ALBERT-LUDWIGS-UNIVERSITÄT
FREIBURG

INSTITUT FÜR INFORMATIK

Arbeitsgruppe Autonome Intelligente Systeme

Prof. Dr. Wolfram Burgard

Zielgerichtete Kollisionsvermeidung für mobile Roboter in dynamischen Umgebungen

Diplomarbeit

Cyrill Stachniss

März 2002 – August 2002

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung meiner Diplomarbeit unterstützt haben. Besonders Herrn Prof. Dr. Wolfram Burgard danke ich für die Zeit, die er sich für mich genommen hat. Er stand mir bei Fragen jederzeit zur Seite und hat mich unterstützt, wo immer es ihm möglich war. Auch bei Maren Bennewitz, Dirk Hähnel, Moritz Tacke und Dirk Zitterell möchte ich mich für die vielen kleinen Tipps und die gute Arbeitsatmosphäre bedanken, nicht zu vergessen Christoph Petz für seine Verbesserungsvorschläge während der schriftlichen Ausarbeitung der Arbeit. Schließlich bedanke ich mich bei meiner Familie für die Unterstützung, die sie mir während meines gesamten Studiums entgegengebracht hat.

Inhaltsverzeichnis

1. Einleitung

Die mobile Robotik hat in den letzten Jahren einen deutlichen Aufschwung erlebt, der sich vermutlich auch noch einige Zeit weiter fortsetzen wird. Für die meisten Aufgaben innerhalb dieses Themengebietes ist ein zuverlässiges und robustes Navigationssystem für eine kollisionsfreie Fahrt unverzichtbar. Bereits Latombe nennt dies *„eminently neccessary since, by definition, a robot accomplishes tasks by moving in the real world“* [LATOMBE 1991]. Schon seit den Anfängen der mobilen Robotik in den 70er Jahren stehen Navigationsaufgaben im Interesse der Forschung. Während die ersten Roboter noch versuchten einer weißen ununterbrochenen Linie zu folgen, konnten sie sich gegen Ende der 80er Jahre eigenständig in selbst erstellten Karten zufriedenstellend lokalisieren, Zielpunkte anfahren und Hindernissen ausweichen (vgl. [MORAVEC 1999]). Bis in die heutige Zeit werden die Verfahren zum Planen von Trajektorien stets verbessert und erweitert. Ein moderner Roboter kann sich auch in bevölkerten und dadurch sich stets verändernden Umgebungen sicher bewegen. Trotz intensiver Forschung lassen sich bei vielen Ansätzen zur Kollisionsvermeidung noch immer Umgebungen konstruieren, in denen ein Roboter keinen Pfad zu einem praktisch erreichbaren Zielpunkt findet. Dies liegt häufig an einer unzureichenden Modellierung der physikalischen Eigenschaften des Roboters, wie beispielsweise dem Brems- und Beschleunigungsverhalten. Aus dem gleichen Grund weisen viele der derzeit eingesetzten Systeme auch Defizite in Standardsituationen auf, so dass sie zwar zum Zielpunkt gelangen, dafür aber nicht den schnellstmöglichen Weg wählen. Die Komplexität der Berechnung einer optimalen Trajektorie wächst exponentiell mit der Anzahl der beteiligten Parameter. Daher sind heutige Computer nicht leistungsfähig genug, um alle physikalischen Eigenschaften eines Roboters exakt zu modellieren und diese bei der Berechnung der Trajektorie zu berücksichtigen. Jedoch ist heute schon absehbar, dass man mit der Hardware der nächsten Jahre und einigen Approximationen in der Berechnung schon sehr nah an die optimale Lösung des Problems der Pfadplanung und Kollisionsvermeidung für mobile Roboter herankommen wird.

Mit unserer Arbeit möchten wir einen neuen Ansatz zur zielgerichteten Kollisionsvermeidung vorstellen, der die kinematischen Einschränkungen wie Brems- und Beschleunigungsverhalten eines Roboters bereits in der Planung berücksichtigt und trotzdem auf aktuellen Computern lauffähig ist. Innerhalb eines lokalen Fensters, dessen Größe sich der zur Verfügung stehenden Rechenleistung anpasst, wird dabei im vollen Konfigurationsraum nach dem zeitoptimalen Pfad mit möglichst geringem Kollisionsrisiko gesucht. Die so entstandenen Teillösungen lassen sich in den übrigen Raum integrieren, so dass sie sich dann zu einer globalen Lösung zusammensetzen. Dadurch erreicht ein Roboter in vielen Situationen schneller seinen Zielpunkt als unter Verwendung der meisten bisherigen Ansätze. Außerdem besitzt das Verfahren keine lokalen Mi-

nima, die in vielen populären Ansätzen zur Pfadplanung und Kollisionsvermeidung vorhanden sind, was eine weitere Stärke unseres Systems darstellt.

1.1. Die Zielsetzung

Die Aufgabe dieser Arbeit ist die Entwicklung eines Ansatzes zur Pfadplanung und Kollisionsvermeidung für mobile Roboter. Der Roboter bewegt sich in einer ihm bekannten Umgebung, die sich allerdings zur Laufzeit ändern kann, d.h. neue Hindernisse oder Personen müssen erkannt und umfahren werden. Das System soll auf echten Robotern eingesetzt werden, nicht nur in Simulationen. Dafür müssen alle Berechnungen in einem relativ kurzen Zeitfenster ausgeführt werden, ansonsten könnte der Roboter nicht schnell genug auf Veränderungen der Welt reagieren. Das der Arbeit zu Grunde liegende Kontrollsystem erlaubt eine Reaktionszeit von ca. $250\ ms$, d.h. in dieser Zeit sollten die Berechnungen ausgeführt werden können. Der Roboter, der seine Position innerhalb der Umgebung kennt, soll den anzufahrenden Zielpunkt kollisionsfrei und in möglichst kurzer Zeit erreichen. Im Vergleich zur bisher verwendeten Kollisionsvermeidung [FOX et al. 1997] sollen dabei die kinematischen Einschränkungen wie Brems- und Beschleunigungsverhalten, denen der Roboter physikalisch unterworfen ist, berücksichtigt werden. Dadurch versprechen wir uns in bestimmten Situationen ein verbessertes und vorausschauendes Fahrverhalten des Roboters in seiner Umgebung. Ein Teil der Aufgabe ist das Finden eines Verfahrens zur Handhabung des komplexen Suchraums.

1.2. Der Aufbau dieser Arbeit

Diese Arbeit ist wie folgt gegliedert: Nach dieser Einleitung werden in Kapitel 2 bisherige Arbeiten vorgestellt, die Überschneidungspunkte mit unserer Arbeit aufweisen bzw. ähnliche Ziele mit anderen Ansätzen verfolgen. Das darauf folgende Kapitel stellt dann grundlegende Algorithmen vor, die in unserer Arbeit Verwendung finden. Außerdem wird die Repräsentation der Umgebung erklärt, sowie eine mathematische Formulierung für die Bewegung eines Roboters hergeleitet. Kapitel 4 beschreibt dann Schritt für Schritt das Verfahren zur zielgerichteten Kollisionsvermeidung, mit dem die Trajektorie des Roboters geplant wird, und stellt somit das Kernstück der entwickelten Software vor. In Kapitel 5 wird dann auf das zu Grunde liegende Kontrollsystem für mobile Roboter, die so genannte *Bee-Software*, eingegangen und die Integration der neu entwickelten Komponente beschrieben. Anschließend wird in Kapitel 6 das Verhalten des Steuerungsmoduls in Experimenten untersucht und Vergleiche zu bisherigen Ansätzen gezogen. Das letzte Kapitel fasst schließlich die gewonnenen Erkenntnisse zusammen und gibt einen Ausblick auf künftige Arbeiten und mögliche Erweiterungen.

2. Verwandte Arbeiten

Um einen Überblick über den aktuellen Stand der Forschung im Bereich der Pfadplanung und Kollisionsvermeidung für mobile Roboter zu geben, werden wir einige der grundlegenden Verfahren kurz vorstellen und Gemeinsamkeiten sowie Unterschiede zu unserer Arbeit aufzeigen.

Unabhängig davon, wie einzelne Systeme die Fahrkommandos für den Roboter ermitteln, existieren zwei Klassen von Ansätzen bezüglich der Repräsentation der Umgebung. Die erste Klasse geht von einer sich nicht ändernden Umgebung des Roboters aus. Dieser besitzt oder erstellt eine statische Karte seiner Welt, in der er sich dann bewegt. Bei dieser Sicht der Dinge lassen sich viele Navigationsaufgaben auf Basis einmaliger Berechnungen schnell lösen, allerdings können Ansätze dieser Art nicht mit sich zur Laufzeit ändernden Umgebungen umgehen, so dass eine Fahrt in beispielsweise einer bevölkerten Umgebung nahezu unmöglich ist.

Die zweite Klasse von Verfahren berücksichtigt auch Veränderungen der Welt. Anhand von Sensoren erkennt der Roboter einzelne Objekte oder berücksichtigt sogar die Art deren Bewegung. Diese Systeme repräsentieren ihre Umgebung wesentlich näher an der Realität als die Gruppe zuvor, jedoch muss dafür deutlich mehr Rechenleistung zur Verfügung gestellt werden. Natürlich existieren auch Mischformen der beiden Klassen. Dabei wird die Welt bei der Planung der Trajektorie des Roboters als statisch angesehen, jedoch erkennt der Roboter in einer lokalen Umgebung einzelne Hindernisse und korrigiert den anhand der statischen Karte berechneten Pfad.

Unabhängig von der Repräsentation der Umgebung lassen sich Ansätze zur Pfadplanung und Kollisionsvermeidung in Gruppen einteilen, je nach dem wie sie die günstigste Trajektorie für den Roboter bestimmen. Vier Gruppen von Ansätzen, die sich im Laufe der Jahre etabliert haben, möchten wir nun im Folgenden kurz skizzieren und unsere Arbeit dann innerhalb dieser Gruppen einordnen:

Eine Gruppe arbeitet auf Basis so genannter *Roadmaps* [LATOMBE 1991]. Dabei werden zwischen markanten Punkten innerhalb einer Umgebungskarte eindimensionale Kurven gelegt. Jede dieser Kurven bildet einen gültigen Pfad zwischen zwei Positionen. Diese *Roadmaps* können beispielsweise die Geraden zwischen allen Eckpunkten aller Hindernisse sein, die kein anderes Hindernis schneiden, oder auch die Kanten eines Voronoidiagramms [OTTMANN und WIDMAYER 1996], bei denen die Hindernisse als Bezugspunkte dienen [CHOSET et al. 1997]. Dann ergibt sich der abzufahrende Pfad durch eine Verbindung der Position des Roboters zu einer *Roadmap*, von dort aus entlang dieser, bis sie wieder verlassen wird, um den Zielpunkt zu erreichen. Dieses Verfahren ist eines der ersten überhaupt, welches innerhalb der mobilen Robotik

verwendet wurde [NILSSON 1969]. Das Hauptproblem liegt im Finden günstiger *Roadmaps*, für das es eine Vielzahl von Verfahren gibt [O'DUNLAING und YAP 1982, CANNY 1988, LATOMBE 1991]. Über randomisierte Methoden versuchen beispielsweise Hsu et al. die komplette Neuberechnung der *Roadmaps* bei neu erkannten Hindernissen zu vermeiden und das Verfahren somit effizienter zu machen [HSU et al. 2000].

Eine weitere Gruppe von Ansätzen baut auf der *Potentialfeld-Methode* auf, die ursprünglich von Khatib entwickelt wurde [KHATIB 1986]. Diese basiert auf der Idee der Potentialfelder, ähnlich dem elektrischen Coulomb-Feld in der Physik. Dabei besitzen Hindernisse ein abstoßendes, nahe ihnen häufig exponentiell anwachsendes Potential, das den Roboter auf Distanz hält. Der Zielpunkt dagegen besitzt ein anziehendes, meist quadratisches Potential, wodurch der Roboter zu diesem hingezogen wird. Durch das Verfolgen des negativen Gradienten der Summe der einzelnen Potentialfelder ergibt sich die Richtung, in der sich der Roboter zu bewegen hat. Allerdings können Potentialfelder lokale Minima besitzen, die nicht identisch mit dem Zielpunkt sind. Dadurch bewegt sich der Roboter unter Umständen zu einer Position hin, aus der er nicht mehr entkommen kann. Um mit diesem generellen Problem der *Potentialfeld-Methode* umzugehen, wird häufig versucht eine Strategie zu entwickeln, die den Roboter aus einem lokalen Minimum herausführt [RATERING und GINI 1993]. Eine andere Idee ist das Verhindern eines dauerhaften Minimums durch ein zusätzliches, schwaches und sich zufällig änderndes Potentialfeld, wie Balch und Hybinette dies in einem Potentialfeld für Gruppen von Robotern realisieren [BALCH und HYBINETTE 2000].

Immer wieder werden neue Verfahren vorgestellt, die versuchen die Probleme der *Potentialfeld-Methode* zu reduzieren. Dazu zählt auch das *Vector Field Histogram* (*VFH*) [BORENSTEIN und KOREN 1991] und seine Erweiterung *VFH+* von Ulrich und Borenstein [ULRICH und BORENSTEIN 1998]. Dabei wird über ein vierstufiges Verfahren die zweidimensionale Welt in ein eindimensionales Histogramm transformiert, das die Umgebung repräsentieren soll. Daraus wird dann versucht ein geeignetes Fahrkommando für den Roboters zu ermitteln. Die letzte Erweiterung *VFH** [ULRICH und BORENSTEIN 2000] kombiniert das *VFH+* mit der $A^\star$-Suche, um Probleme mit lokalen Minima zu vermeiden. Xu und Yang versuchen in ihrer aktuellen Arbeit die *Potentialfeld-Methode* mit den dynamischen Einschränkungen eines Roboters zu kombinieren [XU und YANG 2002]. Allerdings wird im derzeitigen Ansatz nur die Translationsgeschwindigkeit, nicht aber die Rotationsgeschwindigkeit des Roboters bei der Planung berücksichtigt.

Das dritte Verfahren zur Pfadplanung, welches wir hier erwähnen möchten, ist die so genannte *Cellular Decomposition* [SCHWARZ und SHARIR 1983, LATOMBE 1991]. Dabei wird der freie Raum in Zellen unterteilt, die eine möglichst einfache Struktur besitzen. Dies können beispielsweise Dreiecke, gleichmäßige Quadrate oder auch durch Quadtrees entstandene Zellen sein. Eine zusammenhängende Teilmenge von Zellen, die Start- und Zielpunkt enthält, bildet dann einen so genannten Kanal, in dem sich ein gültiger Pfad befindet. Die Verknüpfung dieser einzelnen Zellen lässt sich über einen Graphen beschreiben, in dem dann nach einem Pfad gesucht werden kann (vgl. [CHATILA 1982, SMITH und CHEESEMAN 1986, SCHWARZ et al. 1987, KONOLIGE 2000, CHOSET et al. 2000]). Zum Finden des kostengünstigsten Weges innerhalb sol-

cher Graphen existieren viele Methoden, angefangen bei der $A^\star$-Suche [RUSSEL und NORVIG 1994], der *Value Iteration* [SUTTON und BARTO 1998], hierarchischen Methoden [HOLTE et al. 1996], *Learning Real-Time* $A^\star$ [FURCY und KÖNIG 2000], *Focussed* $D^\star$ [STENTZ 1995] oder dessen optimierte Variante *Focussed* $D^\star$ *Lite* [KÖNIG und LIKHACHEV 2002]. Je nach Größe des entstehenden Suchraums steht bei der Steuerung eines Roboters in Echtzeit unter Umständen nicht genügend Leistung zur Verfügung, um den Raum ausreichend zu explorieren. Dann müssen Approximationen oder vorhandene Teillösungen verwendet werden, was die Qualität der resultierenden Trajektorie meist negativ beeinflusst.

Eine weitere Gruppe von Ansätzen trennt die Pfadplanung und die Kollisionsvermeidung auf. So wird die Pfadplanung beispielsweise nach dem Prinzip der *Cellular Decomposition* realisiert, der Roboter folgt allerdings nicht exakt dieser so berechneten Trajektorie. Das eigentliche Fahrkommando des Roboters sendet eine meist reaktive Kollisionsvermeidung, die versucht eine *Navigationsfunktion* zu maximieren, um sich dem Zielpunkt zu nähern und Hindernissen auszuweichen. Dadurch muss die aufwendige Pfadplanung wesentlich seltener ausgeführt werden und zur Kollisionsvermeidung wird auf ein deutlich schneller agierendes, meist reaktives System zurückgegriffen. Häufig wird ein Verfahren, basierend auf dem *Dynamic Window Approach* [FOX et al. 1997], verwendet, das nur in der direkten Umgebung des Roboters arbeitet. Eine ähnliche Idee verfolgt auch die *Curvature-Velocity Method (CVM)* [SIMMONS 1996]. Beide Ansätze betrachten nur den Teil der Umgebung, den der Roboter aufgrund seiner aktuellen Geschwindigkeit in einem gewissen Zeitfenster erreichen kann, und reduzieren somit die Größe des Suchproblems. Die möglichen Trajektorien innerhalb der lokalen Umgebung werden über eine *Navigationsfunktion* bewertet und das günstigste Fahrkommando anschließend ausgeführt. Allerdings besitzen viele der eingesetzen *Navigationsfunktionen* lokale Minima oder berücksichtigen nicht das Bremsverhalten des Roboters. Dies kann wiederum zu suboptimalen Trajektorien führen. Zur Verbesserung des Fahrverhaltens von beliebig geformten Robotern, bei denen die Erkennung einer Kollision aufwendiger ist, existieren Erweiterungen des ursprünglichen Ansatzes von Fox et al., um dieses populäre Verfahren nahezu beliebigen Robotertypen zugänglich zu machen [SCHLEGEL 1998, ARRAS et al. 2002]. Ko und Simmons haben mit der *Lane-Curvature Method* versucht, die *CVM* mit einem *Roadmap* ähnlichen Verfahren zu kombinieren und somit das Fahrverhalten von Robotern in Korridoren und Räumen zu verbessern [KO und SIMMONS 1998]. Der Roboter bewegt sich dabei tendenziell entlang von Fahrspuren, die ihn zu seinem Zielpunkt führen sollen.

Ansätze zur Kollisionsvermeidung, die auf dem *Dynamic Window Approach* basieren, weisen oft Nachteile beim Abbiegen in enge Korridore auf. Besonders deutlich wird dies, wenn die Breite der Gänge nur geringfügig größer ist als der Durchmesser des Roboters selbst. Abbildung 2.1 zeigt eine solche Umgebung. Der Zielpunkt liege weit entfernt im Süden und der Roboter bewege sich von Feld c_i ausgehend in Richtung c_0. Da im *Dynamic Window Approach* unter anderem versucht wird, die Geschwindigkeit zu maximieren, meist ohne den Geschwindigkeitsraum vorausschauend zu betrachten, verpasst der Roboter die Abzweigung bei Feld c_0, da er sich dort mit Höchstgeschwindigkeit in Richtung c_{-i} bewegt. Daraufhin bremst er ab, wendet und der

Vorgang wiederholt sich. Dieses Verhalten zeigen auch viele Weiterentwicklungen des *Dynamic Window Approach* aufgrund der fehlenden Vorausschau im Raum der Geschwindigkeiten, so beispielsweise der Ansatz von Brock und Khatib [BROCK und KHATIB 1999], .

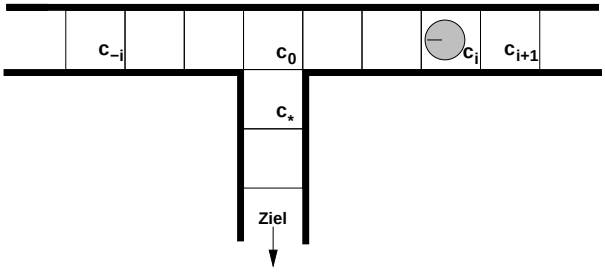

Abbildung 2.1: In der dargestellten Situation können Roboter auf Basis des *Dynamic Window Approach* das Ziel nicht erreichen, da sie zum Abbiegen in dem Feld c_0 nicht früh genug ihre Geschwindigkeit reduzieren.

In jüngster Zeit entstanden Versuche von Mínguez et al. die Grundidee des *Dynamic Window Approach* nochmals zu erweitern. Mittels des so genannten *Nearness Diagram* wird die Umgebung des Roboters analysiert und in Standardsituationen klassifiziert, anhand derer dann die Fahrkommandos an den Roboter gesendet werden [MÍNQUEZ und L.MONTANO 2000, MÍNQUEZ et al. 2001]. Basierend auf dieser Idee entstanden in diesem Jahr wiederum Systeme, die es ermöglichen die dynamischen Eigenschaften des Roboters über den so genannten *Ego-Kinematic Space* auch der *Navigationsfunktion* zugänglich zu machen und so das Fahrverhalten der Roboter weiter zu optimieren. Die Grundidee des *Ego-Kinematic Space* ist ein Raum, in dem sich ein Roboter ohne Einschränkungen frei bewegen kann, da seine physikalischen Eigenschaften durch den Raum selbst modelliert werden. Somit können bestehende *Dynamic Window Approach* Systeme, laut Mínguez, verhältnismäßig einfach um die dynamische Modellierung erweitert werden [MÍNQUEZ et al. 2002].

Unser Ansatz zur zielgerichteten Kollisionsvermeidung gehört zur dritten dieser vier Gruppen. Zusätzlich existiert, ähnlich wie bei den zuletzt genannten Verfahren, eine lokale Umgebung, deren Größe aber mit der vorhandenen Rechenleistung skaliert. Die darin lokal bestimmte Teillösung lässt sich während der Fahrt des Roboters zu einer globalen Lösung zusammensetzen. Durch Kombination des lokalen und globalen Ansatzes sowie der Modellierung der physikalischen Eigenschaften des Roboters erhalten wir bei ausreichend Rechenleistung zu einer aktuellen Situation sogar den zeitoptimalen Pfad, auf dem der Roboter bewegt werden kann.

