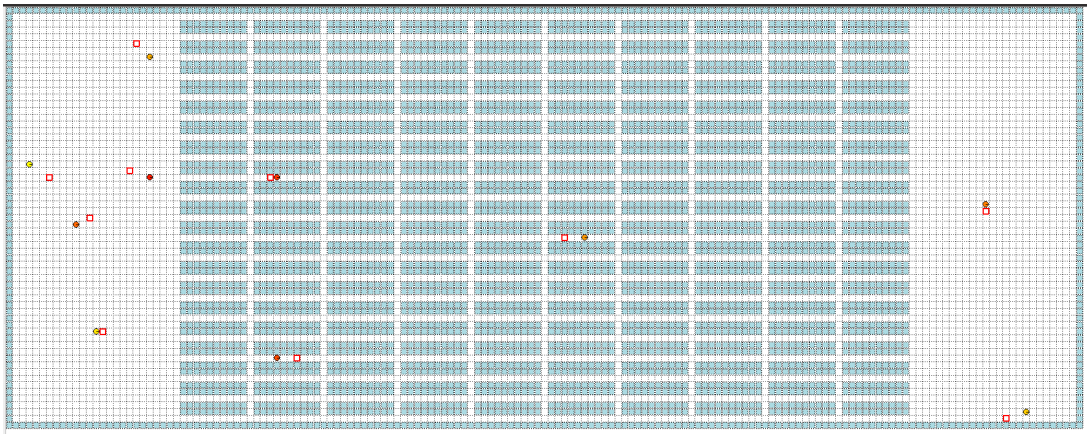

Abbildung 5.2: Der Ausgangszustand der Simulation mit 10 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*)

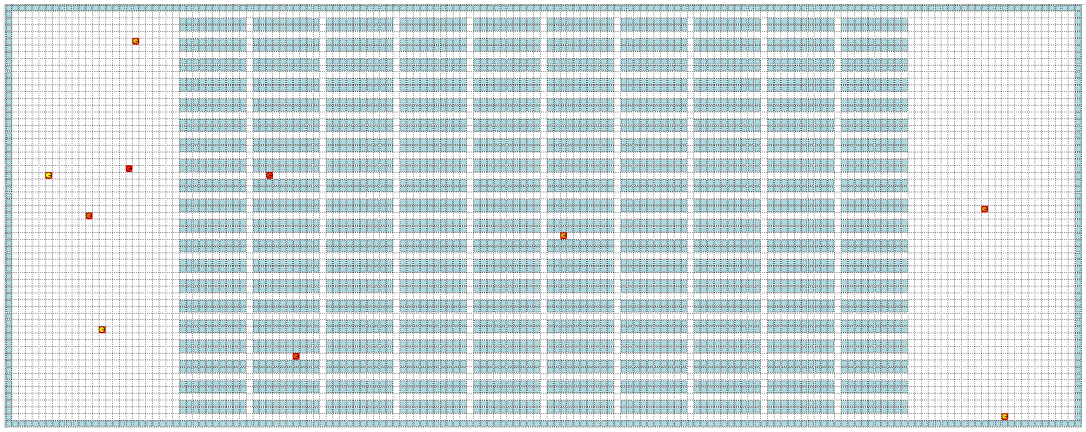

Abbildung 5.3: Der Endzustand nach der Ausführung der Simulation mit 10 Robotern (*warehouse-10-20-10-2-1-even-1.scen*)

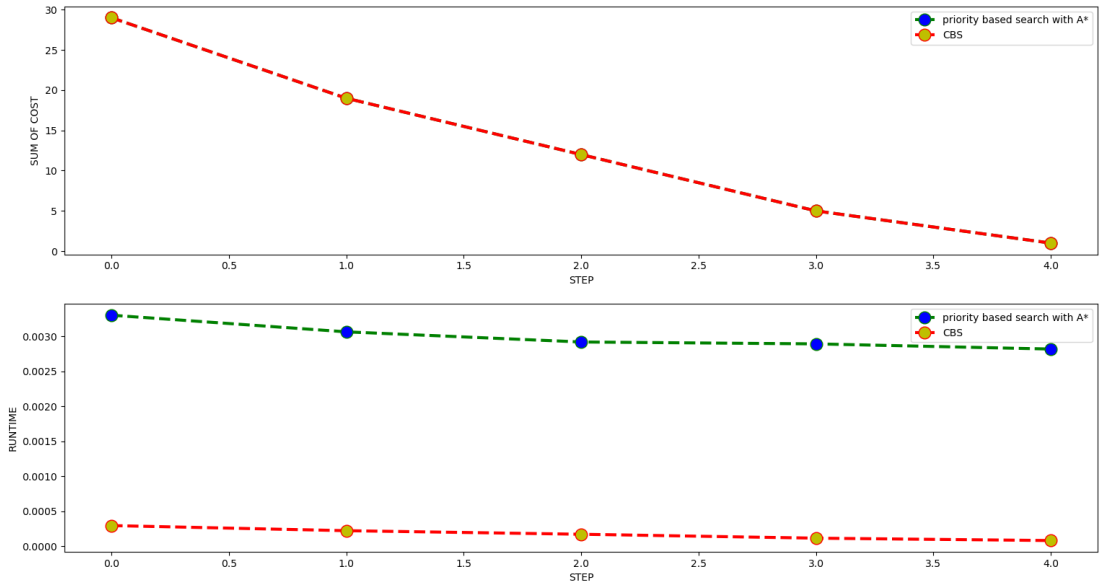

Abbildung 5.4: Die Laufzeit des Algorithmen und die Summen der Kosten sind über die Zeit (*steps*) dargestellt. Nach fünf Zeitschritten erreichen alle 10 Roboter ihre Zielpunkte. Die Summen der Kosten beim PBS werden durch die identischen Daten des CBS überdeckt (*warehouse-10-20-10-2-1-even-1.scen*).

