



Technische Universität Berlin

Ein Raum-Zeit-Scheduler und Trajektorienplaner für ein Schwarmsystem

Masterarbeit

am Fachgebiet Kommunikations- und Betriebssysteme (KBS)

Prof. Dr. Hans-Ulrich Heiß

Fakultät IV Elektrotechnik und Informatik

Technische Universität Berlin

vorgelegt von

Rico Jasper

Betreuer: Daniel Graff

Rico Jasper

Matrikelnummer: 319396

Landsberger Allee 208

10367 Berlin

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum

Unterschrift

Zusammenfassung

Die Vernetzung elektronischer Geräte wird durch den technologischen Fortschritt immer weiter vorangetrieben. Cyber-physische Systeme (CPS) stellen die nächste Evolutionsstufe dieser Entwicklung dar, welche sich aus einer Vielzahl vernetzter Elementen zusammensetzt.

Die steigende Komplexität solcher Systeme erfordert neue Ansätze um diesen Herausforderungen Herr zu werden. *jSwarm* ist ein verteiltes Betriebssystem, welches eine Vielzahl von mobilen Robotern zu einem einzigen System, dem Schwarm, vereint. Durch ein geeignetes Programmiermodell soll die schwerfällige und fehleranfällige Softwareentwicklung verteilter Anwendungen erleichtert werden.

Das Modell sieht vor, verteilte Anwendungen in *Actions* aufzuteilen, die von einem Scheduler in Raum und Zeit eingeplant werden. Ort und Zeitpunkt können durch die Anwendung mithilfe von Constraints eingeschränkt werden. Dadurch gelingt eine Entkopplung der Raum-Zeit-Anforderungen und der eigentlichen Handlung einer Action. Die Auswahl eines Roboters und die Planung der Fahrtrouter übernimmt dabei der Scheduler und ist somit für den Entwickler der Schwarmanwendung transparent.

Gegenstand dieser Arbeit ist die Entwicklung des Raum-Zeit-Schedulers, welcher die Actions sowohl in Zeit als auch Raum einplant. Dazu erfüllt er zwei Kernkompetenzen: Die Jobplanung und die Trajektorienplanung. Die erste Aufgabe umfasst die Auswahl eines Roboters, sowie die Orts- und Zeitbestimmung einer Action. Die Trajektorienplanung bestimmt die Fahrtrouten der Roboter, sodass diese rechtzeitig zum Ausführungsort ihrer Actions gelangen ohne mit bewegten oder unbewegten Hindernissen zu kollidieren.

Summary

The increasing interconnection of electronic devices is driven by technological progress. The next step of this evolution is represented by cyber-physical systems, which consist of a large number of networked elements.

As such systems rise in complexity new approaches are needed to cope with the evoked challenges. *jSwarm* is a distributed operating system which manages a variety of mobile robots as one system: the swarm. The project aims to make software development of distributed applications less cumbersome and error-prone using an appropriate programming model.

Following the model distributed applications are divided into actions which are scheduled in space and time. By attaching constraints to actions the exact location and time can be restricted. This enables to decouple the space-time requirements from the actual operation of an action. Selecting an available robot and pathfinding are taken care of by the scheduler and thus are transparent to the application developer.

Subject of this thesis is the development of the space-time scheduler which schedules actions in both space and time. For this purpose it fulfills two core functions: job planning and trajectory planning. The first function involves selecting a robot, as well as the location and the point in time. The trajectory planning determines the routes of the robots and therefore ensures that they arrive in time at the location without colliding with moving or stationary obstacles.

Inhaltsverzeichnis

Erklärung der Urheberschaft	II
Zusammenfassung	III
Inhaltsverzeichnis	V
1 Einführung	1
1.1 Motivation	1
1.2 Zielsetzung und Abgrenzung	2
1.3 Struktur der Arbeit	3
2 Grundlagen und Stand der Forschung	4
2.1 jSwarm	4
2.2 Path-Velocity-Decomposition	5
3 Problemanalyse	6
3.1 Entwicklung verteilter Raum-Zeit-Anwendungen	6
3.2 Raum- und Echtzeitbedingungen	7
3.3 Fehlersituationen	8
4 Lösungsansatz	9
4.1 Modell des allwissenden Schedulers	9
4.2 Jobplanung	10
4.2.1 Einzelner Job	10
4.2.2 Mehrere Jobs	15
4.2.3 Periodische Jobs	16
4.2.4 Abhängige Jobs	16
4.2.5 Job entfernen	19
4.2.6 Job umplanen	19

4.2.7	Transaktionen	19
4.3	Trajektorienplanung	22
4.3.1	Räumliche Pfadplanung	23
4.3.2	Temporale Pfadplanung	23
4.3.3	Verbotene Regionen	26
4.3.4	Wartezeiten	32
4.4	Knotenauslastung	34
5	Implementierung	36
5.1	Konzepte	36
5.1.1	Kein null	36
5.1.2	Unveränderliche Objekte	37
5.1.3	Fail-Fast	37
5.2	Abhängigkeiten	37
5.2.1	Java 8	37
5.2.2	Java Topology Suite	38
5.2.3	JGraphT	38
5.2.4	la4j	39
5.2.5	straightedge	39
5.3	Typen	39
5.3.1	Gleitkommazahl	39
5.3.2	Zeittypen	39
5.3.3	Pfade	40
5.3.4	Hindernisse	40
5.3.5	Jobs	41
5.4	Komponenten	41
5.4.1	Scheduler	41
5.4.2	Knoten	42
5.4.3	Pathfinder	42
5.5	Schnittstellen	43
5.5.1	Initialisierung des Schedulers	43
5.5.2	Knoten hinzufügen und entfernen	43
5.5.3	Planung eines Jobs	44
5.5.4	Planung periodischer Jobs	45
5.5.5	Planung abhängiger Jobs	46
5.5.6	Umplanen eines Jobs	47

5.5.7 Entfernen eines Jobs	47
5.5.8 Gegenwart setzen	47
5.5.9 Knotenauslastung	48
6 Evaluation	50
6.1 Komplexität	50
6.1.1 Trajektorienplanung	50
6.1.2 Jobplanung	54
6.1.3 Bewertung	56
6.2 Benchmark	58
6.2.1 Trajektorienplanung	58
6.2.2 Jobplanung	62
6.2.3 Bewertung	65
6.3 Problemfälle	66
6.3.1 Allwissenheit	66
6.3.2 Pseudostationäre Hindernisse	66
6.3.3 Orts-Sampling	67
6.3.4 Zeitintervall und Pfadlänge	67
6.3.5 Trajektoriendauer	67
7 Fazit und Ausblick	68
Literaturverzeichnis	69
Abbildungsverzeichnis	70
Tabellenverzeichnis	72
Abkürzungsverzeichnis	73
A Pseudocode	74

Kapitel 1

Einführung

1.1 Motivation

Die technologische Entwicklung der letzten Jahre ist geprägt durch eine zunehmende Vernetzung elektronischer Geräte. Die nächste Evolutionsstufe dieser Entwicklung stellen sogenannte cyber-physicalen Systeme (CPS) dar. **TODO: Zitat** Diese setzen sich aus einer Vielzahl von vernetzten Elementen zusammen, die mithilfe von Sensoren und Aktoren auf die reale Welt Einfluss nehmen können.

Aufgrund der Komplexität der cyber-physicalen Systeme gestaltet sich die Entwicklung von verteilten Anwendungen als schwierig und fehleranfällig. *jSwarm*, ein verteiltes Betriebssystem für mobile Roboter, nimmt sich daher zum Ansatz die Vielzahl an Geräten unter ein einzelnes System zu vereinigen. Ziel dieses Betriebssystems ist die Schaffung einer einheitlichen und zentralen Programmierschnittstelle, die das Entwickeln von verteilten Schwarmanwendungen vereinfacht. Dies ermöglicht weitgehende Transparenz der Heterogenität und Vielfalt der Roboter. Zudem werden essentielle Funktionen zur Koordination, Wegfindung und Wegausführung vom Betriebssystem übernommen.

In einem klassischen Betriebssystem bestimmt der Scheduler, wann ein Prozess auf dem Prozessor ausgeführt wird. Ein Raum-Zeit-Scheduler wählt neben der Zeit zusätzlich einen Ausführungsplatz. Dieser Ort ist relevant für eine Raum-Zeit-Aufgabe die von einem Roboterschwarm bewältigt werden soll.

Im Rahmen dieser Arbeit wird ein Raum-Zeit-Scheduler vorgestellt, welcher in *jSwarm* Anwendung findet. Der Schwarm vereint mehrere physisch unabhängige Roboter (Knoten) unter sich, die in der realen Welt agieren. Zu erledigende Aufgaben bringen Spezifikationen zu Gebiet und Zeitraum mit sich, in dem bzw. zu der sie bearbeitet werden müssen. Eine Aufgabe könnte beispielsweise so aussehen, dass an einem gewissen

Ort ein Bild vom Sonnenuntergang aufgenommen werden soll. Um diese Aufgabe zu erfüllen, muss sich zu gegebener Zeit ein Roboter an diesem Ort befinden, um das Foto schießen zu können.

1.2 Zielsetzung und Abgrenzung

Ziel dieser Arbeit ist die Bereitstellung eines funktionstüchtigen Raum-Zeit-Schedulers. Der Scheduler muss in der Lage sein Raum-Zeit-Aufgaben konfliktfrei in einen bestehenden Schedule einzuplanen, sodass ein gültiger Schedule stets in einen gültigen Schedule transformiert wird. Dazu müssen unter anderen folgende Punkte beachtet werden:

- Ein Roboter kann höchstens eine Aufgabe gleichzeitig bearbeiten.
- Aufgabenspezifikationen an Raum und Zeit müssen erfüllt werden.
- Roboter dürfen nicht mit Hindernissen oder anderen Robotern kollidieren.
- Limitierungen der Roboter (Höchstgeschwindigkeit) müssen beachtet werden.
- Es können nur zukünftige Ereignisse geplant werden.

Zur Einplanung einer Aufgabe ist eine Spezifikation nötig, die Anforderungen an Ort, Zeit und Knoten vorgibt, welche durch den Scheduler bestimmt werden. Dabei muss gewährleistet werden, dass es dem Knoten auch tatsächlich möglich ist, diesen Ort zur angegebenen Zeit zu erreichen, ohne dass andere Aufgaben in Verzug geraten.

Neben den funktionalen Anforderungen muss der Scheduler Einschränkungen hinsichtlich Speicherverbrauch und Berechnungsaufwand genügen. Dazu wird er sowohl in Simulationen (Abschnitt 6.2) als auch in einer theoretischen Komplexitätsanalyse (6.1) untersucht.

Die funktionalen Anforderungen haben jedoch Vorrang gegenüber der Leistung. Da Optimierungen aufwändig und fehleranfällig sind, steht die Korrektheit der Algorithmen im Vordergrund. Des Weiteren wird von einem einfachen Fehlermodell ausgegangen. In einem Schwarmssystem können unzählige Fehlersituationen auftreten. Der Entwicklungsaufwand zur Abdeckung aller Fehlersituationen ist äußerst hoch. Der Fokus liegt daher auf elementaren Scheduler-Funktionen. Eine umfassende zuverlässige Umsetzung zur Handhabung verschiedenster Fehler ist damit Gegenstand zukünftiger Forschungsarbeiten.

1.3 Struktur der Arbeit

Zu Beginn möchte ich im Grundlagenkapitel [2] das Schwarmbetriebssystem **jSwarm** vorstellen, welches den hier entwickelten Scheduler verwendet. Des Weiteren gehe ich im Folgeabschnitt [2.2] auf die Path-Velocity-Decomposition von Kant und Zucker aus [3] ein, die die Basis der **Trajektorienplanung** bildet.

Darauf folgt Kapitel [3], in dem die Problempunkte des Schedulings von Raum-Zeit-Anwendungen identifiziert werden. Diese gilt es anschließend im Folgekapitel [4] zu lösen. Hervorzuheben sind hier die Abschnitte zur **Jobplanung** und **Trajektorienplanung**, welche die beiden Kernaufgaben des Schedulers widerspiegeln.

Details zur Implementierung in Java werden in Kapitel [5] abgehandelt. Hier wird ein allgemeiner Überblick der angewandten Konzepte, verwendeten Bibliotheken und Typen gegeben sowie die Komponenten und Schnittstellen der Umsetzung präsentiert.

In der Evaluation in Kapitel [6] werden die wichtigsten Komponenten sowohl in ihrer Komplexität untersucht als auch deren Zeitverhalten in Benchmarks ermittelt und bewertet. Darüber hinaus werden einige Problemfälle des Schedulers aufgedeckt.

Kapitel [7] schließt die Arbeit mit einem Fazit ab und gewährt einen Ausblick auf zukünftige Arbeiten.

Kapitel 2

Grundlagen und Stand der Forschung

TODO: Roboter/Hardware

2.1 jSwarm

jSwarm ist ein verteiltes Laufzeitsystem um Raum-Zeit-Anwendungen auf einem Roboterschwarm auszuführen. [1][2] Dadurch soll die Heterogenität und Vielfältigkeit der Roboter versteckt werden, indem eine gemeinsame Programmierschnittstelle angeboten wird. Des Weiteren ermöglicht das System die Virtualisierung des Schwarms, sodass mehrere Schwarmanwendungen parallel ausgeführt werden können. Der Programmierer erfährt dabei Transparenz gegenüber Koordination, Synchronisation, Auswahl und Bewegung der Roboter. D.h. er muss sich weder darum kümmern, welcher Roboter zur Erfüllung einer Aufgabe verwendet wird, noch wie er zum Ausführungsort gelangt.

Das Programmiermodell sieht vor, dass Schwarmanwendungen sogenannte *Actions* planen, die innerhalb eines gewissen Areals und während eines bestimmten Zeitintervalls ausgeführt werden. Areal und Zeitintervall bilden gemeinsam ein *Constraint*. Des Weiteren können auch relative Constraints angegeben werden. Diese beziehen sich auf den Ausführungsort und/oder Ausführungszeit einer anderen Action. Eine Schwarmanwendung setzt sich typischerweise aus mehreren Actions zusammen. Die Ergebnisse einer Action können dabei als Eingabe einer anderen Action verwendet werden. Die nötige Koordination und Kommunikation wird von *jSwarm* übernommen.

Die Komponenten des Schwarmsystems teilen sich als lose gekoppelte Dienste auf, die durch ein Nachrichtensystem miteinander kommunizieren (Abbildung 2.1). Dies ermöglicht eine einfache Replikation der Dienste um leichter mit Ausfällen umgehen zu können. Die Dienste lassen sich in zwei Gruppen einordnen: globale Systemdienste und lokale Knotendienste. Die Systemdienste dienen zur Systemverwaltung, Ressourcenal-

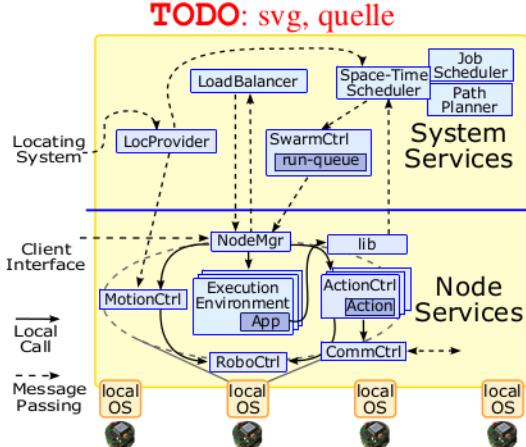


Abbildung 2.1: Architektur

lokation und Steuerung der Roboter. Knotendienste sind für die lokale Verwaltung der Roboter zuständig sowie der Steuerung der Sensoren und Aktoren. Zu jedem Systemdienst existiert immer nur eine aktive Instanz währenddessen Knotendienste auf allen Robotern ausgeführt werden.

TODO: Ergebnisse und Stand

2.2 Path-Velocity-Decomposition

In ihrem Paper *Toward Efficient Trajectory Planning: The Path-Velocity Decomposition* [3] haben Kant und Zucker einen Algorithmus zur Lösung des *Trajectory Planning Problems* (TPP) vorgestellt. Im Gegensatz zu einer statischen Umgebung, in denen sich keine Hindernisse bewegen, wird beim TPP auch die zeitliche Komponente berücksichtigt. Dabei gilt es eine Trajektorie $\vec{x}_\pi : T \rightarrow \mathbb{R}^n$ zu bestimmen, die den Aufenthaltsort $\vec{x} \in \mathbb{R}^n$ eines Roboters in Abhängigkeit der Zeit $t \in T$ beschreibt.

Zur Lösung des TPPs haben Kant und Zucker vorgeschlagen, das Problem in zwei Teile aufzuspalten. Der erste Teil umfasst das statische *Path Planning Problem* (PPP), bei dem lediglich unbewegte Hindernisse $O^s \subset \mathbb{R}^n$ berücksichtigt werden. Der zweite Teil ist das *Velocity Planning Problem* (VPP). Hierbei werden Kollisionen mit bewegten Hindernissen $O^m : T \rightarrow \mathbb{R}^n$ durch Anpassung der Geschwindigkeit vermieden.

Da die Path-Velocity Decomposition ein integraler Bestandteil der Wegfindung des Schedulers darstellt, wird sie zu Gunsten der Kontinuität im Abschnitt **Trajektorienplanung** in Detail vorgestellt.

Kapitel 3

Problemanalyse

3.1 Entwicklung verteilter Raum-Zeit-Anwendungen

Die Entwicklung einer Anwendung für verteilte Systeme gestaltet sich meist als schwierig aufgrund fehleranfälliger Konzepte wie Verteilung und Nebenläufigkeit. [6] Die Einführung von Raum und Zeit erschwert die Situation zusätzlich. Das Ziel von *jSwarm* ist daher die Komplexität von unter anderen Heterogenität, Transparenz oder Nebenläufigkeit zu verbergen. Die Virtualisierung des Schwarms stellt ein weiteres Merkmal dar, sodass mehrere Schwarmanwendungen zeitgleich auf demselben physischen Schwarm ausgeführt werden können. Des Weiteren soll die Anwendung von Raum und Zeit entkoppelt werden. Die Fragen „Wo?“ und „Wann?“ werden somit getrennt von den Fragen „Was?“ und „Wie?“ beantwortet. Diese Trennung wird durch die Formulierung von Raum- und Zeit-Constraints ermöglicht.

Eine Schwarmanwendung wird dazu in Aktionen aufgeteilt, die im weiteren Verlauf Jobs genannt werden. Diese stellen die kleinsten Programmeinheiten dar, die in den Schedule eingefügt werden können. Diese Programmteile klären somit „was“ „wie“ getan wird. Die Beantwortung der übrigen Fragen, „wo“ und „wann“ die Jobs zur Anwendung kommen, gelingt durch die Constraints, die die Schwarmanwendung für die Jobs definiert. Konkret geben die Constraints ein Areal A und ein Zeitintervall T vor, in denen der Ausführungsort $(x, y) \in A$ und die Ausführungszeit $t \in T$ des Jobs liegen müssen.

Neben dem Ort und der Zeit besitzt ein Job noch zwei weitere Eigenschaften, die für den Scheduler relevant sind. Diese sind die Dauer d und der Roboter, dem der Job aufgetragen wird. Während der Ausführung darf sich der Roboter nicht von der Stelle bewegen. Das bedeutet, dass Jobs stets stationär sind. Darüber hinaus kann ein Roboter nur einen Job gleichzeitig durchführen.

Der Schedule des Gesamtsystems setzt sich somit aus den lokalen Schedules der Roboter zusammen. Diese enthalten eine Liste von eingeplanten Jobs, die zu einer bestimmten Zeit an einem festen Ort und für eine vorgegebene Dauer ausgeführt werden müssen.

Die Konsequenzen für den Scheduler sind somit, dass dieser in der Lage sein muss, Jobs verschiedener Schwarmanwendungen unter Berücksichtigung der Constraints einzuplanen. Dazu gehört die Wahl eines geeigneten Ortes, Startzeitpunktes und Roboters.

Neben der Einplanung eines einzelnen Jobs soll es des Weiteren ermöglicht werden eine Jobmenge atomar in den Schedule aufzunehmen. Dies umfasst die Ausführung von periodischen und in Abhängigkeit stehenden Jobs. Es gilt dabei zu verhindern, dass diese nur teilweise eingeplant werden. Gelingt es dem Scheduler die gesamte Menge an Jobs zuzuteilen, so darf kein Element dem Schedule hinzugefügt werden.

3.2 Raum- und Echtzeitbedingungen

Da die Roboter des Schwarmsystems in der realen Welt agieren sollen, sind sie auch deren physikalischen Gegebenheiten unterworfen. Dies betrifft insbesondere Raum und Zeit. Wenn ein Roboter zu einer bestimmten Zeit an einem bestimmten Ort eine Aktion ausführen soll, so muss er auch rechtzeitig dorthin gelangen. Dabei muss beachtet werden, dass der Roboter sich nicht frei durch Raum und Zeit bewegen kann. Beispielsweise kann er seine Höchstgeschwindigkeit nicht überschreiten und sollte sowohl statische als auch dynamische Hindernisse umgehen.

Eine Trajektorie $\vec{x}_\pi : T \rightarrow \mathbb{R}^2$ beschreibt dazu die Bahn des Roboters über dessen Einsatzdauer. Wird der lokale Schedule des Roboters verändert (z.B. durch Einplanung eines Jobs), so hat dies in der Regel auch eine Aktualisierung seiner Trajektorie zur Folge.

Ein weiterer Aspekt der Zeit ist die Unterscheidung zwischen Gegenwart und Zukunft. Es können nur zukünftige Ereignisse geplant werden. Eine zusätzliche Einschränkung ist durch Verarbeitungs- und Nachrichtenlaufzeiten gegeben. Aktionen können daher nicht beliebig früh in der Zukunft geplant werden.

Der Scheduler muss Algorithmen und Schnittstellen zu Verfügung stellen um diese Raum- und Echtzeitbedingungen erfüllen zu können. Seine Trajektorie darf dazu zu keinem Zeitpunkt zu einer Kollision mit einem Hindernis führen. Des Weiteren kann die Trajektorie zu keinem Zeitpunkt vor der Gegenwart manipuliert werden. Außerdem muss bedacht werden, dass ein neu berechneter Teil der Bahn möglicherweise verspätet

oder gar nicht beim Roboter angelangt. Es muss dennoch gewährleistet werden, dass es infolgedessen zu keinen Kollisionen kommt.

3.3 Fehlersituationen

Aufgrund der Größe und Komplexität des Schwarms sind Fehlersituationen unabwendbar. Roboter können spontan ausfallen oder Fehlfunktionen aufweisen. Aktionen werden möglicherweise unvollständig oder gar nicht ausgeführt. Nachrichten treffen verspätet ein oder gehen verloren. Der Scheduler muss Mittel und Wege bereitstellen, damit das Gesamtsystem mit derartigen Situationen umgehen kann. Dies erfordert zudem ein Fehlermodell, welches erwartete Fehler beschreibt.

Kapitel 4

Lösungsansatz

Der Scheduler ist die zentrale Instanz zur Zuteilung von Aufgaben im Schwarm. Der Schwarm setzt sich aus autonomen Ausführungseinheiten, den Knoten, zusammen. Dabei handelt es sich in der Regel um Roboter. Der Scheduler bestimmt welche Aufgabe, auch Job genannt, wann, wo und von welchem Knoten ausgeführt wird. Außerdem berechnet er die Fahrtrouten der Knoten. Somit besitzt er zwei Kernaufgaben, die Jobplanung und die Trajektorienplanung. Die Jobplanung umfasst das Einplanen und Verwalten der Jobs. Die Trajektorienplanung beinhaltet das Berechnen und Aktualisieren der Trajektorien. Beide Aufgaben sind eng miteinander verknüpft, da die Einplanung eines Jobs meist auch eine Trajektorienanpassung erfordert. Ansonsten könnten Jobs am falschen Ort ausgeführt werden.

4.1 Modell des allwissenden Schedulers

Da die Trajektorien der Roboter kollisionsfrei sein sollen, muss der Scheduler Wissen über den Zustand der realen Welt besitzen. Die Welt wird dabei ohne explizite Grenzen in zwei Raumdimensionen (x, y) und einer Zeitdimension (t) abgebildet. Um Ausweichrouten planen zu können, benötigt der Scheduler Standortinformationen der Hindernisse und Roboter. Einfachheitshalber wird davon ausgegangen, dass der Scheduler allwissend über die Hindernisse der Welt ist. Das bedeutet, dass es keine unbekannten Hindernisse geben kann, die ansonsten zu Kollisionen führen könnten. Des Weiteren wird angenommen, dass die Roboter den Verlauf ihrer Trajektorien in der realen Welt exakt verfolgen. Abweichungen und Toleranzen müssen demnach nicht berücksichtigt werden.

Die Informationen über die Hindernisse der Welt sind unveränderlich. Das bedeutet, dass nachträglich keine neuen Hindernisse definiert werden können. Somit wird bei-

spielsweise ausgeschlossen, dass eine vormals kollisionsfreie Route sich nun mit dem neuen Hindernis schneidet.

Im Gegensatz zu Hindernissen können Roboter hinzugefügt und entfernt werden. Um jedoch keine potentiellen Kollisionskonflikte einzuführen, wird die Annahme gemacht, dass inaktive Roboter nie mit anderen Robotern zusammenstoßen. Dies bedeutet, dass aus der Sicht des Schedulers hinzugefügte Roboter aus dem Nichts erscheinen und entfernte Roboter auch dorthin wieder verschwinden. Darüber hinaus dürfen frisch aktivierte Roboter nicht mit anderen Hindernissen oder Robotern kollidieren.

Die zwei zentralen Zustandsinformationen eines Roboters sind seine Liste von Jobs und seine Trajektorie. Die Trajektorie beschreibt seinen Aufenthaltsort in Abhängigkeit der Zeit beginnend von seinem Initialisierungszeitpunkt bis ans „Ende der Zeit“. Die Jobs besitzen eine exakte Start- und Endzeit sowie einen exakten Ausführungsort. Ein Roboter kann immer nur einen Job gleichzeitig ausführen und muss sich zwischen der Start- und Endzeit am Ausführungsort befinden.