Wir teilen den Raum in Quadrate ein und suchen in dem so entstehenden Graphen nach dem zeitoptimalen Pfad. Daher erfolgt die Klassifizierung zur Gruppe der *Cellular Decomposition* Ansätze. Allerdings wird zur Suche nicht wie in den meisten anderen Verfahren (wie beispiels-

weise bei [KONOLIGE 2000]) der zweidimensionale $\langle x, y \rangle$-Raum betrachtet, sondern dieser wird um drei zusätzliche Dimensionen erweitert, nämlich um die Orientierung des Roboters sowie um seine Translations- und Rotationsgeschwindigkeit. Eine *Navigationsfunktion* kommt hier nicht zum Einsatz. Dadurch wächst allerdings auch die Komplexität des Suchproblems stark an. Dies muss ein Verfahren, das einen Roboter in Echtzeit steuern soll, berücksichtigen. Für das Ausführen einer naiven $A^\star$-Suche im $\langle x, y, \theta, v, \omega \rangle$-Raum reicht die zur Verfügung stehende Leistung aktueller Computer nicht aus, so dass neue Wege gefunden werden müssen in einem Suchraum dieser Größe effizient nach einer Lösung zu suchen. Dies geschieht in unserem Algorithmus über die zuvor erwähnte lokale Umgebung, in der die Suche im vollen Konfigurationsraum mit Hilfe geeigneter Heuristiken realisiert werden kann. Es ist zwar deutlich mehr Rechenleistung erforderlich als bei den meisten zuvor genannten Ansätzen, jedoch liegt der resultierende Pfad bei Verwendung aktueller Computersysteme schon relativ nahe an der optimalen Lösung. Zusätzlich besitzt unser Verfahren, das auf der $A^\star$-Suche basiert, keine lokalen Minima wie Ansätze nach der *Potentialfeld-Methode* oder dem *Dynamic Window Approach* mit einer reaktiven *Navigationsfunktion*. Auch in der zuvor angesprochenen Situation, dargestellt in Abbildung 2.1, navigiert ein durch unseren Ansatz kontrollierter Roboter sicher zum Zielpunkt, da seine kinematischen Eigenschaften korrekt modelliert werden.

3. Grundlagen

In diesem Kapitel beschreiben wir einige grundlegende Aspekte unserer Arbeit. Anfangs stellen wir zwei zentrale Algorithmen vor, die für das Verständnis der Arbeit sehr wichtig sind. Danach gehen wir kurz auf die Repräsentation der Umgebung ein und führen den vollen Konfigurationsraum des Roboters ein, in dem die Planung der Trajektorien umgesetzt wird. Anschließend werden Gleichungen hergeleitet, die die Bewegung eines Roboters beschreiben.

3.1. Die $A^\star$-Suche

Der $A^\star$-Algorithmus [RUSSEL und NORVIG 1994] ist eines der grundlegenden Suchverfahren zum Finden des kürzesten Weges von einem Startknoten zu einem Zielknoten innerhalb eines Graphen. $A^\star$ gehört zur Gruppe der informierten Suchverfahren. Im Vergleich zu den uninformierten Suchverfahren, wie z.B. „Gierige Suche“ [OTTMANN und WIDMAYER 1996], bewertet $A^\star$ ein Element n des Suchraums nicht nur anhand der bereits bekannten Kosten der Wegstrecke vom Startelement s aus, sondern schätzt auch die Kosten bis zum gewünschten Zielelement g. Diese Schätzung bezeichnet man als Heuristik $h(n, g)$. Es gilt also für die Bewertung f eines Zustandes n : $f(n) = kosten\_vom\_start(n) + h(n, g)$. Zur Konstruktion einer Heuristik können Vorabinformationen über den Suchraum verwendet werden, um die Kosten von einem beliebigen Zustand zum Zielzustand abzuschätzen. Die Verwendung solcher Heuristiken kann die Suche im Vergleich zu den uninformierten Verfahren deutlich beschleunigen, da meist nur ein kleiner Teil der möglichen Zustände betrachtet werden muss.

Damit der $A^\star$-Algorithmus auch garantiert den kostengünstigsten Pfad zum Zielzustand findet, müssen drei Bedingungen erfüllt sein: Erstens darf jeder Zustand nur endlich viele Nachfolgezustände besitzen, zweitens dürfen die Kosten zwischen zwei Zuständen nicht negativ sein und drittens darf die Heuristik die wirklichen Kosten zum Zielelement nicht überschätzen. Die wahren Kosten müssen eine obere Schranke für die Heuristik bilden.

Die ersten beiden Voraussetzungen können bei vielen Suchproblemen schon im Vorhinein durch Rahmenbedingungen ausgeschlossen werden. Bei der Konstruktion einer Heuristik muss besonders auf den dritten Punkt geachtet werden.

Für jeden Zustand n muss gelten: $h(n, g) \leq h^*(n, g)$, wobei $h^*(n, g)$ die optimale Heuristik ist, die durch die wahren Kosten von n zu g gegeben ist. Erfüllt eine Heuristik dieses letzte Kriterium nennt man sie auch eine zulässige Heuristik.

Es gibt zwei Varianten, den $A^\star$-Algorithmus zu implementieren. Entweder mittels einer Priority Queue oder aber ohne zusätzliche Speicherung der noch zu betrachtenden Knoten. Bei einer geringen Anzahl vom Verknüpfungen zwischen den einzelnen Elementen des Graphen ist die Realisierung mittels Priority Queue die effizientere Variante. Da in unserem Fall nur eine geringe Menge an Nachbarschaftsbeziehungen existiert, haben wir den $A^\star$-Algorithmus mit einer Priority Queue realisiert. Die Grundidee des Suchverfahrens lässt sich dann wie folgt in wenigen Schritten darlegen:

1. Füge das Startelement s in die anfangs leere Priority Queue ein.
2. Extrahiere das Element e niedrigster Priorität aus der Queue.
3. Betrachte alle Nachbarelemente n von e :

   Befindet sich n noch nicht in der Priority Queue, so wird es mit der Priorität $f(n)$ in diese eingefügt. Ansonsten muss geprüft werden, ob der Weg zu n über e günstiger ist als der zuvor gefundene. Ist dies der Fall, werden Kosten und Priorität von n aktualisiert.
4. Solange die Priority Queue noch Elemente enthält und das Zielelement g noch nicht extrahiert wurde, beginne wieder mit Schritt 2.

Falls g nicht gefunden werden kann, existiert kein Weg vom Start- zum Zielelement.

Wenn beim Einfügen in die Queue bzw. beim Aktualisieren der Priorität der Elemente das günstigste Vorgängerelement gespeichert wird, lässt sich daraus der optimale Weg vom Start- zum Zielelement einfach rekonstruieren. Die genaue Darstellung von $A^\star$ findet sich in Algorithmus 1.

3.2. Die deterministische *Value Iteration*

Ein weiteres Verfahren zur Bestimmung der Wegkosten zu einem Zielelement innerhalb eines Graphen ist die so genannte *Value Iteration* [SUTTON und BARTO 1998]. Hierbei interessiert man sich im Vergleich zum $A^\star$-Algorithmus nicht für den optimalen Pfad von einem einzelnen Element zu einem Zielelement, sondern möchte den Weg samt Kosten von jedem beliebigen Element des Suchraums zu einem festen Zielelement bestimmen. Dafür muss der gesamte Suchraum ohne Ausnahme betrachtet werden. Daher wird keine Heuristik wie bei der informierten Suche verwendet.

Wir haben die deterministische Version des Verfahrens, dargestellt in Algorithmus 2, verwendet, bei dem der Nachfolgezustand jeder Aktion bekannt ist und es somit keine unsicheren Aktionen gibt. Dies bedeutet, dass eine gewünschte Aktion, wie beispielsweise das Verfolgen einer Kante zu einem Knoten, auch mit Sicherheit ausgeführt wird und nicht nur mit einer Wahrscheinlichkeit $p < 1.0$. Dadurch ist das Verfahren nahezu identisch zum Algorithmus von Dijkstra [OTTMANN und WIDMAYER 1996] zur Berechnung kürzester Wege in Distanzgraphen.

Algorithmus 1: $A^\star$

Eingabe: Startelement s, Zielelement g, Kostenfunktion c, Heuristik h.
Ausgabe: Der optimale Weg von s nach g oder keine Lösung.

```
initialisiere die Priority Queue PQ;
for all e ∈ Suchraum do
    kosten_vom_start[e] = ∞;
end for
kosten_vom_start[s] = 0;
vorgaenger[s] = s;
füge s mit Priorität h(s, g) in PQ ein;
while PQ nicht leer do
    extrahiere das Element e mit niedrigster Priorität aus PQ;
    if e = g then
        return optimaler Weg von s zu g gegeben durch vorgaenger[].
    end if
    for all n ∈ nachfolger(e) do
        if kosten_vom_start[n] > kosten_vom_start[e] + c(e, n) then
            kosten_vom_start[n] = kosten_vom_start[e] + c(e, n);
            vorgaenger[n] = e;
            if n bereits in PQ gespeichert then
                ersetzte die Priorität von n in PQ durch kosten_vom_start[n] + h(n, g);
            else
                füge n mit Priorität kosten_vom_start[n] + h(n, g) in PQ ein;
            end if
        end if
    end for
end while
return es existiert kein Pfad von s nach g.
```

Die deterministische *Value Iteration* lässt sich sehr ähnlich zur $A^\star$-Suche formulieren. Man beginnt allerdings hier mit dem Zielelement g und expandiert schrittweise den Suchraum. Dies geschieht ohne Verwendung einer Heuristik und das Verfahren wird genau so lange fortgesetzt bis die Priority Queue leer ist. Dadurch ist das Ergebnis der deterministischen *Value Iteration* identisch zum Ausführen der $A^\star$-Suche von *jedem* Element des Suchraums aus zum festen Zielelement g. Jedoch ist sie wesentlich performanter als die mehrfache Ausführung der $A^\star$-Suche. Abbildung 4.3 zeigt die Visualisierung einer entstehenden Kostenverteilung durch eine *Value Iteration*. Die roten Felder stellen Hindernisse dar, hell deutet auf günstige und dunkel auf hohe Kosten zum Ziel hin.

Nach dem Ausführen einer *Value Iteration* sind also die exakten Kosten und der günstigste Weg

Algorithmus 2: *Deterministische Value Iteration*

Eingabe: Zielelement g, Kostenfunktion c.
Ausgabe: Die Kosten und der optimale Weg von jedem Zustand aus zu g, sofern dieser existiert.

```
initialisiere die Priority Queue PQ;
for all e ∈ Suchraum do
    kosten_zum_ziel[e] = ∞;
end for
kosten_zum_ziel[g] = 0;
nachfolger[g] = g;
füge g mit Priorität 0 in PQ ein;
while PQ nicht leer do
    extrahiere das Element e mit niedrigster Priorität aus PQ;
    for all n ∈ vorgaenger(e) do
        if kosten_zum_ziel[n] > kosten_zum_ziel[e] + c(n, e) then
            kosten_zum_ziel[n] = kosten_zum_ziel[e] + c(n, e);
            nachfolger[n] = e;
            if n bereits in PQ gespeichert then
                ersetzte die Priorität von n in PQ durch kosten_zum_ziel[n];
            else
                füge n mit Priorität kosten_zum_ziel[n] in PQ ein;
            end if
        end if
    end for
end while
return Kostenverteilung kosten_zum_ziel[] und Nachfolger-Tabelle nachfolger[].
```

von jedem Element des Raumes aus zum Zielelement bekannt. Die aus dem Algorithmus resultierende Kostenverteilung *kosten_zum_ziel* stellt damit die optimale Heuristik h^* für die Suche mittels $A^\star$ dar. Diese Tatsache wird später verwendet, um aus einer einmalig berechneten *Value Iteration* eine gute Heuristik für die $A^\star$-Suche in einem sich leicht ändernden Suchraum, beispielsweise durch die Integration dynamischer Hindernisse in eine Umgebungskarte, zu konstruieren.

3.3. Die Repräsentation der Umgebung

Eine weitere zentrale Fragestellung lautet: „Wie modelliert man die Umgebung des Roboters?“ Ein verbreitetes Verfahren, das auch in unserer Arbeit Verwendung findet, ist die gitterförmige Diskretisierung der Umgebung in Verbindung mit Belegungswahrscheinlichkeiten [ELFES 1989, MORAVEC 1988]. Hierbei kommt eine zweidimensionale Karte zum Einsatz, die den

Raum in Quadrate unterteilt. Jedes dieser Quadrate trägt eine Wahrscheinlichkeit dafür, dass sich in dessen Inneren ein Hindernis befindet. Daraus ergibt sich eine Karte aus Belegungswahrscheinlichkeiten $P(occ_{x,y}) \in [0.0, 1.0]$, die die Hindernisse der realen Welt modelliert.

Der Roboter hält sich stets auf einem dieser Felder auf und kann von dort aus auf eines der Nachbarfelder gelangen. Abbildung 3.3 zeigt die möglichen Bewegungen einer Roboters, angenommen er befinde sich auf dem mittleren Feld.

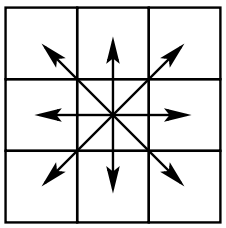

Abbildung 3.1: Die möglichen Bewegungen eines Roboters in der zweidimensionalen Ebene.

Typische Diskretisierungen liegen im Bereich von $10\,cm \times 10\,cm$. Da der Roboter aber im Allgemeinen größer ist als ein solches Feld ist, müssen zusätzliche Vorkehrungen zur Vermeidung von Kollisionen mit Hindernissen getroffen werden. Hierzu werden alle Objekte in der Karte um einen konstanten Wert vergrößert, der mindestens so groß sein muss wie der Radius des Roboters. Dadurch ist die Annahme, der Roboter befinde sich in genau einem Feld, legitim, wenn es um das Detektieren von möglichen Kollisionen geht. Es genügt dann genau das Feld zu überprüfen, in dem sich der Mittelpunkt des Roboters befindet [UDUPA 1977].

Für die bisherigen Annahmen wäre es ausreichend gewesen eine binäre Belegungskarte zu verwenden, da nur zwischen belegten und freien Feldern unterschieden wurde. Allerdings bietet eine kontinuierliche Wahrscheinlichkeitsverteilung Vorteile bei der Kollisionsvermeidung, die wir anhand der folgenden Problematik aufzeigen möchten.

3.3.1. Die *Faltung* der Umgebungskarte

Ein Roboter soll einerseits Hindernisse großzügig umfahren, andererseits aber auch enge Passagen, in denen nur wenig Platz für den Roboter vorhanden ist, durchfahren können. Um diese Vorgabe zu realisieren, werden die einzelnen Felder des Raumes mit Kosten belegt, die bei der Suche nach dem optimalen Pfad mittels $A^\star$ berücksichtigt werden. Es wird dabei nicht nur die Länge eines Weges bewertet, sondern zusätzlich auch die Kosten, die zum Überfahren der Felder auf diesem Weg aufgewendet werden müssen. Diese Kosten eines Feldes (x, y) sind abhängig von dessen Belegungswahrscheinlichkeit $P(occ_{x,y})$. Wenn nun aber $P(occ_{x,y})$ nicht wie bisher ausschließlich von einem möglichen Hindernis an der Stelle (x, y) abhängt, sondern auch von

möglichen Hindernissen auf benachbarten Feldern, lassen sich Felder in der Nähe von Hindernissen mit höheren Kosten belegen als Felder in deren Umgebung sich kein Hindernis befindet. Somit wird erreicht, dass das Durchfahren enger Passagen möglich ist, da kein kostengünstigerer Weg existiert, der Roboter aber gleichzeitig einen größeren Abstand zu Hindernissen aufrecht erhält, falls genügend Platz vorhanden ist.

Um eine solche Kostenverteilung zu erhalten wird eine so genannte *Faltung* der Karte vorgenommen. Je öfter sie durchgeführt wird, desto mehr ist $P(occ_{x,y})$ von weiter entfernten Feldern abhängig. Die *Faltung* wird spalten- und zeilenweise durchgeführt. Die zeilenweise Faltung ist definiert durch:

$$P(occ_{x_i,y}) = \frac{1}{4} \cdot P(occ_{x_{i-1},y}) + \frac{1}{2} \cdot P(occ_{x_i,y}) + \frac{1}{4} \cdot P(occ_{x_{i+1},y})$$

mit $i = 1 \ldots n-2$, sowie an Randfeldern (x_0, y) durch

$$P(occ_{x_0,y}) = \frac{2}{3} \cdot P(occ_{x_0,y}) + \frac{1}{3} \cdot P(occ_{x_1,y})$$

und den gegenüberliegenden Randfeldern (x_{n-1}, y) durch

$$P(occ_{x_{n-1},y}) = \frac{1}{3} \cdot P(occ_{x_{n-2},y}) + \frac{2}{3} \cdot P(occ_{x_{n-1},y})$$

Die spaltenweise *Faltung* erfolgt entsprechend über $P(occ_{x,y_i})$. Durch Anwendung der oben beschriebenen Gleichungen erhöhen sich die Belegungswahrscheinlichkeiten ursprünglich freier Felder und es verringern sich die der ursprünglich belegten Felder. Um Letzteres zu verhindern, wird nach einer mehrfachen *Faltung* das Maximum aus $P(occ_{x,y})$ vor und nach dieser Operation gebildet.

Abbildung 3.2 zeigt die Belegungswahrscheinlichkeiten einer Reihe von Feldern vor und nach zweifacher Anwendung der *Faltung*, Abbildung 3.3 zeigt diese nach zusätzlicher Bildung des Maximums.

3.4. Der vollständige Konfigurationsraum des Roboters

Bisher wurde die Repräsentation der Welt nur mittels einer zweidimensionalen Karte beschrieben. Damit lässt sich die Umgebung des Roboters zu einem Zeitpunkt beschreiben, allerdings ist es nicht ausreichend, den Zustand des Roboters in diesen beiden Dimensionen zu repräsentieren.

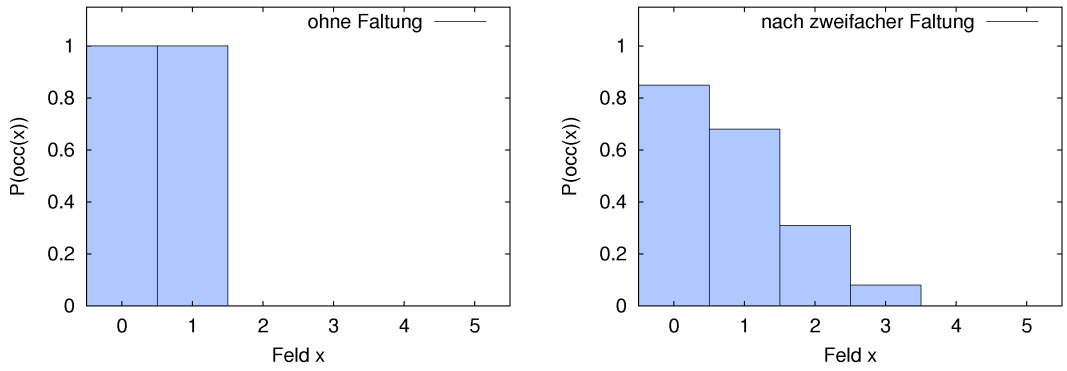

Abbildung 3.2: Die Belegungswahrscheinlichkeiten $P(occ_{x_i,y})$ ohne und nach zweifacher *Faltung*. Die Felder (x_0, y) und (x_1, y) seien belegt, die restlichen Felder frei.

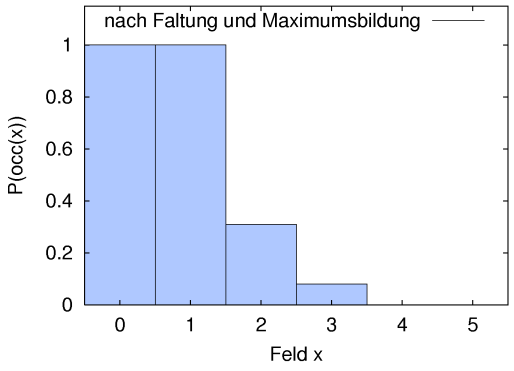

Abbildung 3.3: Die Belegungswahrscheinlichkeiten $P(occ_{x_i,y})$ nach zweifacher *Faltung* und Bildung des Maximums aus den Wahrscheinlichkeiten vor und nach der Operation. Wiederum sind (x_0, y) und (x_1, y) durch Hindernisse belegt, die restlichen Felder frei.

Der verwendete Robotertyp wird hier als rotationssymmetrisch angenommen. Als Referenzmodell dient der B21r Roboter von Real World Interfaces (RWI), der mit einem Synchro-Drive Fahrwerk ausgestattet ist (siehe Anhang A.1). Abbildung 3.4 zeigt die Anordnung der Räder eines 4-Rad-Synchro-Drive Antriebs. Dieser kann sich frei drehen und dadurch können so konstruierte Roboter beliebige Trajektorien abfahren.

Der Roboter besitzt neben der Position in der Karte noch zusätzlich eine Orientierung θ sowie einen zweidimensionalen Geschwindigkeitvektor, der seine Bewegung beschreibt. Dieser Geschwindigkeitvektor besteht einmal aus der Translationsgeschwindigkeit v, mit der sich der Roboter in Richtung der Orientierung θ bewegt, sowie eine Rotationsgeschwindigkeit ω, mit

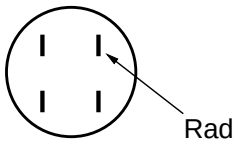

Abbildung 3.4: Die Anordnung der Räder eines Roboters mit 4-Rad-Synchro-Drive Antrieb.

der er seine Orientierung verändert. Die Rotations- und Translationsbeschleunigung kann, wie im Abschnitt 3.5.2 noch genauer erläutert wird, bei der Planung als phasenweise konstant angesehen werden und muss deshalb hier nicht berücksichtigt werden. Die Beschleunigungen des Roboters haben später bei der Planung einen Einfluss auf die möglichen Zustandsänderungen des Roboters innerhalb eines gewissen Zeitintervalls.