5.2 Simulation mit 30 Robotern

In den Abb. 5.5 und 5.6 sind der Ausgangszustand und der Endzustand der Simulation bei einer Konfiguration mit 30 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) dargestellt. In der Abb. 5.7 sind die Ergebnisse der Simulation mit der Angabe der Laufzeit der Algorithmen und der Güte der Lösungen über die Zeit beschrieben. Beide Algorithmen lösen die einfache Aufgabe unter 0.004 Sekunden für jeden Zeitschritt. Die berechneten Pfade müssen nicht übereinstimmten, da es mehrere optimale Lösungen gibt. Jedoch sind die beiden Pfade gleich gut. Die beiden Algorithmen finden die Pfade bei einer Konfiguration mit 10 Robotern für alle scene-Dateien.

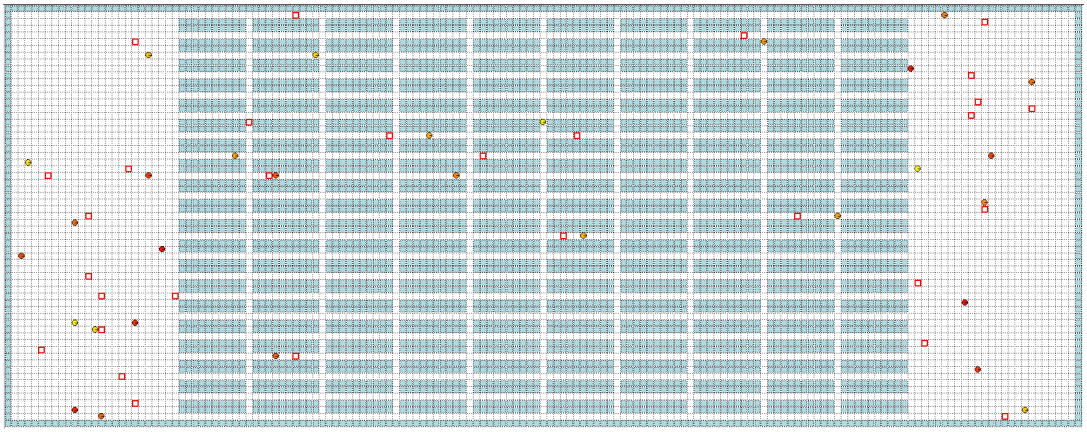

Abbildung 5.5: Der Ausgangszustand der Simulation mit 30 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*)

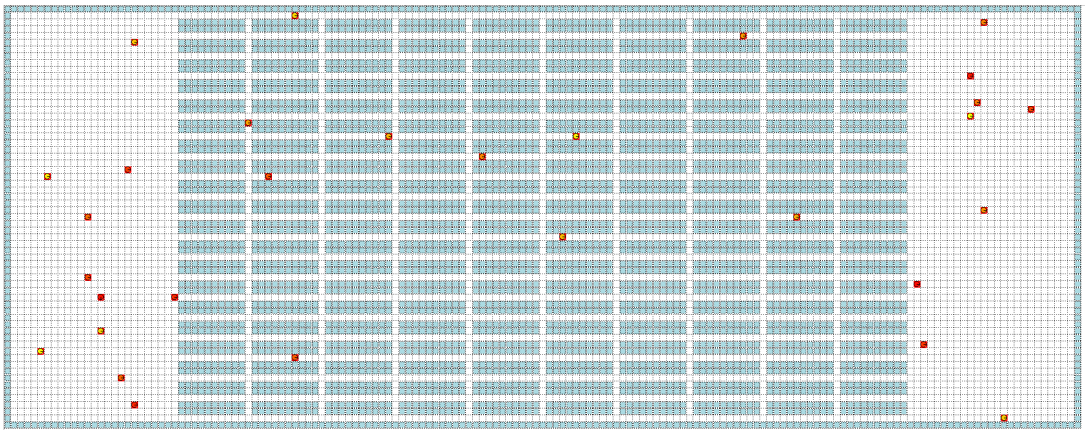

Abbildung 5.6: Der Endzustand nach der Ausführung der Simulation mit 30 Robotern (*warehouse-10-20-10-2-1-even-1.scen*)