Das Fehlermodell geht davon aus, dass Roboter nicht spontan ausfallen und keine Fehlfunktionen aufweisen. Andernfalls wäre die Allwissenheit des Schedulers gefährdet, da man nicht in jedem Fall den Standort eines defekten Roboters bestimmen könnte. Aufgaben können jedoch stets abgebrochen und Roboter ohne unabgeschlossene Jobs entfernt werden. Somit existiert ein gewisser Raum an Möglichkeiten um außerhalb des Schedulers einige Fehlersituationen aufzulösen. Nachrichtenverlust oder -verzögerungen werden jedoch unterstützt.

4.2 Jobplanung

4.2.1 Einzelner Job

Bei der Zuteilung einer Aufgabe muss der Scheduler drei Variablen belegen: Ort, Zeit und Knoten. Diese müssen zum einen den gestellten Bedingungen einer Aufgabenspezifikation genügen und zum anderen realisierbar sein.

Der hier beschriebene Ansatz belegt nach und nach die Variablen mit konkreten Werten und überprüft sie auf Umsetzbarkeit. Erweist sich ein Wert einer zu Beginn belegten Variable als unergiebig, so wird ihr nach dem Backtracking-Prinzip ein neuer Wert zugewiesen bis entweder eine gültige Belegung gefunden wurde oder alle Kombinationsmöglichkeiten erschöpft sind. Die Prozedur SCHEDULE_SINGLE aus Algorithmus A.1 beschreibt dieses Vorgehen in Pseudocode.

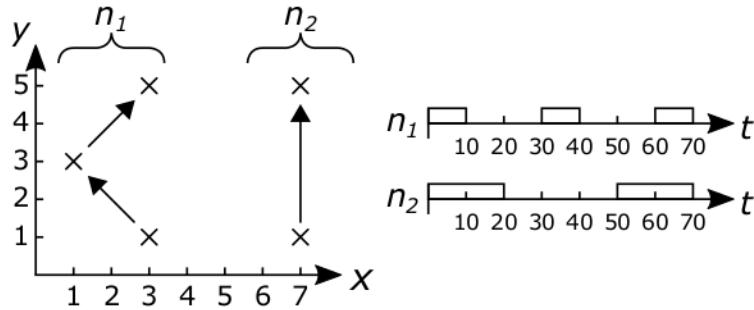


Abbildung 4.1: Ausgangssituation

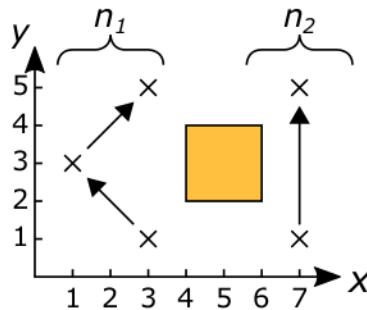


Abbildung 4.2: Aufgabenareal

Im Normalfall befinden sich schon vor der Einplanung einer neuen Aufgabe bereits andere Aufgaben im Schedule. Für das folgende Beispiel wird in Abbildung 4.1 die Ausgangssituation dargestellt. Es existieren zwei Knoten n_1 und n_2 . Beide haben eine Höchstgeschwindigkeit von $v_{max} = 0.5$. n_1 wurden bereits drei Aufgaben zugewiesen, n_2 dagegen zwei.

Aufgabenspezifikation

Der Scheduler weist eine Aufgabe einem Knoten zu, der diese an einem bestimmten Ort zu einer bestimmten Zeit ausführt. Der Aufgabe geht eine Spezifikation voraus, die Anforderungen an den Ort und den Zeitpunkt stellt. Außerdem gibt diese die Dauer d zur Bearbeitung der Aufgabe an. Der Ort (x, y) muss innerhalb eines definierten Areals $A \subseteq \mathbb{R}^2$ liegen und der Zeitpunkt t innerhalb eines Zeitintervalls $T = [t_{min}, t_{max}]$, sodass $(x, y) \in A$ und $t \in T$. Der Scheduler nimmt solch eine Spezifikation entgegen und erzeugt daraus eine konkrete Aufgabe, die er einem Knoten zuweist.

Im Beispiel besitzt das Areal die Form eines Rechtecks mit den Eckpunkten $(4, 2)$ und $(6, 4)$ (siehe Abbildung 4.2). Das Zeitintervall ist durch $[10, 60]$ gegeben. Die Aufgabendauer beträgt 10.

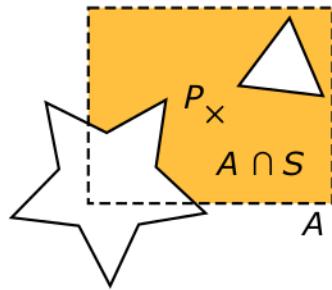


Abbildung 4.3: Hindernisse durchgezogen, Areal A gestrichelt, Schnittmenge $A \cap S$ gelblich und Punkt P

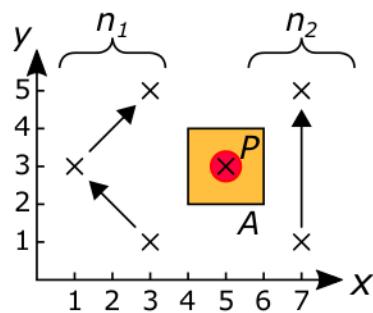


Abbildung 4.4: Konkreter Ort

Ort

Der erste Schritt von der Spezifikation zur konkreten Aufgabe ist die Ortsbestimmung. Es wird innerhalb des spezifizierten Areals A ein Punkt P ermittelt, welcher sich nicht innerhalb eines statischen Hindernisses befindet. Dieser Punkt ist also ein Element aus der Schnittmenge der freien Fläche S und des Areals der Spezifikation (siehe Abbildung 4.3):

$$P \in S \cap A \quad (4.1)$$

Dynamische Hindernisse werden an dieser Stelle nicht berücksichtigt, da ein genauer Zeitpunkt noch nicht bestimmt wurde. Möglicherweise führt der gewählte Ortspunkt zu keiner realisierbaren Belegung der übrigen Variablen, wenn beispielsweise dynamische Hindernisse den Weg versperren. In solch einem Fall muss ein neuer alternativer Ortspunkt gewählt werden. Die Anzahl der Alternativen wird dabei beschränkt, um eine endlose Suche zu vermeiden.



Abbildung 4.5: Pausen in Grün

Im Falle des Aufgabenareals aus dem Beispiel wird der Punkt (5, 3) gewählt (siehe Abbildung 4.4).

Knotenpause

Da ein Ortspunkt gewählt wurde, kann nun nach einem Knoten gesucht werden, welcher diesen Punkt innerhalb des erlaubten Zeitintervalls erreichen kann.

Ein Knoten führt ihm aufgetragene Aufgaben in einer Liste nach und nach aus. Er kann nur eine Aufgabe zur selben Zeit bearbeiten. Es kann nur ein Knoten für die neue Aufgabe ausgewählt werden, der ausreichend Zeit dazu hat.

Im Normalfall muss ein Knoten vor und nach der einzuplanenden Aufgabe ebenfalls Aufgaben erledigen. Er muss genügend Zeit haben, um nach Abschluss der vorherigen Aufgabe zur neuen zu fahren, diese abzuarbeiten, und rechtzeitig die nachfolgende zu beginnen. Zudem muss die neue Aufgabe weiterhin innerhalb des spezifizierten Zeitraums begonnen werden.

Die Zeit zwischen zwei Aufgaben eines Knotens wird als Pause bezeichnet. Die neue Aufgabe kann demnach nur in eine Pause eingeplant werden. In Abbildung 4.5 sind die Pausen der Knoten aus dem Beispiel dargestellt. Formal muss eine Pause drei Bedingungen erfüllen:

1. Die Aufgabe kann innerhalb des Zeitintervalls begonnen werden.

$$t_{max} - t_1 \geq \frac{s_1}{v_{max}} \quad (4.2)$$

2. Die Aufgabe kann innerhalb des Zeitintervalls abgeschlossen werden

$$t_2 - t_{min} \geq \frac{s_2}{v_{max}} + d \quad (4.3)$$

3. Die Pausendauer ist ausreichend lang für Fahrzeiten und Aufgabendauer.

$$t_2 - t_1 \geq \frac{s_1 + s_2}{v_{max}} + d \quad (4.4)$$

s_1 und s_2 sind die Strecken der Luftlinien zu und von der neuen Aufgabe. v_{max} ist die Höchstgeschwindigkeit des Knotens. t_1 und t_2 geben das Zeitintervall $[t_1, t_2]$ der Pause an. d ist die Dauer der neuen Aufgabe. Das Zeitintervall, in dem die neue Aufgabe gestartet werden muss, ist durch $[t_{min}, t_{max}]$ gegeben.

Die ersten beiden Bedingungen stellen sicher, dass die neue Aufgabe während des vorgegebenen Zeitintervalls begonnen wird. Die dritte Bedingung ist wahr, wenn zwischen Ende und Beginn der bereits eingeplanten Aufgaben genügend Zeit für den Fahrweg und der Aufgabendauer vorhanden ist.

Jeder Knoten besitzt Pausen, in denen er keine Aufgabe bearbeitet. In diesem Sinne sind Wege zu Aufgabenorten ebenfalls Bestandteile von Pausen. Nur in jenen Pausen, welche die drei Bedingungen erfüllen, kann die neue Aufgabe eingeplant werden. Wurde solch eine Pause gefunden, so muss nun noch geprüft werden, ob auch gültige Wege zu und von dem Aufgabenort existieren.

Alle drei Bedingungen werden nur von der großen Pause von n_2 erfüllt. Diese dauert von $t_1 = 20$ bis $t_2 = 50$ an. Die Strecke zum neuen Aufgabenort beträgt $s_1 = \sqrt{8} \approx 2.83$. Dieselbe Strecke muss (zufälligerweise) auch zur nachfolgenden Aufgabe zurückgelegt werden: $s_2 = s_1 \approx 2.83$. Die früheste Startzeit ist $t_{min} = 10$, die späteste $t_{max} = 60$, die Dauer $d = 10$. Diese Werte werden zur Prüfung in die Bedingungen eingesetzt.

Stehen mehrere Pausen zur Verfügung, die alle Bedingungen erfüllen, so lässt sich bei der Wahl der Pause eine Heuristik nach dem geringsten Umweg Δs anwenden. Dazu berechnet man die Strecke s_{12} zwischen den Aufgabenorten des Vorgängers und Nachfolgers und subtrahiert die oberen Strecken s_1 und s_2 :

$$\Delta s := s_{12} - (s_1 + s_2) \quad (4.5)$$

Δs bezeichnet somit die Strecke unter Vernachlässigung von Hindernissen, die der Knoten zusätzlich fahren muss, um zur neuen Aufgabe zu gelangen. Somit werden jene Pausen bevorzugt, deren Luftlinie nah an dem neuen Aufgabenort vorbeiführt. Hindernisse werden jedoch vernachlässigt, da dies ansonsten eine Wegfindung erfordern würde.

Die gewählte Pause des Beispiels hat somit einen Umweg von $\Delta s \approx 1.66$.

Wegplanung

Bis hierhin wurden ein Ortspunkt und eine Pause eines bestimmten Knoten als Kandidaten für die Einplanung einer Aufgabe gewählt. Ob dies jedoch auch möglich ist, ergibt

sich erst durch die Wegplanung. Wie bereits im Unterabschnitt **Knotenpause** erwähnt, besitzt die einzuplanende Aufgabe im Regelfall einen Vorgänger und einen Nachfolger. Daher muss der Knoten vom Vorgänger zur neuen Aufgabe fahren, diese ausführen, und anschließend zum Nachfolger fahren. Es sind jeweils Trajektorien für beide Fahrwege zu ermitteln, welche Kollisionen mit Hindernissen vermeiden.

Der konkrete Startzeitpunkt wird erst durch die Wegplanung bestimmt. Die Aufgabenspezifikation und die gewählte Pause geben dazu den Zeitrahmen vor. Innerhalb dieses Rahmens versucht die Wegfindung eine Trajektorie zu bestimmen, die so früh wie möglich zum Aufgabenort führt. Allerdings muss hierbei beachtet werden, dass der Knoten auch genügend Zeit zum Ausführen der Aufgabe hat, ohne dass ihm ein dynamisches Hindernis in die Quere kommt.

Für den zweiten Fahrweg muss die Wegplanung eine Trajektorie liefern, die als Zielzeitpunkt den Start des Nachfolgers verwendet. Es wird hier also nicht nach dem schnellstmöglichen Weg gesucht.

Dieses Vorgehen kann auch dem Algorithmus **A.4** entnommen werden. Details zur Bestimmung der Trajektorien werden im Abschnitt **Trajektorienplanung** erläutert.

Gegenwart und eingefrorener Horizont

Wie im Abschnitt **3.2** geschildert, können nur zukünftige Ereignisse geplant werden. Die Gegenwart markiert eine Zeitgrenze, vor der keine Änderungen am Schedule möglich sind. Aufgrund der Berechnungsdauer und der Nachrichtenlaufzeit muss diese Grenze zusätzlich in die Zukunft verschoben werden. Diese Grenze bezeichne ich als *eingefrorenen Horizont*.

Bei der Job- und Trajektorienplanung muss dieser Horizont beachtet werden. Aufgabenspezifikationen werden daher implizit eingeschränkt, sodass der frühest mögliche Startzeitpunkt t_{min} effektiv bei $t'_{min} = \max(t_{min}, t_{FH})$ liegt, wobei t_{FH} den eingefrorenen Horizont bezeichnet. Des Weiteren werden bei der Wahl der Pause nur jene Pausen betrachtet, deren Endzeitpunkte t_2 hinter t_{FH} liegen. Ihre Startzeitpunkte t_1 werden verschoben, falls sie vor t_{FH} liegen, sodass $t'_1 = \max(t_1, t_{FH})$.

4.2.2 Mehrere Jobs

Neben der Planung eines einzelnen Jobs soll es auch ermöglicht werden mehrere Jobs atomar einzuplanen. Das bedeutet, dass der Scheduler entweder alle angegebenen Jobs im Schedule aufnimmt oder keinen. Das Scheduling einer Menge von Jobs lässt sich

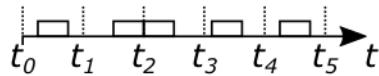


Abbildung 4.6: Periodischer Schedule

zurückführen auf das mehrfache Einplanen einzelner Jobs wie im Abschnitt 4.2.1 beschrieben.

Die beiden folgenden Abschnitte **Periodische Jobs** und **Abhängige Jobs** präsentieren zwei Möglichkeiten mehrere Jobs atomar in den Schedule zu übernehmen.

4.2.3 Periodische Jobs

Jobs können in einer endlichen Wiederholung periodisch eingeplant werden. Ähnlich wie im Abschnitt 4.2.1 wird in einer Spezifikation eine Dauer d und ein Areal A angegeben. Die Dauer bezieht sich dabei auf eine einzelne Wiederholung. Die Zeitrestriktion wird durch eine Startzeit t_0 und eine Periodendauer $d_P \leq d$ festgelegt. Die Anzahl der Wiederholungen wird durch n definiert und muss endlich sein, da für jede Wiederholung die Umsetzbarkeit geprüft wird und gegebenenfalls Trajektorien berechnet werden.

Die erste Periode beginnt zum Zeitpunkt t_0 . Pro Periode muss eine Wiederholung J_i mit $i \in \{1, \dots, n\}$ erfolgen (siehe Abbildung 4.6). Somit muss ein Job J_i im Zeitintervall $[t_{i-1}, t_i - d]$ starten, wobei $t_i := t_0 + (i - 1) \cdot d_P$. Die Wiederholungen können entweder allesamt am selben Ort $P \in A$ ausgeführt werden oder jede Wiederholung J_i an einem unabhängigen Ort $P_i \in A$ (siehe Algorithmus A.2).

4.2.4 Abhängige Jobs

Eine weitere Möglichkeit mehrere Jobs atomar einzuplanen ist das Scheduling abhängiger Jobs. Dazu werden wie im Abschnitt 4.2.1 Aufgabenspezifikationen der Jobs und zusätzlich ein Abhängigkeitsgraph übergeben. Dieser zyklenfreier gerichtete Graph enthält alle Jobs als Knoten. Eine Kante von Job C zu Job A gibt dabei an, dass C erst begonnen werden kann nachdem A abgeschlossen wurde. Dies ermöglicht C die Ergebnisse von A als Eingabeparameter zu verwenden.

Ein Anwendungsszenario könnte so aussehen, dass zwei Bälle in einen Eimer geworfen und herausgeholt werden sollen. Dies erfordert drei Aufgaben: Job A und B werfen die Bälle und C holt sie aus dem Eimer. Das Holen kann jedoch nicht erfolgen, solange die Bälle nicht auch geworfen wurden (siehe Abbildung 4.7). Die Aufgabenspezifikationen der Aufgaben ist Tabelle 4.1 zu entnehmen.

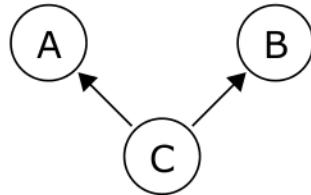


Abbildung 4.7: Abhängigkeitsgraph

Job	t_{min}	t_{max}	d
A	20	70	5
B	0	100	5
C	0	100	10

Tabelle 4.1: Aufgabenspezifikationen

Zur Einplanung werden die Jobs topologisch sortiert, sodass erst A oder B und anschließend C in den Schedule aufgenommen werden. Dabei muss beachtet werden, dass sich die Anforderungen an die Ausführungszeiten implizit durch die Zuteilung ändern. Wenn A beispielsweise zum Zeitpunkt $t_A = 40$ gestartet wird, dann kann C frühestens zur Zeit $t_C = 45$ begonnen werden. Aber auch B muss eingeschränkt werden, da eine Ausführung nach $t_B = 95$ zu keiner gültigen Belegung von t_C führt. Die Aufgabenspezifikation muss deshalb sowohl vor als auch während der Belegung der Variablen eingeschränkt werden.

Die Gleichungen unten geben an, wie die neuen Werte der frühest- und spätestmöglichen Startzeiten zu berechnen sind. Die α -Funktion aus Gleichung 4.6 dient als Hilfsfunktion um rekursiv den impliziten frühestmöglichen Endzeitpunkt einer Spezifikation s zu bestimmen. dep gibt dazu alle Spezifikationen, von der s abhängig ist, zurück ($\text{dep}(s_C) = \{s_A, s_B\}$). β aus Gleichung 4.7 berechnet den impliziten spätestmöglichen Startzeitpunkt, wobei dep^{-1} alle abhängigen Spezifikationen angibt ($\text{dep}^{-1}(s_A) = \{s_C\}$).

$$\alpha(s) := \max \alpha(\text{dep}(s)) \cup \{s.t_{min}\} + s.d \quad (4.6)$$

$$\beta(s) := \min [\beta(\text{dep}^{-1}(s)) - s.d] \cup \{s.t_{max}\} \quad (4.7)$$

$$s.t'_{min} := \alpha(s) - s.d \quad (4.8)$$

$$s.t'_{max} := \beta(s) \quad (4.9)$$

Vor der Zuweisung konkreter Startzeiten sehen die impliziten Spezifikationen aus wie in Tabelle 4.2. Wenn einer Aufgabe einer Spezifikation bereits ein konkreter Start-

Job	t'_{min}	t'_{max}	d
A	20	70	5
B	0	95	5
C	25	100	10

Tabelle 4.2: Implizite Aufgabenspezifikationen

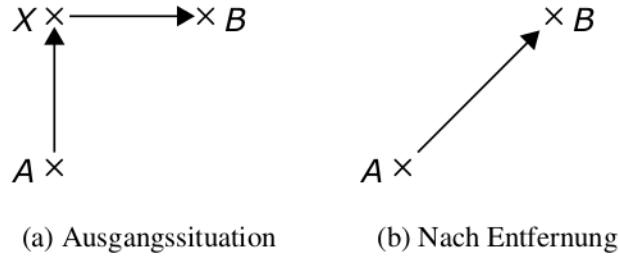


Abbildung 4.8: Entfernung eines Jobs

zeitpunkt t zugewiesen wurde, dann lassen sich die angegebenen Formeln ebenfalls anwenden indem $t_{min} = t_{max} = t$ gesetzt werden.

Nach der topologischen Sortierung könnte die Reihenfolge, in der das Einplanen erfolgt, entweder A, B, C oder B, A, C lauten. Je eher ein Job eingeplant wird, desto mehr Möglichkeiten stehen für eine gültige Variablenbelegung offen. Darum sollten zunächst die Jobs eingeplant werden, deren implizite Deadline am ehesten ist. Somit steigt die Wahrscheinlichkeit, dass alle Jobs erfolgreich in den Schedule aufgenommen werden können. Demnach sollte die Reihenfolge A, B, C gewählt werden.

Vor der Ausführung von C sollte überprüft werden, ob die Würfe A und B auch erfolgreich waren. Wenn jedoch C direkt nach A oder B anschließt, bleibt keine Zeit zum Informationsaustausch. Zwischen zwei abhängigen Aufgaben kann deshalb eine Art Sicherheitsabstand d_m definiert werden. Die Formeln 4.6 und 4.7 müssen entsprechend angepasst werden:

$$\alpha(s) := \max [\alpha(\text{dep}(s)) + d_m] \cup \{s.t_{min}\} + s.d \quad (4.10)$$

$$\beta(s) := \min [\beta(\text{dep}^{-1}(s)) - s.d - d_m] \cup \{s.t_{max}\} \quad (4.11)$$

Der geschilderte Algorithmus zur Einplanung mehrerer abhängiger Jobs ist in A.3 in Pseudocode dargestellt.

4.2.5 Job entfernen

Ein bereits eingeplanter Job kann auch wieder aus dem Schedule entfernt werden. In Abbildung 4.8a sind die Ausführungsorte der drei Aufgaben A , B und X dargestellt. Wird X einfach entfernt, so verläuft die Trajektorie von A zu B immer noch über das entfernte X . Daher wird die Trajektorie neu geplant, sodass sie direkt von A zu B verläuft wie in Abbildung 4.8b. Bei der Pfadfindung muss allerdings beachtet werden, dass die Trajektorie sich nur ab der Zeit nach dem eingefrorenen Horizont t_{FH} ändern darf.

Das Entfernen eines Jobs kann allerdings auch fehlschlagen, da die Pfadfindung scheitern kann.¹ Darum existiert auch eine Schedulerfunktion, die das Entfernen ohne Neuberechnung der Trajektorie ermöglicht.

4.2.6 Job umplanen

Das Umplanen eines Jobs wird durch eine atomare Komposition des Entfernens (Abschnitt 4.2.5) und erneuten Planens (Abschnitt 4.2.1) erreicht. Dies ermöglicht eine Änderung der Aufgabenspezifikation, sodass sich Ort, Zeit oder zugeteilter Knoten des Jobs ändern können. Falls die Pfadfindung daran scheitert eine neue Trajektorie zu berechnen, so werden keine Änderungen vorgenommen.

4.2.7 Transaktionen

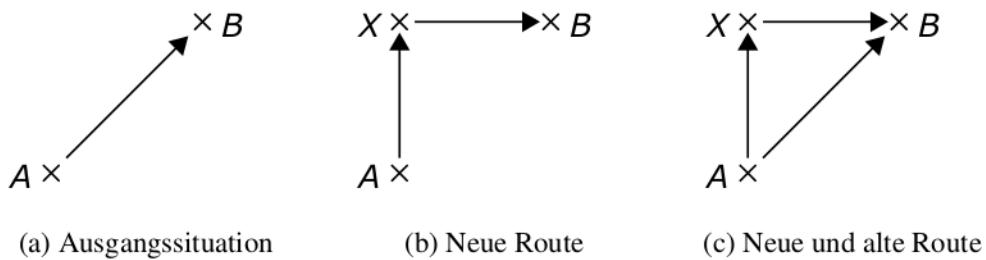
Wenn der Schedule durch das Planen oder Entfernen eines Jobs aktualisiert wird, so müssen auch die betroffenen Knoten über Nachrichten darüber informiert werden. Allerdings können diese Nachrichten verloren gehen oder verspätet eintreffen. Erst wenn der Knoten die Aktualisierung bestätigt, ist gewiss, ob er auch die neue Fahrtroute übernommen hat. Wenn in der Zeit der Ungewissheit Änderungen vorgenommen werden sollen, dann ist nicht bekannt, welche Route der Knoten nun abfahren wird.

Nebenläufiges Einplanen

Das Beispiel in Abbildung 4.9a zeigt die Route eines Knotens, die von A zu B führt ($A \rightarrow B$). In einer Aktualisierung wurde nun Job X eingeplant, sodass die Route nun $A \rightarrow X \rightarrow B$ lautet (siehe Abbildung 4.9b). Dies wird nun zum physischen Roboter kommuniziert. Dieser könnte die Änderungen jedoch ablehnen² oder nie erhalten.

¹Die neue Route könnte durch ein dynamisches Hindernis blockiert sein. Die Wegfindung ist nicht in der Lage solchen Hindernissen räumlich auszuweichen.

²Die Ablehnung des Roboters könnte zum Beispiel eintreten, wenn die Nachricht zu spät eintrifft und die Änderungen bereits veraltet sind.

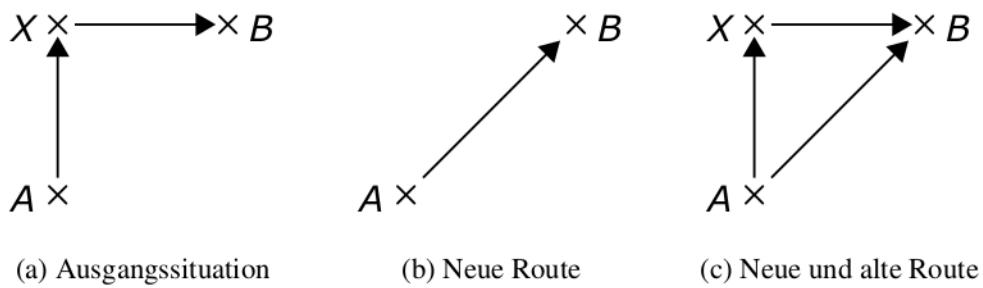
Abbildung 4.9: Einplanung des Jobs X

Wenn nun unabhängig von der Einplanung des Jobs X eine Trajektorie eines anderen Roboters berechnet werden soll, so führt das zu einem Problem: Die Trajektorie könnte sich mit einer der beiden Routen $A \rightarrow B$ oder $A \rightarrow X \rightarrow B$ schneiden. Eine einfache Lösung des Problems könnte vorsehen, dass keine Änderungen am Schedule vorgenommen werden dürfen bis eine Bestätigung (ACK) oder Ablehnung (NAK) des ersten Roboters eingetreten ist. Allerdings würde somit die Nachrichtenübermittlung hier schnell zum Flaschenhals werden. Gewünscht ist jedoch, dass auch andere Operationen nebenläufig durchgeführt werden können, die den Schedule verändern während auf ein ACK/NAK gewartet wird. Ermöglicht wird dies durch Transaktionen.