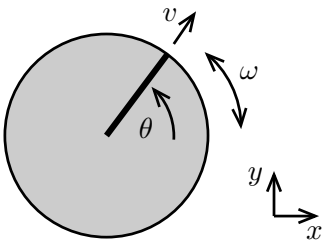

Abbildung 3.5: Die fünf Parameter $\langle x, y, \theta, v, \omega \rangle$ liefern eine vollständige Beschreibung für den Zustand eines Roboters.

Man interessiert sich bei der Konstruktion von Trajektorien für einen Roboter nicht unbedingt für den kürzesten Weg, denn meist möchte man den Roboter möglichst schnell von einem Ort zu einem anderen bewegen. Die Zeit, die der Roboter benötigt, um zu einem Zielpunkt zu gelangen, ist neben der Minimierung des Kollisionsrisikos die eigentlich interessante Größe. Nicht immer kann ein Roboter den kürzesten Weg auch am schnellsten abfahren. Man stelle sich eine Situation wie in Abbildung 3.6 vor, in der ein Roboter aus einem breiten Korridor durch eine enge Tür in einen Raum abbiegen soll. Beim Abfahren des kürzesten Weges müsste der Roboter nahe der Tür seine Fahrt deutlich verlangsamen, um sicher abbiegen zu können. Würde er allerdings einen kleinen Umweg, wie im rechten Bild gezeigt, in Kauf nehmen, könnte er auf diesem Weg mit einer höheren Geschwindigkeit fahren und würde den Zielpunkt früher erreichen.

Dieses Beispiel zeigt sehr deutlich, dass es nicht ausreicht, ausschließlich im $\langle x, y \rangle$-Raum den kürzesten Pfad zu suchen. Man muss den vollen fünfdimensionalen Konfigurationsraum in Betracht zu ziehen, um den optimalen Pfad bezüglich der Fahrzeit zu ermitteln.

Dadurch steigt die Komplexität des Suchproblems aber erheblich an und es ist mit den bis-

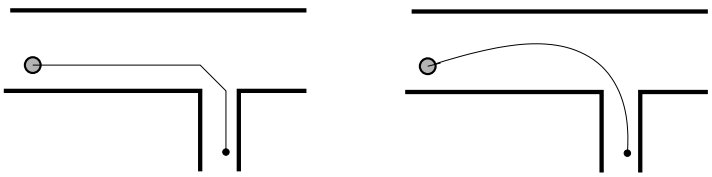

Abbildung 3.6: Das linke Bild zeigt den kürzesten Pfad zum Ziel. Das rechte Bild zeigt einen längeren Weg, der aber mit einer höheren Geschwindigkeit abfahren werden kann, so dass der Roboter hier schneller zum Ziel gelangt.

herigen Mitteln nicht mehr möglich einen optimalen Pfad mittels $A^\star$ im $\langle x, y, \theta, v, \omega \rangle$-Raum innerhalb eines kurzen Zeitintervalls zu finden. Deutlich wird dies, wenn man sich die Größe des Suchraums vor Augen führt: Man betrachte dazu eine typische Karte mit einer Größe von $10\,m \times 20\,m$ bei einer Diskretisierung von $10\,cm \times 10\,cm$, 32 möglichen Orientierungen (je $11.25°$), einer Geschwindigkeitsdiskretisierung von $10\,cm/s$ bzw. $11.25°/s$ und den maximalen Geschwindigkeiten von $70\,cm/s$ bzw. $45°/s$ im und entgegen dem Uhrzeigersinn. Die Größe des Suchraums dieses Problems ist gegeben durch das Produkt der möglichen Zustände der einzelnen Dimensionen. Dies sind dann $100 \times 200 \times 32 \times 8 \times 9 = 46\,080\,000$ Zustände.

An diesem Beispiel lässt sich deutlich erkennen, dass man effiziente Methoden benötigt, um den Suchraum zu explorieren und ihn auf geeignete Bereiche einzuschränken. Reale Roboter müssen *online* gesteuert werden, d.h. es existiert ein fest vorgegebenes, meist recht kurzes Zeitfenster, in dem die Berechnungen auszuführen sind. Spätestens dann wird eine geeignete Einschränkung des Suchraums zu einem der entscheidenden Punkte um eine qualitativ gute Lösung zu erhalten.

3.5. Die Bewegungsgleichungen

In diesem Abschnitt wird gezeigt, wie man die Bewegung eines Roboters mit Synchro-Drive Antrieb mathematisch beschreiben kann. Es wird davon ausgegangen, dass Translationsgeschwindigkeit v und Rotationsgeschwindigkeit ω unabhängig voneinander gesetzt werden können. Beginnend mit exakten Bewegungsgleichungen [FENG et al. 1994] kann man diese vereinfachen unter der Annahme, dass die Geschwindigkeiten als zeitweise konstant anzusehen sind. Dann besteht die Trajektorie eines Roboters aus mehreren aneinander anschließenden Kreissegmenten. Die Approximation der ursprünglichen Bewegungsgleichungen wurde von Fox et al. durchgeführt [FOX et al. 1997] .

3.5.1. Die allgemeinen Bewegungsgleichungen

Die Ausgangskonfiguration des Roboters zum Zeitpunkt t_0 sei gegeben. Nun gilt es diese Konfiguration zu einem späteren Zeitpunkt t_n zu bestimmen. Für $x(t_n)$ und $y(t_n)$ gilt dann in Abhängigkeit von $\theta(t)$ und $v(t)$:

$$x(t_n) = x(t_0) + \int_{t_0}^{t_n} v(t) \cdot \cos\theta(t)\, dt \tag{3.1}$$

$$y(t_n) = y(t_0) + \int_{t_0}^{t_n} v(t) \cdot \sin\theta(t)\, dt \tag{3.2}$$

Allerdings hängt in den Gleichungen (3.1) und (3.2) die Orientierung $\theta(t)$ von der Orientierung zum Zeitpunkt t_0, der Rotationsgeschwindigkeit $\omega(t_0)$ sowie der Rotationsbeschleunigung $\dot{\omega}(t)$ im Zeitintervall $[t_0, t_n]$ ab. Ähnliches gilt für die Geschwindigkeit $v(t)$, die von $v(t_0)$ und $\dot{v}(t)$ während des Zeitintervalls $[t_0, t_n]$ abhängt. Durch Substitution von $v(t)$ und $\omega(t)$ ergibt sich dadurch für die x-Koordinate:

$$\begin{aligned} x(t_n) = x(t_0) &+ \int_{t_0}^{t_n} \left(v(t_0) + \int_{t_0}^{t} \dot{v}(\hat{t})\, d\hat{t} \right) \\ &\cdot \cos\left(\theta(t_0) + \int_{t_0}^{t} \left(\omega(t_0) + \int_{t_0}^{\hat{t}} \dot{\omega}(\tilde{t})\, d\tilde{t} \right) d\hat{t} \right) dt \end{aligned} \tag{3.3}$$

und entsprechend für die y-Koordinate:

$$\begin{aligned} y(t_n) = y(t_0) &+ \int_{t_0}^{t_n} \left(v(t_0) + \int_{t_0}^{t} \dot{v}(\hat{t})\, d\hat{t} \right) \\ &\cdot \sin\left(\theta(t_0) + \int_{t_0}^{t} \left(\omega(t_0) + \int_{t_0}^{\hat{t}} \dot{\omega}(\tilde{t})\, d\tilde{t} \right) d\hat{t} \right) dt \end{aligned} \tag{3.4}$$

Die Gleichungen (3.3) und (3.4) hängen nun nur noch vom Anfangszustand des Roboters zum Zeitpunkt t_0 sowie den Beschleunigungen ab. Durch die digitale Hardware der Roboter ist das System allerdings Einschränkungen unterworfen: Bei den meisten Robotern können nur diskrete Beschleunigungen gesetzt werden und dies auch nur in bestimmten Zeitintervallen. Sei n nun die Anzahl an Kommandos, die nacheinander an den Roboter gesendet werden. Dann kann davon ausgegangen werden, dass $\dot{v}(t)$ und $\dot{\omega}(t)$ zwischen zwei Kommandos, also für $t \in [t_i, t_{i+1}]$ mit $i = 0 \ldots n-1$, konstant sind. Sei nun $\Delta_t^i = t - t_i$. Dann können, unter der Annahme zeitweise konstanter Beschleunigungen, die Bewegungsgleichungen auf die folgende Form vereinfacht werden:

$$x(t_n) = x(t_0) + \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} \left(v(t_i) + \dot{v}_i \cdot \Delta_t^i\right) \cdot \cos\left(\theta(t_i) + \omega(t_i) \cdot \Delta_t^i + \frac{1}{2}\dot{\omega}_i \cdot (\Delta_t^i)^2\right) dt \tag{3.5}$$

sowie

$$y(t_n) = y(t_0) + \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} \left(v(t_i) + \dot{v}_i \cdot \Delta_t^i\right) \cdot \sin\left(\theta(t_i) + \omega(t_i) \cdot \Delta_t^i + \frac{1}{2}\dot{\omega}_i \cdot (\Delta_t^i)^2\right) dt \tag{3.6}$$

3.5.2. Die Approximation der Bewegungsgleichungen

Die Gleichungen (3.5) und (3.6) beschreiben nun die Bewegung eines Roboters. Allerdings sind die Gleichungen recht unhandlich und die Bewegungen, die sie beschreiben, sind im Allgemeinen komplex. Auch Schnittpunktbestimmungen und andere geometrische Operation lassen sich nur mit viel Rechenaufwand bestimmen. Aus diesen Gründen wird eine weitere Approximation der Bewegungsgleichungen vorgenommen. Ähnlich wie bei den Beschleunigungen im vorherigen Abschnitt nimmt man nun auch die Geschwindigkeiten als stückweise konstant an, d.h. $v(t)$ und $\omega(t)$ seien konstant für $t \in [t_i, t_{i+1}]$.

Die später aus dieser Vereinfachung resultierenden Bewegungsgleichungen (3.7) und (3.8) konvergieren gegen die ursprüngliche Darstellung in Gleichung (3.5) bzw. (3.6) falls die Zeitintervalle $[t_i, t_{i+1}]$ gegen Null gehen.

Wenn die Zeitintervalle genügend klein gewählt werden, kann der Term $v(t_i) + \dot{v}_i \cdot \Delta_t^i$ aus Gleichung (3.5) bzw. (3.6) durch eine konstante Geschwindigkeit $v_i \in [v(t_i), v(t_{i+1})]$ approximiert werden. Genauso kann $\theta(t_i) + \omega(t_i) + \frac{1}{2}\dot{\omega}_i(\Delta_t^i)^2$ durch $\theta(t_i) + \omega_i \cdot \Delta_t^i$ mit $\omega_i \in [\omega(t_i), \omega(t_{i+1})]$ ersetzt werden.

Die Gleichungen vereinfachen sich dann zu:

$$x(t_n) = x(t_0) + \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v_i \cdot \cos\left(\theta(t_i) + \omega_i \cdot (\hat{t} - t_i)\right) d\hat{t}$$

und

$$y(t_n) = y(t_0) + \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} v_i \cdot \sin\left(\theta(t_i) + \omega_i \cdot (\hat{t} - t_i)\right) d\hat{t}$$

was dann wiederum auf die endgültigen Bewegungsgleichungen

$$x(t_n) \quad = \quad x(t_0) + \sum_{i=0}^{n-1} (F_x^i(t_{i+1})) \tag{3.7}$$

mit

$$F_x^i(t) \quad = \quad \begin{cases} -\frac{v_i}{\omega_i}(\sin\theta(t_i) - \sin(\theta(t_i) + \omega_i \cdot (t - t_i))), \, \omega_i \neq 0 \\ v_i \cos(\theta(t_i)) \cdot t, \, \omega_i = 0 \end{cases}$$

und entsprechend für $y(t_n)$:

$$y(t_n) \quad = \quad y(t_0) + \sum_{i=0}^{n-1} (F_y^i(t_{i+1})) \tag{3.8}$$

mit

$$F_y^i(t) \quad = \quad \begin{cases} \frac{v_i}{\omega_i}(\cos\theta(t_i) - \cos(\theta(t_i) + \omega_i \cdot (t - t_i))), \, \omega_i \neq 0 \\ v_i \sin(\theta(t_i)) \cdot t, \, \omega_i = 0 \end{cases}$$

führt.

Die Bewegungen, die durch diese Gleichungen beschrieben werden, sind aneinander anschließende Geradenstücke und Kreissegmente. Falls $\omega_i = 0$ ist, bewegt sich der Roboter entlang einer Geraden. Ansonsten beschreiben die Gleichungen eine kreisförmige Bewegung. Dies wird deutlich, wenn man

$$\begin{aligned} M_x^i &= -\frac{v_i}{\omega_i} \cdot \sin\theta(t_i) \\ M_y^i &= \frac{v_i}{\omega_i} \cdot \cos\theta(t_i) \end{aligned}$$

betrachtet. Denn es gilt die folgende Kreisgleichung:

$$\left(F_x^i - M_x^i\right)^2 + \left(F_y^i - M_y^i\right)^2 \quad = \quad \left(\frac{v_i}{\omega_i}\right)^2$$

Das i-te Stück der Trajektorie des Roboters ist also im Falle von $\omega_i \neq 0$ ein Kreissegment um den Punkt (M_x^i, M_y^i) mit dem Radius $r_i = \frac{v_i}{\omega_i}$. In [FOX et al. 1997] wird des weiteren gezeigt, dass der Approximationsfehler linear in der Größe der Zeitintervalle und der wirklichen Geschwindigkeitsänderung ist.

Zusammenfassend gilt also, dass man durch die Annahme, die Geschwindigkeiten des Roboters seien stückweise konstant, die Bewegungsgleichungen deutlich vereinfachen kann. Die neuen Gleichungen beschreiben Geraden bzw. Kreissegmente und für kleine Zeitintervalle konvergieren sie gegen die nicht approximierte Formulierung.

4. Das Verfahren zur zielgerichteten Kollisionsvermeidung

Dieses Kapitel beschäftigt sich mit dem Problem der Fahrtplanung mit integrierter Kollisionsvermeidung und beschreibt somit die zentralen Punkte unserer Arbeit. Der Roboter befindet sich an einer bekannten Position und soll sich zu einem Zielpunkt hinbewegen, dies in möglichst kurzer Zeit, bei möglichst geringem Kollisionsrisiko. Während seiner Fahrt muss er Hindernisse umfahren, sowohl bekannte als auch erst zur Laufzeit detektierte.

4.1. Rahmenbedingungen

Die von uns entwickelte Software muss vorgegebene Rahmenbedingungen einhalten und verlässt sich auf gewisse Informationen über den Zustand der Welt. Diese möchten wir vorweg kurz skizzieren. Ein Übersicht über die Architektur des Kontrollsystems, das für die Informationen über den Weltzustand verantwortlich ist, wird in Kapitel 5 gegeben.

4.1.1. Das Zeitfenster

Mit Hilfe des eingesetzten Kontrollsystems ist es möglich, alle $250\,ms$ ein neues Kommando an das Fahrwerk des Roboters zu senden. An dieses Zeitfenster war auch die zuvor eingesetzte Kollisionsvermeidung gebunden und es hat sich in diversen Experimenten im Laufe der letzten Jahre als ausreichend kurz erwiesen. Nach besagter Zeit liegen auch neue Messungen der Sensoren vor, wie beispielsweise vom laserbasierten Entfernungsmesser (siehe Anhang A.3). Daher sollte innerhalb dieses Zeitfensters das nächste Fahrkommando für den Roboter ermittelt werden. Es stehen also für alle Tätigkeiten, die regelmäßig während der Fahrt durchgeführt werden müssen, nur $250\,ms$ zur Verfügung. Für die Einhaltung dieses Zeitintervalls hat die Software selbst Sorge zu tragen.

4.1.2. Der Weltzustand

Dem System liegt eine Karte der Umgebung vor, in der statische Hindernisse eingetragen sind. Zusätzliche Hindernisse muss der Roboter anhand seiner Sensoren selbst erkennen. Die Umgebungskarte liefert auch die Basis für ein einheitliches Koordinatensystem, in dem alle Be-

rechnungen ausgeführt werden. Es ist davon auszugehen, dass der Zustand $\langle x, y, \theta, v, \omega \rangle$ des Roboters nach jedem Zeitintervall innerhalb dieser Karte bekannt ist.

4.2. Die Planung des Pfades

Hier wird nun der eigentliche Kernpunkt, die Planung der Trajektorie des Roboters, erklärt. Zum besseren Verständnis gehen wir vorerst davon aus, dass genügend Rechenzeit zur Verfügung steht und ein Weg zum Zielpunkt des Roboters existiert. Dann kann der Prozess der Pfadplanung, wie in Abbildung 4.1 gezeigt, in die folgenden Schritte unterteilt werden:

4.2.1. Die Schätzung der nächsten Position

Zu Anfang wird die aktuelle Position des Roboters mit Hilfe eines Lokalisierungssystems wie beispielsweise *Localize*, welches in Abschnitt 5.1.3 genauer beschrieben wird, ermittelt. Dazu wird die letzte Messung der Odometriedaten betrachtet. Je nach Alter der Messung, das sich meist im Bereich von ca. $5\,ms$ bis $100\,ms$ bewegt, wird die jetzige Position des Roboters extrapoliert. Der Geschwindigkeitsvektor ist durch die Odometriesensoren bereits bekannt. Diese Positionsdaten werden dann mittels der Korrekturparameter von *Localize* in das Koordinatensystem der Umgebungskarte transformiert. Somit ist die aktuelle Position des Roboters bekannt.

Allerdings werden in den nächsten Schritten Berechnungen durchgeführt, die $250\,ms$ andauern. Erst dann wird das neue Kommando an den Roboter gesendet. Daher ist es nötig, die Position des Roboters nach Ende des Zeitintervalls zu schätzen und von diesem Ort aus zu planen. Die Schätzung der Position geschieht über die Bewegungsgleichungen aus Abschnitt 3.5. Um diesen Vorgang schneller ausführen zu können, greifen wir dabei auf eine Tabelle zurück, in der die Positionen gespeichert sind, an die der Roboter gelangt, wenn er sich mit seiner aktuellen Geschwindigkeit ein Zeitintervall lang bewegt.

4.2.2. Das Aktualisieren der Umgebungskarte

Im Normalfall besitzt man eine Karte der Umgebung, die einige statische Hindernisse enthält. Diese dient einerseits zur Lokalisierung des Roboters [BURGARD et al. 1996], andererseits können in eine solche Karte auch Hindernisse eingetragen werden, die der Roboter anhand seiner Sensoren nicht detektieren kann. Dies können beispielsweise Glasscheiben sein, falls der Roboter ausschließlich mit laserbasierten Entfernungsmessern[1] ausgestattet ist. Diese statischen Hindernisse werden während des gesamten Verfahrens nicht mehr aus der Karte entfernt. Die Belegungswahrscheinlichkeit eines Feldes kann daher nicht unter diesen Ausgangswert fallen.

[1] Laserbasierte Entfernungsmesser erkennen Glasscheiben im Allgemeinen nicht, da ein Großteil der Lichtwellen durch das Glas nicht reflektiert wird.

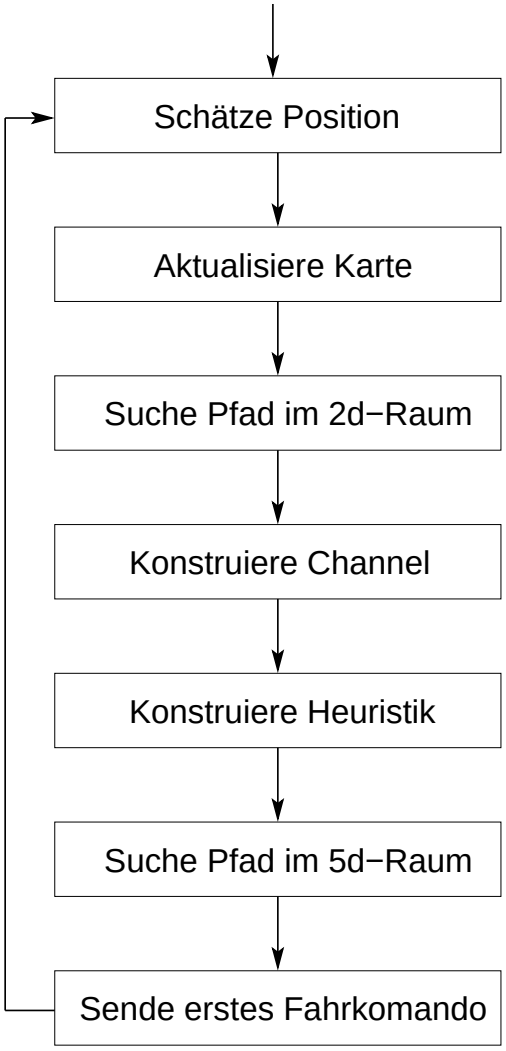

Abbildung 4.1: Dieses Flussdiagramm zeigt die grundlegenden Schritte unseres Algorithmus. Es wird jedoch vorerst davon ausgegangen, dass ein Pfad zum Zielpunkt existiert und genügend Rechenzeit für alle Operationen zur Verfügung steht.

Die Verwendung einer statischen Karte in Verbindung mit Belegungswahrscheinlichkeiten bringt noch einen weiteren Vorteil für eventuelle Erweiterungen. So könnten beispielsweise bevorzugte Trajektorien von Menschen in einer Umgebung ermittelt werden [BENNEWITZ et al. 2002] und diese Bereiche in der Karte dann mit einer minimal höheren Belegungswahrscheinlichkeit versehen werden. Dadurch würde der Roboter tendenziell Trajektorien bevorzugen, die diese Bereiche meiden, ohne dass man einen Eingriff in die Software zur Pfadplanung vornehmen müsste.