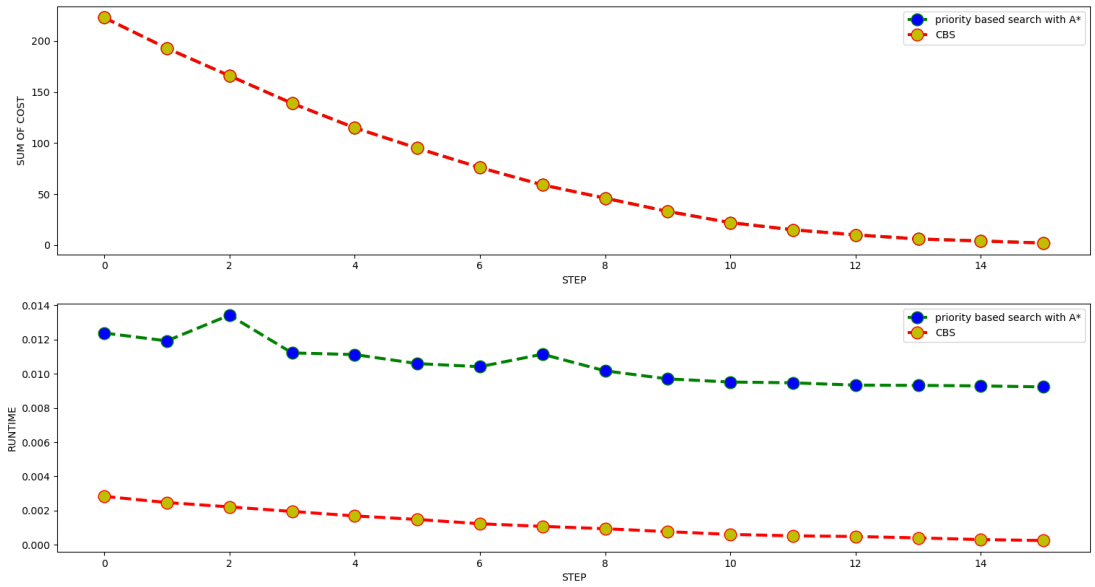

Abbildung 5.7: Die Laufzeit der Algorithmen und die Summen der Kosten sind über die Zeit (*steps*) dargestellt. Nach 15 Zeitschritten erreichen alle 30 Roboter ihre Zielpunkte. Die Summen der Kosten beim PBS werden durch die identischen Daten des CBS überdeckt (*warehouse-10-20-10-2-1-even-1.scen*).

5.3 Simulation mit 50 Robotern

In den Abb. 5.8 und 5.9 sind der Ausgangszustand und der Endzustand der Simulation bei einer Konfiguration mit 50 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) dargestellt. In der Abb. 5.10 sind die Ergebnisse der Simulation mit der Angabe der Laufzeit der Algorithmen und der Güte der Lösungen über die Zeit beschrieben. Beide Algorithmen lösen die einfache Aufgabe unter 0.003 Sekunden für jeden Zeitschritt. Die berechneten Pfade müssen nicht übereinstimmten, da es mehrere optimale Lösungen gibt. Jedoch sind die beiden Pfade gleich gut. Mit der steigenden Menge an Robotern nähern sich langsam die Laufzeiten von PBS und CBS an. Die beiden Algorithmen finden die Pfade bei einer Konfiguration mit 10 Robotern für alle scene-Dateien.

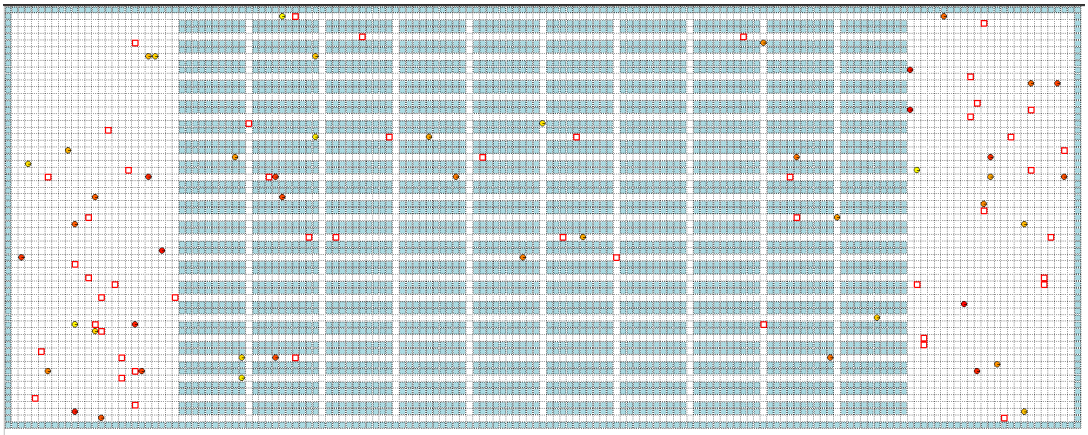

Abbildung 5.8: Der Ausgangszustand der Simulation mit 50 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*)

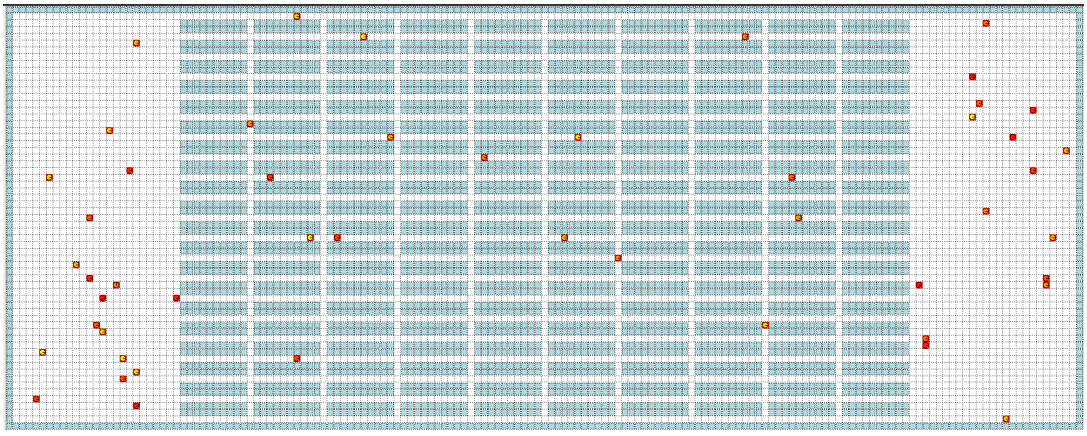

Abbildung 5.9: Der Endzustand nach der Ausführung der Simulation mit 50 Robotern (*warehouse-10-20-10-2-1-even-1.scen*)