Eine Transaktion ist eine Scheduleroperation, wie das Einplanen oder Entfernen von Jobs. Eine Transaktion kann durch ein sogenanntes *Commit* angewandt oder durch *Abort* abgebrochen werden. Bis zum Abschluss durch Commit oder Abort stellt eine Transaktion eine Alternative des Schedules dar.

Damit es zu keinen Kollisionen kommt, werden bei der Wegfindung nicht nur die Routen im Schedule berücksichtigt, sondern auch die alternativen Routen der Transaktionen. Ein Roboter kann somit als zwei unabhängige Hindernisse auftreten (siehe Abbildung 4.9c). Diese Vorgehensweise stellt in den Fällen beider Routen sicher, dass kein anderer Roboter mit dem ersten kollidiert. Sobald der Knoten eine der beiden Routen bestätigt hat, wird die andere Route verworfen, sodass er nur als *ein* Hindernis auftritt.

Bei der nebenläufigen Planung mehrerer Jobs auf einem Knoten muss auch beachtet werden, dass Trajektorienabschnitte nicht beliebig aktualisiert werden dürfen. Generell dürfen Trajektorien nicht in den Zeiträumen verändert werden, in denen der Knoten Jobs ausführt. Unbestätigte Routen dürfen allerdings auch nicht aktualisiert werden. Bis eine Transaktion nicht bestätigt oder abgebrochen wurde, ist deren finaler Verlauf nicht bekannt. Angenommen man möchte neben X noch eine weitere Aufgabe Y zwischen A und B einplanen, so könnten die finalen Routen wie folgt aussehen:

Abbildung 4.10: Entfernung des Jobs X

- X und Y werden nicht übernommen
 $A \rightarrow B$
- Nur X wird übernommen
 $A \rightarrow X \rightarrow B$
- Nur Y wird übernommen
 $A \rightarrow Y \rightarrow B$
- X und Y werden übernommen
 $A \rightarrow X \rightarrow Y \rightarrow B$ oder
 $A \rightarrow Y \rightarrow X \rightarrow B$

Dies übersteigt jedoch die Anzahl von zwei Routenalternativen während beide Transaktionen noch unbestätigt sind. Falls eine weitere Aufgabe Z ebenfalls neben X und Y eingeplant werden sollte, so wären noch mehr Alternativen zu betrachten. Um diese Komplexität zu vermeiden, wird die Trajektorie zwischen A und B während der Transaktion von X gesperrt sodass Y erst nach Abschluss Transaktion eingeplant werden kann. Die Sperrung betrifft jedoch nur fremde Transaktionen. Die eigene Transaktion kann die Trajektorie zwischen A und B mehrfach aktualisieren, um beispielsweise mehrere Jobs X_1, X_2 dazwischen einzuplanen.

Nebenläufiges Entfernen

Da das Entfernen eines Jobs ebenfalls Trajektoriänderungen auslöst, findet auch diese Operation in Form einer Transaktion statt. Hier könnte eine Situation so aussehen, dass eine Aufgabe X , die zwischen A und B eingeplant wurde, nun gelöscht werden soll (siehe Abbildung 4.10). Wie auch im vorherigen Abschnitt **Nebenläufiges Einplanen** wird hier die Trajektorie zwischen A und B gesperrt. Zusätzlich wird auch X zum Löschen gesperrt. Das bedeutet, keine andere Transaktion kann diesen Job nochmals entfernen.

4.3 Trajektorienplanung

Die zweite Kernaufgabe des Schedulers ist die Trajektorienplanung. Damit ein Roboter eine Aufgabe am korrekten Ort erledigen kann, muss ihn seine Trajektorie auch dort hin führen. Durch das Einplanen und Entfernen von Jobs ändert sich seine Trajektorie jedoch immerzu. Der Trajektorienplaner sorgt dafür, dass der Roboter rechtzeitig und ohne kollisionsfrei zur rechten Zeit am rechten Ort angelangt.

Die Path-Velocity-Decomposition aus Abschnitt 2.2 des Grundlagenkapitels bildet die Basis der Trajektorienplanung. Zum Teil geben die folgenden Abschnitte die Inhalte aus [3] für ein besseres Verständnis wieder, greifen aber auch neue Punkte auf. Unter 4.3.3 wird genauer auf die Berechnung der verbotenen Regionen eingegangen und 4.3.4 zeigt, wie besonders langsame Trajektorienabschnitte vermieden werden können.

Ziel der Trajektorienplanung ist, einen Knoten von einem Startpunkt $(\vec{x}_I, t_I) \in \mathbb{R}^2 \times T$ zu einem Zielpunkt $(\vec{x}_F, t_F) \in \mathbb{R}^2 \times T$ zu führen ($T = [t_I, t_F]$). Hindernisse dürfen dabei nicht mit dem Knoten kollidieren. Die Hindernisse werden dazu als zeitabhängige Punktmenge $O : T \rightarrow \mathcal{P}(\mathbb{R}^2)$ dargestellt, wogegen der Knoten nur als punktgroß³ $\vec{x}_\pi : T \rightarrow \mathbb{R}^2$ angenommen wird. $\vec{x}_\pi(t)$ gibt damit dessen Aufenthaltsort zur Zeit t an. Eine kollisionsfreie Route liegt dann vor, wenn $\vec{x}_\pi(t) \cap O(t) = \emptyset, \forall t \in T$.

Hindernisse werden in ihrer Beweglichkeit unterschieden, sodass sich O aus statischen O^s und dynamischen Hindernissen O^m zusammensetzt: $O(t) = O^s \cup O^m(t)$. Diese Unterscheidung motiviert die Dekomposition in eine räumliche Pfadplanung, die allen statischen Hindernissen umfährt, und eine temporale Pfadplanung, die dagegen dynamischen Hindernissen durch Geschwindigkeitsanpassung ausweicht.

Das Ergebnis der räumlichen Planung ist ein kontinuierlicher Pfad $\pi \subset \mathbb{R}^2$, welcher \vec{x}_I und \vec{x}_F verbindet. π stellt somit die Menge an Punkten dar, welche der Roboter befahren muss, um sein Ziel zu erreichen. Die temporale Planung legt dagegen fest, wann der Roboter diese Punkte beschreitet. π wird dazu in Abhängigkeit des Bogenmaßes s des Pfades parametrisiert: $\vec{x}_\pi(s)$. Für s gilt es eine zeitliche Zuordnung $s : T \rightarrow S$ zu finden, wobei $S = [s_I, s_F]$ das Bogenmaßintervall des Pfades π bezeichnet. Durch Komposition erhält man die Trajektorie $\vec{x}_\pi(t) = (\vec{x}_\pi \circ s)(t)$.

Die Aufgabe der temporalen Planung ist, eine Zuordnung $\vec{x}_\pi : T \rightarrow \pi$ zwischen der Zeit t und den Punkten des Pfades π zu finden, welche allen dynamischen Hindernissen entgeht.

³Die Punktgröße vereinfacht die folgenden Betrachtungen. Da die Roboter des Schwarmes jedoch keine Punktgröße aufweisen, müssen die Hindernisse gepuffert werden.

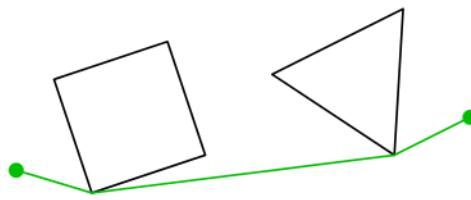


Abbildung 4.11: Kürzester Pfad

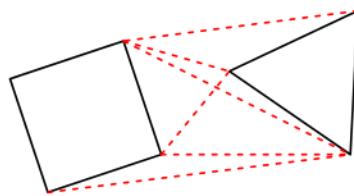


Abbildung 4.12: Sichtlinien zwischen Polygonen

4.3.1 Räumliche Pfadplanung

Der erste Schritt der Wegfindung vernachlässigt jegliche temporalen Aspekte zur Lösung des Path-Planning-Problems (PPP). Daher werden bei der Bestimmung des Pfades π lediglich die statischen Hindernisse O^s betrachtet, welche als Polygone auftreten.

Im zweidimensionalen Raum verläuft der kürzeste Pfad von Eckpunkt zu Eckpunkt der Hindernisse (siehe Abbildung 4.11).⁴ Zur Bestimmung dieses Pfades bildet man einen Graphen, der den Startpunkt \vec{x}_I , den Endpunkt \vec{x}_F und die Eckpunkte V der Hindernisse als Knoten enthält. Die Menge der Kanten des Graphen umfasst jene Knoten, die zueinander sichtbar sind (siehe Abbildung 4.12). Das heißt, ihre Sichtlinie wird nicht durch Hindernisse blockiert.⁴ Als Kantengewicht dient der euklidische Abstand des jeweiligen Punktpaares. Zu dem so konstruierten Graph kann nun die kürzeste Route zwischen Start- und Endpunkt ermittelt werden, welche den Pfad π darstellt.

4.3.2 Temporale Pfadplanung

Während die Kollisionsfreiheit mit statischen Hindernissen durch die Berechnung von π an dieser Stelle garantiert werden kann, muss nun noch das Geschwindigkeitsprofil durch die Zuordnung s bestimmt werden, um dynamischen Hindernissen zu entgehen (Velocity-Planning-Problem, kurz VPP). Bei dynamischen Hindernissen handelt es sich

⁴Die Hindernisse enthalten nicht ihren eigenen Rand ($O^s \cap \partial O^s = \emptyset$), sodass die Polygonkanten als befahrbar gelten.

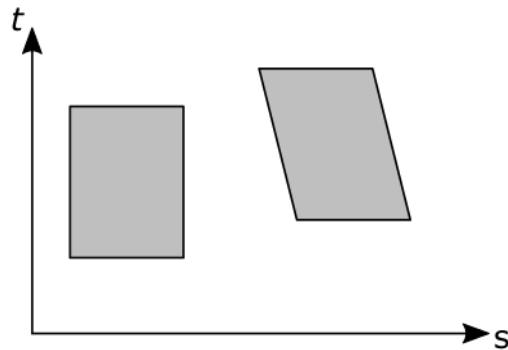


Abbildung 4.13: Verbotene Regionen

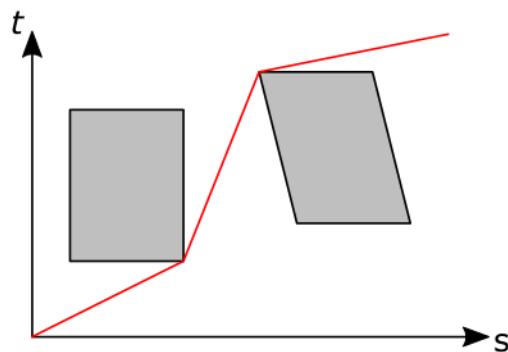


Abbildung 4.14: Geschwindigkeitsprofil

ebenfalls um Polygone, die allerdings mit der Zeit verschoben werden. Grundlage zur Bestimmung des Geschwindigkeitsprofils bieten sogenannte verbotene Regionen, deren Berechnung im folgenden Abschnitt 4.3.3 ausführlich beschrieben wird.

Verbotene Regionen sind Gebiete des $s \times t$ -Raums, die nicht befahren werden dürfen. Sie geben an zu welchen Zeiten t der $x \times y$ -Pfad π aufgrund dynamischer Hindernisse unpassierbar ist (siehe Abbildung 4.13). s stellt hierbei den Bogenlängenparameter der Ortsfunktion $\vec{x}_\pi(s)$ dar.

Durch die Wahl eines Geschwindigkeitsprofils $s : T \rightarrow S$, welches keine verbotenen Regionen im $s \times t$ -Raum schneidet, werden alle dynamischen Hindernisse durch Justierung der Geschwindigkeit entlang der Route gemieden. Der nachfolgende Schritt bestimmt nun einen neuen Pfad $\sigma \subset T \times \mathbb{R}$ innerhalb dieses Raumes. Dieser legt das Geschwindigkeitsprofil des Roboters fest (siehe Abbildung 4.14). Man kann ihn auf ähnlicher Weise bestimmen wie bereits die Lösung des PPPs. Allerdings muss dabei zusätzlich beachtet werden, dass der Pfad nicht rückwärts in der Zeit verlaufen kann und eine maximale Geschwindigkeit nicht überschritten wird. Da es sich bei einer Dimension im $s \times t$ -Raum um die Zeit handelt, werden folgende Regeln hinzugefügt:

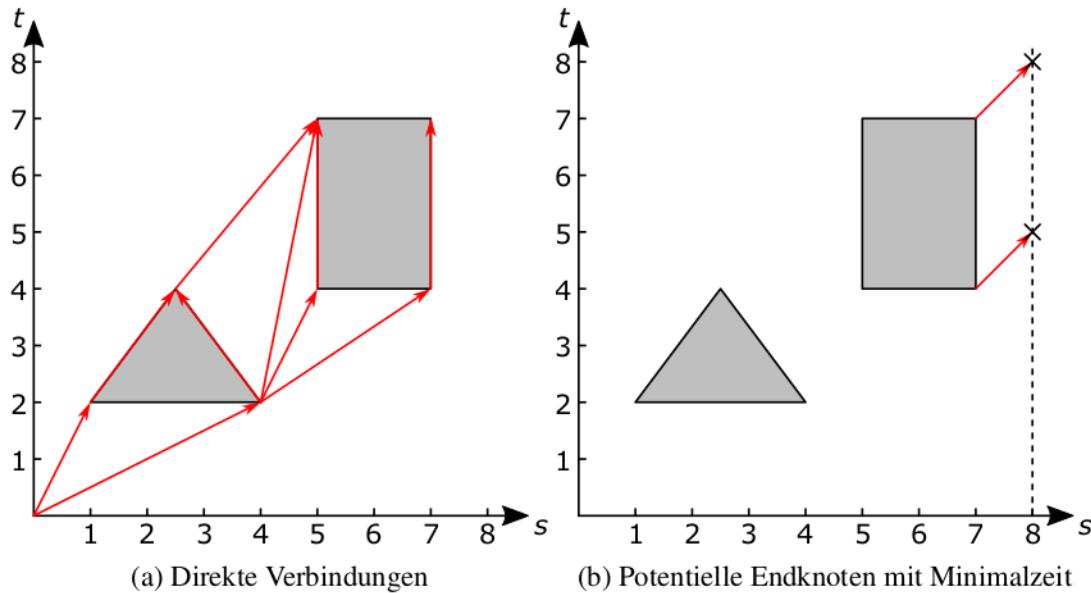


Abbildung 4.15: Graphkonstruktion des Geschwindigkeitsprofils

1. Der Roboter kann sich nur in der Zeit vorwärts bewegen. Darum sind zwei Knoten des Graphen nur dann sichtbar, wenn die Zeit des ersten vor dem des zweiten liegt.
 2. Der Roboter kann sich nicht unendlich schnell bewegen, sondern besitzt eine Höchstgeschwindigkeit.

Formal sehen die Regeln so aus:

1. $t_i < t_j$
 2. $\left| \frac{s_j - s_i}{t_j - t_i} \right| \leq v_{max}$

Wobei $n_i = (s_i, t_i)$ der erste und $n_j = (s_j, t_j)$ der zweite Knoten ist.

Wie schon beim PPP lässt sich so ein Graph konstruieren (Abbildung 4.15a) und somit der Pfad bestimmen, welcher das Geschwindigkeitsprofil beschreibt.

Um eine Route mit minimaler Fahrzeit zu bestimmen, müssen dem Graphen zusätzliche Knoten und Kanten hinzugefügt werden (siehe Abbildung 4.15b). Diese Knoten liegen auf einer Geraden mit $s = s_F$ und dienen als potentielle Endknoten. Jeder dieser Knoten ist mit einem anderen Knoten des Graphen verbunden, welcher entweder der Startpunkt (s_I, t_I) oder einer der Eckpunkte der verbotenen Regionen ist. Die Kanten dieser Paare haben einen Anstieg entsprechend der Höchstgeschwindigkeit v_{max} des Roboters. Zur Bestimmung des Pfades σ verwenden die Kanten deren Zeitkosten als Gewicht. Die Suche nach der kostengünstigsten Route, welche den Startknoten mit einem der Endknoten verbindet, führt somit zur frühestmöglichen Ankunftszeit des Roboters am Ziel.

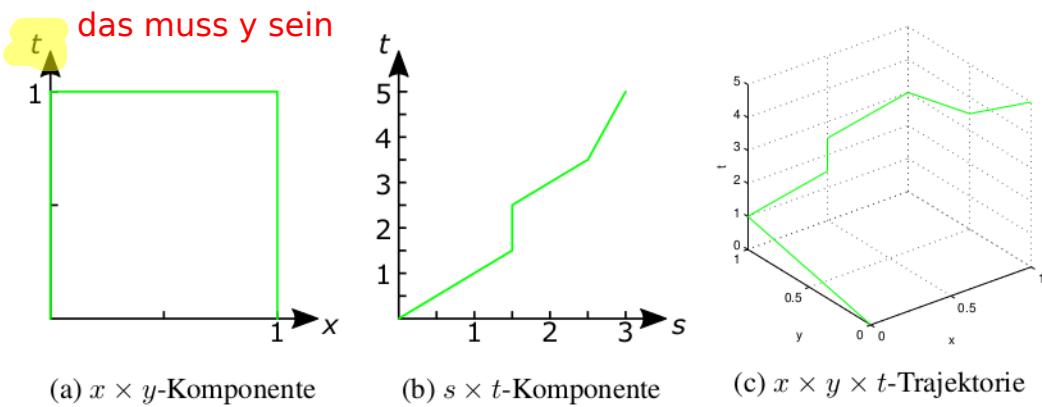


Abbildung 4.16: Trajektorie

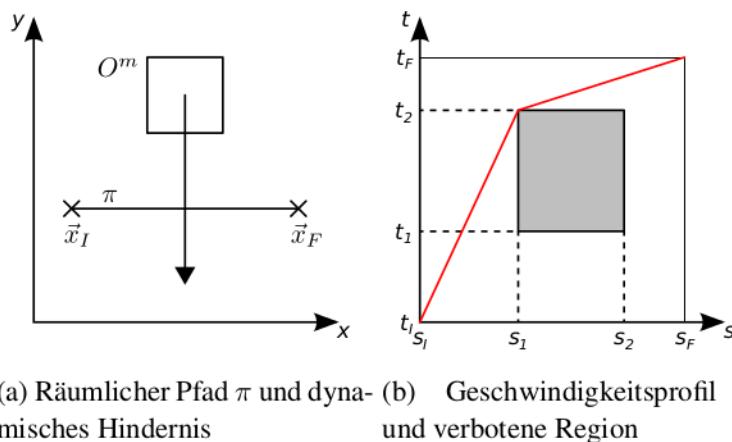


Abbildung 4.17: Abbildung eines dynamischen Hindernisses als verbotene Region

Aus dem räumlichen Pfad π und dem temporalen Pfad σ kann man nun eine dreidimensionale Trajektorie τ des $x \times y \times t$ -Raums berechnen.⁵ Dazu werden die Daten des Geschwindigkeitsprofils genutzt um Zeitwerte für die Eckpunkte des räumlichen Pfades zu bestimmen. In gleicher Weise werden Ortspunkte zu den Eckpunkten des Geschwindigkeitsprofils anhand der Daten des räumlichen Pfades ermittelt. Das Ergebnis ist ein $x \times y \times t$ -Pfad, welcher die Eckpunkte beider Pfadkomponenten enthält (siehe Abbildung 4.16).

4.3.3 Verbotene Regionen

In Abbildung 4.17 wird dargestellt, wie ein dynamisches Hindernis im $s \times t$ -Raum zu einer verbotenen Region transformiert wurde. Im linken Bild 4.17a ist der Pfad π zu sehen. Der Roboter muss vom Punkt \vec{x}_I zu \vec{x}_F gelangen ohne das eingezeichnete Hindernis zu

⁵ $\tau = \{ (x, y, t) \mid t \in T \wedge \vec{x}_\pi(t) = (x, y) \}$

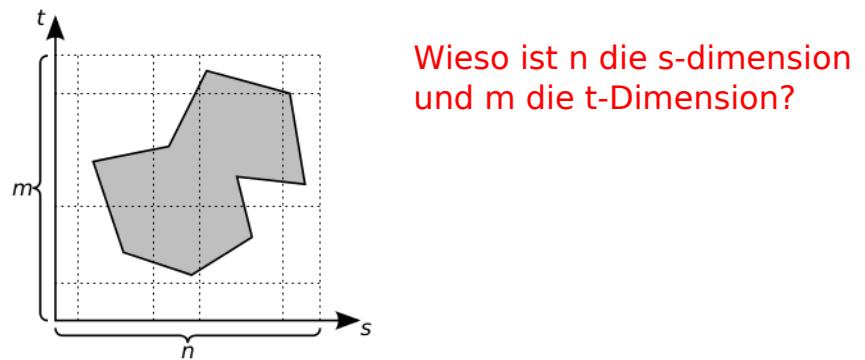


Abbildung 4.18: Kachelmuster der segmentweisen Berechnung verbotener Regionen

Definition einführen: Robotersegment, Hindernissegment

schneiden, welches den Abschnitt $[s_1, s_2]$ des Pfades im Zeitraum $[t_1, t_2]$ blockiert. Dies ist anhand des rechten Bildes 4.17b deutlich zu entnehmen. Der Roboter entgeht einer Kollision, indem er seine Geschwindigkeit soweit verlangsamt, dass das Hindernis die Strecke genau dann freigibt, wenn er am Punkt $\vec{x}_\pi(s_1)$ angelangt. Danach beschleunigt der Roboter, um sein Ziel rechtzeitig zu erreichen.

Zur Berechnung der verbotenen Regionen werden alle dynamischen Hindernisse separat betrachtet. Wie der Roboter selbst, verfolgt das Hindernis eine Trajektorie, die in lineare Segmente unterteilt ist. Die gesamte Region wird segmentweise berechnet. Bei n Trajektoriensegmenten des Roboters und m Segmenten des Hindernisses entsteht so ein $n \times m$ großes Kachelmuster, wobei jede Kachel einen Ausschnitt der ganzen Region enthält (siehe Abbildung 4.18).

$\vec{x}_i^R := (x_i^r, y_i^r)^T$ bezeichnet den i -ten Eckpunkt des Roboterpfades und $\vec{X}_j^m := (x_j^m, y_j^m, t_j^m)^T$ den j -ten Eckpunkt der Hindernistrajektorie. Das i -te Robotersegment legt dabei das Bogenlängenintervall fest: $S_i := [s_i^r, s_{i+1}^r] \ni s$; und das j -te Hindernissegment das Zeitintervall: $T_j := [t_j^m, t_{j+1}^m] \ni t$.⁶ Des Weiteren wird $\vec{v}_j^m := \Delta t_j^{m-1} (\Delta x_j^m, \Delta y_j^m)^T$ definiert als der Geschwindigkeitsvektor des Hindernisses während des j -ten Segments.⁷

Es werden drei Fälle unterschieden, um die Kachel einer verbotenen Region zu bestimmen (siehe Abbildung 4.19):

- Der reguläre Fall, in dem weder $\Delta \vec{x}_i^r$ noch $\Delta \vec{v}_j$ gleich null sind und zudem linear unabhängig:

$$\det \begin{bmatrix} \Delta \vec{x}_i^r & \Delta \vec{v}_j \end{bmatrix} \neq \vec{0} \quad (4.12)$$

⁶ s_i und t_i sind gegeben durch $s_i := \sum_{k=1}^{i-1} \|\vec{x}_{i+1}^r - \vec{x}_i^r\|$ und $t_j := t_j^m$

⁷ $\Delta(\cdot)_i := (\cdot)_{i+1} - (\cdot)_i$

Ich verstehe nicht,
was genau ein
Robotersegment
ist, da wir doch
noch gar keine
Geschwindigkeiten
für den Roboter
haben ..

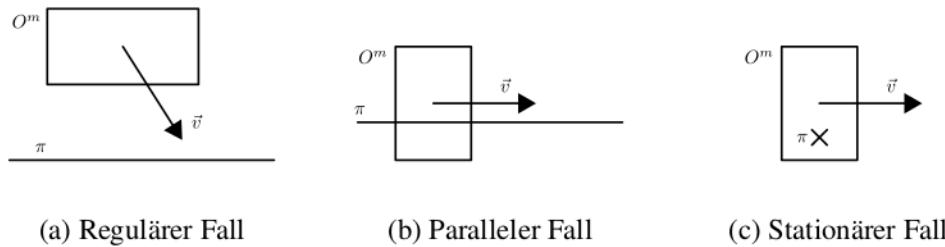
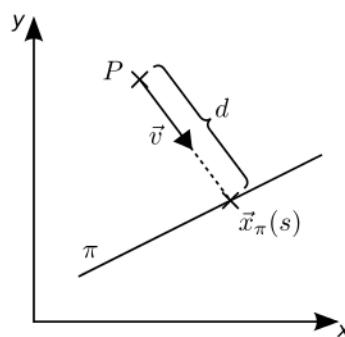


Abbildung 4.19: Fallunterscheidung zur Berechnung verbotener Regionen

Abbildung 4.20: Bewegter Hindernispunkt P und Segment des Pfades π

- Der parallele Fall, in dem der Geschwindigkeitsvektor \vec{v}_j^m parallel zum Segment $\Delta \vec{x}_i^r$ verläuft oder null ist:

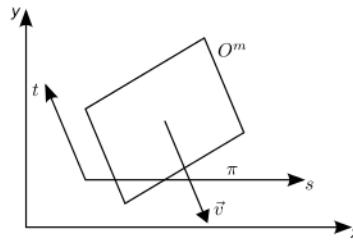
$$\det \begin{bmatrix} \Delta \vec{x}_i^r & \Delta \vec{v}_j \end{bmatrix} = \vec{0} \wedge \Delta \vec{x}_i^r \neq \vec{0} \quad (4.13)$$

- Der stationäre Fall, in dem das Robotersegment keine Länge besitzt:

$$\Delta \vec{x}_i^R = \vec{0} \quad (4.14)$$

Regulärer Fall

Im regulären Fall wird die verbotene Region durch Transformation der Polygonpunkte $O^m (t_j^m)$ erzeugt. Zum Verständnis wird ein einzelner Hindernispunkt P betrachtet. Dieser Punkt besitzt die Geschwindigkeit \vec{v}_j^m und wird zu einem bestimmten Zeitpunkt t auf eine bestimmte Stelle s des Streckenabschnitts des Roboterpfades treffen. In der Abbildung 4.20 ist zu sehen, dass die Stelle anhand des Schnittpunkts mit dem verlängerten Geschwindigkeitsvektor zu ermitteln ist. Um den Zeitpunkt t zu bestimmen, muss zunächst die Entfernung d zwischen dem Startpunkt des Punktes und dem Schnitt-

Abbildung 4.21: $s \times t$ -Basis

punkt berechnet werden. Teilt man die Entfernung durch die absolute Geschwindigkeit, so erhält man den Zeitpunkt t .