Das ausschließliche Verwenden einer statischen Karte ist selbstverständlich nicht ausreichend für eine Kollisionsvermeidung in dynamischen, d.h. sich ändernden, Umgebungen. Diese Veränderungen können durch Personen hervorgerufen werden, die sich im Umfeld des Roboters bewegen, oder auch durch statische Hindernisse wie geschlossene Türen oder ähnliches, die nicht in der ursprünglichen Karte eingetragen wurden.

Um solche Hindernisse in die Planung mit einzubeziehen, müssen die Daten der Sensoren des Roboters ausgewertet werden. Wir haben hierzu in unserer Arbeit nur Daten von Lasersensoren der Firma SICK verwendet, da diese kaum Fehlmessungen liefern und sehr präzise arbeiten. Theoretisch kann auch jede andere Art von Sensoren zusätzlich integriert werden. Das ausschließliche Verwenden von Lasersensoren hat sich in unseren Versuchen aber als ausreichend erwiesen. Diese Lasersensoren liefern eine Menge von Abständen, in denen sich Hindernisse relativ zum Sensor befinden. Die von uns verwendeten Sensoren besitzen eine Auflösung von $1/2^\circ$ sowie 1° bei einem Öffnungswinkel von 180°. Dies ergibt 360 bzw. 180 Messwerte, die durch ein externes Softwaremodul ca. alle $250\,ms$ zur Verfügung gestellt werden. Aus der aktuellen Position des Roboters (x_r, y_r) und dessen Orientierung θ_r kann damit eine Menge von Hindernispunkten (x_{obs_i}, y_{obs_i}) wie folgt bestimmt werden:

$$\begin{pmatrix} x_{obs_i} \\ y_{obs_i} \end{pmatrix} = \begin{pmatrix} x_r \\ y_r \end{pmatrix} + dist_i \cdot \begin{pmatrix} cos(\theta_r + \alpha_i) \\ sin(\theta_r + \alpha_i) \end{pmatrix}$$

mit α_i Winkel des Laserstrahls i von θ_r aus gesehen und $dist_i$ dem entsprechend gemessenen Abstand.

An den Stellen (x_{obs_i}, y_{obs_i}) und den benachbarten Feldern innerhalb eines konstanten Radius wird daraufhin die Belegungswahrscheinlichkeit in der Umgebungskarte auf 1.0 gesetzt. Dies dient, genau wie bei der Vorverarbeitung der statischen Umgebungkarte in Abschnitt 3.3.1, zur Vermeidung von Kollisionen aufgrund der Größe des Roboters. Zusätzlich wird um das so erweiterte Hindernis noch ein zusätzlicher Rand mit hoher Belegungswahrscheinlichkeit $P(occ_{x,y})$ gelegt. Dieser Rand ermöglicht es dem Roboter sich dem Hindernis anzunähern, belegt diese Aktion aber mit hohen Kosten, ähnlich dem Ergebnis einer *Faltung*.

Besonders bei sich bewegenden Hindernissen wie Personen ist es extrem wichtig, die Belegungswahrscheinlichkeit wieder zu reduzieren, falls die Sensoren kein Hindernis mehr detektieren können. Ansonsten würde schon nach sehr kurzer Zeit die Karte durch in Wirklichkeit nicht mehr existierende Hindernisse unbrauchbar werden. In unserer Arbeit verwenden wir zwei

verschiedene Kriterien zum Reduzieren der Belegungswahrscheinlichkeit eines Feldes. Erstens besitzt jedes Hindernis eine begrenzte Lebensdauer und wird nach Ablauf dieser Frist gelöscht, falls es nicht wieder detektiert wurde. Dazu erhält es einen Zeitstempel und wird beim Einfügen einmal in die Karte aufgenommen und zusätzlich in einem Ringpuffer gespeichert. Somit werden in jedem Schritt auch die Hindernisse einer alten Messung aus dem Puffer entfernt. Dabei wird der Zeitstempel aus dem Ringpuffer mit dem aus der Karte verglichen. Sind diese identisch, wird das Hindernis aus der Karte entfernt, ansonsten wurde es durch eine neuere Sensormessung als noch aktuell bestätigt und bleibt erhalten.

Das zweite Kriterium berücksichtigt die durch den Laserstrahl überstrichenen Felder. Hierzu wird der Strahl vom Hindernis ausgehend zum Roboter zurückverfolgt und jedes überstrichene Feld wird auf seine ursprüngliche Belegungswahrscheinlichkeit zurückgesetzt. Zusätzlich markiert man die bereits überstrichenen Felder in einem Booleschen Array. Überstreicht einer der Strahlen ein Feld, das bereits durch einen anderen Strahl markiert wurde, muss dieser Strahl nicht weiter verfolgt werden, da er sich von nun an nur noch in Feldern bewegt, die bereits aktualisiert wurden. Dies funktioniert natürlich nur, wenn der Strahl vom Hindernis ausgehend und nicht vom Roboter ausgehend verfolgt wird. Diese Technik reduziert den Aufwand zur Bestimmung der zu löschenden Hindernisse in den meisten Fällen deutlich, da benachbarte Strahlen aufgrund der hohen Auflösung des Laserscanners häufig schon im gleichen Feld enden oder in großen Teilen die gleichen Felder überstreichen. Abbildung 4.2 illustriert diesen Sachverhalt.

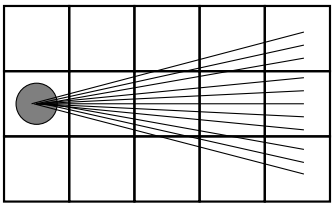

Abbildung 4.2: Benachbarte Laserstrahlen überstreichen oft identische Felder, wodurch das Löschen dynamischer Hindernisse durch das Protokollieren besuchter Felder performant realisiert werden kann.

Somit wird die an sich statische Karte nun um dynamische Hindernisse erweitert. Wir möchten hier nochmals erwähnen, dass $P(occ_{x,y})$ an den Stellen der zu löschenden Hindernisse wieder auf den zuvor erwähnten Ausgangswert zurückgesetzt wird, nicht aber auf 0.0. Ansonsten würde man statische Hindernisse wie nicht detektierbare Glasscheiben unberechtigterweise aus der Karte entfernen. Außerdem ergeben sich dadurch Vorteile bei der Konstruktion einer Heuristik für die $A^\star$-Suche, wie im folgenden Abschnitt genauer erläutert wird.

4.2.3. Die Konstruktion der Heuristik für die $A^\star$-Suche im $\langle x, y\rangle$-Raum

Eine Heuristik, die die Kosten eines Feldes zu einem Zielpunkt gut abschätzen kann, erhöht die Ausführungsgeschwindigkeit der $A^\star$-Suche erheblich. Bei deren Konstruktion kann nun die im letzten Absatz angesprochene Eigenschaft des Suchraums ausgenutzt werden. Danach besitzen die Kosten eines jeden Feldes einen Ausgangswert, der nie unterschritten wird.

Sobald der Roboter einen neuen Zielpunkt anfahren soll, wird eine *Value Iteration* auf der statischen, zweidimensionalen Umgebungskarte, die keine dynamischen Hindernisse enthält, ausgeführt. Die aus Abschnitt 3.2 bekannte Kostenverteilung $kosten\_zum\_ziel$ wird gespeichert und als Heuristik verwendet. Da die Kosten eines Feldes beim Integrieren und Entfernen dynamischer Hindernisse nie unter den ursprünglichen Anfangswert fallen, sondern maximal anwachsen können, ist diese Heuristik zulässig, d.h. die wirklichen Kosten werden nie überschätzt. Die *Value Iteration* muss nicht wie die anderen Schritte dieses Kapitels in jedem Durchgang neu ausgeführt werden. Es reicht aus sie nur nach dem Setzen eines neuen Zielpunktes auszuführen.

Würde man die *Value Iteration* beim Setzen eines neuen Zielpunktes auf der aktuellen Karte inklusive dynamischer Hindernisse ausführen, erhielte man nach dem nächsten Löschen eines Hindernisses eine nicht mehr zulässige Heuristik, da diese die Kosten dieses Feldes zum Zielpunkt überschätzen würde. Daher ist die $kosten\_zum\_ziel$-Funktion die beste Heuristik, die man in in der zweidimensionalen Umgebung anhand einer einmaligen Berechnung konstruieren kann.

Die Suche nach dem optimalen Weg in einer dynamischen Umgebung muss sehr oft ausgeführt werden. Das bedeutet, dass die Ausführung von $A^\star$ zeitkritisch ist. Sie darf ein bestimmtes Zeitintervall nicht überschreiten. Aus diesem Grund lohnt sich das einmalige Ausführen der *Value Iteration* zum Finden einer guten Heuristik. Wie noch in Abschnitt 6.1 gezeigt wird, macht die Verwendung der $kosten\_zum\_ziel$-Funktion die $A^\star$-Suche schneller als eine Heuristik basierend auf der euklidischen Distanz, da die Kosten eines Feldes besser abgeschätzt werden können. Abbildung 4.3 zeigt die Visualisierung einer entstehenden Kostenverteilung.

4.2.4. Die $A^\star$-Suche im $\langle x, y\rangle$-Raum

Da die Ausführung einer $A^\star$-Suche im gesamten $\langle x, y, \theta, v, \omega\rangle$-Raum zu zeitaufwendig wäre, wird zuerst der Pfad des Roboters in der $\langle x, y\rangle$-Ebene geplant. Dadurch erhält man den kürzesten Pfad zwischen Roboterposition und Zielpunkt oder stellt fest, dass ein solcher Weg überhaupt nicht existiert, ohne den gesamten fünfdimensionalen Konfigurationsraum durchsuchen zu müssen. Als Heuristik für die $A^\star$-Suche wird die im vorherigen Abschnitt besprochene Kostenverteilung $kosten\_zum\_ziel$ verwendet.

Der resultierende Pfad im $\langle x, y\rangle$-Raum wird in den folgenden Schritten dazu verwendet den vollen Konfigurationsraum für die Planung auf einen günstigen Teilbereich einzuschränken.

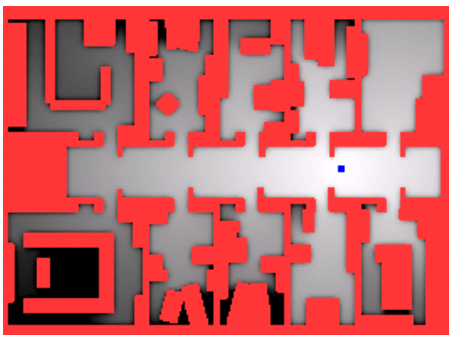

Abbildung 4.3: Die Kostenverteilung $kosten\_zum\_ziel$ einer *Value Iteration*. Die roten Felder stellen Hindernisse dar, hell deutet auf günstige und dunkel auf hohe Kosten zum Ziel hin.

4.2.5. Die Einschränkung des vollen Konfigurationsraums

Wie soeben angedeutet, wird nun der $\langle x, y, \theta, v, \omega \rangle$-Konfigurationsraum auf einen kleineren Raumbereich eingeschränkt. Wenn man sich die Pfade ohne Einschränkung des Raumes offline berechnen lässt, stellt man fest, dass die überfahrenen Felder meist dicht am Pfad der zweidimensionalen Suche liegen. Daher erscheint es sinnvoll zur Einschränkung des Raumbereichs auf den im vorherigen Abschnitt berechneten Pfad zurückzugreifen. Hierzu wird ein so genannter *Channel* um einen Teil dieses Pfades gelegt, beginnend mit der geschätzten Roboterposition. Ein *Channel* umfasst alle Punkte der Umgebung, die weniger als eine Konstante vom zuvor berechneten Pfad entfernt liegen. Er wird schrittweise entlang dieses Pfades gelegt, bis er eine bestimmte Länge erreicht hat. Das zuletzt aufgenommene Wegelement dient nun als Zwischenzielpunkt für die folgende $A^\star$-Suche innerhalb dieses Raumbereichs.

Die zur Verfügung stehende Rechenleistung bestimmt die Länge und Breite des *Channel*. Beide Parameter können durch das System vergrößert oder verkleinert werden, wodurch sich das Verfahren an die vom System zur Verfügung gestellte Leistung adaptiert.

Falls der Zwischenzielpunkt identisch mit dem Endzielpunkt ist, kann die Orientierung θ des Roboters beliebig sein, Translations- und Rotationsgeschwindigkeit müssen aber am Zielpunkt identisch 0 sein. Handelt es sich dagegen um einen echten Zwischenzielpunkt, darf v beliebige Werte annehmen. Die Orientierung θ dagegen wird auf die Richtung, gegeben durch den optimalen Pfad im $\langle x, y \rangle$-Raum am Zwischenzielpunkt, gesetzt und ω gleich Null gewählt damit der Roboter tendenziell dem $\langle x, y \rangle$-Pfad folgt. Im diesem Fall kann aber das Finden exakt dieses Elements im fünfdimensionalen Suchraum recht zeitaufwendig werden. Daher ist es für die Performanz der Suche meist günstiger kleine Werte der Rotationsgeschwindigkeit ω zuzulassen sowie eine geringe Abweichung in der Orientierung θ.

4.2.6. Die Heuristik für die $A^\star$-Suche im $\langle x, y, \theta, v, \omega \rangle$-Raum

Die Wahl einer guten Heuristik für die Suche mittels $A^\star$ im fünfdimensionalen Konfigurationsraum ist sehr wichtig, damit nur möglichst kleine Teile des extrem großen Suchraums exploriert werden müssen. Zwei grundlegende Kenngrößen, die bei der Suche berücksichtigt werden, sind einmal die Kosten der zu überfahrenden Felder sowie die Länge der zu fahrenden Strecke. Da man sich aber für die Fahrzeit interessiert, muss die Streckenlänge mit der maximal erreichbaren Geschwindigkeit des Roboters kombiniert werden. Um diese Größen möglichst genau zu ermitteln, wird eine *Value Iteration* im $\langle x, y \rangle$-Raum innerhalb des *Channel* inklusive der dynamischen Hindernisse durchgeführt. Dabei werden Streckenlänge und Kosten des günstigsten Weges gespeichert. Für die $A^\star$-Suche zum Zwischenzielpunkt in zwei Dimensionen ergäbe sich hieraus die optimale Heuristik. Im fünfdimensionalen Raum ist dies allerdings nicht mehr der Fall, sie bietet aber einen guten Ausgangspunkt für das weitere Vorgehen.

Angenommen, der Roboter fährt mit der Geschwindigkeit $v_{aktuell}$ und beschleunigt bis er v_{max} erreicht hat. Basierend auf der Gleichung zur Berechnung der gefahrenen Wegstrecke bei gleichmäßiger Beschleunigung a legt der Roboter in der Zeit t die Strecke s zurück, gegeben durch

$$s = v_{aktuell} \cdot t + \frac{1}{2} \cdot a \cdot (min(t, t_a))^2 + (v_{max} - v_{aktuell}) \cdot max(0, t - t_a)$$

wobei t_a die Zeit ist, in der der Roboter von $v_{aktuell}$ auf v_{max} beschleunigt, gegeben durch

$$t_a = \frac{||v_{max} - v_{aktuell}||}{a}, a > 0.$$

Speichert man diese Werte einmalig in einer Tabelle ab, kann die minimale Fahrzeit für eine Strecke schnell ermittelt werden. Ähnlich verhält es sich mit der Rotation. Hier benötigt der Roboter auch eine gewisse Zeit, um sich aus seiner aktuellen Orientierung zu der gewünschten Orientierung des Zielpunktes zu drehen. Das Maximum dieser beiden Zeiten ist damit kleiner oder gleich der minimalen Zeit, die der Roboter in Wirklichkeit benötigt, um die Trajektorie zum Zwischenzielpunkt abzufahren.

Außerdem werden im Vorhinein die relativen Koordinaten von Feldern, die der Roboter beim Ausführen eines Fahrkommandos überstreicht, tabelliert, entsprechend der Bewegungsgleichungen aus Abschnitt 3.5. Mit Hilfe dieser Tabelle und der Information über die Feldkosten können dann wiederum die Kosten einer Trajektorie gut und effizient abgeschätzt werden. Wichtig ist dabei jedoch, die Kosten durch die Heuristik nicht zu überschätzen und die Art, wie die Feldkosten berücksichtigt werden, hat einen Einfluss darauf.

Zur Bestimmung der Kosten wird die zu fahrende Trajektorie anhand der Diskretisierung der Umgebung in Wegstücke unterteilt. Man erhält damit eine Menge von Teilstücken des Pfades, die jeweils innerhalb eines Feldes liegen. Die intuitive Variante, die Kosten eines Feldes mit der Länge des zugehörigen Teilstücks zu gewichten, führt aber unter Umständen zu Problemen bei der Kostenabschätzung. Bei der Abschätzung innerhalb der Suche geht man immer davon aus,

dass der Roboter sich von der Mitte des ersten zur Mitte des zweiten Feldes bewegt. In Wirklichkeit bewegt sich der Roboter unter Umständen aber nur geringfügig über die Grenze zwischen dem ersten und dem zweiten Feld, so dass damit die Kosten der Heuristik über den realen Kosten liegen. Abbildung 4.4 zeigt genau diesen Sachverhalt: Links sieht man die Annahme der zweidimensionalen Planung und damit auch der Heuristik, dass sich der Roboter zur Mitte des Feldes hin bewegt, rechts dagegen eine mögliche reale Bewegung zum Rand des Feldes hin. Die Kosten dieser Aktion liegen in der linken Variante über denen der rechten, wodurch es zu besagter Überschätzung durch die Heuristik kommen kann.

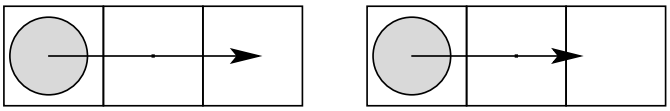

Abbildung 4.4: Die Fahrt eines Roboters in ein benachbartes Feld. Links fährt der Roboter zur Mitte des Feldes, so wie auch die Heuristik basierend auf den zweidimensionalen Informationen die Kosten abschätzt. Rechts dagegen sieht man eine Bewegung zum Rand des Feldes, wodurch die Kosten niedriger ausfallen und es zur Kostenüberschätzung durch die Heuristik kommt.

Daher bewerten wir in unserem Verfahren jedes Feld, das der Roboter überstreicht mit einer Kostengewichtung von Eins, unabhängig davon, wie lange sich der Roboter auf diesem Feld aufhält. Dadurch ist gewährleistet, dass die Kosten durch die Heuristik nicht überschätzt werden.

Uns fiel in Versuchen auf, dass die Suche im fünfdimensionalen Raum deutlich länger dauert, wenn die Planung zu genau einem Zielzustand durchgeführt wird, d.h. der Roboter exakt mit der gewünschten Geschwindigkeit und Orientierung an einem speziellen Ort ankommen muss. Meist kann die Suche deutlich beschleunigt werden, wenn man kleine Abweichungen vom Zielzustand zulässt, d.h. beispielsweise einen Abstand von $20\,cm$ zum Zielpunkt oder eine Abweichung in der Orientierung von 10°. Expandiert man einen solchen Zustand bei der $A^\star$-Suche, kann diese abgebrochen werden und es muss nicht weiter nach dem exakten Zielzustand gesucht werden. Diese Optimierung führt aber zu einem Problem: Ein Zustand, der durch diese Änderungen als Zielzustand zu betrachten ist, wird durch die Heuristik mit Kosten größer Null belegt, was einer Überschätzung durch die Heuristik entspricht. Diese ist damit nicht mehr zulässig. Um dieses Problem zu umgehen, kann aber die Heuristik für diese unscharfe Art der Suche angepasst werden. Hierzu belegt man die hindernisfreien Felder c_i in entsprechendem Radius um den Zielpunkt während der *Value Iteration* innerhalb des *Channel* mit den Kosten $kosten\_zum\_ziel[c_i] = 0$. Dadurch liegen die geschätzten Kosten nicht mehr über den realen Kosten und die Heuristik bleibt für die $A^\star$-Suche zulässig.

4.2.7. Die $A^\star$-Suche im $\langle x, y, \theta, v, \omega \rangle$-Raum

In diesem Schritt wird nun die $A^\star$-Suche zum Zwischenzielpunkt im $\langle x, y, \theta, v, \omega \rangle$-Raum innerhalb des *Channel* aus Abschnitt 4.2.5 ausgeführt. Als Ergebnis erhält man eine Sequenz von Geschwindigkeitvektoren (v, ω), die den Roboter auf dem optimalen Pfad von seiner aktuellen Position aus zum Zwischenzielpunkt bewegen.

Da der $A^\star$-Algorithmus einen diskreten Suchgraphen benötigt, dürfen für alle fünf Dimensionen auch nur diskrete Werte zugelassen werden. Beispielsweise sind die Translationsgeschwindigkeiten beschränkt auf 0, 10, 20, ..., $70\,cm/s$. Tabelle 4.1 zeigt die von uns verwendete Diskretisierung der Welt. Da der Raum aller Zustände extrem groß ist, wird dieser nicht über ein fünfdimensionales Array repräsentiert, sondern die einzelnen Zustände werden erst auf Anforderung konstruiert und dann in einer Hash-Tabelle [OTTMANN und WIDMAYER 1996] gespeichert. Die Zustandsübergänge sind nicht direkt erkennbar, wie etwa bei der Suche in der zweidimensionalen Ebene. Die Menge aller Nachbarn n, vom aktuellen Zustand akt aus gesehen, ergibt sich durch Anwenden der Bewegungsgleichungen aus Abschnitt 3.5.2. Durch Ausführen des Geschwindigkeitsvektors (v_n, ω_n) führt dann zu:

$$\langle x_{akt}, y_{akt}, \theta_{akt}, v_{akt}, \omega_{akt} \rangle \overset{(v_n, \omega_n)}{\longrightarrow} \langle x_n, y_n, \theta_n, v_n, \omega_n \rangle$$

wobei $|v_{akt} - v_n| \leq a_v$ und $|\omega_{akt} - \omega_n| \leq a_\omega$ mit a_v, a_ω die maximal zulässigen Geschwindigkeitsänderungen.