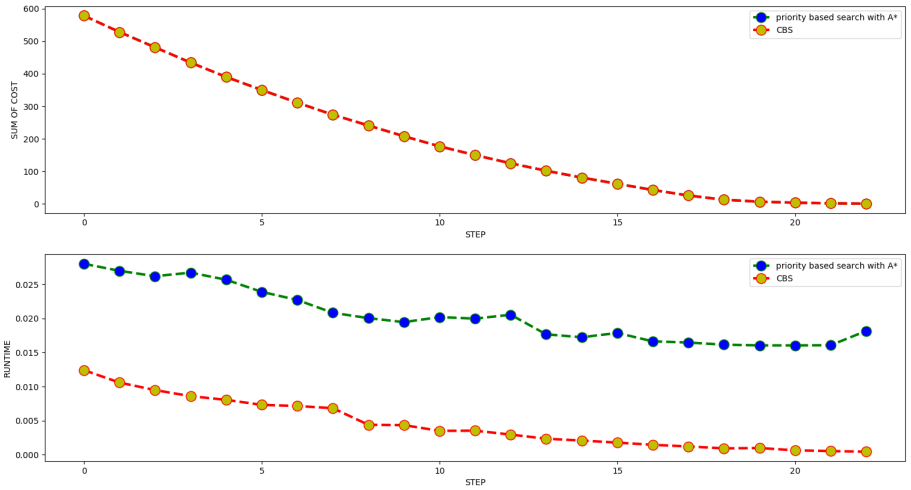

Abbildung 5.10: Die Laufzeit der Algorithmen und die Summen der Kosten sind über die Zeit (*steps*) dargestellt. Nach 22 Zeitschritten erreichen alle 50 Roboter ihre Zielpunkte. Die Summen der Kosten beim PBS werden durch die identischen Daten des CBS überdeckt (*warehouse-10-20-10-2-1-even-1.scen*).

5.4 Simulation mit 60-90 Robotern

Mit der steigenden Menge an Robotern sinkt die Effizienz von CBS im Vergleich zu PBS. Ab einer Menge von 60 Robotern findet der CBS nicht immer eine Lösung innerhalb der festgelegten Zeitgrenze von einer Minute.

Für die Daten aus *warehouse-10-20-10-2-1-even-1.scen* (*buckets*: 0-5) schafft der CBS es nicht die Lösung bei 60 oder mehr Robotern rechtzeitig zu ermitteln. Der PBS findet jedoch die Lösung innerhalb von weniger als 0.1 Sekunden für jeden Zeitschritt (Abb. 5.11) .

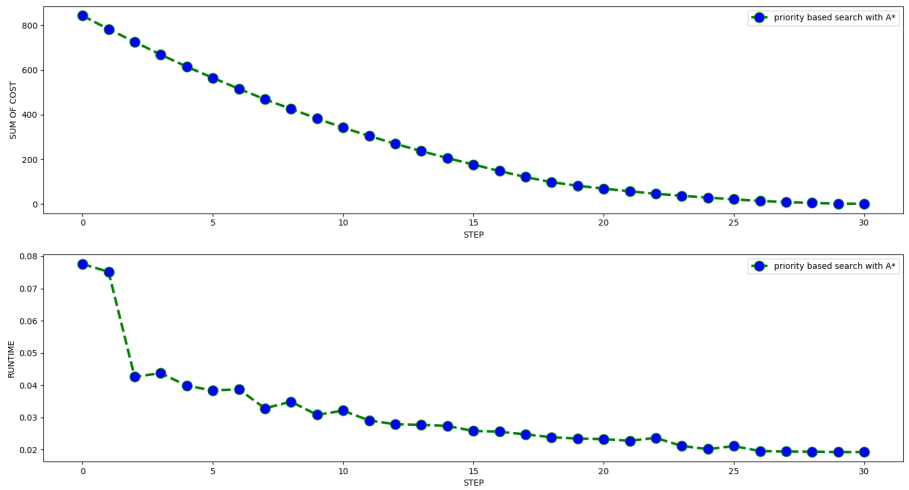

Abbildung 5.11: Die Laufzeit des Algorithmen (PBS) und die Summen der Kosten sind über die Zeit (*steps*) dargestellt. Nach genau 30 Zeitschritten erreichen alle 60 Roboter ihre Zielpunkte (*warehouse-10-20-10-2-1-even-1.scen*).

In den Abb. 5.12 und 5.13 sind der Ausgangszustand und der Endzustand der Simulation bei einer Konfiguration mit 70 Robotern (*warehouse-10-20-10-2-1-even-4.scen*) dargestellt. In der Abb. 5.14 sind die Ergebnisse der Simulation mit der Angabe der Laufzeit der Algorithmen und der Güte der Lösungen über die Zeit beschrieben. Beim ersten Schritt braucht der CBS deutlich länger, etwa 12 Sekunden, für die Suche nach den optimalen Pfaden für die Roboter. PBS dagegen findet bereits nach 0.1-0.2 Sekunden eine ebenso gute Lösung. Die Entfernung der Roboter von den jeweiligen Zielpunkten bestimmt direkt die Dauer der Suche nach der optimalen Lösung. Beim ersten Zeitschritt sind die Roboter am weitesten von ihren Zielpunkten entfernt. Daher benötigt die Berechnung beim ersten Schritt in der Regel die meiste Zeit. Die Laufzeit des Algorithmus sinkt beim CBS drastisch nach dem ersten Schritt, wie man der Abb. 5.14 entnehmen kann.

Abbildung 5.12: Der Ausgangszustand der Simulation mit 70 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-4.scen*)

Abbildung 5.13: Der Endzustand nach der Ausführung der Simulation mit 70 Robotern (*warehouse-10-20-10-2-1-even-4.scen*)

CBS und PBS wurden zusätzlich für Konfigurationen mit 90 Robotern mit den fünf *scene*-Dateien getestet. CBS konnte in keinem der fünf Fälle eine Lösung unter

einer Minute finden. PBS konnte jedes der Probleme für beliebigen Zeitschritt unter 0.2 Sekunden lösen (Abb. 5.15). Die Summen der Kosten sind bei den vier dargestellten Graphen vergleichbar. Damit sind die Roboter zu Beginn der Simulation in der Summe etwa gleich weit von ihren Zielpunkten entfernt und die Laufzeiten des Algorithmus können für die vier Fälle miteinander verglichen werden.