Eine andere Herangehensweise ist, den Streckenabschnitt und den negativen Geschwindigkeitsvektor als eine Basis des $s \times t$ -Raumes aufzufassen (siehe Abbildung 4.21). Mit Hilfe dieser Basis lassen sich die Koordinaten (s, t) einfach in (x, y) -Koordinaten umrechnen:

$$\begin{bmatrix} x - x_i^r \\ y - y_i^r \end{bmatrix} = (s - s_i^r) \cdot \vec{e}_s - (t - t_j^m) \cdot \vec{v}_j^m \quad (4.15)$$

Wobei \vec{e}_s der Einheitsvektor des Streckenabschnitts des Roboters ist.⁸ Diese Gleichung lässt sich in das folgende Gleichungssystem umwandeln, dessen Lösung zum transformierten Polygon im $s \times t$ -Raum führt:

$$\begin{bmatrix} x - x_i^r \\ y - y_i^r \end{bmatrix} = \begin{bmatrix} \vec{e}_s & -\vec{v}_j^m \end{bmatrix} \begin{bmatrix} s - s_i^r \\ t - t_j^m \end{bmatrix} \quad (4.16)$$

Die Beschneidung ist sehr dürtig erklärt. Insbesondere die zeitliche Beschneidung ist mir unklar. Solltest du nicht die komplette Geometrie beachten?

Da jedoch das Polygon die Strecke des Roboters nicht immer in seiner Gänze durchläuft, muss es beschnitten werden (siehe Abbildung 4.22). Zum einen wird der Schnittbereich durch den Streckenabschnitts S_i des Roboters begrenzt und zum anderen durch das Zeitintervall T_j . Die Beschneidung legt somit das Ausmaß der aktuellen Kachel fest.

Paralleler Fall

Nicht in jedem Fall kann die Berechnung der verbotenen Region eines Hindernisses wie vorherigen Abschnitt Regulärer Fall erfolgen. Bewegt sich das Hindernis parallel zur Strecke des Roboters oder befindet sich im Stillstand, so spannen Strecken- und Ge-

⁸Der Einheitsvektor lässt sich durch die Gleichung $\vec{e}_s = \frac{\Delta x_i^r}{\|\Delta x_i^r\|}$ bestimmen.

An welcher Stelle gehst du auf die Zeitparameter der Robotertrajektorie ein? Das bleibt hier komplett außen vor.

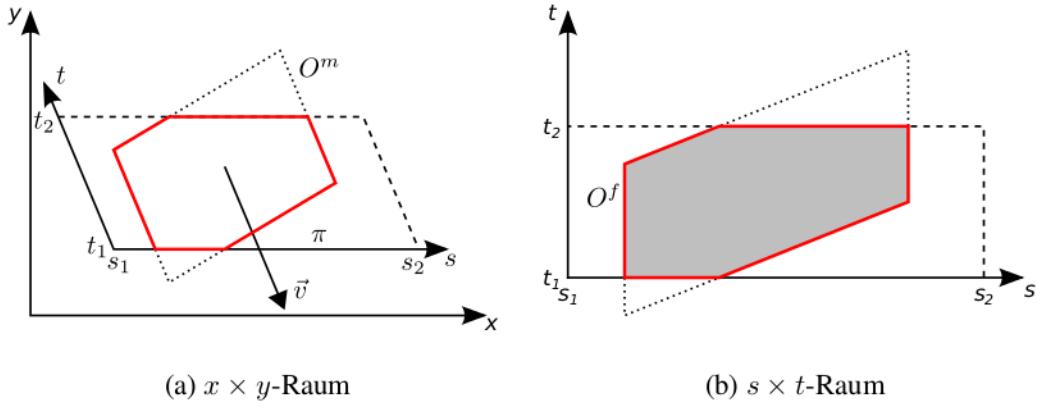
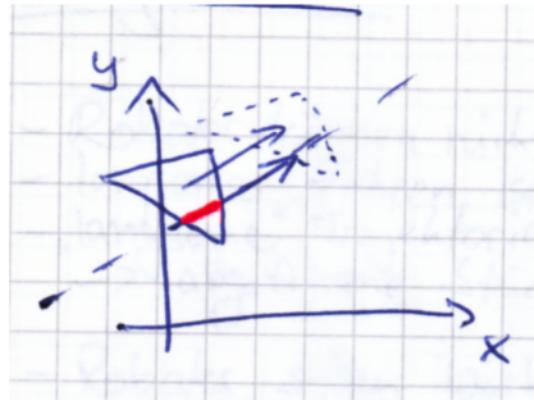


Abbildung 4.22: Beschneidung des Polygons

Abbildung 4.23: Schnittmenge der Geraden g mit dem Hindernis O^m

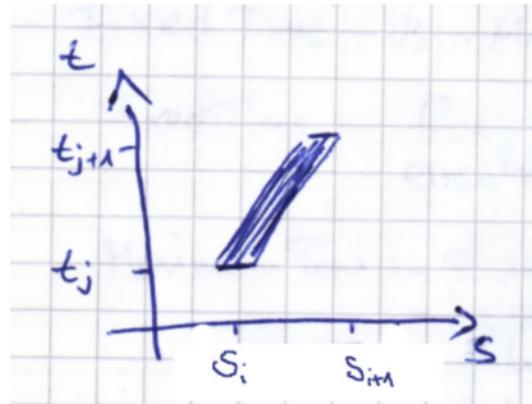
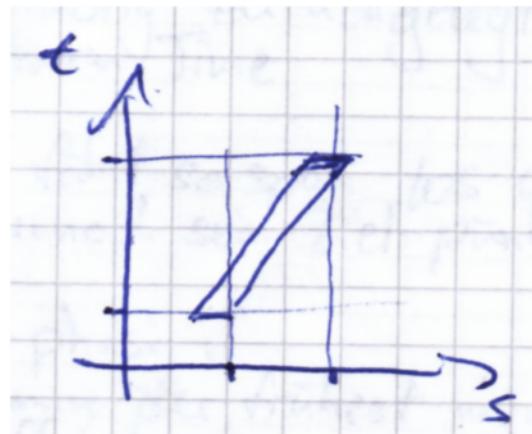
schwindigkeitsvektor keine Basis auf. Der parallele Fall deckt hierbei jene Situationen ab, in denen der Streckenabschnitt des Roboters eine Länge $\Delta s_i^r > 0$ aufweist.

Betrachten wir eine Gerade g auf der sich der Pfadabschnitt des Roboters befindet. Allgemein lässt sich eine Mengenfunktion $S : T_j \rightarrow \mathcal{P}(\mathbb{R})$ definieren, die zu einem Zeitpunkt t verbogene Bogenlängen s angibt:

$$S(t) := [(O^m(t) \cap g) - \vec{x}_i^r(t)] \cdot \vec{e}_s + s_i^r \quad (4.17)$$

Da sich das Hindernis O^m parallel zum Pfad bewegt, schneidet es zu jeder Zeit $t \in T_j$ die Gerade in Bezug zum Ort $\vec{x}^m(t)$ stets an derselben Stelle (siehe Abbildung 4.23). Daher kann man für den parallelen Fall die Menge $S(t)$ auch wie folgt berechnen:

$$S'(t) = S(t_j) + \vec{v}_j^m \cdot \vec{e}_s (t - t_j) \quad (4.18)$$

Abbildung 4.24: Verschiebung der Schnittlinie im $s \times t$ -RaumAbbildung 4.25: Beschneidung der Schnittfläche $S(T_j)$

Somit bestimmt man die Schnittmenge nur für den Zeitpunkt t_j und verschiebt diese dann in Abhängigkeit der Zeit t (siehe Abbildung 4.24).

Da die Strecke jedoch endlich ist und die Funktion S die Schnittpunkte für die gesamte Gerade g angibt, muss die Menge wie schon im regulären Fall beschnitten werden (siehe Abbildung 4.25).

Stationärer Fall

Der letzte Fall findet seine Anwendung bei stationären Pfadsegmenten des Roboters. Sowohl der reguläre als auch der parallele Fall sind hier ungeeignet, da sich kein Einheitsvektor \vec{e}_s bestimmen lässt. Stattdessen befindet sich der Roboter über den Zeitraum T_j an derselben Position $\vec{x}_i^r = \vec{x}_{i+1}^r$. Demnach ist zu bestimmende Kachel der verbetenen Region nur eine vertikale Linie der Länge Δt_j^m an der Stelle $s_i^r = s_{i+1}^r$.

Wenn auch das Streckensegment des Hindernisses stationär ist, muss lediglich überprüft werden, ob der Punkt $\vec{x}_i^r \in O^m(t_j)$ ist. Falls dies zutrifft, ist die Region ein Punkt

Ich verstehe nicht, was der Sinn hier ist. Wenn der Roboter stationär ist, also sich nicht bewegen soll, wie soll ich dann sein Geschwindigkeitsprofil festlegen? Dies ist doch konstant 0 ..
Mir fehlt die Idee bei dieser Beschreibung.

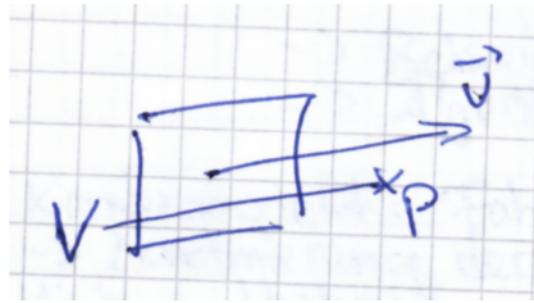


Abbildung 4.26: Spur V des Punktes \vec{x}_i^r

an der Stelle (s_i^r, t_j) . Andernfalls muss bestimmt werden, zu welchen Zeiten die Stelle s_i^r blockiert wird.

Zur Bestimmung dieser Zeiten wird das Hindernis O^m zur Zeit t_j betrachtet. Der Punkt \vec{x}_i^r berührt nur jene Punkte des Hindernisses, welche zum Zeitpunkt t_j auf der Spur $V := \{ \vec{x}_i^r - \vec{v}_j^m t \mid t \in T_j \}$ liegen (siehe Abbildung 4.19). Die Schnittpunkte der Spur V mit dem Hindernis $O^m(t_j)$ geben die Zeitpunkte $T_{FR} \subseteq T_j$ wieder, an denen der Punkt des Roboterpfades blockiert wird:

$$T_{FR} = [(V \cap O^m(t_j)) - \vec{x}_i^r] \cdot \left(-\vec{v}_j^m / (\vec{v}_j^m)^2 \right) + \vec{t}_j^m \quad (4.19)$$

4.3.4 Wartezeiten

Im Abschnitt 4.3.2 werden zur Erstellung des Geschwindigkeitsprofils Start (s_I, t_I) und Ziel (s_F, t_F) über die Eckpunkte der verbotenen Regionen durch gerade Strecken verbunden (siehe Abbildung 4.14). Ein Nachteil dieses Vorgehens ist jedoch, dass beliebig langsame Geschwindigkeiten $v > 0$ auftreten können. Zum Stehen kommt der Roboter dagegen im Regelfall allerdings nur, wenn er eine Aufgabe durchführt. Sehr langsame Geschwindigkeiten können jedoch ineffizient oder sogar nicht realisierbar sein.

Stattdessen ist ein Verhalten vorzuziehen, bei dem der Roboter erst an Ort und Stelle wartet, um anschließend mit einer höheren Geschwindigkeit v_L zum Ziel zu gelangen. Das Warten ist auch im Falle von Trajektoriänderungen vorteilhaft. Beispielsweise könnte ein Roboter eine lange Pause zwischen zwei Aufgaben haben. Die Einplanung einer neuen Aufgabe innerhalb der Pause ist daher wahrscheinlich. Würde der Roboter bereits zur nächsten Aufgabe unterwegs sein, bevor es zur Anpassung des Fahrweges kommt, so wäre er bereits einen unnötigen Umweg gefahren.

Zur Realisierung der Wartezeiten ist eine Modifikation des Graphen notwendig, da dieser ursprünglich wie in Abbildung 4.27a üblicherweise keine stationären Kanten enthält. In Abbildung 4.27b ist dagegen ein Graph zu sehen, der Kanten und Knoten nach

TODO: Abbildung

(a) Regulärer Graph ohne Wartezeiten

TODO: Abbildung

(b) Ergänzung des Graphen

Abbildung 4.27: Graphen ohne und mit Wartezeit

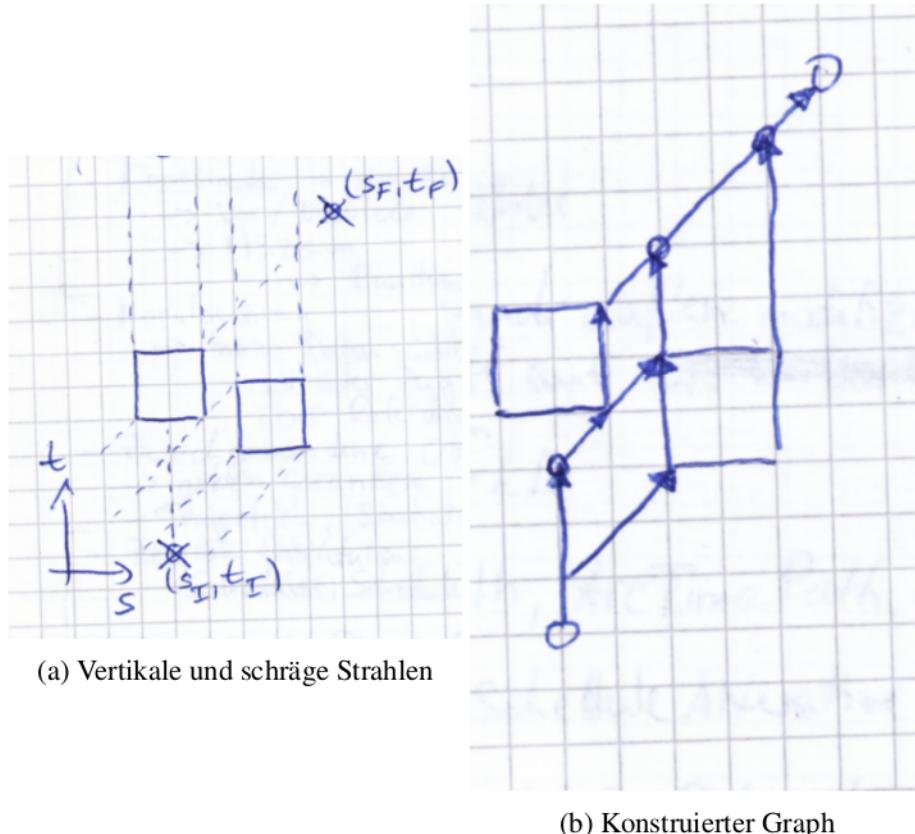


Abbildung 4.28: Graphkonstruktion

dem oben geschilderten Prinzip mit Wartezeiten besitzt. Um den letztgenannten Graphen zu erzeugen geht man wie folgt vor (siehe auch Abbildung 4.28):

Als Ausgangspunkt dient ein Graph der lediglich als Knoten Start- und Endpunkt sowie die Eckpunkte der verbotenen Regionen enthält.

1. Beginnend von jedem Knoten wird ein vertikaler Strahl nach oben in t -Richtung gezogen.
2. Beginnend von jedem Knoten wird ein schräger Strahl mit Anstieg v_L nach unten gegen t - und s -Richtung gezogen.
3. Die Schnittpunkte aller vertikalen und schrägen Strahlen bilden neue Knoten insofern keiner der jeweiligen Strahlen bereits mit einer Region kollidierte.
4. Es werden Kanten hinzugefügt, die die neuen Knoten mit den Ursprüngen der jeweiligen Strahlen verbindet.

Um zu verhindern, dass sehr kurze Wartezeiten eingeführt werden, kann ein Schwellwert d_{th} definiert werden. In diesem Fall werden nur jene Wartezeiten in den Graph aufgenommen, die den Schwellwert nicht unterschreiten. Allerdings kann es nun dazu kommen, dass ein Pfad aufgrund dieser Regel nicht gefunden werden kann. Durch das zusätzliche Hinzufügen von Kanten nach der alten Vorschrift aus Abschnitt 2.2 kann hier Abhilfe geschaffen werden. Als Kantengewicht empfiehlt sich die Fahrzeit. Dadurch werden Pfade mit geringer Fahrzeit bevorzugt.

Wenn das Ziel durch keine Zeitvorgabe spezifiziert wurde, sondern die frühestmögliche Ankunftszeit t_{min} gewünscht ist, so geht man hier in zwei Phasen vor. Zunächst wird t_{min} ermittelt, indem ein Pfad minimaler Dauer ohne Wartezeiten bestimmt wird wie bereits im Abschnitt 4.3.2 vorgeschlagen. $(s_F, t_F = t_{min})$ geben dann den festen Zielpunkt in der zweiten Phase vor, in der der Graph wie oben beschrieben konstruiert wird.

4.4 Knotenauslastung

Der Scheduler erlaubt die Auskunft über die Auslastung der Knoten. Diese Informationen sind beispielsweise für den Lastverteiler des Schwarmbetriebssystems nützlich, der nicht Teil des Schedulers ist. Dazu werden im Folgenden verschiedene Arten von Lasten definiert. Diese unterscheiden grundsätzlich zwei Arten: Jobausführung und Bewegung. Ein Roboter kann nach dieser Unterscheidung drei Zustände einnehmen. Entweder er führt einen Job aus, befindet sich in Bewegung oder in Stillstand (ohne Job).

Die erste Art gibt Auskunft über die Jobauslastung L_J eines Knotens innerhalb eines Zeitintervalls T . Resultat ist ein Wert zwischen 0 und 1. Führt ein Knoten während des gesamten Zeitraums T keinen Job aus, so liefert L_J den Wert 0. Ist er dagegen vollbeschäftigt, lautet der Wert 1. Die konkrete Berechnung ist gegeben durch:

$$L_J(T) := \frac{\text{dur}(J \cap T)}{\text{dur}(T)} \quad (4.20)$$

J ist eine Menge über alle Zeitpunkte, zu denen der Knoten durch ein Job beschäftigt ist:
 $J := \{ t \mid \text{Knoten bearbeitet einen Job zum Zeitpunkt } t \}$. Die Hilfsfunktion dur gibt die Dauer einer Menge aus Zeitpunkten zurück:

$$\text{dur}(T) := \int_T dt \quad (4.21)$$

Analog lässt sich die Bewegungsauslastung definieren. Die Menge M erstreckt sich hier über alle Zeitpunkte, zu der sich der Knoten in Bewegung befindet: $M := \{ t \mid v(t) > 0 \}$

$$L_M(T) := \frac{\text{dur}(M \cap T)}{\text{dur}(T)} \quad (4.22)$$

Auch die stationäre Auslastung L_S folgt diesem Schema. Bei S handelt es sich um die Zeit, in der sich der Roboter weder bewegt noch einen Job ausführt ($S \cap J = \emptyset$ und $S \cap M = \emptyset$). Daher gilt $S \cup J \cup M = T$.

$$L_S(T) := \frac{\text{dur}(S \cap T)}{\text{dur}(T)} = 1 - L_J(T) - L_M(T) \quad (4.23)$$

Aus der Auslastung L_J und L_M lässt sich die Gesamtauslastung definieren:

$$L(T) := \frac{\text{dur}((M \cup J) \cap T)}{\text{dur}(T)} = L_J(T) + L_M(T) \quad (4.24)$$

Da sich ein Roboter nie in Bewegung befindet, wenn er gerade ein Job ausführt, sind die Mengen J und M disjunkt, sodass $J \cap M = \emptyset$. Daher lässt sich $L(T)$ auch Summe von L_J und L_M notieren.

Die letzte Variante berechnet die Geschwindigkeitsauslastung L_v . Dazu wird die Geschwindigkeit des Roboters während des Zeitintervalls T der Höchstgeschwindigkeit gegenüber gestellt. Eine Auslastung von 1 wird hier nur dann erreicht, wenn der Roboter sich während T stets mit maximaler Geschwindigkeit fortbewegt. Nach einer anderen Interpretation wird hier der zurückgelegte Weg gegenüber dem maximal möglichen Weg verglichen.

$$L_v(T) := \frac{\int_T v(t) dt}{v_{max} \cdot \text{dur}(T)} \quad (4.25)$$

Kapitel 5

Implementierung

Die Umsetzung des Lösungskonzepts erfolgte vollständig in Java 8. Die Ausführung ist daher weitestgehend plattformunabhängig, da für die meisten Betriebsumgebungen eine JVM zur Verfügung steht. Als Build-Tool dient Maven, welches Abhängigkeiten zu externen Bibliotheken automatisch auflöst und das Kompilieren des Quellcodes zu einer JAR-Datei ermöglicht.

Der Scheduler wurde als Bibliothek konzipiert und ist daher für sich nicht als eigenständiges Programm ausführbar. Jedoch finden sich im Quellcode einige Beispielprogramme, die die Verwendung des Schedulers demonstrieren¹²

Die in Eigenentwicklung erstellten Quellen befinden sich unter der Packagestruktur `de.tu_berlin.mailbox.rjasper`. Das Subpackage `st_scheduler` enthält den Raum-Zeit-Scheduler. Die Unterteilung der beiden Kernaufgaben spiegelt sich in dessen Unterordnern `scheduler` (Jobplanung) und `world` (Trajektorienplanung) wider.

5.1 Konzepte

5.1.1 Kein null

Im Allgemeinen akzeptiert die Schnittstellen des Schedulers keine `null`-Argumente. Ebenso werden auch keine `null`-Objekte zurückgegeben. Stattdessen werden leere Objekte verwendet. Beispielsweise ein Pfad ohne Knoten oder eine leere Liste. Dies sorgt für weniger Fallunterscheidungen und eine leichter verständliche Logik.

¹Die Beispiele befinden sich im Package `de.tu_berlin.mailbox.rjasper.st_scheduler.example`
²**TODO:** linebreak

5.1.2 Unveränderliche Objekte

An vielen Stellen werden unveränderliche Objekte genutzt. Beispiele dafür sind `LocalDateTime`, `ImmutableList` oder `ImmutablePolygon`. Dies gewährleistet zum einen die Sicherheit, dass sich Objekte nicht unerwartet ändern und zudem ohne Gefahr direkt zurückgegeben werden können.

Dies ist von großem Vorteil, da sich an den meisten Stellen Objekte, einmal erstellt, nicht ändern. Das Klonen von Objekten beim Initialisieren oder setzen (Setter-Methoden) entfällt somit ebenso wie das Verschachteln in unmodifizierbaren Wrappern beim Zurückgeben (Getter-Methoden).

Viele Objekte verwenden „Immutable“-Varianten um ihre Attribute zu speichern. Beispielsweise liegt die Form eines `DynamicObstacles` als `ImmutablePolygon` vor.

5.1.3 Fail-Fast

Bei der Entwicklung des Schedulers wurde darauf Wert gelegt, Fehler an dessen inneren und äußeren Schnittstellen frühzeitig zu erkennen. Somit werden beispielsweise unzulässige Parameterwerte und `null`-Objekte zurückgewiesen. Die falsche Verwendung von Schnittstellen oder Ursachen von Bugs können so schneller identifiziert werden. Falls es zu einem Fehler kommt, werden in der Regel Ausnahmen wie `NullPointerException`, `IllegalArgumentException` oder `IllegalStateException` geworfen.

5.2 Abhängigkeiten

Die Implementierung ist in Java 8 geschrieben und von mehreren externen Java-Bibliotheken abhängig, die allesamt über Maven eingebunden werden. Mit Ausnahme der `straightedge`-Bibliothek, sind alle Abhängigkeiten öffentlich über das Maven Repository³ zugänglich.

5.2.1 Java 8

Java 8 ist die neueste Version der Programmiersprache und enthält gegenüber den Vorgängern umfangreiche Neuerungen. Im Mittelpunkt steht die Einführung von Lambda-

³<http://www.mvnrepository.com/>

Ausdrücken. Diese ermöglichen Funktionen kompakt zu definieren, die früher umständlich durch anonyme innere Klassen erzeugt wurden:

```
1 JButton button = new JButton("Button");
2 // before Java 8: anonymous inner class
3 button.addActionListener(new ActionListener() {
4     @Override
5     public void actionPerformed(ActionEvent e) {
6         System.out.println("button_clicked");
7     }
8 });
9 // Java 8: lambda expression
10 button.addActionListener(e ->
11     System.out.println("button_clicked"));
```

Lambda-Ausdrücke motivierten auch die neue Stream-Schnittstelle, die Operationen wie Map oder Reduce ermöglicht:

```
1 List<Integer> list = Arrays.asList(1, 2, 3);
2 list.stream().map(n -> 2*n);
```

5.2.2 Java Topology Suite

Die Trajektorienplanung ist stark auf die *Java Topology Suite* (JTS) angewiesen. Diese Bibliothek implementiert die *Simple Features Specification for SQL*, welche grundlegende Geometrien wie Punkte, Linien oder Polygone spezifiziert. Des Weiteren stellt JTS ein umfangreiches Angebot fundamentaler Funktionen für zweidimensionale Geometrien zur Verfügung. Somit lassen sich beispielsweise verbotene Regionen, wie in Abschnitt 4.3.3 beschrieben, transformieren und beschneiden.