Um die Berechnungen zur Bestimmung jedes einzelnen Nachbarn nicht immer wieder ausführen zu müssen, werden die Ergebnisse zuvor einmalig tabelliert. So kann zu einem Geschwindigkeitsvektor (v_n, ω_n) als Eingabe der Nachfolgezustand $\langle x_n, y_n, \theta_n, v_n, \omega_n \rangle$ relativ zum aktuellen Zustand $\langle x_{akt}, y_{akt}, \theta_{akt}, v_{akt}, \omega_{akt} \rangle$ schnell ermittelt werden.

Parameter	**Diskretisierung**
x-Koordinate*	$10\ cm$
y-Koordinate*	$10\ cm$
Orientierung θ	$\pi/16\ (11.25°)$
Translationsgeschwindigkeit v	$10\ cm/s$
Rotationsgeschwindigkeit ω	$\pi/16s\ (11.25°/s)$

Tabelle 4.1: Diese Parameter zeigen die von uns verwendeten Diskretisierungen der Welt bei der Planung der Trajektorien. Die mit * gekennzeichneten Werte sind von der Umgebungskarte abhängig, wenn nicht anders erwähnt, wurden aber die aufgeführten Werte verwendet.

Auch müssen bei der Betrachtung der Kosten eines Zustandsübergangs stets die während des Zeitintervalls überstrichenen Felder berücksichtigt werden. Würde man dies ignorieren, könnte

ein Roboter unter Umständen versuchen eine dünne Wand zu durchfahren, da das Hindernis übersehen werden könnte, wie in Abbildung 4.5 dargestellt. Die Berechnung, welche Felder bei der Ausführung eines Fahrkommandos überstrichen werden, kann, ähnlich wie die Bestimmung des Zielfeldes zuvor, über eine Tabelle effizient realisiert werden.

Abbildung 4.5: Der Roboter im linken der drei Felder bewege sich mit einer Geschwindigkeit von zwei Feldern pro Zeitintervall nach rechts. Wenn die bei der Bewegung überstrichenen Felder nicht berücksichtigt werden, kann es zu einer Kollision des Roboters mit dem Hindernis im mittleren Feld kommen, da dies übersehen wird.

4.2.8. Die Nutzung der verbleibenden Rechenzeit

Nun müssen noch empfangene Nachrichten des Kontrollsystems des Roboters, das in Kapitel 5 genauer erläutert wird, abgearbeitet werden. Falls das Zeitintervall noch nicht komplett ausgenutzt wurde, können dann zusätzliche Arbeiten erledigt werden. Dies ist beispielsweise das Aktualisieren der Umgebungkarte im Benutzerinterface und das Abarbeiten zusätzlicher GUI-Nachrichten. Falls auch diese Arbeiten erledigt wurden, ohne dass die $250\,ms$ ausgenutzt wurden, wird die verbleibende Zeit gewartet.

4.2.9. Das Ausführen des ermittelten Fahrtkommandos

Nun befindet sich der Roboter ungefähr an der in Abschnitt 4.2.1 geschätzten Position, von der aus die Berechnungen gestartet wurden. Das erste Kommando des Pfades aus Geschwindigkeitsvektoren aus Abschnitt 4.2.7 wird an die Hardware des Roboters gesendet und es wird wieder mit Abschnitt 4.2.1 begonnen.

4.3. Beispiel einer geplanten Trajektorie

Dieser Abschnitt soll die soeben vorgestellten Planungsschritte durch ein Beispiel verdeutlichen. In Abbildung 4.6 sieht man eine geplante Trajektorie für einen Roboter. Die roten Flächen stellen Hindernisse in der Umgebung dar. Die Felder nahe diesen Hindernissen tragen, bedingt durch die *Faltung* der Karte, hohe Feldkosten, dargestellt durch einen schwarzen bzw. grauen Rand. Der Roboter wird durch den gelben Punkt dargestellt, der Zielpunkt durch den türkisen Punkt. Zwischen diesen Punkten befindet sich eine durchgezogene Linie. Sie zeigt das Ergebnis der

$A^\star$-Suche im $\langle x, y\rangle$-Raum. Den im $\langle x, y, \theta, v, \omega\rangle$-Raum geplanten Pfad zeigt die gepunktete Linie. Dieser Pfad liegt innerhalb der blau gezeichneten Region, die den *Channel* und damit die Einschränkung des vollen Konfigurationsraums zeigt.

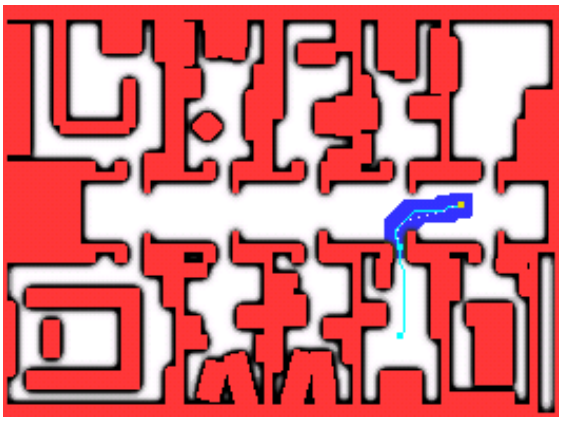

Abbildung 4.6: Eine von unserem Algorithmus geplante Trajektorie. Der blaue Bereich zeigt den *Channel*, der gelbe Punkt den Roboter, der türkise Punkt innerhalb des *Channel* den Zwischenzielpunkt. Die durchgezogene Linie zeigt die Trajektorie aus der $A^\star$-Suche im $\langle x, y\rangle$-Raum, die am globalen Zielpunkt endet, und die gepunktete Linie zeigt den Pfad der $A^\star$-Suche im $\langle x, y, \theta, v, \omega\rangle$-Raum zum Zwischenzielpunkt.

Abbildung 4.7 zeigt eine Sequenz von geplanten Trajektorien. Im Vergleich zu Abbildung 4.6 war der Roboter hier starken kinematischen Einschränkungen unterworfen. So konnte er sich nur extrem langsam drehen ($\omega_{max} = 11,25°/s$), dagegen aber relativ schnell geradeaus fahren ($v_{max} = 70\ cm/s$). Zusätzlich waren seine maximal möglichen Beschleunigungen sehr gering ($a_\omega = 11,25°/s^2$, $a_v = 10\ cm/s^2$). Dadurch weicht die im vollen Konfigurationsraum geplante Trajektorie verhältnismäßig stark vom Pfad der zweidimensionalen Suche ab.

Unter *http://www.informatik.uni-freiburg.de/~burgard/animations/stachniss-iros02/* finden sich zusätzliche Beispiele, unter anderem auch Videos, Animationen und Bildsequenzen zur Fahrt verschiedener Roboter unter Verwendung unseres neuen Ansatzes.

4.4. Die algorithmische Darstellung des Verfahrens

Dieser Abschnitt zeigt mit Algorithmus 3 die soeben erklärten Schritte zum Finden des besten zulässigen Fahrkommandos, um den Roboter von seiner aktuellen Position aus in Richtung Zielzustand zu bewegen.

Im Folgenden wird auf einzelne Punkte noch genauer eingegangen und kleine Veränderungen am Verfahren vorgenommen, da nicht alle in Abschnitt 4.2 gemachten Annahmen in der realen Welt zulässig sind. Einige Situationen müssen dann, wie in Abschnitt 4.5 erklärt wird, abgefangen

und gesondert behandelt werden, um eine kollisionsfreie Fahrt des Roboters gewährleisten zu können.

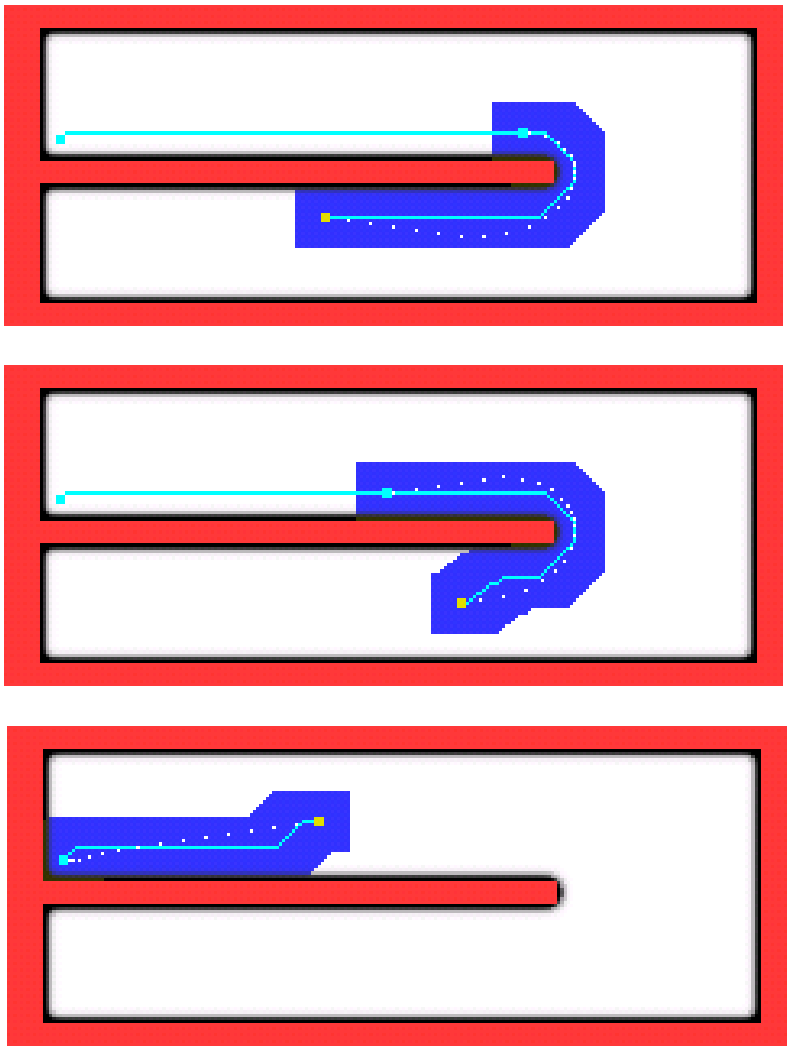

Abbildung 4.7: Hier sieht man einige von unserem Algorithmus geplante Trajektorien. Wie in Abbildung 4.6 zeigt der blaue Bereich den *Channel*, der gelbe Punkt den Roboter, der türkise Punkt den Zwischenzielpunkt bzw. den globalen Zielpunkt. Weiter zeigt die durchgezogene Linie den Pfad aus der $A^\star$-Suche im $\langle x, y\rangle$-Raum und die gepunktete Linie das Ergebnis der $A^\star$-Suche im $\langle x, y, \theta, v, \omega\rangle$-Raum. Im Vergleich zur Abbildung 4.6 wurden diese Trajektorien für einen Roboter mit starken kinematischen Einschränkungen geplant ($\omega_{max} = 11,25°/s$, $a_\omega = 11,25°/s^2$, $v_{max} = 70\,cm/s$, $a_v = 10\,cm/s^2$). Dadurch unterscheidet sich der im vollen Konfigurationsraum geplante Pfad verhältnismäßig stark von dem im $\langle x, y\rangle$-Raum geplanten.

Algorithmus 3: Nächstes Fahrkommando

Eingabe: Umgebungskarte $karte$, Zielpunkt $ziel$
Ausgabe: Das nächste auszuführende Kommando (v, ω)

```
position = SchaetzePositionNachZeitintervall();
karte = AktualisiereUmgebungskarte(karte);
pfad2d = A*_2d(karte, position, ziel);
if pfad2d wurde nicht gefunden then
    return derzeitig existiert kein Weg zum Zielpunkt;
end if
channel_groesse = SchaetzeGuenstigeChannelGroesse();
channel = KonstruiereChannel(pfad2d, channel_groesse);
zwischen_ziel = ErmittleZwischenZielpunkt(pfad2d, channel);
heuristik5d = ValueIteration(karte, channel, zwischen_ziel);
pfad5d = A*_5d(karte, channel, position, zwischen_ziel, heuristik5d);
if pfad5d erfolgreich berechnet then
    return erstesFahrkommando(pfad5d);
else
    return (v = 0, ω = 0);    // Stoppen des Roboters
end if
```

4.5. Ausnahmebehandlung bei der Planung der Trajektorie

Nicht immer kann der Pfad des Roboters ermittelt werden, wie dies die letzten Abschnitte vermuten lassen. Während der Pfadplanung können verschiedenen Probleme auftreten, auf deren Behandlung wir im Folgenden eingehen werden:

- Der Roboter wird fälschlicherweise innerhalb eines Hindernisses lokalisiert, wodurch der $A^\star$-Algorithmus ohne Lösung terminiert.
- Die $A^\star$-Suche im $\langle x, y\rangle$-Raum konnte keinen Pfad zum Zielpunkt finden, da alle Wege versperrt sind oder der Zielpunkt selbst innerhalb eines Hindernisses liegt.
- Die Suche nach einer Trajektorie im $\langle x, y, \theta, v, \omega\rangle$-Raum hat nach $250\,ms$ noch keine Lösung gefunden. Der $A^\star$-Algorithmus hat sich daher selbst abgebrochen.

4.5.1. Lokalisierung des Roboters innerhalb eines Hindernisses

Dieses Phänomen kann zwei Ursachen haben. Einerseits können Sensoren fehlerhafte Daten geliefert haben, so dass ein Hindernis an der aktuellen Position des Roboters vermutet wird. Die

andere Möglichkeit besteht darin, dass die vom Lokalisierungsmodul (siehe Kapitel 5) ermittelten Korrekturparameter fehlerhaft sind, während sich der Roboter nahe eines Hindernisses befindet. Fälschlicherweise wird dann unter Umständen die Position des Roboters auf die eines Hindernisses abgebildet. Dieses Problem lässt sich allerdings recht einfach lösen: Man transformiert dazu die Umgebungskarte so, dass die Kosten eines Hindernisfeldes nicht mehr unendlich hoch sind, so dass das Betreten des Feldes generell verboten ist. Durch einen sehr hohen aber endlichen Wert kann der Roboter, sofern es keine andere Möglichkeit gibt, ein Hindernisfeld überfahren.

Falls ein Zwischenzielpunkt aus dem vorhergehenden Berechnungsschritt existiert und sich dieser nicht innerhalb eines Hindernisses befindet, wird er als weiterhin gültig angenommen und die angepasste $A^\star$-Suche im $\langle x, y, \theta, v, \omega \rangle$-Raum innerhalb des *Channel* des letzten Berechnungsschrittes ausgeführt.

Falls allerdings der letzte Zwischenzielpunkt innerhalb eines Hindernisses liegt oder noch gar nicht berechnet wurde, wird das nächste freie Feld in der Umgebung des Roboters als neuer Zwischenzielpunkt gewählt und anschließend die $A^\star$-Suche ausgeführt.

4.5.2. Die $A^\star$-Suche im $\langle x, y \rangle$-Raum scheitert

In diesem Falle sind alle möglichen Pfade des Roboters zum Zielpunkt durch Hindernisse blockiert. Der Roboter wird daraufhin angehalten und es werden alle dynamischen Hindernisse aus der Karte entfernt, nur die letzte Messung wird in die Karte integriert. Dadurch werden eventuell nicht mehr gültige Hindernisse aus der Karte entfernt. Kann nun immer noch kein Pfad zum Zielpunkt gefunden werden, stoppt der Roboter und wartet eine gewisse Zeit lang, in der Hoffnung, dass der Pfad wieder frei wird. Es kann an dieser Stelle auch eine Nachricht an ein optional vorhandenes Sprachausgabesystem gesendet werden, welches dann eventuell vorhandene Personen auffordert, den Weg zu räumen. Falls auch dies zu keinem Erfolg führt, wird die Planung abgebrochen, der Zielpunkt kann zum gegenwärtigen Zeitpunkt nicht erreicht werden.

4.5.3. Zeitüberschreitung bei der Suche im vollen Konfigurationsraum

Dies ist der am häufigsten auftretende „Fehler“. Meist wurde der *Channel* aus Abschnitt 4.2.5 im Verhältnis zur verfügbaren Rechenleistung zu groß gewählt, die Heuristik konnte die Kosten bei der Suche nicht gut genug abschätzen oder von einer anderen, auf dem Rechner ausgeführten Applikation, wurden größere Arbeiten ausgeführt. Daher wird die Größe des *Channel* für den nächsten Schritt reduziert. Jedoch muss dem Roboter schon in diesem Zyklus ein Fahrkommando gesendet werden, d.h. es muss in möglichst kurzer Zeit ein alternatives Kommando ermittelt werden. Falls im vorherigen Durchgang die Planung im fünfdimensionalen Raum erfolgreich war, wird versucht, das nächste Fahrkommando aus der alten Berechnung zu verwenden. Dazu muss allerdings getestet werden, ob der alte Pfad nicht in der Zwischenzeit zu einer Kollision

mit einem neuen Hindernis führt.

Falls aber auch schon im letzten Durchgang aufgrund einer Zeitüberschreitung die Planung abgebrochen wurde, wird der Roboter aus Sicherheitsgründen angehalten und der *Channel* anschließend auf eine kleine Größe zurückgesetzt. Aus diesem definierten Zustand wird dann die Planung wieder aufgenommen.

Führt dies immer noch zu keinem Ergebnis besitzt der Roboter die Möglichkeit sich entlang des Pfades aus der zweidimensionalen Suche zu bewegen, bis eine Planung im vollen Konfigurationsraum wieder möglich ist. Um sich auf dem Pfad aus der Suche im $\langle x, y \rangle$-Raum zu bewegen, fährt der Roboter einfach die einzelnen Geradenstücke des Pfades ab. Abbildung 4.8 zeigt das Vorgehen des Roboters, der sich hier im Feld $A2$ befinde. Bis zum Feld $C2$ kann er eine gerade Strecke zurücklegen. Dannach muss er anhalten, sich um $45°$ drehen und kann dann seine Fahrt nach $D1$ fortsetzen. Nach einem weiteren Stopp mit anschließender Drehung kann er dann den Zielpunkt im Feld $E1$ erreichen.

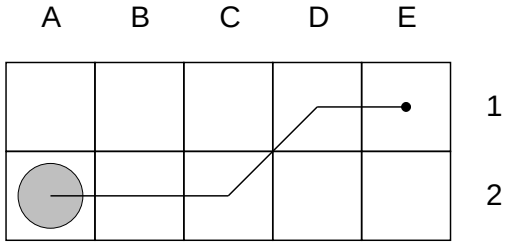

Abbildung 4.8: In einigen Fällen muss ein alternatives Fahrkommando ermittelt werden, da die Suche im vollen Konfigurationsraum nicht erfolgreich war. Eine Möglichkeit ist das Abfahren des Pfades aus der $A^\star$-Suche im $\langle x, y \rangle$-Raum. Der Roboter befinde sich im Feld $A2$. Er fährt eine gerade Strecke bis zum Feld $C2$. Dort hält er an und dreht sich um $45°$ und fährt anschließend ins Feld $D1$. Nach einem weiteren Stopp und einer Drehung kann er dann zum gewünschten Zielpunkt im Feld $E1$ fahren.

Alternativ könnte man hier auf andere reaktive Systeme, wie beispielsweise den *Dynamic Window Approach* zurückgreifen, da auch diese Ansätze in sehr kurzer Zeit ein zulässiges Fahrkommando ermitteln können.