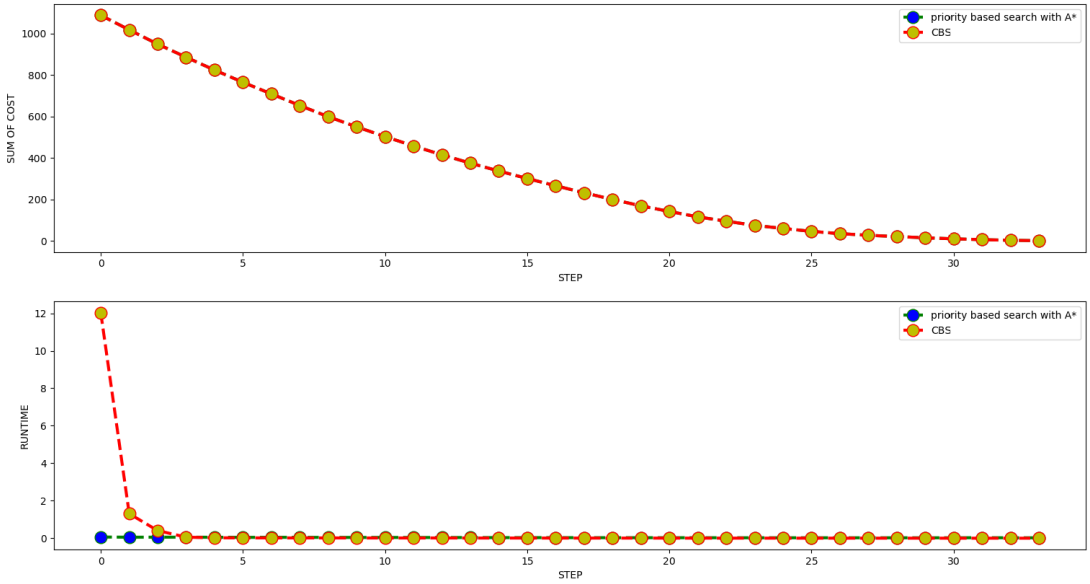

Abbildung 5.14: Die Laufzeit der Algorithmen und die Summen der Kosten sind über die Zeit (*steps*) dargestellt. Nach 33 Zeitschritten erreichen alle 70 Roboter ihre Zielpunkte. Die Summen der Kosten beim PBS werden durch die identischen Daten des CBS überdeckt (*warehouse-10-20-10-2-1-even-4.scen*).

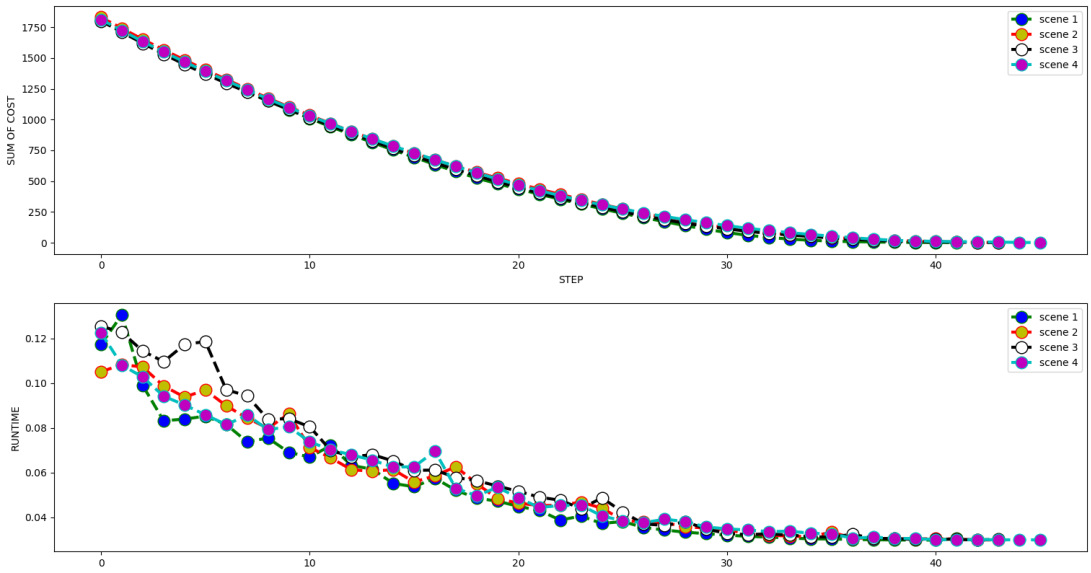

Abbildung 5.15: Die Laufzeit von PBS und die Summen der Kosten für die ersten vier *scene*-Dateien bei einer Konfiguration mit 90 Robotern.

5.5 Simulation mit 100 Robotern

In den Abb. 5.16 und 5.17 sind der Ausgangszustand und der Endzustand der Simulation bei einer Konfiguration mit 100 Robotern (*warehouse-10-20-10-2-1-even-1.scen*) dargestellt. In den Abb. 5.18 und 5.19 sind die Ergebnisse der Simulation mit der Angabe der Laufzeit des PBS-Algorithmen und der Güte der Lösung über die Zeit beschrieben. Zu Vergleichszwecken wurde die Simulation für die gleiche Problemstellung mehrmals durchgeführt. PBS schafft es für die gegebene Umgebung die Pfade für 100 Roboter unter von 0.3 Sekunden für jeden Zeitschritt zu berechnen.