5.2.3 JGraphT

Zur Berechnung der Geschwindigkeitsprofile der Trajektorienplanung werden die Graph-Algorithmen der *JGraphT*-Bibliothek verwendet. Auch die Auflösung des Abhängigkeitsgraphen aus Abschnitt 4.2.4 gelingt mit Hilfe dieser Bibliothek.

5.2.4 la4j

Die Lösung der Gleichung 4.16 erfolgt durch die *la4j*-Bibliothek. Sie stellt Typen, wie Matrizen und Vektoren, sowie Algorithmen der linearen Algebra Verfügung. Ihre Funktionen kommen auch bei der Berechnung der übrigen Fälle verbotener Regionen zur Anwendung.

5.2.5 straightedge

Die räumliche Wegfindung, welche lediglich stationäre Hindernisse umgeht und die Zeitdimension nicht berücksichtigt, wird von der *straightedge*-Bibliothek bewältigt. Hindernisse werden durch 2D-Polygone beschrieben. Die Suche erfolgt durch die Erstellung eines Navigationsgraphen und der Anwendung des A*-Algorithmus.

5.3 Typen

5.3.1 Gleitkommazahl

Raumordinaten werden stets als 64-bit Gleitkommazahlen (**double**) gespeichert. Auch Zeitangaben erfolgen in gewissen Teilen des Programmcodes ebenfalls als **double**, um Operationen wie Multiplikation mit anderen Größen zu ermöglichen. Generell werden alle physikalischen Werte wie Entfernung, Zeit oder Geschwindigkeit in SI-Einheiten ($m, s, \frac{m}{s}$) angegeben.

Beim Umgang mit Gleitkommazahlen gilt zu beachten, dass sich bei jeder Verdopplung des Wertes die Präzision halbiert. Möchte man beispielsweise für Zeitangaben eine Präzision im Nanosekundenbereich erhalten, so dürfen die Werte einen Absolutbetrag von etwa 97 Tagen nicht überschreiten.

5.3.2 Zeittypen

Im Allgemeinen werden Zeitpunkte als `LocalDateTime` und Zeitdauern als `Duration` dargestellt. Beide Klassen bieten grundlegende arithmetische Operationen, wie das Addieren oder Subtrahieren einer Dauer. Beide Klassen haben eine fixe Präzision von einer Nanosekunde.

An einigen Programmstellen des Schedulers müssen `LocalDateTime` oder `Duration` in **double**-Werte umgewandelt werden. Die Genauigkeit eines **doubles**

kann jedoch problematisch werden, wenn eine Zeitspanne von 97 Tagen überschritten wird. In solch einem Fall kann eine Umrechnung nicht exakt abgebildet werden. **double**-Werte können zudem den Wertebereich der anderen Klassen übersteigen. Der letzte Zeitpunkt einer `LocalDateTime` ist der `31.12.999999999` um `23:59:59.999999999` und die längste Duration beträgt etwa 292 Milliarden Jahre. Gegenüber steht die maximale Dauer eines **doubles** von etwa 5.7×10^{300} Jahren.

5.3.3 Pfade

Um die Bahnen der Hindernisse und Roboter in Raum und Zeit zu beschreiben, wird das Interface `Trajectory` verwendet. Objekte dieses Interfaces speichern eine Liste von dreidimensionalen Punkten (x, y, t), welche die geradlinig verbundenen Eckpunkte der Bahn angeben. `Trajectory` wird von zwei Klassen implementiert.

Die `SimpleTrajectory` speichert zwei gleich lange Listen aus zweidimensionalen Punkten der JTS-Bibliothek (`Point`) und Zeitpunkten (`LocalDateTime`), die gemeinsam die Ordinaten der Raumzeitpunkte bilden.

Die zweite Klasse ist die `DecomposedTrajectory`. Sie wird durch zwei zweidimensionale Pfade `SpatialPath` und `ArcTimePath` gebildet. Der `SpatialPath` entspricht dem räumlichen Pfad π und `ArcTimePath` dem temporalen Pfad σ . Bei beiden handelt es sich um Listen von JTS-Punkten.

Die vier Klassen `SimpleTrajectory`, `DecomposedTrajectory`, `SpatialPath` und `ArcTimePath` implementieren allesamt das Interface `Path` und bieten Funktionen wie Interpolation, Konkatenation oder die Bestimmung von Teilstücken an.

5.3.4 Hindernisse

Hindernisse werden durch die beiden Klassen `StaticObstacle` und `DynamicObstacle` dargestellt. Ein statisches Hindernis setzt sich lediglich aus ein zweidimensionales JTS-Polygon, welches dessen Form angibt, zusammen. Dynamische Hindernisse besitzen zudem noch eine Trajektorie, die in Abhängigkeit der Zeit die Verschiebung derer Formen anzeigt.

Die Klasse `World` setzt sich aus statischen und dynamischen Hindernissen zusammen, um somit die Beschaffenheit der Wirklichkeit abzubilden.

5.3.5 Jobs

Eingeplante Jobs werden durch die Klasse `Job` implementiert. Objekte dieser Klasse sind unveränderlich und geben Auskunft über Ort, Startzeit, Dauer und Knoten. Erzeugt wird ein `Job` durch den Scheduler, der festlegt, wann, wo und von welchen Knoten der Auftrag ausgeführt wird.

5.4 Komponenten

5.4.1 Scheduler

Die `Scheduler`-Klasse ist die zentrale Komponente des Raum-Zeit-Schedulers. Sie bietet die wichtigsten Schnittstellen an und enthält den gesamten Zustand. Dazu gehört die Abbildung der realen Welt (`World`) und Roboter sowie die Jobs. Über seine Schnittstellen kann man unter anderen Knoten (Roboter) hinzufügen oder entfernen, Jobs einplanen oder zurücknehmen, die Uhrzeit aktualisieren und Parameter festlegen.

Der Plan, welcher die Jobs aller Knoten enthält, wird durch die `Schedule`-Klasse implementiert. Der `Scheduler` besitzt genau einen `Schedule`, welcher auch die Knoten speichert. Neben dem aktuellen Plan speichert der `Schedule` auch beliebig viele Alternativen (`ScheduleAlternative`). Dabei handelt es sich um Änderungen des `Schedules`, die noch nicht durch ein `Commit` bestätigt oder ein `Abort` abgewiesen wurden. Um den Plan zu manipulieren, werden spezialisierte `Scheduler` eingesetzt, die der zentrale `Schedule` aufruft.

Für das Einplanen von Jobs wird die `schedule`-Methode verwendet, die in drei Ausführungen auftritt: Die Einplanung von

- einzelnen Jobs,
- periodischen Jobs und
- abhängigen Jobs.

Die drei spezialisierten `Scheduler` `SingleJobScheduler`, `PeriodicJobScheduler` und `DependentJobScheduler` suchen nach einer gültigen Belegung der Variablen des Ortes, Zeitpunktes und Knotens der Jobs. Die anderen beiden `Scheduler` greifen dabei auf den `SingleJobScheduler` zurück. Das Aufnehmen oder Löschen eines Jobs in bzw. aus dem `Schedule` erfordert eine Trajektorienplanung, die durch den `JobPlanner` oder `JobRemovalPlanner` eingeleitet wird.

5.4.2 Knoten

Roboter und andere Arbeitseinheiten werden vom Scheduler unter dem Begriff Knoten vereinigt. Objekte der Klasse `Node` speichern sowohl die physikalischen Eigenschaften als auch die Joblisten und Trajektorien der Knoten.

Zu den physikalischen Eigenschaften zählen die Form, die wie auch bei den Hindernissen durch ein JTS-Polygon dargestellt wird, die Höchstgeschwindigkeit und der initiale Raum-Zeit-Punkt, der Ort und Zeit angibt ab dem der Knoten für den Scheduler beginnt zu existieren.

Die Jobliste und Trajektorie eines Knotens bilden den Schedule des Knotens ab. Die Jobs liegen sortiert nach Startzeit in einem Suchbaum vor. Die Trajektorie setzt sich aus mehreren Abschnitten zusammen, die sich ebenfalls in einem Suchbaum innerhalb eines speziellen Objekts (`TrajectoryContainer`) befinden.

`Node`-Objekte bieten Methoden zur Manipulation des Schedules an und werden daher vom Scheduler nicht öffentlich zugänglich gemacht. Um dem Scheduler neue Knoten hinzuzufügen, muss zunächst eine `NodeSpecification` erzeugt werden, die die physikalischen Eigenschaften des Knotens festlegt. Auf bestimmte Informationen des Knotens kann über eine `NodeReference` zugegriffen werden, die keine Manipulation des Knotens zulässt.

5.4.3 Pathfinder

Da der Trajektorienplaner nur mit punktgroßen Robotern umgehen kann, müssen zuvor alle Hindernisse mit dem Radius des Roboters gepuffert werden. Die JTS-Bibliothek stellt dazu bereits eine Funktion bereit, die geometrische Formen entsprechend transformieren kann.

Die Planung von Trajektorien erfolgt stets in zwei Phasen. Zunächst wird der räumliche Pfad π mithilfe des `StraightEdgePathfinders` bestimmt, welcher auf die `straightedge`-Bibliothek zurückgreift. In der zweiten Phase wird das Geschwindigkeitsprofil bestimmt. Dazu existieren die beiden Implementierungen `LazyFixTimePathfinder` und `LazyMinimumTimePathfinder`. Der erste dient zur Berechnung des Profils dessen Start- und Zielzeitpunkte vorgegeben sind. Beim zweiten ist lediglich die Startzeit festgelegt. Die Zielzeit soll dagegen minimiert werden.

Die Berechnung des Geschwindigkeitsprofils erfolgt in drei Schritten. Ziel ist die Vermeidung jeglicher dynamischer Hindernisse. Als erstes werden die verbotenen Regionen im $s \times t$ -Raum vom `ForbiddenRegionBuilder` berechnet. Jedes Hindernis

muss dabei einer Transformation unterzogen werden. Viele Hindernisse kommen jedoch womöglich gar nicht in die Nähe des Pfades π . Diese können daher wie im Pseudocode A.7 in Zeile 5, 9 und 14 vorzeitig aussortiert werden.

Anschließend erstellt ein sogenannter „Mesher“ den Navigationsgraphen. Dafür existieren zwei Varianten; der `LazyFixTimeMesher` und der `LazyMinimumTimeMesher`, die jeweils beim entsprechenden Pathfinder Anwendung finden. Diese greifen wiederum auf Konnektoren zurück, die anhand der verbotenen Regionen Knoten und Kanten des Graphen erstellen oder löschen. Der `SimpleVertexConnector` verbindet beispielsweise alle Eckpunkte der verbotenen Regionen miteinander insofern sie zueinander sichtbar sind und die Kriterien aus Abschnitt 2.2 erfüllen. Der `MinimumTimeVertexConnector` erstellt dagegen mögliche Zielknoten um die Suche nach dem schnellsten Pfad zu ermöglichen. Als Kantengewicht wird die Fahrtzeit verwendet. Stationäre Pfadsegmente verursachen keine Kosten.

Nach der Erstellung des Graphen erfolgt die Bestimmung des kürzesten Weges⁴ durch Dijkstras Algorithmus der *JGraphT*-Bibliothek.

5.5 Schnittstellen

5.5.1 Initialisierung des Schedulers

Die Initialisierung des Schedulers erfordert die Übergabe eines unveränderlichen `World`-Objekts, welches die statischen und dynamischen Hindernisse enthält. Die Welt des Schedulers kann nachträglich nicht mehr geändert werden.

```

1 // Scheduler constructor
2 public Scheduler(World world)
3 // World constructor
4 public World(
5     ImmutableCollection<StaticObstacle> staticObstacles,
6     ImmutableCollection<DynamicObstacle> dynamicObstacles)
```

5.5.2 Knoten hinzufügen und entfernen

Um einen Knoten beim Scheduler zu registrieren, muss man zunächst eine `NodeSpecification` erstellen, die sich aus dessen ID und physikalischen Eigen-

⁴Der kürzeste Weg bezeichnet hier den Pfad mit den geringsten Kosten.

schaften zusammensetzt. Mithilfe der ID lässt sich der Knoten wieder auffinden. Es gilt dabei zu beachten, dass eine ID nur einem Knoten zugeordnet sein kann.

```

1 // NodeSpecification constructor
2 public NodeSpecification(
3     String nodeId,
4     ImmutablePolygon shape,
5     double maxSpeed,
6     ImmutablePoint initialLocation,
7     LocalDateTime initialTime)
8 // Scheduler methods

```

Das Hinzufügen des Knotens kann eine Ausnahme auslösen, falls der angegebene Ort und Zeitpunkt zu einer Kollision mit einem Hindernis oder anderen Knoten führen würde. Zum Löschen eines Knotens ist dessen ID notwendig. Zudem dürfen ihm keinerlei unerledigten Aufgaben zugeordnet sein.

```

1 public NodeReference addNode(NodeSpecification spec)
2     throws CollisionException
3 public void removeNode(String nodeId)

```

Die Schnittstelle erlaubt es nicht, direkt auf den erstellten Knoten des Schedulers zugreifen. Um dennoch bestimmte Funktionen aufrufen zu können, ermöglicht die Methode `getNodeReference` die Rückgabe einer Referenz. Eine `NodeReference` bietet verschiedene delegierende Methoden an, um Informationen (z.B. Auslastung) zur Verfügung zu stellen.

```

1 public NodeReference getNodeReference(String nodeId)

```

5.5.3 Planung eines Jobs

Zur Einplanung eines einzelnen Jobs wird eine Spezifikation benötigt.

```

1 public ScheduleResult schedule(JobSpecification spec)

```

Diese grenzt den konkreten Ort und Startzeitpunkt durch ein Areal und Zeitintervall ein. Die Dauer ist dagegen fest vorgegeben. Die Spezifikation enthält auch eine ID um den Job später adressieren zu können. Bereits verwendete IDs sind jedoch unzulässig

```

1 public static <G extends Geometry & ImmutableGeometry>
2     JobSpecification createSF(
3         UUID jobId,

```

```

4     G locationSpace,
5     LocalDateTime earliestStartTime,
6     LocalDateTime latestFinishTime,
7     Duration duration)

```

Zur Erzeugung einer `JobSpecification` existieren drei statische Funktionen `createSF`, `createSS` und `createFF`. Sie unterscheiden sich in der Semantik des Zeitintervalls. Das Suffix „SF“ gibt an, dass sich das Intervall vom frühstmöglichen Startzeitpunkt bis zum spätestmöglichen Endzeitpunkt erstreckt. Damit enthält das Intervall auch die Dauer des Jobs. Die anderen beiden Methoden beziehen sich dagegen entweder auf das Intervall des Start- („SS“) oder Endzeitpunktes („FF“).

Die Rückgabe der `schedule`-Methode ist ein `ScheduleResult`. Dieser Typ dient dazu, zu überprüfen, ob die Einplanung erfolgreich war und welche Änderungen am Schedule vorgenommen wurden. Im Falle eines Erfolgs muss die Planung jedoch noch bestätigt werden. Das `ScheduleResult` stellt dazu eine Transaktions-ID zur Verfügung, mit der die Änderungen übernommen oder verworfen werden können. Der Scheduler bietet hierfür die folgenden Methoden an:

```

1 // total commit
2 public void commit(UUID transactionId)
3 // partial commit
4 public void commit(UUID transactionId, String nodeId)
5 // total abort
6 public void abort(UUID transactionId)
7 // partial abort
8 public void abort(UUID transactionId, String nodeId)

```

Die Bestätigung oder Abweisung kann auch partiell für einzelne Knoten erfolgen. Dies ermöglicht die Freigabe von knotenspezifischen Sperren (engl. Locks), welche durch die Transaktion entstanden sind.

5.5.4 Planung periodischer Jobs

Auch periodische Jobs werden mithilfe einer Spezifikation eingeplant. Allerdings handelt es sich hier um eine `PeriodicJobSpecification`:

```
1 public ScheduleResult schedule(PeriodicJobSpecification spec)
```

Ihre Parameter sind zum Teil mit der `JobSpecification` aus dem vorherigen Abschnitt [5.5.3](#) zu vergleichen.

```

1 // PeriodicJobSpecification constructor
2 public <G extends Geometry & ImmutableGeometry>
3   PeriodicJobSpecification(
4     ImmutableList<UUID> jobIds,
5     G locationSpace,
6     boolean sameLocation,
7     Duration duration,
8     LocalDateTime startTime,
9     Duration period)

```

Bei der periodischen Planung werden beliebig viele Einzeljobs nacheinander in den Schedule aufgenommen. Für jeden dieser Jobs muss eine ID angegeben werden. Die Größe der `jobIds`-Liste gibt die Jobanzahl vor. Der erste Job erhält die erste ID, der zweite die zweite und so weiter. Der `locationSpace` gibt auch hier das Areal vor, wo die Aufgabe ausgeführt werden kann. Neu ist der boolesche Parameter `sameLocation`. Wenn dieser wahr ist, so werden alle Jobs am selben Ort durchgeführt. Andernfalls können die Jobs an verschiedenen Orten , jedoch im selben Areal, zugewiesen werden. Die Periodendauer wird durch `period` definiert. Die erste Periode beginnt zur `startTime` und jede weitere setzt an ihrem Vorgänger an. Innerhalb jeder Periode muss ein Job eingeplant werden.

Falls die Einplanung aller Jobs nicht möglich ist, so wird keiner in den Schedule aufgenommen.

5.5.5 Planung abhängiger Jobs

Bei der Planung abhängiger Jobs können mehrere Jobs durch einen Scheduler-Aufruf auf einmal eingeplant werden. Jeder Job wird durch eine `JobSpecification` spezifiziert. Der `dependencies`-Parameter ist ein einfacher Graph, der die Abhängigkeiten der einzuplanenden Jobs definiert. Als Knoten dienen die Job-IDs. Die Kanten des Graphen zeigen auf die Jobs, von der ein Job selbst abhängt. Alle Jobs der `specs` müssen auch im Graphen vorliegen und umgekehrt.

```

1 public ScheduleResult schedule(
2   Collection<JobSpecification> specs,
3   SimpleDirectedGraph<UUID, DefaultEdge> dependencies)

```

Durch die Angabe des Graphen wird gewährleistet, dass die Abhängigkeiten eines Jobs stets abgeschlossen sind bevor er selbst startet. Im Abschnitt 4.2.4 wurde ein weiterer Parameter d_m eingeführt, durch den man einen zeitlichen Sicherheitsabstand festlegen

kann. Der Job wird dann erst frühestens d_m nach Abschluss der letzten Abhängigkeit ausgeführt. In der Implementierung handelt es sich bei d_m um einen globalen Parameter des Schedulers, der dort als `interDependencyMargin` bezeichnet wird:

```
1 public void setInterDependencyMargin(
2     Duration interDependencyMargin)
```

Wie schon bei der [Planung periodischer Jobs](#) werden entweder alle oder keine Jobs eingeplant.

5.5.6 Umplanen eines Jobs

Die Umplanung eines Jobs erfolgt ähnlich wie die Einplanung mit dem Unterschied, dass bereits ein Job mit der ID existieren muss, die in der Spezifikation angegeben wird. Auch das Umplanen erfordert eine Bestätigung oder Abweisung.

```
1 public ScheduleResult reschedule(JobSpecification spec)
```

5.5.7 Entfernen eines Jobs

Ein Job kann auf zwei Weisen entfernt werden. Entweder man verwendet `unschedule`, welches auch eine Neuplanung der Trajektorien durchführt. Oder ohne Anpassung der Trajektorie durch `removeJob`. Zum Entfernen sollte man zunächst die `unschedule`-Variante probieren, da so die Fahrtwege optimiert werden können. Da die Trajektorienplanung allerdings fehlschlagen kann und weil die Anpassung eventuell nicht durch den Roboter bestätigt wird, existiert auch die zweite Variante. Führt die erste nicht zum gewünschten Ergebnis, so kann somit auf `removeJob` zurückgegriffen werden.

```
1 public ScheduleResult unschedule(UUID jobId)
2 public void removeJob(UUID jobId)
```

5.5.8 Gegenwart setzen

Der Großteil der genannten Methoden des Schedulers sind zeitabhängig. Das bedeutet, das Verhalten kann sich unterscheiden, wenn die Zeit der Gegenwart ändert. Beispielsweise kann eine Aufgabe nicht eingeplant werden, falls deren Startzeitpunkt bereits in der Vergangenheit liegt.

Die Gegenwart wird als Variable des Schedulers (`presentTime`) gespeichert. Diese muss allerdings explizit durch Aufruf gesetzt werden:

```
1 public void setPresentTime(LocalDateTime presentTime)
```

Dabei ist es unzulässig, die bereits gesetzte Zeit zu verringern. Damit Methoden wie `schedule` oder `unschedule` ordnungsgemäß funktionieren, sollte diese Zeit vor Aufruf aktualisiert werden. Auf ein implizites Aktualisieren auf die Systemzeit wurde verzichtet, um Tests und Simulationen zu ermöglichen. Das Verhalten des Schedulers wird somit allein durch die Aufrufe der Schnittstellen bestimmt.

Eine weitere wichtige Größe in Zusammenhang der Gegenwart t_{pr} ist der eingefrorene Horizont t_{FH} . Er gibt an bis zu welcher Zeit keine Änderungen am Schedule mehr möglich sind. Die Zeit des eingefrorenen Horizonts wird implizit gesetzt, wenn die Gegenwart aktualisiert wird. Ein entscheidender Wert dazu ist die Dauer $d_{FH} = t_{FH} - t_{pr}$. Diese Dauer wird durch die folgende Methode festgelegt:

```
1 public void setFrozenHorizonDuration(  
2   Duration frozenHorizonDuration)
```

Im Normalfall handelt es sich bei der Dauer um einen Wert, der nur einmal festgelegt wird und sich anschließend nicht verändert. Mit

```
1 public LocalDateTime getFrozenHorizonTime()
```

lässt sich der Zeitpunkt t_{FH} abfragen. Es wird gewährleistet, dass sich dieser Wert niemals reduziert, selbst wenn sich der Wert d_{FH} durch `setFrozenHorizonDuration` verringert.

5.5.9 Knotenauslastung

Die folgenden Schnittstellen werden von der Klasse `NodeReference` zu Verfügung gestellt um Auskunft über die Knotenauslastung zu geben. Mithilfe der Knoten-ID gibt der Scheduler durch Aufruf der `getNodeReference`-Methode die entsprechende Referenz zurück.

```
1 public Duration calcJobDuration(  
2   LocalDateTime from, LocalDateTime to)  
3 public Duration calcMotionDuration( ... )  
4 public double calcJobLoad( ... )  
5 public double calcMotionLoad( ... )  
6 public double calcLoad( ... )  
7 public double calcStationaryIdleLoad( ... )  
8 public double calcVelocityLoad( ... )
```

Alle oben genannten Methoden betrachten ein gegebenes Zeitintervall. `calcJobLoad` berechnet den relativen Zeitanteil innerhalb des Gesamtintervalls, in dem ein Job ausgeführt wird. `calcMotionLoad` bezieht sich dagegen auf den Bewegungsanteil. `calcLoad` stellt die Summe der beiden Werte dar. `calcStationaryIdleLoad` gibt dagegen den Anteil des Stillstands außerhalb einer Jobausführung an. `calcVelocityLoad` berechnet die Geschwindigkeitsauslastung hinsichtlich der Höchstgeschwindigkeit.

Bei den Rückgabewerten all dieser Methoden muss beachtet werden, dass sich die Berechnungen lediglich auf den aktuellen Schedule stützen. Dessen Alternativen, durch Transaktionen hervorgerufen, können nicht berücksichtigt werden.

Kapitel 6

Evaluation

6.1 Komplexität

Die Komplexitätsanalyse (Zeitkomplexität) erfolgt im Gegensatz zum Lösungs- und Implementierungsteil in umgekehrter Richtung (Bottom-Up), sodass aufbauend auf Teilergebnissen die Komplexitätsbestimmung übergeordneter Komponenten gelingt. Als Grundlage dient der Pseudocode aus Anhang A

6.1.1 Trajektorienplanung

Die Trajektorienplanung eines Jobs wird durch die Prozedur `plan_job` aus Algorithmus A.4 vorgenommen. Die Trajektorien werden durch die Funktionen `calc_ft_trajectory` und `calc_mt_trajectory` bestimmt. Diese berechnen dazu zunächst die verbotene Region des räumlichen Pfades mithilfe von `calc_forbidden_region`. Anschließend wird der Navigationsgraph von `connect_directly`, `connect_lazy` und `connect_finish_candidates` konstruiert. Des Weiteren werden grundlegenden Funktionen betrachtet, die ebenfalls zur Komplexität beitragen.

Geometrische Mengenoperationen

Zwei grundlegende Funktionen sind die Schnittmengen- und Differenzmengenoperation (\cap , \setminus) für geometrische Objekte. Realisiert werden diese durch die JTS-Bibliothek. Da derartige Funktionen nicht trivial und äußerst komplex sind, kann an dieser Stelle über die eigentliche Komplexität nur gemutmaßt werden.

Ein zentraler Schritt bei derartigen Mengenoperationen ist die Bestimmung der Schnittpunkte der linearen Liniensegmente des Randes zweier Geometrien. Anschlie-

ßend wird eine neue Geometrie durch das geeignete Zusammenfügen beider Ränder die resultierende Geometrie bestimmt.

Die Schnittpunkte können durch paarweise Betrachtung der Randsegmente berechnet werden. Ich gehe daher von einer Komplexität von $\mathcal{O}(m \cdot n)$ ¹, wobei m und n die Zahl der Eckpunkte beider Geometrien bezeichnet. Bei der Zusammenfügung des Randes nehme ich stattdessen nur $\mathcal{O}(m + n)$ an. Bei übrigen Operationen wie Fallunterscheidungen gehe ich nicht davon aus, dass solche die Komplexität von $\mathcal{O}(m \cdot n)$ verschlechtern. Kombiniert verbleibt somit eine Komplexität von

$$\mathcal{O}(m \cdot n). \quad (6.1)$$

CALC_FORBIDDEN_REGION

Von zentraler Bedeutung für die Komplexität sind die drei ineinander verschachtelten **for**-Schleifen (siehe Algorithmus A.7). Die oberste Schleife iteriert über die individuellen Hindernisse, die zweite über deren Trajektoriensegmente und die innerste über die Pfadsegmente des Roboters. Die Hauptberechnung stellt die Transformation der Hindernisse in den $s \times t$ -Raum dar.