4.6. Das sichere Stoppen des Roboters

Wie im vorherigen Abschnitt erklärt, muss der Roboter in bestimmten Situationen gestoppt werden, da kein mit Sicherheit kollisionsfreier Pfad zum gewünschten Zielpunkt gefunden werden kann. Allerdings reicht es im Allgemeinen nicht aus, den Geschwindigkeitsvektor (v, ω) des Roboters auf Null zusetzen, da dieser aufgrund seiner Trägheit nicht sofort zum Stehen kommt. Durch die Verzögerung kann es theoretisch zu einer Kollision kommen, so dass es günstiger wäre, das Hindernis durch einen Richtungswechsel zu umfahren, ähnlich wie dies auch von der

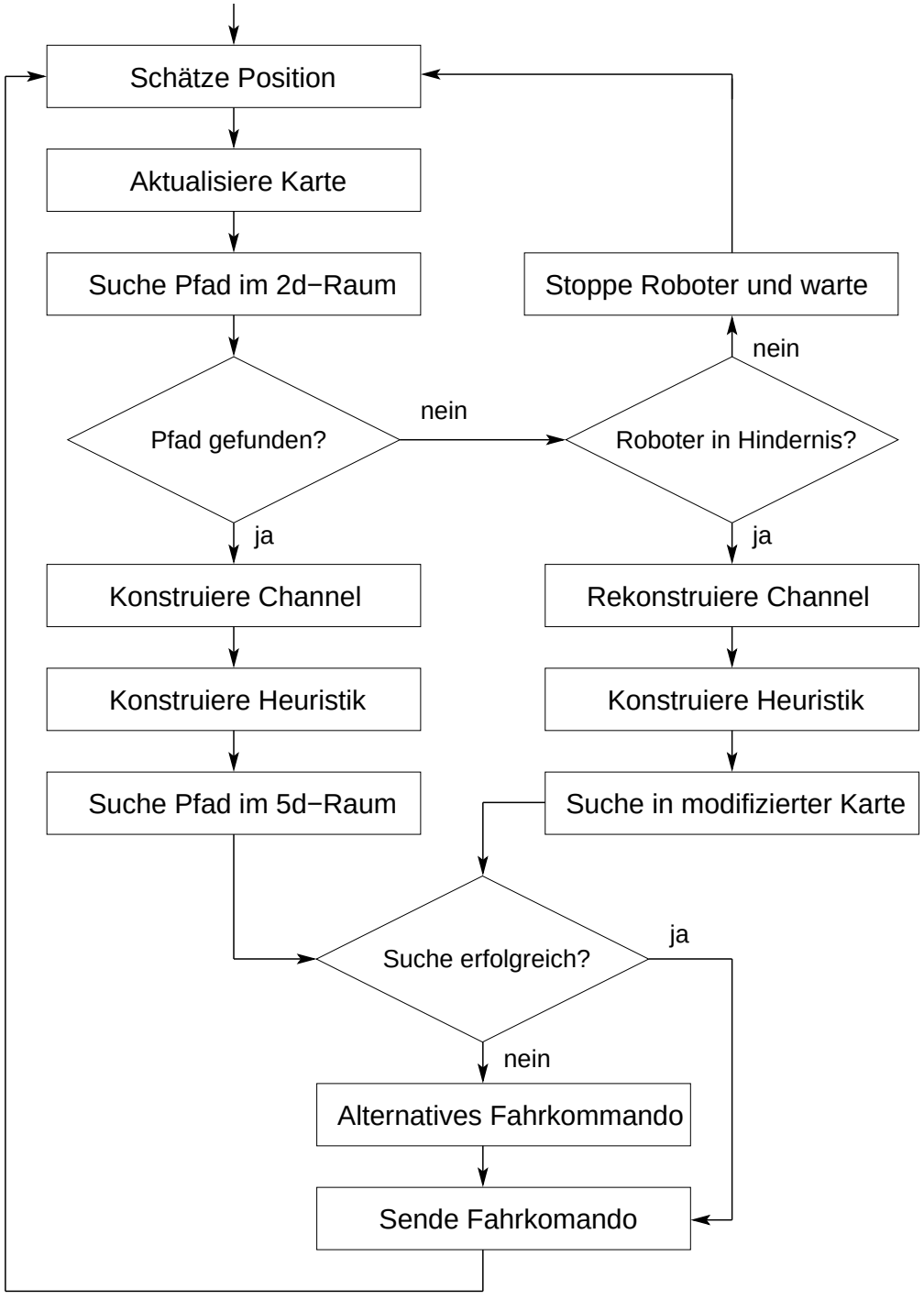

Abbildung 4.9: Das Flussdiagramm zeigt die grundlegenden Schritte des vollständigen Algorithmus zur Pfadplanung.

Kollisionsvermeidung von Fox et al. realisiert wird [FOX et al. 1997].

Um eine kollisionsfreie Fahrt des Roboters gewährleisten zu können, führen wir eine $A^\star$-Suche im fünfdimensionalen Raum aus, wobei der Zielzustand nur durch den Geschwindigkeitsvektor $(v, \omega) = (0, 0)$ gegeben ist, Ort und Orientierung können beliebige Werte annehmen. Eine solche Suche kann extrem schnell ausgeführt werden, da nur sehr wenige Zustände des an sich großen Suchraums betrachtet werden müssen bis einer der vielen möglichen Zielzustände gefunden wird. Die berechneten Fahrkommandos führen dann zu einer kollisionsfreien Fahrt, die mit dem Stillstand des Roboters endet. Abbildung 4.10 zeigt einen Roboter, der nur den Geschwindigkeitsvektor auf Null setzt, so dass es zu einer Kollision mit einem Hindernis kommt. In Abbildung 4.11 dagegen kann der Roboter durch ein Ausweichmanöver eine Kollision vermeiden.

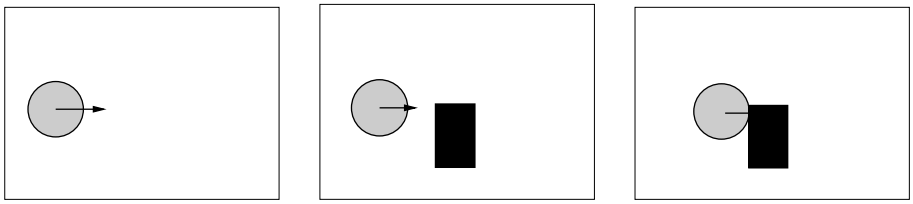

Abbildung 4.10: Der Roboter stoppt durch einfaches Setzten des Geschwindigkeitsvektors auf Null. Aufgrund der Trägheit des Roboters kommt es zu einer Kollision mit dem neu entdeckten Hindernis.

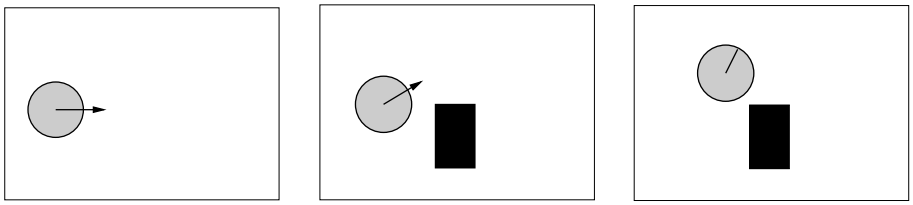

Abbildung 4.11: Der Roboter bremst und führt während dessen ein Ausweichmanöver aus, wodurch eine Kollision mit dem neu erkannten Hindernis vermieden werden kann.

4.7. Zeitüberschreitungen in Abhängigkeit des Ortes

Trotz eines kleinen *Channel* kann es gelegentlich zu Zeitüberschreitungen bei der $A^\star$-Planung kommen. Nun stellt sich die Frage, in wie weit diese von der Position des Roboters in der Umgebungskarte abhängen oder ob die Position darauf keinerlei Einfluss besitzt und die Zeitüberschreitungen beispielsweise durch andere Programme, die auf dem Computer laufen, zustande kommen. Wie später in Experimenten gezeigt wird, gibt es Positionen im Raum, an denen

die Planung im vollen Konfigurationsraum länger dauert und somit Zeitüberschreitungen wahrscheinlicher sind als an den übrigen Positionen.

Das Wissen über solche Felder kann bei der Planung berücksichtigt und somit beispielsweise die Größe des *Channel* im Vorhinein reduziert werden, um den Aufwand bei der $A^\star$-Suche zu reduzieren. Unsere Software legt daher zu jeder Umgebungskarte eine so genannte *Timeout-Karte* an, in der die Anzahl der Besuche und der aufgetretenen Zeitüberschreitungen eines Feldes gespeichert werden. Zusätzlich wird noch die Größe des *Channel* bei der letzten Zeitüberschreitung für jedes Feld gespeichert. Diese Karte wird bei jeder Fahrt des Roboters aktualisiert und passt sich somit der Umgebung, wie auch der Leistung des zur Planung verwendeten Computers an.

Wenn der Roboter sich an einer Stelle befindet, die bereits in der *Timeout-Karte* vermerkt ist, betrachtet er zur Bestimmung der Größe des *Channel* den dort gespeicherten Wert. Dieser gibt Auskunft über deren Größe bei der letzten Zeitüberschreitung. Aus der Anzahl der Besuche und der Zeitüberschreitungen eines Feldes kann eine Wahrscheinlichkeit für diese Art von „Fehler“ an dieser Position bestimmt werden. In Abhängigkeit der errechneten Wahrscheinlichkeit wird die Größe des *Channel* basierend auf dem Wert aus der *Timeout-Karte* nochmal um eine Konstante verkleinert. Wie im nächsten Kapitel gezeigt wird, reduziert sich durch dieses Vorgehen die Anzahl der Zeitüberschreitungen, da an den entscheidenden Positionen im Raum der volle Konfigurationsraum durch den *Channel* stärker als sonst eingeschränkt werden kann. Abbildung 6.16 zeigt eine solche *Timeout-Karte*. An den blau gekennzeichneten Stellen traten Zeitüberschreitungen auf und man sieht deutlich eine Häufung an bestimmten Positionen.

5. Details zur Implementierung

Nahezu jedem Roboter liegt ein Kontrollsystem zugrunde, das die Hardware des Roboters ansteuert und verschiedene Aufgaben bewältigen kann. Die von uns entwickelte Software muss daher in das bestehende, modular gestaltete Kontrollsystem integriert werden. Diese Kapitel soll einen Überblick über die wichtigsten Module geben.

5.1. Die *Bee-Software*

Hinter dem Namen *Bee-Software* verbirgt sich eine Sammlung von über 20 Modulen zur Steuerung mobiler Roboter. Diese einzelnen Module werden über eine Middleware-Architektur namens *TCX* [FEDOR 1993] verbunden. Basierend auf dem *TCP/IP*-Protokoll organisiert der *TCX-Server* den Informationsaustausch zwischen den einzelnen Modulen. Diese klinken sich zur Laufzeit in den *TCX-Server* ein und bilden so ein lose gekoppeltes, verteiltes und ortsunabhängiges System. Somit ist auch der Austausch von Modulen problemlos möglich.

Diese Technologie erlaubt es, rechenintensive Arbeiten auf leistungsfähige Computer auszulagern. Dagegen können hardwarenahe Tätigkeiten direkt auf dem Rechner des Roboters ausgeführt werden, der über ein Funknetzwerk mit den anderen Computern verbunden ist.

Durch die modulare Struktur unterstützt die *Bee-Software* diverse Robotertypen, indem einfach das entsprechende Steuerungsmodul ausgetauscht wird. Uns standen zwei mobile Roboter zur Verfügung, einmal der B21r Roboter `Albert` (siehe Anhang A.1) sowie der Pioneer1 Roboter `Ludwig` (siehe Anhang A.2), beide dargestellt in Abbildung 5.1.

Im Folgenden möchten wir nun die wichtigsten Komponenten der *Bee-Software* kurz beschreiben. Abbildung 5.2 zeigt den Aufbau der Architektur, wobei die in unserer Arbeit entwickelte Software namens *Newplanner* das ursprüngliche Modul *Colli-Server* zur Kollisionsvermeidung ersetzt und auch die Funktionalität des *Plan*-Moduls zur Pfadplanung mit übernimmt.

5.1.1. Die Kommunikation mit der Hardware via *Base-Server*

Der *Base-Server* stellt die erste Softwareebene über der Basishardware des Roboters dar. Er steuert die Motoren und hat Zugriff auf die Messungen einiger Sensoren. Des weiteren stellt er ein Koordinatensystem relativ zur Startposition des Roboters zur Verfügung. Die Bewegung innerhalb dieses Koordinatensystems wird über Sensoren an den Rädern des Roboters ermittelt.

Abbildung 5.1: Die mobilen Roboter `Ludwig` (links) und `Albert` (rechts).

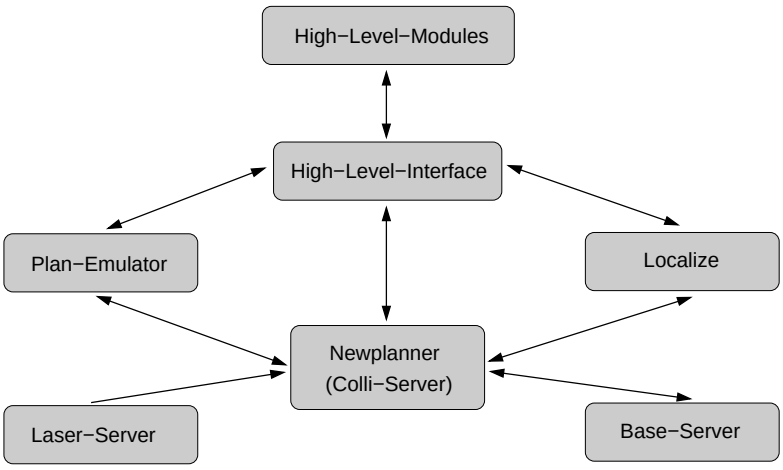

Abbildung 5.2: Die wichtigsten Komponenten der *Bee-Software* im Datenflussdiagramm.

Falls unterschiedliche Robotertypen zum Einsatz kommen, muss für jeden Typ ein angepasster *Base-Server* entwickelt werden. Oberhalb dieser Ebene unterscheiden einige Module noch die verschiedenen Robotertypen, dies dient aber nur zur Optimierung des Fahrverhaltens. Die eigentliche Schnittstelle zur Steuerungssoftware ist durch dieses Modul gegeben.

5.1.2. Das Auslesen der Lasersensoren mit dem *Laser-Server*

Zusätzlich zur Basishardware nutzen viele Roboter Lasersensoren zur Erkennung von Hindernissen (siehe Anhang A.3). Der *Laser-Server* bietet eine Schnittstelle zum Zugriff auf die Hardware der laserbasierten Entfernungsmesser. Er stellt in Zeitabständen von ca. $250\,ms$ Abstandsmessungen kombiniert mit einem Zeitstempel zur Verfügung. Diese Informationen werden verwendet, um einerseits Hindernisse zu erkennen und andererseits die Position des Roboters innerhalb einer Karte zu schätzen.

5.1.3. Die Positionsbestimmung via *Localize*

Das Wissen über die genaue Position des Roboters bei der Wegplanung ist besonders in engen Passagen essentiell. Man verwendet dafür eine Umgebungskarte mit eigenem Koordinatensystem. Bedingt durch unebenen Boden, Schlupf, Messungenauigkeiten und ähnliche Einflüsse der realen Welt ist es nicht möglich die Position des Roboters während einer längeren Fahrt nur mithilfe der an den Rädern angebrachten Sensoren zu bestimmen. Durch Auswertung zusätzlicher Sensordaten, wie die des Laserscanners, ist es aber möglich die Position des Roboters näherungsweise zu bestimmen.

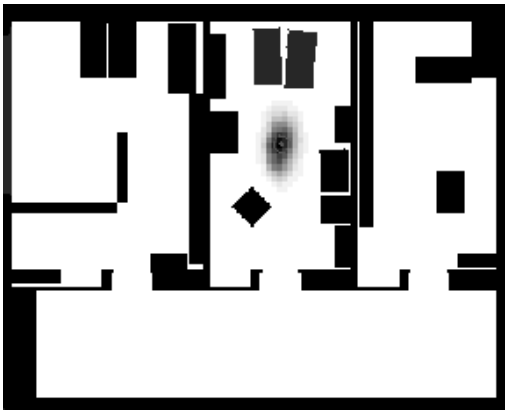

Abbildung 5.3: Die Schätzung der Roboterposition durch *Localize*.

Die *Bee-Software* stellt hierzu ein Modul namens *Localize* zur Verfügung. Es wertet Odometrie- sowie Laserdaten aus und benutzt zur Positionsbestimmung eine Version der Markov Loka-

lisierung [BURGARD et al. 1996]. Diese liefert die Roboterposition als Wahrscheinlichkeitsverteilung im Raum. Nach jeder Bewegung des Roboters wird die Aufenthaltswahrscheinlichkeit anhand neuer Odometrie- und Laserdaten aktualisiert. Falls die Umgebung keine zu großen Selbstähnlichkeiten ausweist, liefert das Verfahren eine Wahrscheinlichkeitsverteilung, die um die tatsächliche Position des Roboters gehäuft ist. Abbildung 5.3 zeigt die Visualisierung einer solchen Verteilung.

Anhand des wahrscheinlichsten Ortes werden Korrekturparameter errechnet, mit denen man die Position aus dem Koordinatensystem des *Base-Server* in das der Umgebungskarte transformieren kann.

5.1.4. Die Wegplanung und Kollisionsvermeidung

Colli-Server und *Plan*-Modul

In der skizzierten Architektur des Systems wird die Planung eines Pfades zwischen Positionen innerhalb der Umgebungskarte vom *Plan*-Modul [THRUN et al. 1998] ausgeführt. Die Kollisionsvermeidung in der direkten Umgebung des Roboters erledigt der *Colli-Server* [FOX et al. 1997]. Letzterer ist eine sehr grundlegende Komponente der Architektur und stellt dem System viele zusätzliche Funktionen zur Verfügung, wie beispielsweise die Verteilung von Sensor- und Odometriedaten an andere Module.

Newplanner

Um die von uns entwickelte Software *Newplanner* zur Wegplanung und Kollisionsvermeidung in das bestehende System zu integrieren, müssen die beiden zuvor genannten Module *Colli-Server* und *Plan* entfernt werden.

Um aber die Kompatibilität zu anderen bestehenden Modulen zu wahren, sollten sämtliche Aufgaben der beiden ursprünglichen Module von *Newplanner* übernommen werden. Einige Funktionen müssen allerdings, aufgrund des neuen Ansatzes, deaktiviert werden, um eine kollisionsfreie Fahrt des Roboters zu gewährleisten. So darf beispielsweise ein fremdes Modul nicht mehr die aktuelle Geschwindigkeit des Roboters bestimmen.

Aufrund der gestiegenen Rechenleistung moderner Computer ist es nicht mehr notwendig Wegplanung und Kollisionsvermeidung zu trennen, um sie auf verschiedenen Rechnern ausführen zu können. Aus Kompatibilitätsgründen wird aber ein weiteres Modul, der *Plan-Emulator* benötigt. Dieser gibt sich innerhalb der *Bee-Software* als *Plan*-Modul aus und fungiert dann als eine Art Proxy-Server, indem er Anfragen anderer Module an *Newplanner* weiterleitet.

Zusammengefasst sind die Aufgaben von *Newplanner*:

- Planung eines kollisionsfreien Weges zwischen zwei Positionen,
- Ausführung des geplanten Pfades,
- Erkennung von Hindernissen,
- Verteilung von Sensor- und Statusinformationen an andere Module, sowie die
- Bereitstellung einer Schnittstelle zur Steuerung des Roboters durch andere Module.

Um einen Eindruck der Software zu vermitteln, zeigt Abbildung 5.4 eine Momentaufnahme von *Newplanner* während der Planung einer Trajektorie.

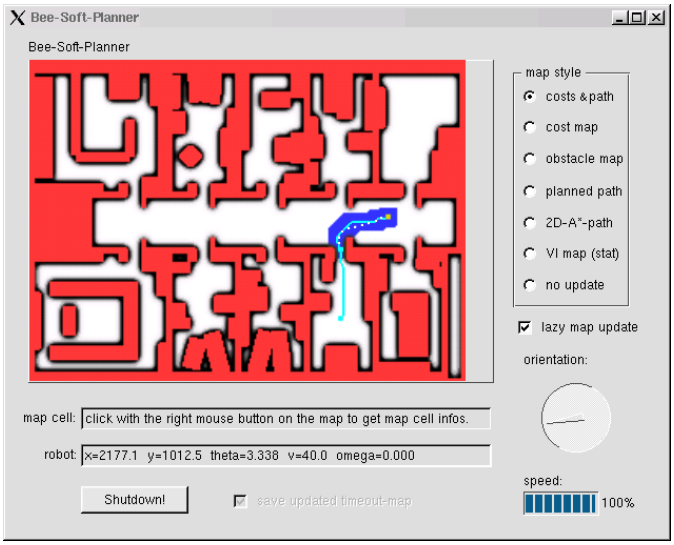

Abbildung 5.4: Eine Momentaufnahme von *Newplanner*. Die Karte zeigt die Umgebung des Roboters, der durch den gelben Punkt gekennzeichnet ist. Innerhalb diese Karte kann der Benutzer Zielpunkte anwählen und somit den Roboter steuern.

5.1.5. Das High-Level-Interface *HLI*

Um die Arbeit für Anwendungsentwickler zu vereinfachen, integriert das High-Level-Interface [HÄHNEL et al. 1998] die Schnittstellen der verschiedenen Module in ein einziges Interface und führt Informationen verschiedener Quellen zu einem einheitlichen Statusbericht zusammen. Der Anwendungsentwickler muss sich daher nicht mehr mit einer Menge von Modulen auseinander setzen, sondern kann auf eine einheitliche Schnittstelle zurückgreifen.

Außerdem generiert *HLI* Rückmeldungen über den Ausgang von Aktionen, während die meisten anderen Module der *Bee-Software* nicht über eine solche Funktion verfügen. Durch das Überwachen der verschiedenen Module ist es jedoch oft möglich, Aussagen über den Ausgang einzelner Aktionen zu machen.

Somit bietet *HLI* eine deutlich benutzerfreundlichere Programmierschnitstelle als die Sammlung der einzelnen Module an sich. Viele optionale Komponenten setzen daher auf dem High-Level-Interface auf, so dass die Kompatibilität der neuen Software mit dem High-Level-Interface der *Bee-Software* ein wichtiger Aspekt für deren Akzeptanz ist.

5.1.6. Der *Simulator*

Um Experimente häufiger ausführen und die Einflüsse bestimmter Parameter besser vergleichen zu können, stellt die *Bee-Software* einen Simulator zur Verfügung. Dieser generiert, basierend auf einer Karte, Sensor- und Odometriedaten eines virtuellen Roboters. Dieser wird samt Beschleunigungsverhalten nachempfunden, selbst sich bewegende Hindernisse können generiert werden. *Newplanner* kann auch dessen Nachrichten interpretieren und sendet die Fahrkommandos auf Wunsch nicht an den *Base-Server*, sondern an den *Simulator*.

5.2. Die Neuerungen im Kontrollsystem

5.2.1. Die Veränderungen durch *Newplanner*

Wie im Abschnitt 5.1.4 beschrieben, wird das Modul *Colli-Server* komplett aus der Architektur entfernt, da dessen Aufgaben in Zukunft von *Newplanner* wahrgenommen werden. Diese lassen sich in drei Kategorien unterteilen: Einerseits Funktionen, die nahezu identisch übernommen werden können, zweitens Operationen, die in ihrer Umsetzung modifiziert werden müssen und drittens Funktionen, die nicht mehr unterstützt werden können.