Abbildung 5.16: Der Ausgangszustand der Simulation mit 100 Robotern vor der Ausführung (*warehouse-10-20-10-2-1-even-1.scen*)

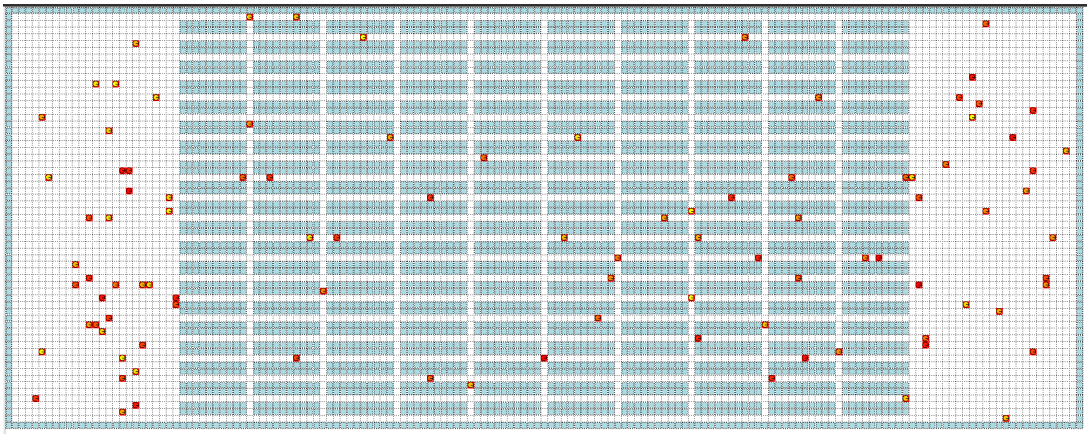

Abbildung 5.17: Der Endzustand nach der Ausführung der Simulation mit 100 Robotern (*warehouse-10-20-10-2-1-even-1.scen*)

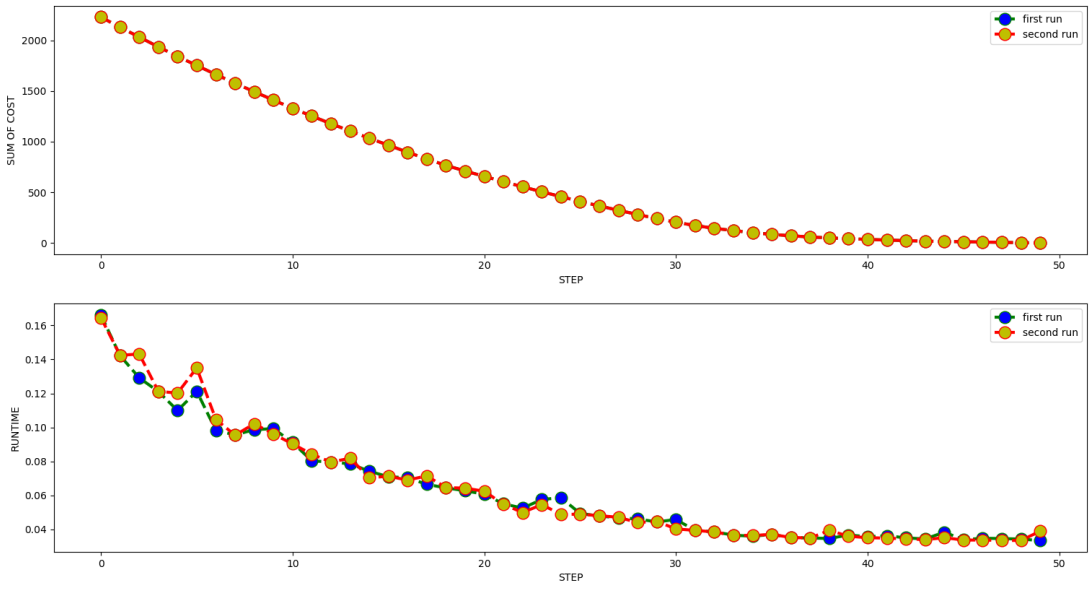

Abbildung 5.18: Die Laufzeit von PBS und die Summen der Kosten sind über die Zeit (*steps*) für zwei Versuche dargestellt. Nach 49 Zeitschritten erreichen alle 100 Roboter ihre Zielpunkte (*warehouse-10-20-10-2-1-even-1.scen*).

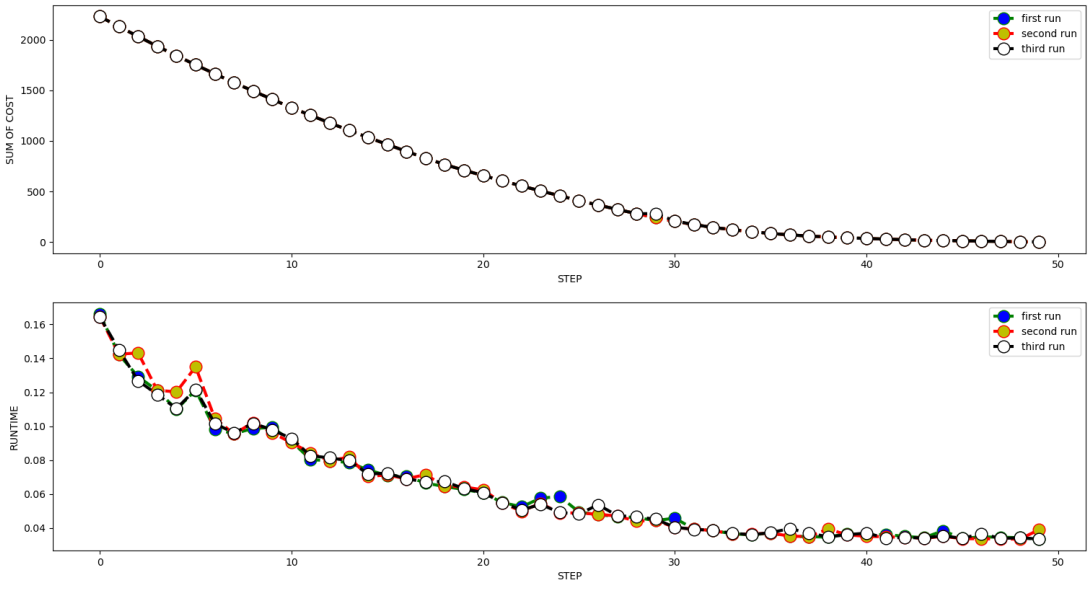

Abbildung 5.19: Die Laufzeit von PBS und die Summen der Kosten sind über die Zeit (*steps*) für drei Versuche dargestellt. Nach 49 Zeitschritten erreichen alle 100 Roboter ihre Zielpunkte (*warehouse-10-20-10-2-1-even-1.scen*).