Bei der Transformation werden drei Fälle unterschieden (regulär, parallel und stationär). Interessant sind die Zeilen 19, 23 und 33, da diese von der Anzahl der Eckpunkte von $O_k^m(t)$ abhängen.

Einfachheitshalber wird diese Anzahl im Rahmen der Komplexitätsanalyse mit $\underline{O}_k^m(t)$ bezeichnet. Dies wird mit weiteren Parametern in analoger Weise fortgeführt, sodass $\underline{\pi}$ und $\underline{\tau}$ die Segmentanzahl des Pfades π bzw. der Trajektorie τ angeben.

Die innerste Schleife wird $\underline{\pi}$ -mal ausgeführt und hat daher eine $\mathcal{O}(\underline{\pi} \cdot \underline{O}_k^m(t))$ -Komplexität. $O_k^m(t)$ stellt ein Teilstück des $x \times y \times t$ -Raumes dar, welches von der Punktmenge O^m eingenommen wird. Die beiden ersten Schleifen in Zeile 3 und 8 iterieren über diese Teilstücke, sodass im gesamten die verbotene Region von O_k^m berechnet wird. Daher lautet die Gesamtkomplexität aller Schleifen $\mathcal{O}(\underline{\pi} \cdot \underline{O}^m)$, wobei \underline{O}^m die Anzahl der dreidimensionalen Eckpunkte von O^m angibt.

Bisher wurde ignoriert, dass jeder Schleifendurchlauf vorzeitig in den Zeilen 6, 10 und 15 übersprungen werden kann. Oftmals kann anhand eines Tests auf Hüllenüberschneidung bereits geschlossen werden, dass ein Hindernis für eine Kachel des $s \times t$ -Raumes keine verbotene Region verursacht. Ob diese Abkürzung allerdings auch einen Einfluss auf die Komplexität hat, ist schwer zu bestimmen. Nicht in jedem Fall wird für ein Hindernis auch eine Transformation durchgeführt. Es steht die Frage offen, ob die

¹ \mathcal{O} wird innerhalb der Komplexitätsanalyse stellvertretend für Θ verwendet.

Zunahme an Hindernissen, Trajektorien- oder Pfadsegmenten auch zu einer Steigerung des Berechnungsaufwands in den Zeilen 19, 23 und 33 führt. Entscheidend ist hier die genaue Lage der Hindernisse, Trajektorien und Pfade. Ohne empirische Daten von konkreten Anwendungsfällen ist die Entwicklung eines geeigneten Modells jedoch kaum möglich. Daher verzichte ich auf die Berücksichtigung der Zeilen 6, 10 und 15.

Die Gesamtkomplexität beträgt daher

$$\mathcal{O}(\underline{\pi} \cdot \underline{O^m}). \quad (6.2)$$

CONNECT_DIRECTLY, CONNECT_LAZY und CONNECT_FINISH_CANDIDATES

In Zeile 2 der CONNECT_DIRECTLY-Prozedur aus Algorithmus A.8 werden die Eckpunkte der verbotenen Region als Graphknoten aufgefasst, deren Anzahl $\underline{O^f}$ benannt. Über die Knoten wird paarweise Iteriert und verursacht dadurch $\mathcal{O}(\underline{O^f}^2)$. In der innersten Schleife ist lediglich der VISIBLE-Aufruf von Bedeutung, da dieser $\mathcal{O}(\underline{O^f})$ bewirkt. Insgesamt lautet die Komplexität daher:

$$\mathcal{O}(\underline{O^f}^3) \quad (6.3)$$

Die Prozedur CONNECT_LAZY berechnet in Zeile 13 und 14 Strahlen (CAST_MOTION_RAYS, CAST_STATIONARY_RAYS), die zur Erzeugung zusätzlicher Knoten und Kanten verwendet werden. Für jeden Eckpunkt der verbotenen Region wird dazu ein Strahl berechnet, der sich bis zum ersten Kollision mit einer Hinderniskante fortsetzt. Die Kollisionsbestimmung hat dabei einen Aufwand von $\mathcal{O}(\underline{O^f})$. CAST_MOTION_RAYS und CAST_STATIONARY_RAYS verursachen deshalb jeweils eine Komplexität von $\mathcal{O}(\underline{O^f}^2)$.

Die Schnittpunktberechnung der Strahlen ab Zeile 15 beruht auf zwei verschachtelten Schleifen, die alle Strahlenpaare durchlaufen. Der Rumpf der innersten Schleife hat einen konstanten Aufwand. Deshalb wird auch hier $\mathcal{O}(\underline{O^f}^2)$ verursacht und der Gesamtaufwand lautet:

$$\mathcal{O}(\underline{O^f}^2) \quad (6.4)$$

Die letzte der drei Prozeduren durchläuft alle Knoten in Zeile 27 einmal. Die Sichtbarkeitsprüfung im Rumpf in Zeile 29 sorgt auch hier für einen Aufwand von insgesamt

$$\mathcal{O}(\underline{O^f}^2). \quad (6.5)$$

CALC_FT_TRAJECTORY und CALC_MT_TRAJECTORY

Die Funktion CALC_FT_TRAJECTORY führt einige der bisher betrachteten Prozeduren und Funktionen nacheinander aus, sodass deren Komplexitäten lediglich zusammengeführt werden müssen. Der Aufwand zur Berechnung der verbotenen Region in Zeile 1 ist durch $\mathcal{O}(\underline{O^m} \cdot \underline{\pi})$ aus [6.2] gegeben. CONNECT_DIRECTLY in Zeile 4 übertrifft CONNECT_LAZY und verursacht $\mathcal{O}(\underline{O^f}^3)$ [6.3]. Dijkstras Algorithmus hat allgemein einen Aufwand von $\mathcal{O}(V \log V + E)$, wobei V die Knotenzahl und E die Kantenzahl angeben. An dieser Stelle führe ich unter der Annahme, dass die Knoten eng vermascht sind, die Vereinfachung durch, dass $E = V^2 = \underline{O^f}^2$ ist. Somit führt auch Zeile 8 zu einem quadratischen Aufwand. Die letzte Zeile verursacht $\mathcal{O}(\underline{O^f} + \underline{\pi})$.

Insgesamt führt das zum Gesamtaufwand von

$$\mathcal{O}(\underline{O^m} \cdot \underline{\pi} + \underline{O^f}^3). \quad (6.6)$$

CALC_MT_TRAJECTORY (Algorithmus A.6) stellt eine Alternative Version von CALC_FT_TRAJECTORY und unterscheidet sich lediglich in den Zeilen 6 und 8. CONNECT_FINISH_CANDIDATES führt den Aufwand $\mathcal{O}(\underline{O^f}^2)$ ein. DETERMINE_FINISH prüft welcher Zielknoten erreichbar und wählt den mit der geringsten Zeit aus. Da es sich dabei um eine Traversierung handelt, kann der Aufwand den des Dijkstra-Algorithmus nicht übersteigen. Die Gesamtkomplexität ist daher identisch zu (6.6).

PLAN_JOB

Die Trajektorienplanung eines neuen Jobs umfasst die Planung von zwei Raumpfaden und zwei Trajektorien. Algorithmus A.4 berechnet die Pfade π_1 und π_2 in den Zeilen 2-3. Implementiert wird die Funktion CALL_SPATIAL_NODE durch die *straightedge*-Bibliothek. Die generelle Umsetzung dürfte wie schon bei der Trajektorienberechnung die Konstruktion eines Navigationsgraphen und eines Graphenalgorithmus wie Dijkstra beinhalten.

Der Navigationsgraph für die räumliche Wegfindung muss nur einmalig durchgeführt werden, und wird hier daher nicht betrachtet. Damit reduziert sich die Berechnung auf den Graphenalgorithmus. Unterstellt man auch hier Dijkstra, so kann man wie schon im vorigen Abschnitt quadratischen Aufwand unterstellen. Allerdings kommt es auch hier auf die Beschaffenheit der statischen Hindernisse an. Handelt es sich beispielsweise um identische abgeschlossene Räume, so würde die Kantenzahl linear mit den Räumen und somit den Knoten zunehmen. Sind es dagegen viele kleine Hindernisse, die die Sicht kaum beeinträchtigen, so ist auch hier eine quadratische Knoten-zu-Kanten-Relation zu

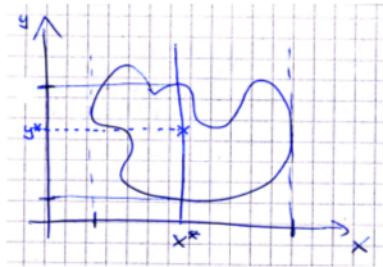


Abbildung 6.1: Extraktion eines Punktes aus einem Areal

vermuten. Da es an einem Modell mangelt, gehe ich von einer quadratischen Relation aus, sodass es zum Aufwand $\mathcal{O}(\underline{O}^{s^2})$ kommt. Zusammen mit der Trajektorienberechnung sieht die Komplexität daher wie folgt aus:

$$\mathcal{O}(\underline{O}^{s^2} + \underline{O}^m \cdot \pi + \underline{O}^{f^3}) \quad (6.7)$$

6.1.2 Jobplanung

Zur Einplanung von Jobs existieren die drei Methoden SCHEDULE_SINGLE, SCHEDULE_PERIODIC und SCHEDULE_DEPENDENT. Die beiden letztgenannten greifen dabei auf SCHEDULE_SINGLE zurück, welche sich wiederum der PLAN_JOB-Prozedur der Trajektorienplanung bedient.

SCHEDULE_SINGLE

SCHEDULE_SINGLE aus Algorithmus A.1 berechnet in Zeile 2 die Differenzmenge des spezifizierten Aufgabenareals mit statischen Hindernissen O^s . Diese Operation nimmt $\mathcal{O}(L \cdot \underline{O}^s)$ in Anspruch, wobei L der Eckpunktzahl des Aufgabenareals entspricht.

In der nächsten Zeile werden l Ortspunkte dem Differenzgebiet extrahiert. Die Funktion SAMPLE_LOCATIONS ist in Zeile 15 definiert. Dort wird eine **while**-Schleife bis zu l mal durchlaufen. Im Rumpf benötigt Zeile 21 mit SAMPLE_LOCATION eine Zeit von $\mathcal{O}(L)$. Diese Funktion extrahiert einen Punkt aus einer Menge von Polygonen. Dazu wird zunächst ermittelt über welchen Intervallen einer Dimension sich das Polygon erstreckt (siehe Abbildung 6.1). Wurden die Intervalle für die x -Richtung bestimmt, so wählt man daraus nun ein x^* -Wert aus. Durch diesen Wert lässt sich nun eine Linie $\{(x^*, y) \mid y \in \mathbb{R}\}$ definieren, die mit dem Polygon geschnitten wird. Aus der resultierenden Schnittmenge muss nun lediglich ein Punkt ausgewählt werden. Sowohl zum Bestimmen der Intervalle als auch der Schnittmenge müssen alle Punkte des Polygons einmal betrachtet werden. Daher ist mit einem Zeitaufwand von $\mathcal{O}(L)$ zu rechnen. Zei-

le 23 ruft DEVIDE_AREA(a) auf. Diese teilt die aktuelle *area* in bis zu vier *subareas* auf. Die Aufteilung gelingt in $\mathcal{O}(L)$. Damit ergibt sich für SAMPLE_LOCATIONS ein Aufwand von $\mathcal{O}(L \cdot l)$.

In Zeile 4 wird die äußere zweier verschachtelter Schleifen l mal ausgeführt. Die innere Schleife iteriert über S Knotenpausen, die von FIND_SLOTS ermittelt wurden. Diese Funktion iteriert über alle Knoten und deren Pausen. Dadurch entsteht ein Zeitaufwand von $\mathcal{O}(n \cdot s)$. n bezeichnet die Anzahl der Knoten und s die durchschnittliche Pausenzahl.

Innerhalb der inneren Schleife aus der Prozedur SCHEDULE_SINGLE wird in Zeile 7 nun der Job eingeplant. PLAN_JOB benötigt bekanntermaßen nach (6.7) $\mathcal{O}(\underline{Q}^{s^2} + \underline{Q}^m \cdot \underline{\pi} + \underline{Q}^{f^3})$. Falls die Planung erfolgreich war, wird der Prozeduraufruf in Zeile 9 beendet. Ist eine Einplanung für alle Ortspunkte und Pausen jedoch unmöglich, so wird PLAN_JOB $l \cdot S$ mal ausgeführt. Daher gehe ich davon aus, dass es im Mittel zu $l \cdot S$ Aufrufen von PLAN_JOB kommt.

Durch Zusammenfügung der Teilergebnisse gelangt man zu folgenden Aufwand der SCHEDULE_SINGLE-Prozedur:

$$\mathcal{O}\left(L \cdot \underline{Q}^s + L \cdot l + l \cdot n \cdot s + l \cdot S \cdot \left(\underline{Q}^{s^2} + \underline{Q}^m \cdot \underline{\pi} + \underline{Q}^{f^3}\right)\right) \quad (6.8)$$

SCHEDULE_PERIODIC

Die Einplanung mehrerer periodischer Jobs kommt in zwei Varianten daher. Die erste plant die Jobs alle am selben Ort ein und ist im Algorithmus A.2 unter der Bezeichnung SCHEDULE_PERIODIC_SAME_LOCATION zu finden. Wie auch schon bei der Prozedur SCHEDULE_SINGLE wird hier in den Zeilen 2 und 3 das spezifizierte Aufgabenareal beschnitten ($\mathcal{O}(L \cdot \underline{Q}^s)$) und Ortspunkte extrahiert ($\mathcal{O}(L \cdot l)$). In l Durchläufen wird versucht j Jobs einzuplanen. Der Aufwand des SCHEDULE_SINGLE-Aufrufs in Zeile 10 ist gegenüber (6.8) reduziert auf $\mathcal{O}(\underline{Q}^s + n \cdot s + S \cdot (\underline{Q}^{s^2} + \underline{Q}^m \cdot \underline{\pi} + \underline{Q}^{f^3}))$, da die Spezifizierung *spec* nur einen Ortspunkt als Aufgabenareal enthält.²

Für die Prozedur SCHEDULE_PERIODIC_SAME_LOCATION ergibt sich somit

$$\mathcal{O}\left(L \cdot \underline{Q}^s + L \cdot l + l \cdot j \cdot \left(\underline{Q}^s + n \cdot s + S \cdot \left(\underline{Q}^{s^2} + \underline{Q}^m \cdot \underline{\pi} + \underline{Q}^{f^3}\right)\right)\right). \quad (6.9)$$

Die zweite Variante SCHEDULE_PERIODIC_INDEPENDENT_LOCATION plant die Jobs an unabhängigen Orten ein. Gegenüber der ersten Variante reduziert sich die Proze-

²Der Aufwand ließe sich hier noch weiter reduzieren, da SCHEDULE_SINGLE in den Zeilen 2 und 3 mehrfach dieselben Rechnungen wiederholt.

dur auf die j -malige Ausführung von SCHEDULE_SINGLE³. Der benötigte Zeit beträgt daher

$$\mathcal{O} \left(j \cdot \left(L \cdot \underline{O^s} + L \cdot l + l \cdot n \cdot s + l \cdot S \cdot \left(\underline{O^{s2}} + \underline{O^m} \cdot \pi + \underline{O^{f3}} \right) \right) \right). \quad (6.10)$$

SCHEDULE_DEPENDENT

SCHEDULE_DEPENDENT ist die letzte der drei Varianten, mit der sich Jobs in den Schedule einplanen lassen. Diese fügt mehrere Jobs unter Berücksichtigung ihrer Abhängigkeiten untereinander ein.

Der erste Schritt dieser Planung ist Zeile 2 des Algorithmus A.3 zu entnehmen. Dort werden die übergebenen Spezifikationen zunächst normalisiert. Die Normalisierung erfolgt in der Prozedur NORMALIZE_SPECS in Zeile 15. In Zeile 16 werden dazu zuerst die Abhängigkeiten topologisch in $\mathcal{O}(D + j)$ sortiert. D bezeichnet die Anzahl der Abhängigkeiten (bzw. Kanten) und j die Jobanzahl (Knoten). Zeile 17 kopiert die Spezifikationen in $\mathcal{O}(j)$. Die beiden Schleifen aus Zeile 18 und 23 überschreiben das Startzeitintervall der Spezifikationen und sind ähnlich aufgebaut. Dabei Abhängigkeiten aller Jobs besucht. Somit verursachen beide Schleifen jeweils einen Aufwand von $\mathcal{O}(D + j)$. Zum Schluss werden die normalisierten Spezifikationen nochmals nach ihrer Deadline sortiert, wodurch $\mathcal{O}(j \log j)$ eingeführt wird. Die NORMALIZE_SPECS-Prozedur besitzt damit die Komplexität $\mathcal{O}(D + j \log j)$.

In Zeile 3 beginnt die Schleife, der SCHEDULE_DEPENDENT-Prozedur, welche in j Durchgängen über alle Aufgabenspezifikationen iteriert. Der SCHEDULE_DEPENDENT-Aufruf betrachtet alle Abhängigkeiten des aktuellen Jobs wodurch insgesamt alle Abhängigkeiten betrachtet werden ($\mathcal{O}(D)$). Zeile 8 enthält die SCHEDULE_SINGLE-Anweisung, die den Aufwand in (6.8) verursacht. Falls ein Job nicht eingeplant werden kann, so wird die Schleife abgebrochen. Andernfalls wird sie voll durchlaufen.

Die Gesamtkomplexität beläuft sich in der Summe auf

$$\mathcal{O} \left(D + j \log j + j \cdot \left(L \cdot \underline{O^s} + L \cdot l + l \cdot n \cdot s + l \cdot S \cdot \left(\underline{O^{s2}} + \underline{O^m} \cdot \pi + \underline{O^{f3}} \right) \right) \right). \quad (6.11)$$

6.1.3 Bewertung

Betrachtet man die Ergebnisse der drei SCHEDULE-Prozeduren, so zeigt sich als ausschlaggebender Term des Berechnungsaufwandes $\underline{O^{s2}} + \underline{O^m} \cdot \pi + \underline{O^{f3}}$, welcher sei-

³Auch hier ist eine Optimierung möglich, sodass die Rechnungen aus den Zeilen 2 und 3 in SCHEDULE_SINGLE nur einmalig erfolgen.

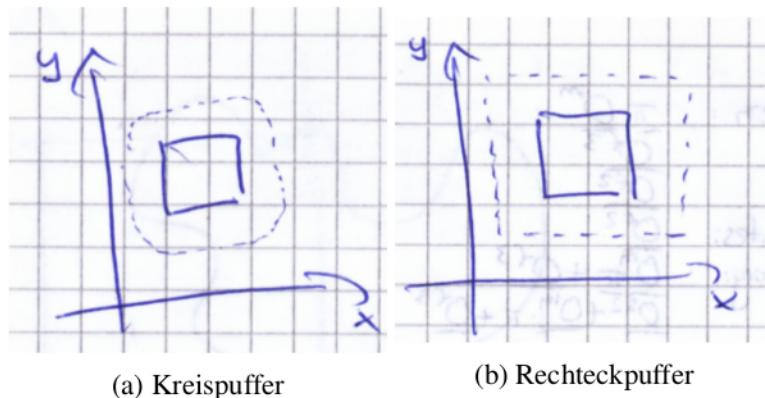


Abbildung 6.2: Pufferung von Hindernissen durch Minkowski-Summe

nen Ursprung in der PLAN_JOB-Prozedur hat. Die beiden Faktoren l und S bestimmen wie häufig diese Prozedur aufgerufen wird. Die Anzahl der Ortskandidaten l ist in der konkreten Implementierung nach oben durch 5 beschränkt. Das heißt aus einem Areal werden von SAMPLE_LOCATIONS nur bis zu fünf Ortspunkte extrahiert. S ist dagegen nur durch die Gesamtzahl aller Knotenpausen ($n \cdot s$) beschränkt. Die genaue Zahl hängt von der Situation ab. Beeinflusst wird sie durch die Aufgabendichte der Roboter, ihrer Entfernung zum Aufgabenareal und das gegebene Zeitintervall der Spezifikation. Je größer das Zeitintervall, je mehr Roboter, je geringer ihre Entfernung und je höher ihre Aufgabendichte, desto höher fällt S aus.

Die einförmige Komplexität $\mathcal{O}(\underline{O}^{s^2})$ ist situationsabhängig möglicherweise überschätzt. Die größte Last $\mathcal{O}(\underline{O}^{f^3})$ wird potentiell von den verbotenen Regionen verursacht. Es ist daher besonders wichtig dafür zu sorgen, dass diese in ihrer Beschaffenheit möglichst einfach ausfallen. Wie groß \underline{O}^f auftritt, hängt von mehreren Faktoren ab. Generell gilt, je mehr Hindernisse den Roboterpfad kreuzen, desto mehr Regionen treten hervor. Eine große Dichte an Robotern, ein langer Pfad und ein großes Zeitintervall begünstigen solch eine Situation. Jedoch trägt auch der Detailgrad der Form der dynamischen Hindernisse entscheidend zu \underline{O}^f bei. Roboter sollten daher lediglich als Rechteck dargestellt werden. Eine weitere Maßnahme liegt in der Implementierung der Pufferung von Hindernissen. Sowohl statische als auch dynamische Hindernisse werden gepuffert, um zur Berechnung der Trajektorie von einem punktgroßen Roboter ausgehen zu können. Zurzeit wird die Pufferung von der JTS-Bibliothek übernommen, welche die Minkowski-Summe des Hindernisses mit einem gefüllten Kreis bildet (Abbildung 6.2a). Ein Kreis ist zwar genauer, was die Pufferung anbelangt, steigert aber auch den Detail und führt daher zu Mehraufwand. Ein Rechteck würde dagegen ebenfalls genügen und den Detailgrad nur gering anheben (Abbildung 6.2b).

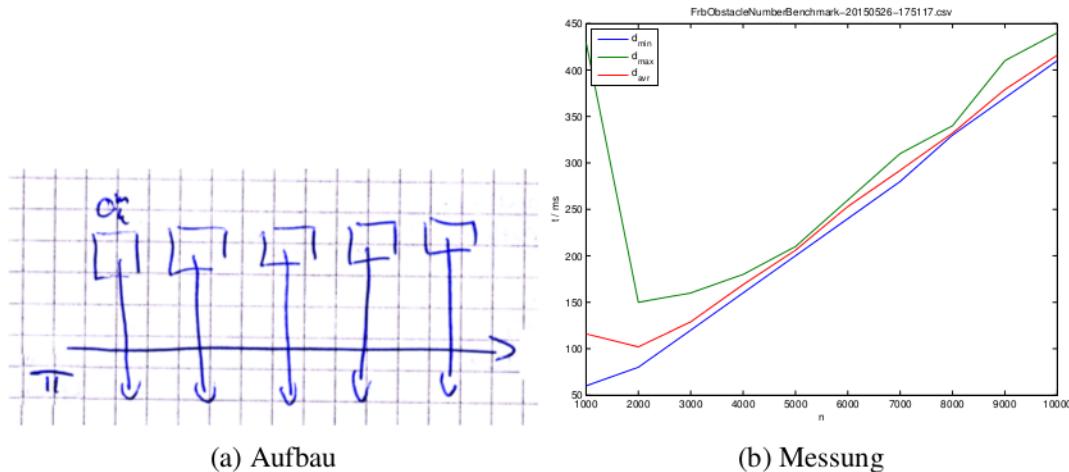


Abbildung 6.3: Testfall: Anzahl bewegter Hindernisse

Eine weitere Optimierung ist bei SCHEDULE_PERIODIC möglich. Dessen Varianten rufen jeweils SCHEDULE_SINGLE auf, welche mehrfach dieselbe Berechnung aus Zeile 2 und 3 des Algorithmus A.1 wiederholen.

6.2 Benchmark

Neben einer Komplexitätsanalyse wurden die Komponenten des Schedulers auch einem Benchmark-Test unterzogen. Als Testsystem diente ein PC mit einem Intel Core i5-3470 (3.2 GHz) unter Windows 7 (64-Bit) und Java in der Version 8 Update 25.

In 16 Testfällen wurden die Berechnungszeiten des Schedulers und seiner Komponenten ermittelt. Jeder Einzeltest erlaubt das Festlegen einer Problemgröße, beispielsweise die Zahl der Roboter. Die Messung der Berechnungsdauer für verschiedene Problemgrößen desselben Testfalls ermöglicht somit die Prüfung der Ergebnisse der Komplexitätsanalyse. Pro Testfall und Problemgröße wurde die Messung 20 mal wiederholt und der geringste Messwert gewählt um äußere Einflüsse zu minimieren.

6.2.1 Trajektorienplanung

CALC_FORBIDDEN_REGION

Das Zeitverhalten der CALC_FORBIDDEN_REGION-Funktion wurde in mehreren Testfällen für vier Problemgrößen untersucht.