Zur ersten Gruppe zählt beispielsweise der Not-Halt oder das Senden von Statusberichten. Nachrichten, die den Roboter auffordern eine gewisse Wegstrecke von der aktuellen Position aus zurückzulegen, zählen zur zweiten Gruppe und werden durch das Setzen eines neuen Zielpunktes in entsprechendem Abstand realisiert. Nicht umsetzbar dagegen sind Funktionen, die in die Bewegung des Roboters eingreifen und die an den Roboter zu sendende Geschwindigkeit direkt beeinflussen. Das Zulassen solcher Operationen würde in dem neuen Ansatz der Pfadplanung die kollisionsfreie Fahrt des Roboters gefährden.

Wie in Abschnitt 5.1.5 kurz erwähnt, senden die meisten Module der *Bee-Software* keine Informationen über Erfolg oder Misserfolg einer Aktion. Um die Benutzerfreundlichkeit zu erhöhen, versendet *Newplanner* Rückmeldungen über den Ausgang ausgeführter Aktionen an andere Mo-

dule. Dafür musste eine neue Schnittstelle[1] geschaffen werden. Über diese werden beispielsweise Nachrichten versendet, wenn ein Zielpunkt erreicht wurde oder auch die Planung abgebrochen wurde, da der Zielpunkt nicht erreicht werden konnte. Des weiteren existiert die Möglichkeit, sich über das Setzen neuer Zielpunkte informieren zu lassen. So kann beispielsweise ein Modul feststellen, ob der Benutzer die aktuelle Planung unterbrochen und einen neuen eigenen Zielpunkt gesetzt hat, und dadurch die folgenden Aktionen darauf abstimmen.

5.2.2. Die Emulation des *Plan*-Moduls

Um die zuvor beschriebene Kompatibilität zu anderen bestehenden Modulen zu erhalten, muss auch die Schnittstelle des ursprünglichen *Plan*-Moduls bereitgestellt werden. Diese Aufgabe übernimmt der *Plan-Emulator*. Durch ihn empfangene Nachrichten werden konvertiert und an *Newplanner* weitergeleitet. Zusätzlich werden Statusberichte und Informationen über noch anzufahrende Zielpunkte an verbundene Module verteilt. Da *Newplanner* nur die Planung zu einem einzelnen Zielpunkt unterstützt, wurde der *Plan-Emulator* um eine Queue von Zielpunkten erweitert. Hat der Roboter seinen Zielpunkt erreicht und befinden sich weitere Zielpunkte in der Queue des Emulators, werden diese nacheinander an *Newplanner* weitergereicht.

5.3. Das Kontrollsystem *CARMEN*

Neben der *Bee-Software* gibt es noch weitere Kontrollsysteme zur Steuerung von mobilen Robotern. Das *Carnegie Mellon Robot Navigation Toolkit (CARMEN)* ist ein Kontrollsystem, das der *Bee-Software* vom Aufbau her sehr ähnlich ist. Es basiert auf der Middleware-Architektur namens *IPC* [SIMMONS und JAMES 2001], über die einzelne Softwarekomponenten Nachrichten miteinander austauschen können. Identisch zur *Bee-Software* werden Laser-, Odometrie- und Lokalisierungsinformationen über einzelne Module zur Verfügung gestellt.

Um unsere Software zusätzlichen Roboterplattformen zugänglich zu machen, kann *Newplanner* auch mit *CARMEN* zusammen arbeiten. Die derzeitige Version von *CARMEN* besitzt jedoch eine reaktive Kollisionsvermeidung, auf der alle Planungsmodule aufsetzen und die nicht deaktiviert werden kann. Falls dieses Modul eine potentielle Kollision entdeckt, werden die an den Roboter zu sendenden Fahrkommandos nicht ausgeführt und dieser gestoppt. Es ist also nicht möglich, die volle Kontrolle des Roboters auf *Newplanner* zu übertragen. Aus diesem Grund wurden alle nun folgenden Versuche auf Basis der *Bee-Software* ausgeführt, da dort die volle Kontrolle des Roboters durch *Newplanner* gewährleistet ist.

[1] `BASE2-messages.h`

6. Experimente

In diesem Kapitel präsentieren wir verschiedene Experimente, die in Simulationen wie auch mit echten Robotern durchgeführt wurden. Anfangs wird gezeigt, dass die in Abschnitt 4.2.3 vorgestellte Heuristik die $A^\star$-Suche beschleunigt, danach werden Fahrten des Roboters in verschiedenen Umgebungen untersucht und die Unterschiede zum bisherigen Navigationssystem analysiert. Anschließend wird das Fahrverhalten des Roboters in Abhängigkeit von der Größe der Raumdiskretisierung und der Art der Einschränkung des Suchraums betrachtet. Das Kapitel endet mit dem Vergleich der Fahrzeiten unseres Ansatzes zur theoretisch optimalen Lösung, gegeben die modellierten, kinematischen Einschränkungen des Roboters und die Diskretisierung der Welt.

6.1. Der Einfluss der Heuristik auf die $A^\star$-Suche

In diesem Abschnitt zeigen wir anhand von experimentellen Daten, dass die in Abschnitt 4.2.3 konstruierte Heuristik, im Vergleich zu einer Abschätzung basierend auf der euklidischen Distanz, die $A^\star$-Suche deutlich beschleunigt. Die optimierte und in unserem Ansatz verwendete Heuristik basiert auf der $kosten\_zum\_ziel$-Funktion, die aus einer *Value Iteration* auf der statischen Umgebungskarte hervorgeht. Abbildung 6.1 zeigt die für die $A^\star$-Suche im $\langle x, y\rangle$-Raum mit zwei verschiedenen Heuristiken benötigte Zeit während der Fahrt eines Roboters zu einem Zielpunkt. In der linken Abbildung wurde für die Abschätzung der Kosten zum Ziel die euklidische Distanz in Verbindung mit den minimalen Feldkosten verwendet, in der rechten Abbildung wurde auf die optimierte Heuristik zurückgegriffen. Abbildung 6.2 zeigt den Weg, den der Roboter während des Tests gefahren ist. Wie man leicht sieht, ist die $A^\star$-Suche mit der optimierten Heuristik deutlich schneller. Während die Berechnung des kürzesten Weges dort unterhalb von $10\,ms$ liegt[1], benötigt die Suche mit der euklidischen Heuristik dagegen zeitweise sogar über ein Drittel der zur Verfügung stehenden Rechenzeit von $250\,ms$.

Ein ähnliches Ergebnis erhält man bei der $A^\star$-Suche im $\langle x, y, \theta, v, \omega\rangle$-Raum innerhalb des *Channel*. Wird auf eine Heuristik, basierend auf der euklidischen Distanz, kombiniert mit der maximalen Geschwindigkeit und den minimalen Feldkosten zurückgegriffen, müssen wesentlich mehr Zustände expandiert werden, als unter Verwendung der optimieren Heuristik aus Abschnitt 4.2.6. Die beiden Histogramme in Abbildung 6.3 zeigen diesen Sachverhalt.

[1] auf einem Pentium-III Laptop mit 800MHz

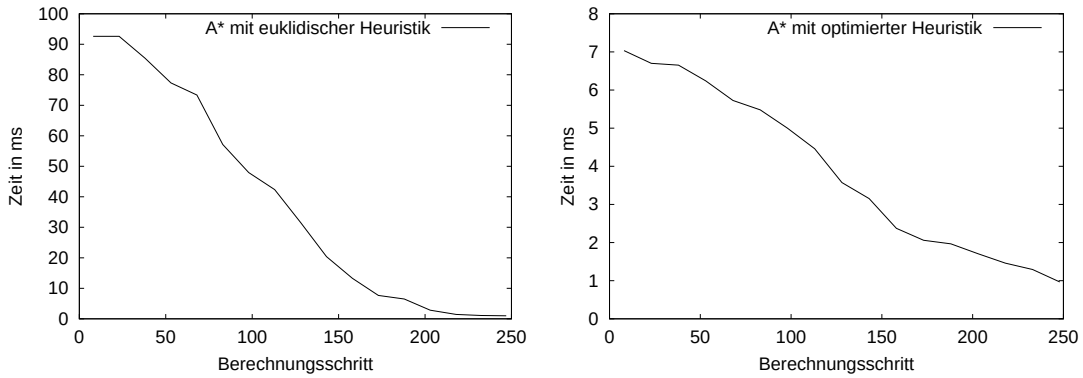

Abbildung 6.1: Die linke Abbildung zeigt die benötigte Zeit, während der Fahrt eines Roboters zu einem Zielpunkt, für die $A^\star$-Suche im $\langle x, y \rangle$-Raum unter Verwendung der euklidischen Heuristik. Die rechte Abbildung zeigt die für den gleichen Pfad benötigte Zeit, falls die in Abschnitt 4.2.3 eingeführte Heuristik, basierend auf der $kosten\_zum\_ziel$-Funktion einer *Value Iteration*, zum Einsatz kommt.

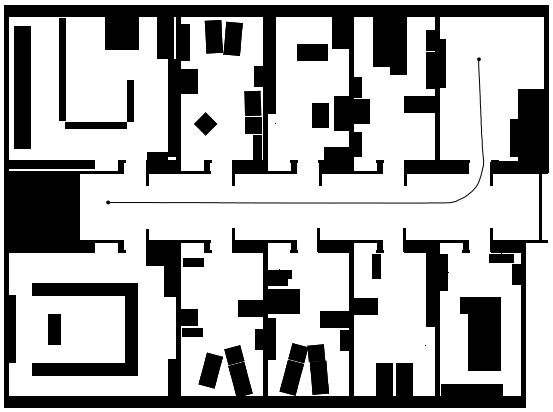

Abbildung 6.2: Der durch den Roboter abgefahrene Pfad beim Vergleich der Heuristiken für die $A^\star$-Suche.

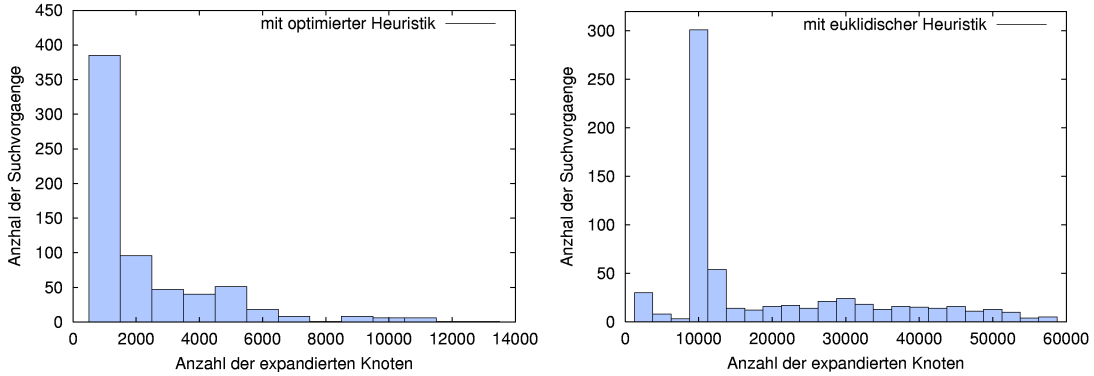

Abbildung 6.3: Beide Abbildungen zeigen die Anzahl der expandierten Knoten bei die $A^\star$-Suche im $\langle x, y, \theta, v, \omega \rangle$-Raum, während einer längeren Fahrt eines Roboters zu verschiedenen Zielpunkten. Links wurde eine Heuristik, basierend auf der *Value Iteration* im *Channel*, verwendet. Im rechten Bild wurde auf die euklidische Distanz kombiniert mit den minimale Feldkosten zurückgegriffen. Man sieht deutlich den Vorteil der optimierten Heuristik, bei der nur ein Bruchteil der Knoten expandiert werden muss.

6.2. Versuche in verschiedenen Umgebungen

In diesem Abschnitt werden die Fahrten der Roboter in verschiedenen Umgebungen untersucht. Zur Verfügung standen die in Abbildung 5.1 gezeigten Roboter `Ludwig` und `Albert`, sowie der in Abschnitt 5.1.6 vorgestellte Simulator der *BEE-Software*. Die wirklichen Roboterfahrten wurden in der Büroumgebung der Abteilung für Autonome Intelligente Systeme der Universität Freiburg, dargestellt in Abbildung 6.4, durchgeführt. Ein besonderes Augenmerk wurde auch auf Situationen gelegt, in denen der in Abschnitt 5.1.4 vorgestellte *Colli-Server* in Verbindung mit dem *Plan*-Modul Probleme aufweist.

Während der gesamten Versuchsreihe wurden die Roboter bei der Fahrt mit *Newplanner* mit den in Tabelle 6.1 angegeben Parametern gesteuert.

6.2.1. Roboterfahrten über größere Distanzen

Bei diesem Versuch wird das Verhalten eines Roboters in Standardsituationen einer Büroumgebung untersucht. Der Roboter fährt dazu mehrere hundert Meter durch bevölkerte Flure und Büroräume. Es befinden sich mehrere, nicht in die statische Karte eingetragene Hindernisse auf dem direkten Wegen des Roboters, außerdem bewegen sich Personen in dessen Umgebung. Abbildung 6.4 zeigt einen kleinen Ausschnitt aus der Trajektorie, die von dem Pioneer 1 Roboter `Ludwig` gefahren wurde. Die grau gezeichneten Hindernisse wurden dabei zur Laufzeit erkannt und umfahren. Die Bildsequenz aus Abbildung 6.5 wurde während dieser Fahrten aufgenommen.

In den mehrmals ausgeführten Versuchen, war die durch den Roboter zurückgelegte Distanz jeweils größer als $300\,m$ und er fuhr mit durchschnittlich $33\,cm/s$, bei einer maximal zugelassenen Geschwindigkeit von $40\,cm/s$. Die Versuche wurden auf beiden zur Verfügung stehenden Robotern ausgeführt und zeigten jeweils ähnlicher Ergebnisse. Während der gesamten Fahrt kam es zu keiner Kollision, weder mit einer Person, noch mit einem statischen Hindernis. Auch wurden ähnliche Versuche im Simulator ausgeführt, hier fuhr der virtuelle Roboter zufällig gewählte Zielpunkte an und legte dabei eine Strecke von über $20\,km$ in knapp 17 Stunden kollisionsfrei zurück, was einer ähnlichen Durchschnittsgeschwindigkeit entspricht.

In Abbildung 6.6 sieht man den Roboter `Ludwig` beim Umfahren mehrerer erkannter Hindernisse während eines solchen Versuches. Ähnliche Experimente wurden auch mit höheren Geschwindigkeiten von bis zu $70\,cm/s$, sowie in verschiedenen simulierten Umgebungen durchgeführt und waren stets kollisionsfrei. Ein Beispielpfad eines Simulatorexperiments ist in Abbildung 6.7 dargestellt.

6.2.2. Das Abbiegen in enge Korridore

Eine für Roboter oft zu bewältigende Aufgabe ist das Durchfahren von Türen und das Abbiegen aus einem Korridor in einen benachbarten Raum. Falls der Raum durch eine normale Tür mit einer Breite von ca. $80\,cm$ bis $100\,cm$ vom Korridor getrennt ist, haben viele Ansätze zur Kollisionsvermeidung Schwierigkeiten, diese Aufgabe gut zu lösen. Ein weit verbreiteter Ansatz ist der *Dynamic Window Approach* [FOX et al. 1997] in Verbindung mit einem Planungsmodul. Zur Ermittlung des besten Fahrkommandos wird dabei versucht, eine Bewertungsfunktion zu maximieren. Einer der Parameter dieser so genannten *Navigationsfunktion* ist die Translationsgeschwindigkeit. Der Roboters versucht dadurch, seine Geschwindigkeit zu erhöhen, um möglichst schnell zum Zielpunkt zu gelangen. In vielen Situationen funktioniert dieses Verfahren recht gut, allerdings fahren Roboter, die durch ein *Dynamic Window Approach* System kontrolliert werden, aufgrund einer zu hohen Geschwindigkeit oft an engen Abzweigungen vorbei und müssen einen weiteren Versuch unternehmen, diese zu durchfahren.

In unserem Ansatz dagegen wird der vollständige Konfigurationsraum aus Abschnitt 3.4 durchsucht. Dieser enthält auch die Geschwindigkeiten des Roboters, wodurch dieser rechtzeitig vor einer Abzweigung abbremst und schon beim ersten Versuch in beispielsweise einen benachbarten Raum abbiegen kann. In Abbildung 6.8 sieht man die Trajektorien eines Roboters beim Abbiegen: Das linke Bild zeigt den Pfad, den der Roboter mit Hilfe des bisherigen Navigationssystems, bestehend aus *Colli-Server* und *Plan*-Modul, gefahren ist. Man sieht deutlich, das der Roboter beim ersten Versuch an der Tür vorbeifährt, da er zu spät seine Geschwindigkeit reduziert. Im rechten Bild dagegen bremst der Roboter unter Verwendung von *Newplanner* rechtzeitig ab und kann dadurch problemlos abbiegen.

Abbildung 6.9 zeigt eine Sequenz von Bildern, in denen der Roboter `Albert` mit Hilfe von *Newplanner* aus einem Gang in einen seitlich gelegenen Raum abbiegt.

Parameter	**Werte**
Maximale Translationsgeschwindigkeit	$40\ cm/s$ bzw. $70\ cm/s$
Maximale Rotationsgeschwindigkeit	$45°/s$
Translationsbeschleunigung	$20\ cm/s^2$
Translationsverzögerung	$20\ cm/s^2$
Rotationsbeschleunigung	$22.5°/s^2$
Rotationsverzögerung	$22.5°/s^2$

Tabelle 6.1: Die aufgeführten Parameter wurden bei den Fahrten mit *Newplanner* in den folgenden Versuchen verwendet. Die Translationsgeschwindigkeit betrug meist $40\ cm/s$, in einigen Versuchen wurde sie auf $70\ cm/s$ erhöht.

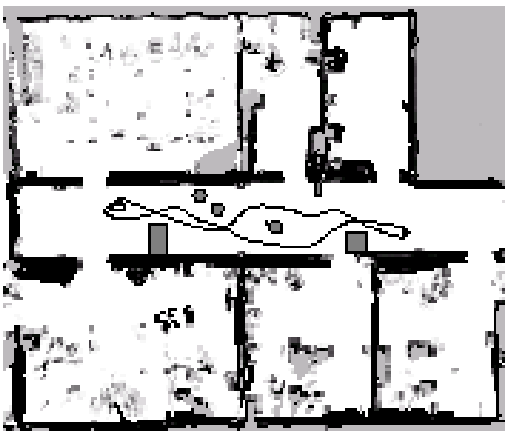

Abbildung 6.4: Ein Teilstück eines langen Pfades, auf dem sich der Roboter `Ludwig` während eines Experimentes in einer Büroumgebung bewegt hat.

Abbildung 6.5: Die Bildsequenz zeigt den Roboter `Ludwig` während der Fahrt innerhalb einer Büroumgebung. Neben unerwarteten statischen Hindernissen bewegten sich auch Menschen in der Umgebung des Roboters.

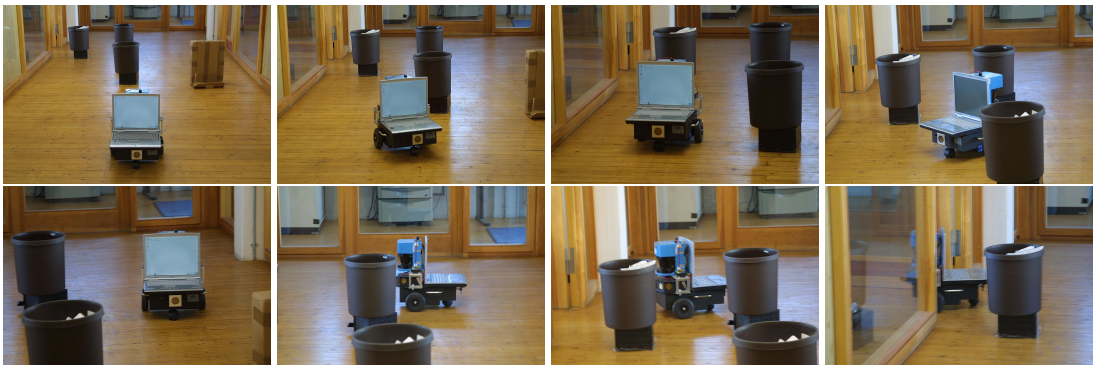

Abbildung 6.6: Die Bildsequenz zeigt den Roboter `Ludwig` beim Umfahren mehrerer Hindernisse.

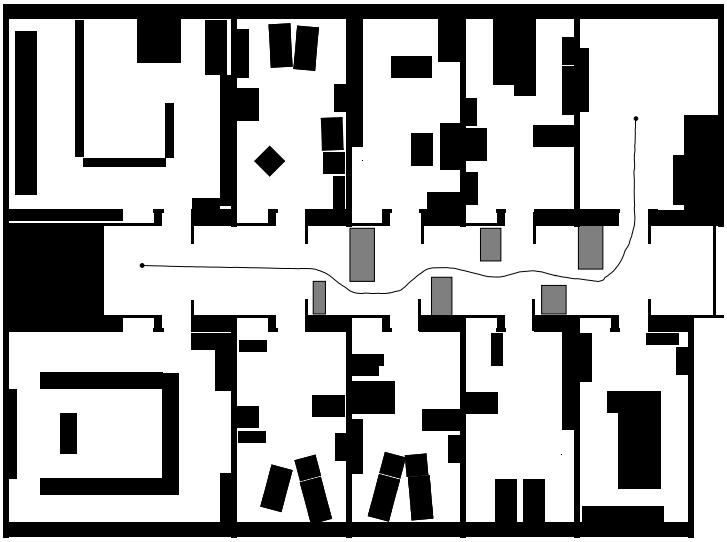

Abbildung 6.7: Hier sieht man die Trajektorie einer Roboterfahrt im Simulator, wieder wurden die grauen Hindernisse zur Laufzeit erkannt und umfahren.