5.6 Auswertung der Ergebnisse

Beide Algorithmen lösen die Problemstellungen mit bis zu 50 Robotern bei der gegebenen Umgebung von unter 0.1 Sekunden. Ab 60 Robotern findet der CBS nicht immer eine Lösung innerhalb der festgelegten Zeitgrenze von einer Minute. PBS kann jedoch die Pfade für 100 Roboter gleichzeitig berechnen.

Der CBS hat gegenüber dem PBS den Vorteil, vollständig und optimal zu sein. Bei der gegebenen Umgebung (Simulation eines Lagerhauses) spielt jedoch der Faktor der Optimalität und der Vollständigkeit des Algorithmus eine geringere Rolle, da es oft mehrere und zugleich gleich gute bzw. optimale Wege zum Ziel gibt. Dies ist besonders dann der Fall, wenn die Zwischengänge im Lagerhaus mehrere Roboter breit sind (bspw. bei *warehouse-10-20-10-2-2.map*). Zusätzlich erhalten die Roboter bei den verwendeten Benchmarks in der Regel ein Ziel in ihrer Nähe. Dies ist möglicherweise auf die generelle konzeptbedingte Planung in automatisierten Lagerhäusen zurückzuführen. Die langen Wege für die Roboter sollen nach Möglichkeit vermieden werden. Jedoch vereinfacht es das Problem hinsichtlich der Suche nach einer optimalen Lösung. Um das Problem zu erschweren, wurde die Map *warehouse-10-20-10-2-1.map* für die experimentellen Versuche ausgewählt. Hier sind die Gänge nur einen Roboter breit. Dennoch findet der PBS bei allen getesteten Konfigurationen gleich gute Lösungen wie der CBS.

Zusammenfassend lässt sich sagen, dass die geringen Zeitvorteile bei kleinen Problemstellungen ($\leq$50 Roboter), die Optimalität und Vollständigkeit von CBS bei den durchgeführten Experimenten im Vergleich zum PBS nicht zur Geltung kommen. Jedoch zeigt PBS bei komplizierteren Problemen ($\geq$60 Roboter) deutlich bessere Ergebnisse. Daher ist der PBS bei der getesteten Umgebung dem CBS vorzuziehen.

Die Ergebnisse aus [7] hinsichtlich der höheren Effizienz von *Priotiry Based Search* gegenüber dem *Conflict Based Search* werden hier bestätigt.

Kapitel 6

Zusammenfassung

In dieser Arbeit wird ein auf *Robot Operating System 2* aufbauendes Framework zur Simulation der Pfadplanung für die klassische *Multi-Agent Path Finding* Problemstellung entwickelt. In das Framework werden mehrere Algorithmen integriert und verschiedene Datensätze simuliert. Am Ende werden die Ergebnisse analysiert und dokumentiert.

Das Framework setzt sich aus den folgenden vier Komponenten zusammen: der Simulation, dem Controller, der Pfadplanung und dem Auswerter. Die Kommunikation zwischen den einzelnen Komponenten erfolgt unter ROS2. Die Simulation führt die Visualisierung der eigentlichen Simulation aus und bietet dem Benutzer eine grafische Schnittstelle für die Interaktion mit dem Programm an. Der Controller kommuniziert mit den anderen Komponenten und dient als Schnittstelle zwischen der Simulation, der Pfadplanung und dem Auswerter. Die Pfadplanung ermittelt die Pfade für die einzelnen Roboter zum Erreichen ihrer Ziele. Der Auswerter speichert die statistischen Daten der Algorithmen, bspw. die Laufzeit und die Güte der Lösung, und plottet diese über die Zeit. Dadurch können die verwendeten Algorithmen analysiert und miteinander verglichen werden.

Folgende zwei Algorithmen werden in das Framework integriert und vorgestellt: *Conflict Based Search* (CBS) und *Priority Based Search* (PBS).

CBS ist ein häufig angewandter MAPF Algorithmus, der optimal und vollständig ist. Die Suche nach der Lösung erfolgt hiebei auf zwei Ebenen. Auf der oberen Ebene (*High-Level Search*) werden Konflikte zwischen den einzelnen Pfaden der Agenten ermittelt und daraus eine Liste an *constrains* für jeden Agenten generiert. Auf der unteren Ebene (*Low-Level Search*) erfolgt die Berechnung der einzelnen Pfade der Agenten unter der Berücksichtgung entsprechender constraints.

PBS ist suboptimal und nicht vollständig, jedoch sehr effizient. Die Suche nach der Lösung erfolgt hierbei ebenenfalls auf zwei Ebenen. Im Gegensatz zu CBS werden beim *High-Level Search* Konflikte durch das Anpassen der Prioritätenliste und nicht durch das Erzeugen von *constraints* aufgelöst. Beim *Low-Level Search* erfolgt die Berechnung der Pfade für die einzelnen Agenten der Reihe nach gemäß der definierten Prioritätenliste.

Die beiden Algorithmen werden mit verschiedenen Daten getestet. Es werden entsprechende Benchmarks für die Simulation eines Lagerhauses, das in großen automatisierten Logistikzentren vorzufinden sind, ausgewählt. Die Simulation wird zu Beginn mit 10 Robotern getestet. Die Menge der Roboter wird schrittweise bis 100 erhöht.