Bei der ersten Größe handelt es sich um die Zahl bewegter Hindernisse, welche den Roboterpfad kreuzen (siehe Abbildung 6.3a). In Abbildung 6.3b zeigen die Messzei-

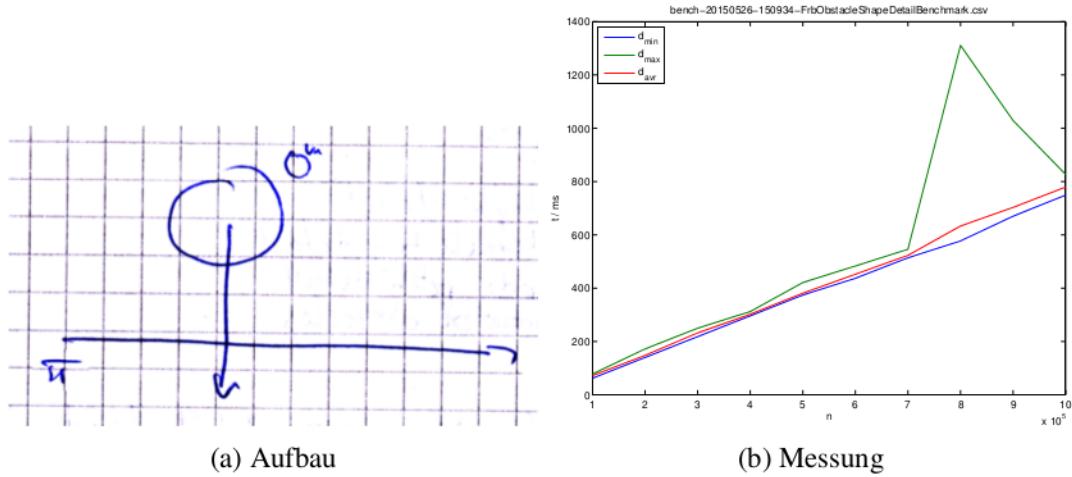


Abbildung 6.4: Testfall: Hindernisdetailgrad

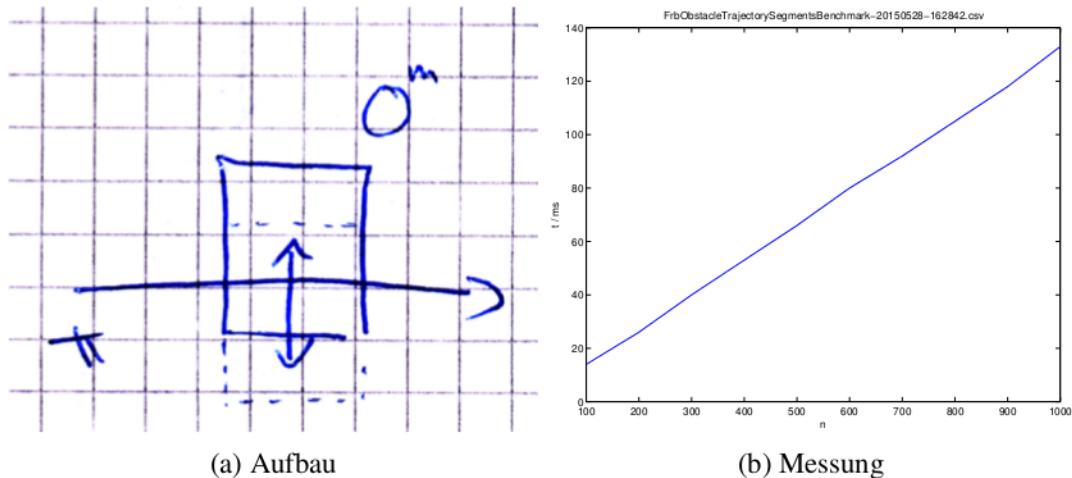


Abbildung 6.5: Testfall: Segmentanzahl der Hindernistrajektorie

ten einen linearen Zusammenhang mit der Zahl der Hindernisse auf. Dies befindet sich im Einklang der Komplexitätsanalyse, da O^m linear mit den Hindernissen steigt. Der Anstieg der Hinderniszahl bewirkt eine höhere Anzahl an Durchläufen der äußersten Schleife in Zeile 3 des Algorithmus [A.7]

Im nächsten Test wird der Detailgrad der Hindernisform erhöht. Dabei handelt es sich um ein Polygon, welches einen Kreis annähert (siehe Abbildung 6.4a). Ausschlaggebend ist jedoch die Anzahl der Eckpunkte des Polygons. Wie auch im vorherigen Testfall, ist hier ein linearer Zusammenhang festzustellen. Ein höherer Detailgrad erfordert einen größeren Aufwand bei der Transformation des Hindernisses in den Zeilen 17 bis 35. Die Komplexität wird auch hier bestätigt, da O^m proportional zunimmt.

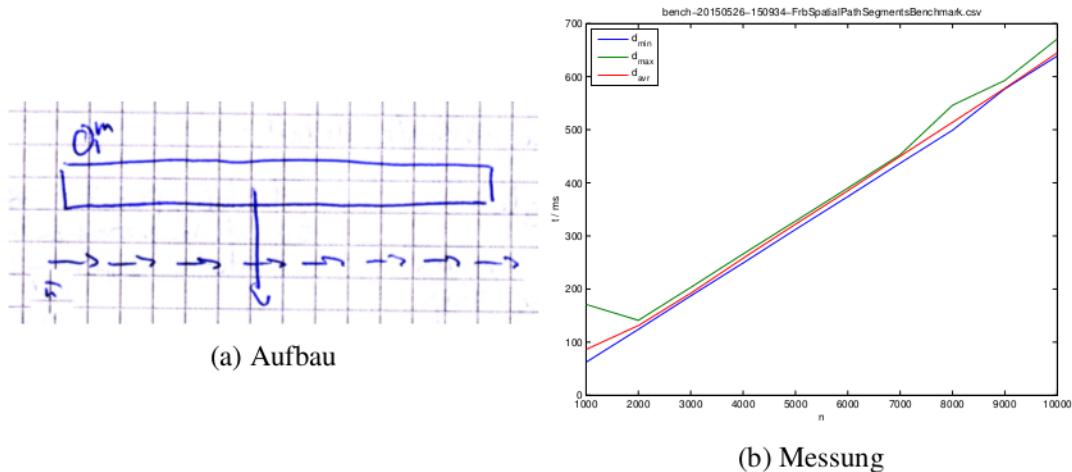


Abbildung 6.6: Testfall: Segmentanzahl des Roboterpfades

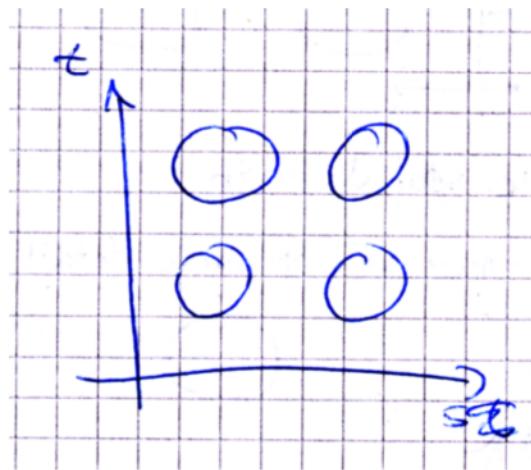
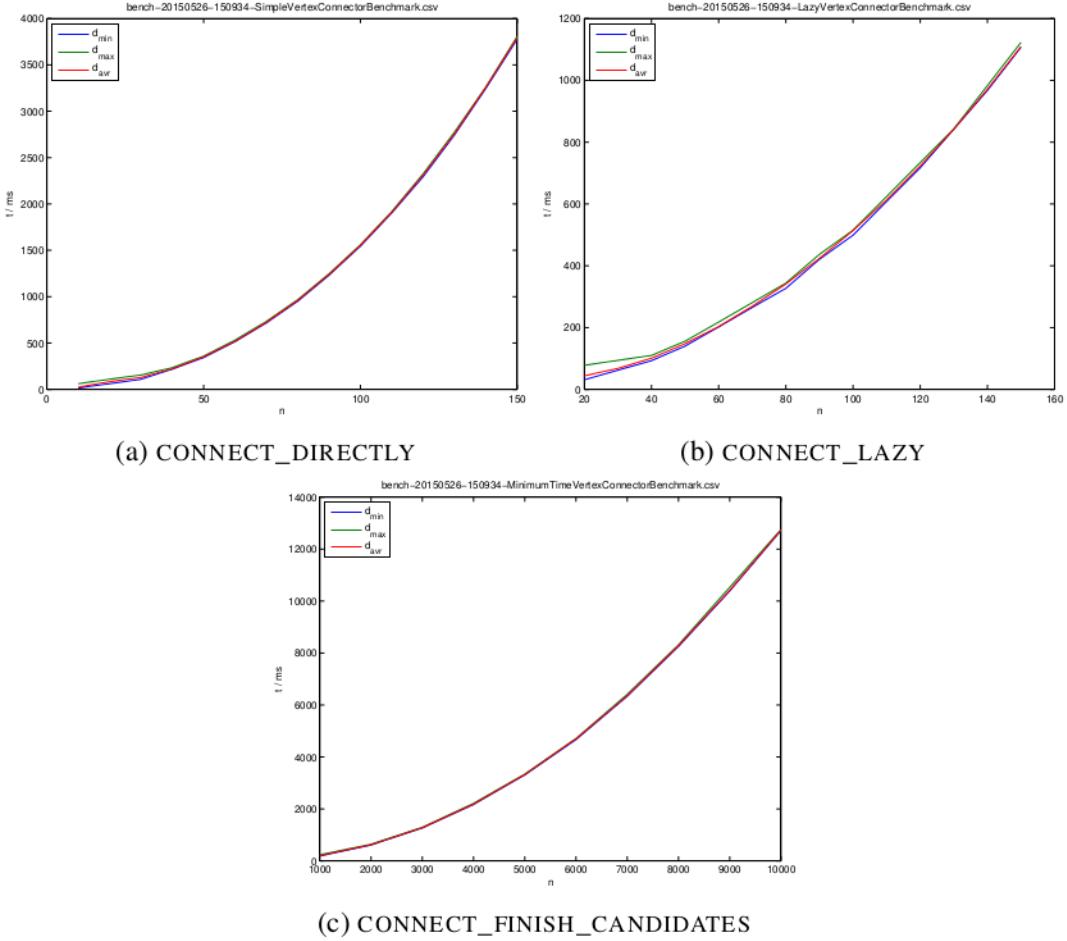


Abbildung 6.7: Detailgrad verbotener Regionen

Die Segmentanzahl der Hindernistrajektorie stellt die dritte Möglichkeit dar O^m zu steigern. In Abbildung 6.5a wandert ein Hindernis wiederholt über den Roboterpfad. Auch hier wächst die Ausführungszeit proportional zur Problemgröße (siehe Abbildung 6.5b), da nun die innere Schleife aus Zeile 8 mehrfach iteriert wird.

Die letzte untersuchte Problemgröße ist die Anzahl der Pfadsegmente des Roboters. Im Testfall überquert ein Hindernis alle Segmente des Pfades (siehe Abbildung 6.6a) und führt auch hier zu einer linearen Zunahme der Berechnungszeit (siehe 6.6b) aufgrund der innersten Schleife in Zeile 13.



CONNECT_DIRECTLY, CONNECT_LAZY und CONNECT_FINISH_CANDIDATES

Für die drei CONNECT-Funktionen um einen Navigationsgraphen aus verbotenen Regionen zu berechnen, wird jeweils der gleiche Testfall verwendet. Dieser prüft den Rechenaufwand in Abhängigkeit des Detailgrades der Regionen. Der Testfall, dargestellt in Abbildung 6.7, enthält vier kreisförmige Polygone, deren Detail mit der Problemgröße zunimmt.

Alle Funktionen weisen ein überproportionales Verhalten auf. Anhand der Komplexitätsanalyse liegt die Annahme nahe, dass es sich um quadratischen bzw. kubischen Aufwand handelt. Visuell lässt sich dies jedoch schwer einschätzen. Möglicherweise sind die untersuchten Problemgrößen im Falle von CONNECT_DIRECTLY noch zu gering um die kubische Komponente offensichtlich zu machen.

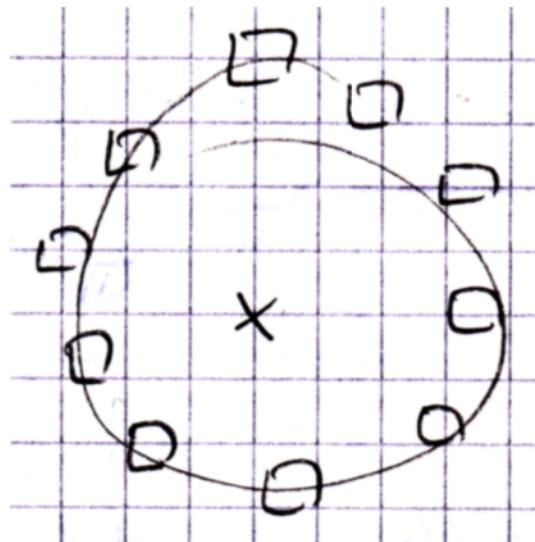
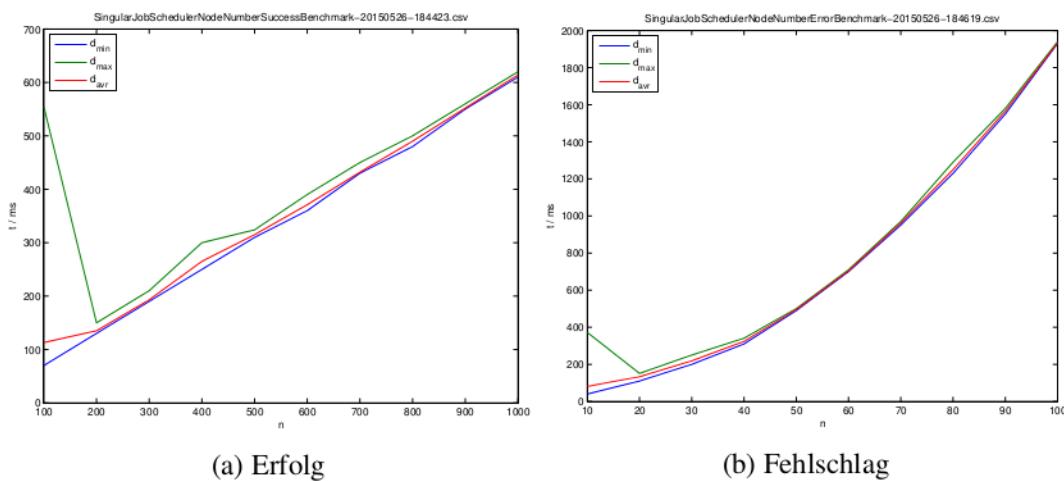


Abbildung 6.9: Knotenanzahl (Aufstellung)



(a) Erfolg

(b) Fehlschlag

Abbildung 6.10: Knotenanzahl (Messung)

6.2.2 Jobplanung

SCHEDULE_SINGLE

Die Testfälle für die SCHEDULE_SINGLE-Funktion untersuchen die Knoten- und Pausenanzahl als Problemgrößen. Dabei wird unterschieden, ob bereits der erste Planungsversuch erfolgreich ist, oder ob alle Variablenkombinationen erschöpft werden müssen.

Beim ersten Fall werden n Knoten in einem Kreis aufgestellt (siehe Abbildung 6.9). Im Zentrum soll eine Aufgabe eingeplant werden, die jeder Roboter in der Lage ist, zu übernehmen. Daher kann bereits dem ersten Roboter die Aufgabe zugeteilt werden. Es ergibt sich eine lineare Beziehung zwischen n und der Ausführungszeit (siehe Abbil-

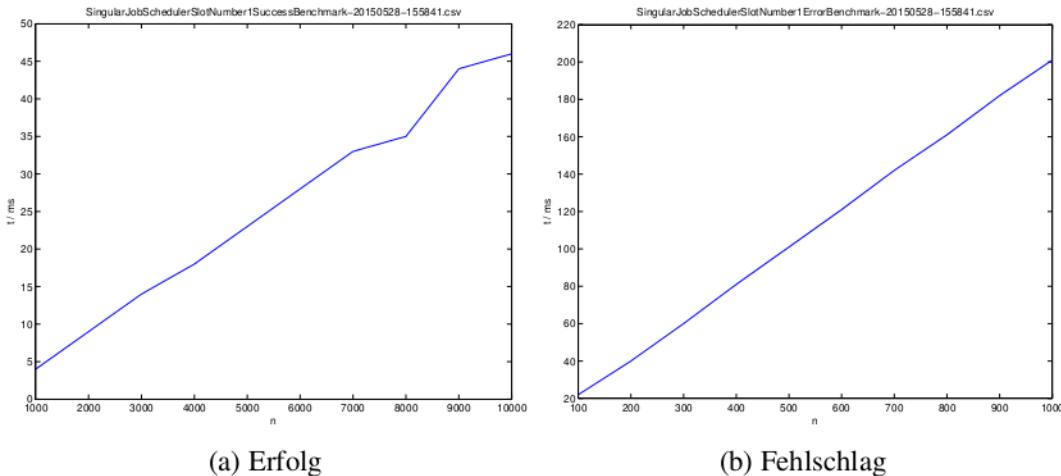


Abbildung 6.11: Pausenanzahl – Einzelter Knoten (Messung)

dung 6.10a) Dies ist primär dadurch zu begründen, dass jeder Knoten auch als dynamisches Hindernis auftritt und somit O^m steigt.

In der zweiten Variante dieses Falls kann die Aufgabe nicht eingeplant werden, da der Ort durch ein fremdes dynamisches Hindernis blockiert wird. Durch die Wiederholung der Planung für jeden Fall steigt der Aufwand nun quadratisch an (siehe Abbildung 6.10b).

Im nächsten Test wird die Anzahl der Pausen eines einzelnen Knoten verändert. Im Erfolgsfall hat die Problemgröße nur einen geringen linearen Einfluss, der selbst bei hohen Werten kaum Auswirkungen zeigt, da die Trajektorienplanung nur einmalig durchgeführt werden muss (siehe Abbildung 6.11a).

Die zweite Variante verhindert auch hier die Einplanung mithilfe einer Blockade eines fremden Hindernisses. Aufgrund dessen kommt es zu mehreren Versuchen eine Trajektorie zum Aufgabenort zu planen. Es zeigt sich daher deutlich ein linearer Aufwand (siehe Abbildung 6.11b).

Der letzte Testfall umfasst zwei Knoten. Nur ein Knoten ist in der Lage den Aufgabenort zu erreichen. Der andere Knoten tritt jedoch als dynamisches Hindernis auf. Modifiziert werden hier nur die Pausen des zweiten Knotens, wodurch seine Trajektorie in mehrere Abschnitte aufgeteilt wird. Deshalb steigt hier die Ausführungszeit linear an, da O^m durch zusätzlichen Trajektorien erhöht wird.

Im Falle der Blockade des Aufgabenortes sieht die Situation ähnlich aus (siehe Abbildung 6.12b). Auch hier trägt die Trajektorie des zweiten Knotens maßgeblich zur Linearität bei. Da nur ein Planungsversuch unternommen wird, verläuft die Kurve ähnlich zum obigen Fall.

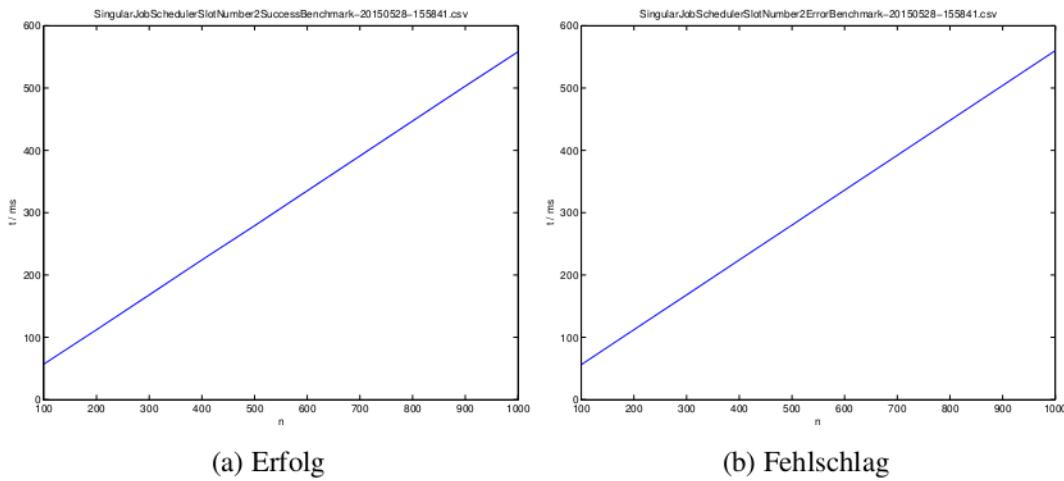


Abbildung 6.12: Pausenanzahl – Zwei Knoten (Messung)

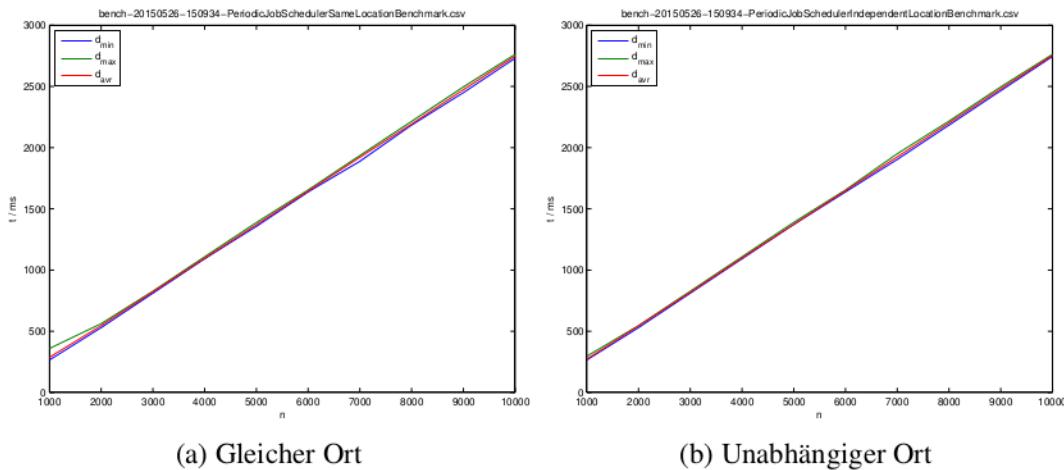


Abbildung 6.13: Periodische Einplanung

SCHEDULE_PERIODIC

Die SCHEDULE_PERIODIC-Prozedur wird hinsichtlich der Anzahl einzuplazender Jobs untersucht. Ein Roboter soll an einem nahe gelegenen Punkt periodische Jobs ausführen. Nach Abbildung 6.13a und 6.13b ergibt sich für die Varianten SCHEDULE_PERIODIC_SAME_LOCATION und SCHEDULE_PERIODIC_INDEPENDENT_LOCATION ein ähnlicher linearer Verlauf.

SCHEDULE_DEPENDENT

Wie auch schon bei der periodischen Einplanung wird die Berechnungsdauer der SCHE-DULE_DEPENDENT-Prozedur in Abhangigkeit der Jobanzahl ermittelt. Als Abhangigkeitsgraph wird einfacheitshalber eine kettenformige Struktur gewählt, sodass alle Jobs

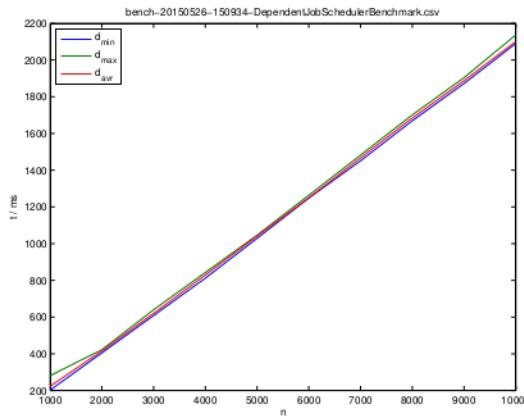


Abbildung 6.14: Einplanung abhängiger Aufgaben

der Reihe nach ausgeführt werden müssen. Ebenso steht hier nur ein Roboter zur Entgegennahme der Aufgaben zur Verfügung. Erwartungsgemäß der Komplexitätsanalyse ergibt sich auch hier ein linearer Aufwand (siehe Abbildung 6.14).

6.2.3 Bewertung

Nach Betrachtung der Ergebnisse des Benchmarks ist festzustellen, dass diese mit den Resultaten der Komplexitätsanalyse konform gehen. Allerdings gilt an dieser Stelle zu bemerken, dass die Testfälle die Laufzeit stets in nur einer Dimension untersuchen und daher nicht alle Aspekte der Komplexität geprüft werden konnten.

Vergleicht man die Messwerte der SCHEDULE-Prozeduren zwischen Erfolg und Misserfolg, so wird deutlich, dass das Zeitverhalten stark davon abhängt, wie früh der Scheduler einen Job einplanen kann. Die Kombinationsvielfalt der Variablenbelegung (Ort, Zeit, Knoten), kann hier schnell zum Verhängnis werden. Es wäre an dieser Stelle besser, nur einen Teil der Kombinationsmöglichkeiten zu prüfen.

Besondere Aufmerksamkeit muss der Erstellung des Navigationsgraphen geschenkt werden. Die Prozeduren CONNECT_DIRECTLY und CONNECT_LAZY weisen hier den höchsten Aufwand auf. Daher ist es umso dringlicher, den Detailgrad der verbotenen Regionen so niedrig wie möglich zu halten. Es bleibt zu untersuchen, wie hoch das Auftreten der Regionen in der Praxis ausfällt.

Abschließend ist festzustellen, dass alle untersuchten Problemgrößen dem Scheduler Grenzen vorgeben. Selbst in den besten Fällen ist ein linearer Zusammenhang zum Aufwand zu beobachten. Um beispielsweise besonders hohe Knotenzahlen zu ermöglichen, sind tiefer greifende Ansätze (z.B. hierarchische Verwaltung) von Nöten.

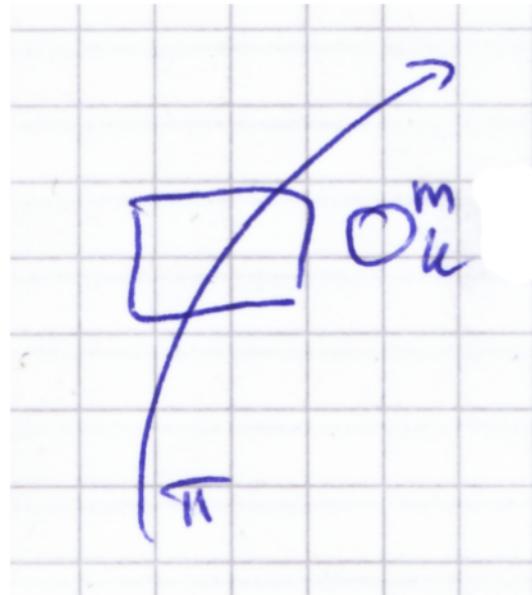


Abbildung 6.15: Pseudostationäres Hindernis blockiert Pfad

6.3 Problemfälle

6.3.1 Allwissenheit

Das Modell der Allwissenheit des Schedulers kann in der Praxis schnell an seine Grenzen stoßen. Der Umgang mit unbekannten Hindernissen oder Ausfällen von Robotern ist nicht möglich. Auch das Hinzufügen und Entfernen von Robotern kann nicht zufriedenstellend abgebildet werden. Die Praxistauglichkeit in einer realen alltäglichen Umgebung ist damit stark beeinträchtigt und reduziert sich damit auf Laborversuche.