Beide Algorithmen lösen die Problemstellungen mit bis zu 50 Robotern in der gegebenen Umgebung von unter 0.1 Sekunden. Ab 60 Robotern findet der CBS nicht immer eine Lösung innerhalb der festgelegten Zeitgrenze von einer Minute. PBS schafft es jedoch die Pfade für 100 Roboter gleichzeitig zu berechnen. Die geringen Zeitvorteile bei kleinen Problemstellungen ($\leq$50 Roboter), die Optimalität und Vollständigkeit von CBS gegenüber dem PBS kommen bei den durchgeführten Experimenten nicht zur Geltung. Jedoch zeigt PBS bei komplizierteren Problemen ($\geq$60 Roboter) deutlich bessere Ergebnisse. Daher ist der PBS bei der getesteten Simulationsumgebung dem CBS vorzuziehen.

Die Ergebnisse aus [7] hinsichtlich der höheren Effizienz von *Priotiry Based Search* gegenüber dem *Conflict Based Search* bestätigen sich ebenenfalls für die in dieser Arbeit gewählten Simulationsumgebung.

6.1 Ausblick

Das Framework kann durch zahlreiche Funktionen erweitert und verbessert werden. Weitere GUI-Funktionen können implementiert werden. Bpsw. könnte der Anwender per Anklicken und Ziehen oder durch einen Eintrag in der Robotertabelle die einzelnen Roboter in der Umgebung neupositionieren.

Zur Zeit erfolgt die Berechnung der Pfade für die Agenten zeitgleich für einen bestimmten Zeitschritt. Dieses Vorgehen kann eventuell durch einen dezentralen Ansatz ersetzt werden, so dass die Berechnung der Pfade für jeden Agenten separat verläuft. Somit könnte jeder Agent bei Bedarf die Berechnung des Pfades initieren bzw. von der Pfadplanung anfordern. Die Bewegung der Roboter erfolgt für einen Zeitschritt simultan. Ein anderer Ansatz, bei dem die Bewegung der Roboter asynchron verläuft und die Zeit nicht als diskret, sondern als kontinuierlich betrachtet wird, kann ausprobiert werden.

In dieser Arbeit wurde der einfache CBS behandelt. Es existieren verschiedene Variationen des Algorithmus, bspw. CBSwP, ECBS usw. Diese Variationen sowie andere Algortihmen, bpsw. ICTS, können in das Framework integiert und mit bereits vorhandenen Algorithmen verglichen werden.

Bis jetzt werden die *Sum of Cost* und die Laufzeit der Algorithmen über die Zeit geplottet. Weitere Auswertungskriterien für die Algorithmen, bswp. die Anzahl generierter und expandierter Knoten für den *High-Level* und den *Low-Level Search*, können ebenenfalls in einem Graphen über die Zeit dargestellt werden.

Die untersuchten Algorithmen können mit weiteren Datensätzen getestet und ausgewertet werden.

Literatur

[1] Zain Alabedeen Ali und Konstantin S. Yakovlev. "Prioritized SIPP for Multi-Agent Path Finding With Kinematic Constraints". In: *CoRR* abs/2108.05145 (2021). arXiv: 2108.05145. URL: https://arxiv.org/abs/2108.05145.

[2] John Doe. *ROS / Introduction*. URL: https://wiki.ros.org/ROS/Introduction. (last modified: 2018-08-08).

[3] Martin Gebser, Philipp Obermeier, Thomas Otto, Torsten Schaub, Orkunt Sabuncu, Van Nguyen und Tran Cao Son. *Experimenting with robotic intra-logistics domains*. 2018. arXiv: 1804.10247 [cs.AI].

[4] Brian Gerkey. *Why ROS2?* Juni 2014. URL: http://design.ros2.org/articles/why_ros2.html. (last modified: 2022-05).

[5] Sven Koenig. *Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery*. URL: https://www.youtube.com/watch?v=eEbssZioY2g. (25.11.2018).

[6] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar und Sven Koenig. "Lifelong Multi-Agent Path Finding in Large-Scale Warehouses". In: *CoRR* abs/2005.07371 (2020). arXiv: 2005.07371. URL: https://arxiv.org/abs/2005.07371.

[7] Hang Ma, Daniel Harabor, Peter J. Stuckey, Jiaoyang Li und Sven Koenig. "Searching with Consistent Prioritization for Multi-Agent Path Finding". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (Juli 2019), S. 7643–7650. DOI: 10.1609/aaai.v33i01.33017643. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4758.

[8] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette und William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[9] Open Robotics. *ROS 2 Foxy Fitzroy Logo Image*. URL: https://github.com/ros-infrastructure/artwork/blob/master/distributions/foxy.png. 2020.

[10] Open Robotics. *The ROS Ecosystem*. URL: https://www.ros.org/blog/ecosystem/. (2021).

[11] *ROS LOGO*. URL: http://wiki.ros.org.

[12] Guni Sharon, Roni Stern, Ariel Felner und Nathan Sturtevant. "Conflict-Based Search For Optimal Multi-Agent Path Finding". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 26.1 (Sep. 2021), S. 563–569. DOI: 10.1609/aaai.v26i1.8140. URL: https://ojs.aaai.org/index.php/AAAI/article/view/8140.

[13] Roni Stern. "Multi-Agent Path Finding – An Overview". In: *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures*. Hrsg. von Gennady S. Osipov, Aleksandr I. Panov und Konstantin S. Yakovlev. Cham: Springer International Publishing, 2019, S. 96–115. ISBN: 978-3-030-33274-7. DOI: 10.1007/978-3-030-33274-7_6. URL: https://doi.org/10.1007/978-3-030-33274-7_6.

[14] Roni Stern u. a. *Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks*. 2019. arXiv: 1906.08291 [cs.AI].

[15] Nathan Sturtevant. *Pathfinding Benchmarks: File Formats*. URL: https://movingai.com/benchmarks/formats.html. (o. J.)