6.3.2 Pseudostationäre Hindernisse

Eine Kernannahme der Path-Velocity-Decomposition ist, dass dynamische Hindernisse sich in Bewegung befinden und einen bestimmten Raum nur kurz in Anspruch nehmen. Bei den Robotern ist dies jedoch nicht immer der Fall. Bearbeiten sie gerade eine Aufgabe oder haben noch ausreichend Zeit um zur nächsten Aufgabe zu fahren, dann bewegen sie sich unter Umständen für einige Zeit nicht. Da die Trajektorienplanung jedoch nicht in der Lage ist dynamischen Hindernissen räumlich auszuweichen, blockieren jene Roboter für andere möglicherweise den Weg, ohne dass diese ausweichen können. In Abbildung 6.15 muss ein Roboter innerhalb des Zeitintervalls T den Pfad π bewältigen. Jedoch blockiert das Hindernis O_k^m einen Abschnitt des Pfades über das gesamte Intervall T .

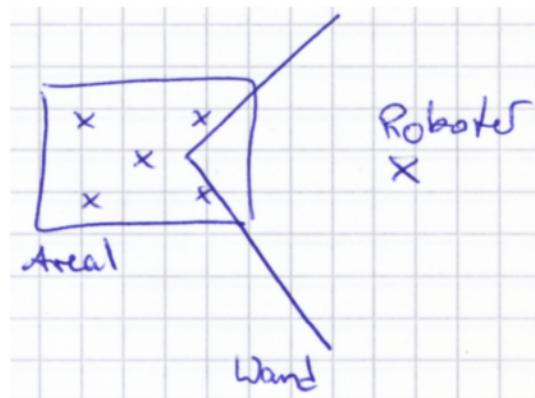


Abbildung 6.16: Ungünstiges Orts-Sampling

6.3.3 Orts-Sampling

Aufgabenspezifikationen ermöglichen es, ein Areal festzulegen, in dem ein Job ausgeführt werden soll. Allerdings werden nur wenige Punkte dieses Areals betrachtet. Im ungünstigsten Fall werden nur solche Punkte betrachtet, für die eine Einplanung unmöglich ist, obwohl das Areal als Ganzes dies erlauben würde. Ein Beispiel dazu ist in Abbildung 6.16 dargestellt, in dem eine Mauer ein Areal so schneidet, dass alle Sampling-Punkte sich auf der linken Seite befinden. Der Roboter rechts hat jedoch nicht ausreichend Zeit um die Mauer zu umfahren. Der linke Teil des Areals ist theoretisch erreichbar, wird jedoch aufgrund des Samplings nicht betrachtet.

6.3.4 Zeitintervall und Pfadlänge

Die Dauer des Zeitintervalls, in der eine Aufgabe eingeplant werden soll, kann die Leistung des Schedulers beeinträchtigen. Zum einen führt eine längere Dauer in der Regel zu mehr zu betrachtenden Knotenpausen. Zum anderen fällt die Berechnung des Geschwindigkeitsprofils umfangreicher aus. Lange Pfade erhöhen ebenfalls die Wahrscheinlichkeit dynamische Hindernisse zu kreuzen und führen daher auch zu mehr Aufwand.

6.3.5 Trajektoriendauer

Die Dauer einer Trajektorie kann problematisch werden, wenn sie 97 Tage überschreitet. Ursache ist eine Umrechnung zu Gleitkommazahlen und wurde bereits im Abschnitt 5.3.1 behandelt. Durch die Umwandlung können Rundungsfehler auftreten, sodass die Dauer einer Trajektorie nicht korrekt berechnet wird. In solch einem Fall scheitert der Scheduler in einem Programmfehler.

Kapitel 7

Fazit und Ausblick

Der Raum-Zeit-Scheduler ermöglicht die Einplanung von Aufgaben durch eine Constraint-basierte abstrakte Spezifikation. Dies verschafft einem Schwarmbetriebssystem den Vorteil, Benutzerprogrammen eine Schnittstelle zu bieten, bei der sich der Programmierer nur noch wenige Gedanken darüber machen zu müssen, wie ein Roboter zu einer bestimmten Zeit an den Ort einer Aufgabe zu gelangen. Die Aufgabenzuweisung und Pfadplanung wird System übernommen und ist gegenüber des Benutzerprogramms transparent. Dazu werden die Constraints der Spezifikation vom Scheduler ausgewertet und eine Kombination Ort, Zeit und Roboter gesucht, welche diese erfüllen. Anschließend wird eine Trajektorienplanung von Start zu Ziel vorgenommen, welche Hindernisse umgeht.

Die zu Beginn gesteckten Ziele werden somit vom Scheduler erfüllt. Seine Schwächen zeigt er jedoch in seiner Skalierbarkeit. Vor allem an der Trajektorienplanung bestehen Verbesserungsmöglichkeiten. Auch das Modell des allwissenden Schedulers ist nicht zukunftsträchtig. Die reale Welt enthält Unbekannte, die nicht vorab erfasst werden können. Zukünftige Arbeiten sollten darauf abzielen, dass die Roboter selbst Ausweichmanöver einleiten. Der Scheduler würde dann nur noch eine grobe Fahrplanung durchführen ohne dynamische Hindernisse zu berücksichtigen. Weitere Entwicklungen sollten in der Richtung des Best-Effort-Prinzips erfolgen, da man in der Welt der Unbekannten keine Garantien geben kann.

In seiner jetzigen Form soll der Scheduler in der nahen Zukunft jedoch dazu dienen, neue Spezifikationen für spätere Entwicklungen zu formulieren. Zunächst steht die Sammlung von Erfahrungen mit dem bestehenden System im Vordergrund. Beispielsweise sollte der Ansatz der Constraint-getriebenen Planung von Aufgaben geprüft werden. Es muss auch die Frage geklärt werden, wie sinnvolle Anwendungen für ein derartiges Schwarmbetriebssystem überhaupt aussehen könnten.

Literaturverzeichnis

- [1] GRAFF, D., J. RICHLING und M. WERNER: *jSwarm: Distributed Coordination in Robot Swarms*. In: *Robotic Sensor Networks (RSN 2014)*, April 2014. 1 4
- [2] GRAFF, D., D. RÖHRIG, R. JASPER, H. PARZYJEGLA und J. RABAЕY: *Operating System Support for Mobile Robot Swarms*. In: *Second International Workshop on the Swarm at the Edge of the Cloud (accepted for publication)*, April 2015. 4
- [3] KANT, K. und S. W. ZUCKER: *Toward Efficient Trajectory Planning: The Path-velocity Decomposition*. *Int. J. Rob. Res.*, 5(3):72–89, September 1986. 3 5 22
- [4] WANGDAHL, G. E., S. M. POLLOCK und J. B. WOODWARD: *Minimum-Trajectory Pipe Routing*. *Journal of Ship Research*, 18(1):46–49, 3 1974. 23
- [5] LOZANO-PEREZ, T.: *Spatial Planning: A Configuration Space Approach*. *Computers, IEEE Transactions on*, C-32(2):108–120, Feb 1983. 23
- [6] GRAFF, DANIEL, JAN RICHLING und MATTHIAS WERNER: *Programming and Managing the Swarm – An Operating System for an Emerging System of Mobile Devices*. In: LIN, KAI, HENG QI, KEQIU LI, IVAN STOJMENOVIC, ALBERT ZOMAYA, HONGYI WU, SONG GUO und SYMEON PAPAVASSILIOU (Herausgeber): *9th IEEE International Conference on Mobile Ad-hoc and Sensor Networks (MSN 2013)*, Seiten 9–16. IEEE Computer Society, December 2013. 6

Abbildungsverzeichnis

2.1 Architektur	5
4.1 Ausgangssituation	11
4.2 Aufgabenareal	11
4.3 Hindernisse durchgezogen, Areal A gestrichelt, Schnittmenge $A \cap S$ gelblich und Punkt P	12
4.4 Konkreter Ort	12
4.5 Pausen	13
4.6 Periodischer Schedule	16
4.7 Abhängigkeitsgraph	17
4.8 Entfernung eines Jobs	18
4.9 Einplanung des Jobs X	20
4.10 Entfernung des Jobs X	21
4.11 Kürzester Pfad	23
4.12 Sichtlinien zwischen Polygonen	23
4.13 Verbotene Regionen	24
4.14 Geschwindigkeitsprofil	24
4.15 Graphkonstruktion des Geschwindigkeitsprofils	25
4.16 Trajektorie	26
4.17 Abbildung eines dynamischen Hindernisses als verbotene Region	26
4.18 Kachelmuster der segmentweisen Berechnung verbotener Regionen	27
4.19 Fallunterscheidung zur Berechnung verbotener Regionen	28
4.20 Bewegter Hindernispunkt P und Segment des Pfades π	28
4.21 $s \times t$ -Basis	29
4.22 Beschneidung des Polygons	30
4.23 Schnittmenge der Geraden g mit dem Hindernis O^m	30
4.24 Verschiebung der Schnittlinie im $s \times t$ -Raum	31
4.25 Beschneidung der Schnittfläche $S(T_j)$	31

4.26 Spur V des Punktes \vec{x}_i^r	32
4.27 Graphen ohne und mit Wartezeit	33
4.28 Graphkonstruktion	33
6.1 Extraktion eines Punktes aus einem Areal	54
6.2 Pufferung von Hindernissen durch Minkowski-Summe	57
6.3 Testfall: Anzahl bewegter Hindernisse	58
6.4 Testfall: Hindernisdetailgrad	59
6.5 Testfall: Segmentanzahl der Hindernistrajektorie	59
6.6 Testfall: Segmentanzahl des Roboterpfades	60
6.7 Detailgrad verbotener Regionen	60
6.9 Knotenzahl (Aufstellung)	62
6.10 Knotenzahl (Messung)	62
6.11 Pausenzahl – Einzelter Knoten (Messung)	63
6.12 Pausenzahl – Zwei Knoten (Messung)	64
6.13 Periodische Einplanung	64
6.14 Einplanung abhängiger Aufgaben	65
6.15 Pseudostationäres Hindernis blockiert Pfad	66
6.16 Ungünstiges Orts-Sampling	67

Tabellenverzeichnis

4.1 Aufgabenspezifikationen	17
4.2 Implizite Aufgabenspezifikationen	18

Abkürzungsverzeichnis

JAR Java Archive

JVM Java Virtual Machine

Anhang A

Pseudocode

Algorithmus A.1 Planung eines Jobs

input: Jobspezifikation

output: Erfolgsstatus

```

1: procedure SCHEDULE_SINGLE(spec)
2:   location_space  $\leftarrow$  spec.location_space \  $O^s$ 
3:   locations  $\leftarrow$  SAMPLE_LOCATIONS(location_space)
4:   for all location  $\in$  locations do
5:     slots  $\leftarrow$  FIND_SLOTS(spec, location)
6:     for all slot  $\in$  slots do
7:       status  $\leftarrow$  PLAN_JOB(location, slot, spec)
8:       if status then
9:         return true
10:      end if
11:    end for
12:  end for
13:  return false
14: end procedure
15: function SAMPLE_LOCATIONS(location_space)
16:   queue  $\leftarrow$  [location_space]                                 $\triangleright$  eine Singleton-Liste
17:   locations  $\leftarrow$   $\emptyset$ 
18:   i  $\leftarrow N$                                           $\triangleright N$  (konstant) beschränkt die Anzahl der Orte
19:   while queue  $\neq \emptyset$   $\wedge i > 0$  do            $\triangleright N$  mal oder bis queue leer
20:     area  $\leftarrow$  POP(queue)
21:     sample  $\leftarrow$  SAMPLE_LOCATION(area)           $\triangleright$  innerer Punkt der subarea
22:     locations  $\leftarrow$  locations  $\cup \{ sample \}$ 
23:     subareas  $\leftarrow$  DEVIDE_AREA(area, sample)  $\triangleright$  teilt area in vier rechteckige
        Untergebiete auf
24:     ADD(queue, subareas)
25:     i  $\leftarrow i - 1$ 
26:   end while
27:   return locations
28: end function
29: function FIND_SLOTS(spec, location)
30:   slots  $\leftarrow \emptyset$ 
31:   for all node  $\in$  nodes do
32:     for all slot  $\in$  slots do
33:       if CHECK(spec, location, slot) then     $\triangleright$  CHECK prüft die Bedingungen
        aus 4.2.1 Knotenpause
34:         slots  $\leftarrow$  slots  $\cup \{ slot \}$ 
35:       end if
36:     end for
37:   end for
38:   return slots
39: end function

```

Algorithmus A.2 Planung periodischer Jobs

input: Periodische Jobspezifikation

output: Erfolgsstatus

```

1: procedure SCHEDULE_PERIODIC_SAME_LOCATION(pspec)
2:   location_space  $\leftarrow$  pspec.location_space \  $O^s$ 
3:   locations  $\leftarrow$  SAMPLE_LOCATIONS(location_space)
4:   for all location  $\in$  locations do
5:     pstart  $\leftarrow$  pspec.t_start                                 $\triangleright$  Beginn der aktuellen Periode
6:     nobreak  $\leftarrow$  true
7:     for pspec.repetitions times do
8:       pend  $\leftarrow$  pstart + pspec.period                 $\triangleright$  Ende der aktuellen Periode
9:       spec  $\leftarrow$  (location, pstart, pend, pspec.duration)     $\triangleright$  Der Job muss
      innerhalb von [pstart, pend] begonnen und abgeschlossen werden.
10:      status  $\leftarrow$  SCHEDULE_SINGLE(spec)
11:      if  $\neg$ status then
12:        nobreak  $\leftarrow$  false
13:        break
14:      end if
15:    end for
16:    if nobreak then
17:      return true
18:    else
19:      UNDO_CHANGES
20:    end if
21:  end for
22:  return false
23: end procedure
24: procedure SCHEDULE_PERIODIC_INDEPENDENT_LOCATION(pspec)
25:   pstart  $\leftarrow$  pspec.t_start                                 $\triangleright$  Beginn der aktuellen Periode
26:   for pspec.repetitions times do
27:     pend  $\leftarrow$  pstart + pspec.period                 $\triangleright$  Ende der aktuellen Periode
28:     spec  $\leftarrow$  (pspec.location_space, pstart, pend, pspec.duration)   $\triangleright$  Der Job
      muss innerhalb von [pstart, pend] begonnen und abgeschlossen werden.
29:     status  $\leftarrow$  SCHEDULE_SINGLE(spec)
30:     if  $\neg$ status then
31:       return false
32:     end if
33:   end for
34:   return true
35: end procedure

```

Algorithmus A.3 Planung abhängiger Jobs

input: Mehrere Jobspezifikationen und Abhängigkeitsgraph

output: Erfolgsstatus

```

1: procedure SCHEDULE_DEPENDENT(specs, dependencies)      ▷ Die Graphknoten
   sind IDs der Jobs.
2:   specs'  $\leftarrow$  NORMALIZE_SPECS(specs, dependencies)
3:   for all spec  $\in$  specs' do
4:     spec'  $\leftarrow$  CONSTRAIN(spec, dependencies)
5:     if spec' = null then
6:       return false
7:     end if
8:     status  $\leftarrow$  SCHEDULE_SINGLE(spec')
9:     if  $\neg$ status then
10:      return false
11:    end if
12:  end for
13:  return true
14: end procedure

15: function NORMALIZE_SPECS(specs, dependencies)           ▷ normalisiert die
   Spezifikationen nach den Formeln 4.8 und 4.9 aus Abschnitt 4.2.4
16:   sorted  $\leftarrow$  TOPOSORT(dependencies)                  ▷ topologische Sortierung
17:   normalized  $\leftarrow$  CLONE(specs)
18:   for all id  $\in$  sorted do
19:     spec  $\leftarrow$  LOOKUP(normalized, id)    ▷ sucht nach der Spezifikation mit der
   angegebenen ID
20:      $T_{dep} \leftarrow \{ s.t_{finish,min} + d_m \mid s \in normalized \wedge s.id \in dep(dependencies, id) \}$ 
21:     spec.tstart,min  $\leftarrow \max T_{dep} \cup \{ spec.t_{start,min}, t_{FH} \}$ 
22:   end for
23:   for all id  $\in$  reversed(sorted) do
24:     spec  $\leftarrow$  LOOKUP(normalized, id)
25:      $T_{dep} \leftarrow \{ s.t_{start,max} - d_m \mid s \in normalized \wedge s.id \in dep^{-1}(dependencies, id) \}$ 
26:     spec.tfinish,max  $\leftarrow \min T_{dep} \cup \{ spec.t_{finish,max} \}$ 
27:   end for
28:   return SORT_BY_DEADLINE(normalized)
29: end function

30: function CONSTRAIN(spec, dependencies)           ▷ schränkt die Spezifikation
   hinsichtlich der eingeplanten Abhängigkeiten weiter ein
31:   if dep(dependencies, spec.id)  $\neq \emptyset$  then
32:     return spec
33:   end if
34:    $t_{dep} \leftarrow \max \{ j.t_{finish} \mid j \in J \wedge j.id \in dep(dependencies, spec.id) \}$       ▷
   Abschlusszeit der letzten Abhängigkeit
35:   tstart,min  $\leftarrow t_{dep} + d_m
36:   if tstart,min  $<$  spec.tstart,min then
37:     return spec                                ▷ Spezifikation unverändert
38:   end if
39:   if tstart,min  $>$  spec.tstart,max then
40:     return null                               ▷ Einplanung nicht möglich
41:   end if
42:   return SUBSTITUTE_EARLIEST_START_TIME(spec, tstart,min)
43: end function$ 
```

Algorithmus A.4 Trajektorienplanung eines neuen Jobs

input: Ort, Knotenpause und Jobspezifikation

output: Erfolgsstatus

```

1: procedure PLAN_JOB(location, slot, spec)
2:   node  $\leftarrow$  slot.node
3:    $\pi_1 \leftarrow \text{CALC\_SPATIAL\_PATH}(\text{slot.}\vec{x}_{\text{start}}, \text{location})$ 
4:    $\pi_2 \leftarrow \text{CALC\_SPATIAL\_PATH}(\text{location}, \text{slot.}\vec{x}_{\text{finish}})$ 
5:    $\tau_1 \leftarrow \text{CALC\_MT\_TRAJECTORY}(\pi_1, \text{slot.}t_{\text{start}})$ 
6:    $\text{job} \leftarrow (\text{node, location, } \tau_1 \cdot t_{\text{finish}}, \text{spec.}d)$ 
7:    $\tau_2 \leftarrow \text{CALC\_FT\_TRAJECTORY}(\pi_2, \text{job.}t_{\text{finish}}, \text{slot.}t_{\text{finish}})$ 
8:    $\tau_{12} \leftarrow \text{STATIONARY\_TRAJECTORY}(\text{location, job.}t_{\text{start}}, \text{job.}t_{\text{finish}})$   $\triangleright$  eine
    stationäre Trajektorie am Ort location von job.t_start bis job.t_finish
9:   UPDATE_TRAJECTORY(node,  $\tau_1 \cup \tau_{12} \cup \tau_2$ )
10:  ADD_JOB(node, job)
11: end procedure

```

Algorithmus A.5 Berechnung einer Trajektorie mit fester Ankunftszeit

input: Raumpfad und Zeitintervall

output: Trajektorie

```

1: function CALC_FT_TRAJECTORY( $\pi, t_{\text{start}}, t_{\text{finish}}$ )
2:   forbidden  $\leftarrow \text{FRB}(O^m, \pi)$ 
3:   graph  $\leftarrow (\emptyset, \emptyset)$ 
4:   CONNECT_DIRECTLY(graph, forbidden)
5:   CONNECT_LAZY(graph, forbidden)
6:   start  $\leftarrow (0, t_{\text{start}})$ 
7:   finish  $\leftarrow (\text{length}(\pi), t_{\text{finish}})$ 
8:    $\sigma \leftarrow \text{DIJKSTRA}(\text{graph, start, finish})$   $\triangleright$  berechnet das Geschwindigkeitsprofil
9:   return COMPOSE( $\pi, \sigma$ )
10: end function

```

Algorithmus A.6 Berechnung einer Trajektorie mit variabler Ankunftszeit

input: Raumpfad und Abfahrtszeit

output: Trajektorie

```

1: function CALC_MT_TRAJECTORY( $\pi, t_{start}$ )
2:    $forbidden \leftarrow \text{CALC\_FORBIDDEN\_REGION}(O^m, \pi)$ 
3:    $graph \leftarrow (\emptyset, \emptyset)$ 
4:   CONNECT_DIRECTLY( $graph, forbidden$ )
5:   CONNECT_LAZY( $graph, forbidden$ )
6:    $candidates \leftarrow \text{CONNECT\_FINISH\_CANDIDATES}(graph, forbidden)$ 
7:    $start \leftarrow (0, t_{start})$ 
8:    $finish \leftarrow \text{DETERMINE\_FINISH}(graph, candidates)$             $\triangleright$  bestimmt den
   Zielknoten aus der Menge  $candidates$  der als erstes erreicht werden kann
9:    $\sigma \leftarrow \text{DIJKSTRA}(graph, start, finish)$   $\triangleright$  berechnet das Geschwindigkeitsprofil
10:  return COMPOSE( $\pi, \sigma$ )
11: end function

```

Algorithmus A.7 Berechnung der verbotenen Region

input: Dynamische Hindernisse und Raumpfad

output: Verbotene Region

```

1: function CALC_FORBIDDEN_REGION( $O^m, \pi$ )
2:    $forbidden \leftarrow \emptyset$ 
3:   for all  $O_k^m \in \text{GET_OBSTACLES}(O^m)$  do
4:      $\tau \leftarrow \text{GET_TRAJECTORY}(O_k^m)$ 
5:     if  $\neg\text{CHECK_ENVELOPE_INTERSECTION}(O_k^m, \pi)$  then     $\triangleright$  prüft ob sich die
       rechteckigen Hüllen zweier Punktmengen schneiden
6:       continue
7:     end if
8:     for all  $\tau^* \in \text{GET_SEGMENTS}(\tau)$  do
9:       if  $\neg\text{CHECK_ENVELOPE_INTERSECTION}(O_k^{m*}, \pi)$  then       $\triangleright O_k^{m*}$ 
          bezeichnet den Ausschnitt des Hindernisses  $O_k^m$  während der Zeit von  $\tau^*$ 
10:        continue
11:      end if
12:       $t \leftarrow \text{GET_START_TIME}(\tau^*)$ 
13:      for all  $\pi^* \in \text{GET_SEGMENTS}(\pi)$  do
14:        if  $\neg\text{CHECK_ENVELOPE_INTERSECTION}(O_k^{m*}, \pi^*)$  then
15:          continue
16:        end if
17:        if regular case then
18:           $mask \leftarrow \text{MAKE_PARALLELOGRAM}(\tau^*, \pi^*)$ 
19:           $f \leftarrow \text{TRANSFORM_REGULAR}(O_k^m(t) \cap mask)$ 
20:        else if parallel case then
21:           $mask_{xy} \leftarrow \text{MAKE_SPATIAL_MASK}(\tau^*, \pi^*)$ 
22:           $mask_{st} \leftarrow \text{MAKE_ARC_TIME_MASK}(\tau^*, \pi^*)$ 
23:           $f \leftarrow \text{TRANSFORM_PARALLEL}(O_k^m(t) \cap mask_{xy}) \cap mask_{st}$ 
24:        else if stationary case then
25:          if  $\tau^*$  is stationary) then
26:            if  $\pi^* \cap O_k^m(t) = \emptyset$  then
27:               $f \leftarrow \emptyset$ 
28:            else
29:               $f \leftarrow$  a fully filled region tile
30:            end if
31:          else
32:             $mask \leftarrow \text{MAKE_TRACE_MASK}(\tau^*, \pi^*)$ 
33:             $f \leftarrow \text{TRANSFORM_STATIONARY}(O_k^m(t) \cap mask)$ 
34:          end if
35:        end if
36:         $forbidden \leftarrow forbidden \cup f$ 
37:      end for
38:    end for
39:  end for
40:  return  $forbidden$ 
41: end function

```

Algorithmus A.8 Erstellung des Navigationsgraphen für Geschwindigkeitsprofile

input: Graph und verbotene Region

```

1: procedure CONNECT_DIRECTLY(graph, forbidden)
2:   vertices  $\leftarrow$  GET_VERTICES(forbidden)
3:   ADD_VERTICES(graph, vertices)
4:   for all src  $\in$  vertices do
5:     for all dst  $\in$  vertices do
6:       if src  $\neq$  dst  $\wedge$  VISIBLE(src, dst, forbidden)  $\wedge$  CAUSAL(src, dst) then
         $\triangleright$  CAUSAL prüft ob zeitliche Relationen erfüllbar sind
7:         ADD_EDGE(graph, (src, dst))
8:       end if
9:     end for
10:   end for
11: end procedure
12: procedure CONNECT_LAZY(graph, forbidden)
13:   motion_rays  $\leftarrow$  CAST_MOTION_RAYS(forbidden)
14:   stationary_rays  $\leftarrow$  CAST_STATIONARY_RAYS(forbidden)
15:   for all mr  $\in$  motion_rays do
16:     for all sr  $\in$  stationary_rays do
17:       if mr  $\cap$  sr  $\neq \emptyset$  then
18:         vertex  $\leftarrow$  point(mr  $\cap$  sr)  $\triangleright$  point fasst die Punktmenge mr  $\cap$  sr
          als einzelnen Punkt auf
19:         ADD_VERTEX(graph, vertex)
20:         ADD_EDGE(graph, (sr.origin, vertex))
21:         ADD_EDGE(graph, (vertex, mr.origin))
22:       end if
23:     end for
24:   end for
25: end procedure
26: procedure CONNECT_FINISH_CANDIDATES(graph, forbidden)
27:   for all v  $\in$  graph.vertices do
28:     c  $\leftarrow$  CALC_CANDIDATE(v)  $\triangleright$  CALC_CANDIDATE berechnet einen
        Zielknotenkandidaten
29:     if VISIBLE(v, c, forbidden)  $\wedge$  CAUSAL(v, c) then
30:       ADD_VERTEX(graph, c)
31:       ADD_EDGE(graph, (v, c))
32:     end if
33:   end for
34: end procedure
35: function VISIBLE(src, dst, forbidden)
36:   return line(src, dst)  $\cap$  forbidden =  $\emptyset$ 
37: end function

```