Abbildung 6.8: Im linken Bild verwendet der Roboter das bisherige Navigationssystem bestehend aus *Colli-Server* und *Plan*-Modul. Man sieht deutlich, dass dieser beim ersten Versuch die Abzweigung am Ende des Ganges verpasst und wenden muss. Im rechten Bild bremst der Roboter dank *Newplanner* rechtzeitig ab und kann schon beim ersten Anlauf in den benachbarten Raum abbiegen.

Abbildung 6.9: Der Roboter `Albert` beim Abbiegen aus einem Korridor durch eine enge Tür in einen benachbarten Raum.

Die Fahrt des Roboters `Albert` aus Abbildung 6.8 wird im Folgenden weiter untersucht. Dazu wurden zwei Zeiten gemessen: Einmal die, die der Roboter benötigt, um das erste, gerade Stück abzufahren und zweitens die Zeit, um von dort aus durch die Tür abzubiegen. Dieser Versuch wurde 50 mal wiederholt und die mittleren Zeiten sind in Abbildung 6.10 aufgeführt. Die Fehlerbalken, die bei den ersten drei Messwerten nicht sichtbar sind, zeigen die $\alpha = 0.05$ Signifikanz. Die beiden ersten Balken zeigen, dass *Newplanner* wie auch das bisherige Navigationssystem zum Abfahren einer geraden Strecke in etwa die gleiche Zeit benötigen, d.h. Parameter wie maximale Geschwindigkeit und Beschleunigungsverhalten identisch gesetzt wurden. Die Analyse der zweiten Strecke, dargestellt im dritten und vierten Balken, zeigt dagegen deutliche Unterschiede. Der Roboter kann die Aufgabe mittels *Newplanner* signifikant schneller lösen, verglichen mit der Fahrt unter Verwendung von *Colli-Server* und *Plan*-Modul. Auch zeigt die Abbildung, dass *Newplanner* immer nahezu die gleiche Zeit benötigt, die des bisherigen Systems dagegen schwankt erheblich. Dies liegt daran, dass der Roboter dort die Tür selten beim ersten, meistens beim zweiten, gelegentlich aber auch erst beim dritten Anlauf durchfahren kann.

Es wurden ähnliche Versuche in anderen Umgebungen durchgeführt, der Roboter musste dabei jeweils zwei Türen durchfahren, um zu seinem Zielpunkt zu gelangen. Allerdings waren in dieser Umgebung die zu durchfahrenden Türen noch enger als im vorherigen Versuch, so dass einige Beschleunigungsparameter des *Colli-Server* nach oben korrigiert werden mussten, damit ein Durchfahren der Türen überhaupt möglich wurde. Ansonsten drehte der Roboter - aufgrund einer befürchteten Kollision - bei, bevor die Tür durchfahren werden konnte. Die gewählten Parameter sind in Tabelle 6.2, die Mittelwerte der Fahrtzeiten in Abbildung 6.11 dargestellt. Wieder zeigen die Fehlerbalken die $\alpha = 0.05$ Signifikanz und die Ergebnisse zeigen ein ähnliches Verhalten des Roboters wie in den vorherigen Versuchen. Dadurch zeigt sich, dass ein durch *Newplanner* gesteuerter Roboter deutlich schneller zu seinem gewünschten Zielpunkt gelangt, falls dafür Türen und enge Passagen durchfahren werden müssen. Falls der optimale Weg zum Ziel hauptsächlich aus einer geraden Strecke besteht, sind die Fahrtzeiten der beiden Ansätze dagegen identisch. Es sei nochmal darauf hingewiesen, dass *Newplanner* bessere Ergebnisse erzielt, obwohl er mit deutlich stärkeren kinematischen Einschränkungen seine Planung durchgeführt hat.

6.2.3. Die Pfadplanung in einem extrem engen Korridor

Wie bereits in Abschnitt 6.2.2 angesprochen, zeigen Ansätze zur Kollisionsvermeidung, die auf dem *Dynamic Window Approach* basieren, oft Nachteile beim Abbiegen in enge Korridore. Besonders deutlich wird dies, wenn die Breite der Gänge nur geringfügig größer ist als der Durchmesser des Roboters selbst. Eine solche Situation wurde bereits in Abbildung 2.1 gezeigt. Dort ist es mit *Dynamic Window Approach*-Methoden kaum möglich den Zielpunkt zu erreichen, da eine Vorausschau im Raum der Geschwindigkeiten fehlt und der Roboter so die Abzweigung verpasst. Dieses Manko zeigt auch der ansonsten elegante *Global Dynamic Window Approach* von Brock und Khatib [BROCK und KHATIB 1999].

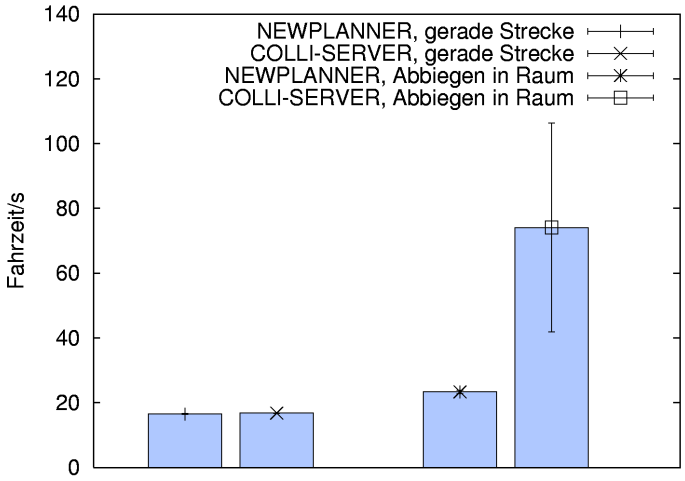

Abbildung 6.10: Die Mittelwerte der benötigten Fahrzeiten für die Trajektorien aus Abbildung 6.8. Einmal beim Befahren des ersten gerade Stückes der Trajektorie in der Mitte des Ganges (links) und beim Abbiegen in den benachbarten Raum (rechts), jeweils unter Verwendung von *Newplanner* und *Colli-Server*.

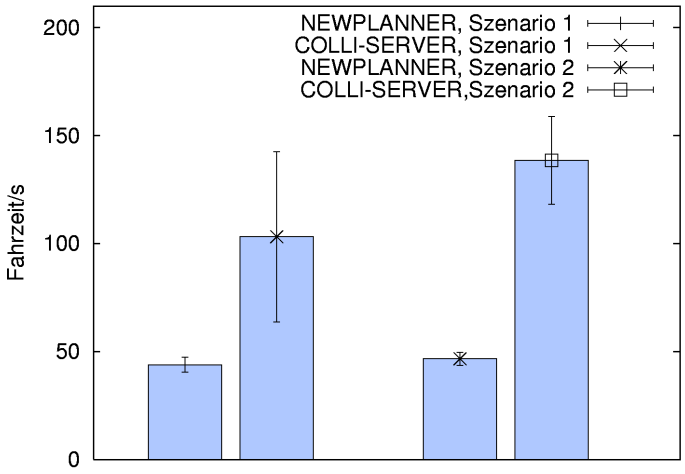

Abbildung 6.11: Wie in Abbildung 6.10 sind die Mittelwerte der benötigten Fahrtzeiten aufgetragen. Es wurden zwei verschiedene Szenarien untersucht, jeweils musste der Roboter zwei Türen passieren. Beide Versuchen wurden 50 mal mit Hilfe des Simulators wiederholt.

Parameter	***Newplanner***	***Colli-Server***
Maximale Translationsgeschwindigkeit	$70\ cm/s$	$70\ cm/s$
Maximale Rotationsgeschwindigkeit	$45°/s$	$45°/s$
Translationsbeschleunigung	$20\ cm/s^2$	$25\ cm/s^2$
Translationsverzögerung	$20\ cm/s^2$	$30\ cm/s^2$
Rotationsbeschleunigung	$22.5°/s^2$	$35°/s^2$
Rotationsverzögerung	$22.5°/s^2$	$35°/s^2$

Tabelle 6.2: Die Parameter des Versuches dargestellt in Abbildung 6.11. Dabei mussten einige Parameter des *Colli-Server* angepasst werden, da der Roboter sonst nicht durch die engen Türen fahren konnte. Selbst mit stärkeren kinematischen Einschränkungen erzielte *Newplanner* noch deutlich bessere Ergebnisse als das bisherige Navigationssystem aus *Colli-Server* und *Plan*-Modul.

Durch *Newplanner* dagegen reduziert der Roboter rechtzeitig seine Geschwindigkeit und kann dadurch das Feld c_* und damit auch den Zielpunkt erreichen. Abbildung 6.12 zeigt die Fahrt eines Roboters im Simulator unter Verwendung von *Newplanner*.

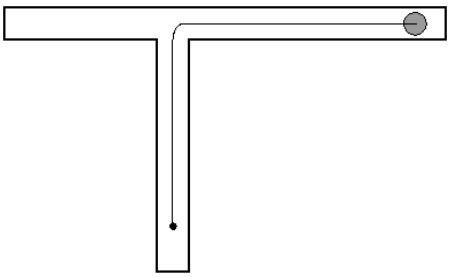

Abbildung 6.12: Wie dieser Simulationsversuch zeigt, kann ein Roboter mittels *Newplanner*, durch rechtzeitiges Reduzieren seiner Geschwindigkeit, in den nach Süden zeigenden Korridor abbiegen.

6.2.4. Lokale Minima des *Dynamic Window Approach*

Ein weiteres Problem des *Dynamic Window Approach* sind lokale Minima, aus denen sich der Roboter nicht mehr befreien kann. Diese können beispielsweise durch dynamische Hindernisse entstehen, die eine Tür teilweise versperren. Ein solche Situation ist in Abbildung 6.13 zu sehen. Der Roboter umfährt im linken Bild die Hindernisse vor der Tür, ist aber zu schnell, um zwischen ihnen in Richtung Tür abzubiegen. Hinter den Hindernissen gerät er dann durch seine *Navigationsfunktion* in ein lokales Minimum. Das rechte Bild dagegen zeigt den Pfad, der mit *Newplanner* zurückgelegt wurde. Durch die Betrachtung des Geschwindigkeitsraums kann der Roboter rechtzeitig bremsen und zwischen den Hindernissen hindurch in den Raum gelangen.

Eine Bildsequenz der Fahrt des Roboters `Albert` in dieser Umgebung findet sich in Abbildung 6.14.

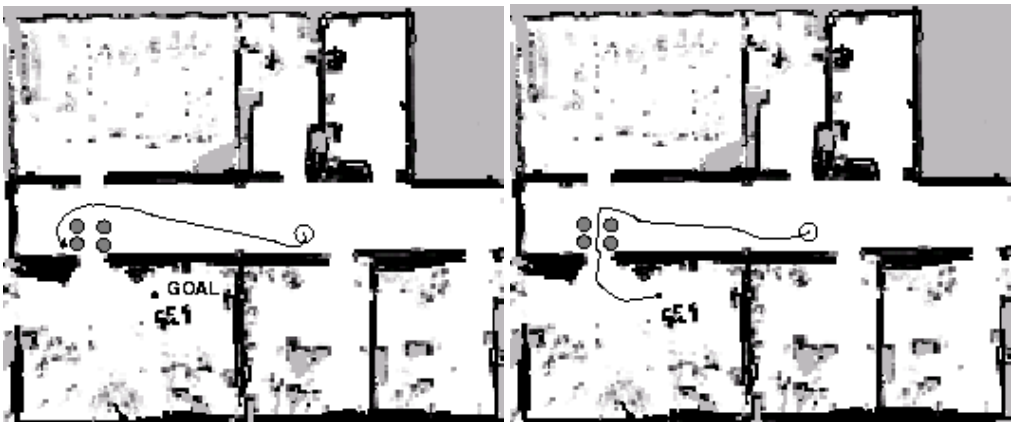

Abbildung 6.13: Im linken Bild sieht man den Pfad des Roboters `Albert` in Verbindung mit dem *Colli-Server*. Dieser umfährt die Hindernisse aufrund seiner hohen Geschwindigkeit und gerät dann in ein lokales Minimum. Im rechten Bild findet *Newplanner* einen Weg zum Ziel durch rechtzeitiges reduzieren seiner Geschwindigkeit. Die passende Bildsequenz der Fahrt findet sich in Abbildung 6.14.

6.3. Die Raumdiskretisierung

In den bisherigen Versuchen wurde immer eine Raumdiskretisierung von $10\,cm \times 10\,cm$ verwendet, d.h. die in Abschnitt 3.3 beschriebenen Quadrate zur Raumaufteilung haben eine Kantenlänge von $10\,cm$. Je kleiner man diese Diskretisierung wählt, desto mehr Zustände entstehen, die bei der $A^\star$-Suche betrachtet werden müssen, d.h. umso leistungsfähiger muss der zu Grunde liegende Computer sein. Bei der Planung der Trajektorie wird immer davon ausgegangen, dass sich der Roboter in der Mitte eines solchen Quadrates befindet, obwohl der Roboter eventuell an einer geringfügig anderen Stelle steht. Der maximale Diskretisierungsfehler ist direkt von der Größe der Quadrate abhängig. Je kleiner die Diskretisierung gewählt wird, desto kleiner ist somit auch der Fehler in der Position des Roboters, was besonders in engen Passagen von Vorteil sein kann. Dagegen wird bei einer gröberen Diskretisierung weniger Rechnenleistung benötigt, allerdings kann der Roboter Probleme in engen Passagen bekommen, da seine Position innerhalb der Umgebung bei der $A^\star$-Suche auf Grund des Diskretisierungsfehlers zu ungenau bestimmt ist.

Ab einer Diskretisierung vom $30\,cm$ ist es nicht mehr möglich durch enge Türen unserer Umgebung zu fahren, wie beispielsweise noch bei einer Auflösung von $10\,cm$ oder $20\,cm$. Durch den Diskretisierungsfehler ist die Position des Roboters in der Planung dann zu ungenau bestimmt.

Abbildung 6.14: Der Roboter `Albert` fährt den in Abbildung 6.13 dargestellten Pfad mit *Newplanner* ab und erreicht somit den Zielpunkt.

Bei einer Auflösungen von $20\ cm$ ist eine Fahrt zwar noch möglich, allerdings in engen Passagen nicht so flüssig wie bei der Standardauflösung. Wegen der groben Diskretisierung sind die Kosten benachbarter Felder in engen Passagen sehr unterschiedlich. Falls der Roboter dann auf Grund eines Diskretisierungsfehlers in ein anderes als das erwartete Feld gelangt, bremst dieser ab und korrigiert seine Position.

Wenn die Auflösung dagegen heraufgesetzt wird, wird der Fehler in der Position des Roboters nochmals kleiner. Wie bei der Standardauflösung navigiert der Roboter bei einer Auflösung von $5\ cm$ sicher durch enge Passagen. Signifikante Unterschiede in der Fahrsicherheit konnten wir jedoch nicht feststellen. Allerdings steigt die benötigte Rechenleistung deutlich an. Auf einem Standard-Laptop ist das Ausführen der Berechnungen dann kaum noch möglich und es kommt zu vielen Zeitüberschreitungen während der $A^\star$-Suche. Auf leistungsfähigeren Computern sind die Berechnungen aber ausführbar und auch der *Channel* besitzt noch eine sinnvolle Größe. Abbildung 6.15 zeigt eine geplante Trajektorie mit einer $5\ cm$ Diskretisierung.

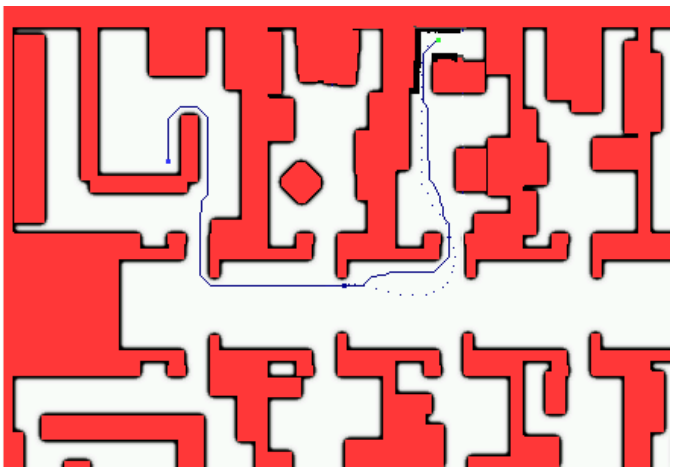

Abbildung 6.15: Hier sieht man eine Trajektorie, die bei einer Diskretisierung der Umgebung von $5\ cm$ geplant wurde. Der gepunktete Pfad zeigt die geplante Trajektorie zum Zwischenzielpunkt basierend auf der $\langle x, y, \theta, v, \omega \rangle$-$A^\star$-Suche, die durchgezogene Linie den Pfad des $\langle x, y \rangle$-$A^\star$zum Zielpunkt.

Die gewählte Auflösung von $10\ cm$ ist also eine sinnvolle Größe, d.h. der Roboter navigiert sicher durch Türen und Korridore und gleichzeitig ist die Leistung eines Standard-Laptops ausreichend, um die notwendigen Berechnungen durchzuführen.

6.4. Zeitüberschreitungen in Abhängigkeit des Ortes

Wie in Abschnitt 4.7 beschrieben, verfügt *Newplanner* über die Möglichkeit so genannte *Timeout-Karten* zu erstellen und zu erweitern, um so Positionen, an denen es oft zu einer Zeitüberschreitung kommt, zu ermitteln. In unseren Experimenten konnten wir kein System erkennen, an denen man solche Positionen im Vorhinein ausmachen könnte. Es fällt jedoch auf, dass sich einige dieser Positionen in der Nähe von Türen befinden. Jedoch scheint dies nicht das einzige Kriterium zu sein. In Abbildung 6.16 sieht man eine Karte der Umgebung des Roboters. Die blau gezeichneten Stellen sind Positionen, an denen bei der Berechnung der Trajektorien eine Zeitüberschreitung vorlag.

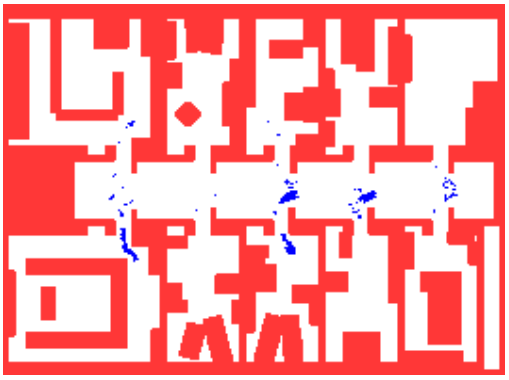

Abbildung 6.16: Diese Karte zeigt Zeitüberschreitungen bei der Berechnung der Trajektorie in Abhängigkeit von der Position des Roboters. Die blaue Punkten markieren Stellen, an denen eine Zeitüberschreitung auftrat.

Zur Beurteilung des Verhaltens des Roboters mit und ohne *Timeout-Karte* fuhr ein Roboter 14 Zielpunkte hintereinander an und der Versuch wurde mehrfach mit beiden Verfahren wiederholt. Dabei wurden Wegstrecke und Zeit, sowie die Anzahl der Zeitüberschreitungen protokolliert. Die benötigte Fahrzeit lag bei beiden Verfahren relativ dicht beieinander, mit *Timeout-Karte* benötigte der Roboter geringfügig weniger Zeit, der Unterschied lag aber nur im Bereich von 1%. Die Anzahl der Zeitüberschreitungen konnte mit der neuen Technik allerdings deutlich reduziert werden. Sie lag unter Verwendung der *Timeout-Karte* bei durchschnittlich 0.5% der Berechnungen, ohne Karte traten diese in 0.9% der Fälle auf. In Abbildung 6.17 sieht man ein Histogramm, welches die Anzahl der Zeitüberschreitungen im Laufe des Experimentes zeigt, einmal mit und einmal ohne die *Timeout-Karte*.

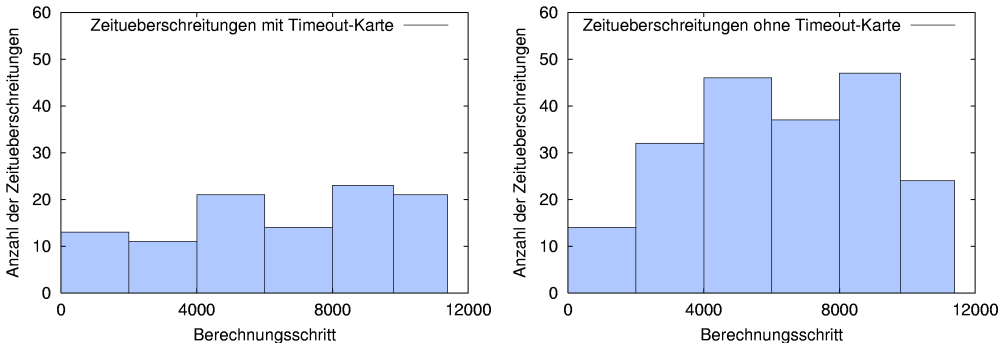

Abbildung 6.17: Dieses Histogramm zeigt die Anzahl der Zeitüberschreitungen während einer Roboterfahrt zu 70 Zielpunkten, die nacheinander angefahren wurde. In der linken Abbildung wurde auf eine gelernte *Timeout-Karte* zurückgegriffen, rechts wurde sie nicht verwendet.

6.5. Zeitüberschreitungen in Abhängigkeit der Pfadkosten

Jeder $A^\star$-Planung im vollen Konfigurationsraum geht eine Suche im $\langle x, y\rangle$-Raum voraus. Daher ist es sinnvoll das Wissen aus diesen Berechnungen für die Planung im fünfdimensionalen Raum zu nutzen. Ein Ergebnis, neben dem Pfad selbst, sind dabei die Kosten der einzelnen Felder auf dem Weg vom Startelement zum Zielelement. Bei der $A^\star$-Suche wird immer versucht den optimalen Pfad zu finden, daher werden zuerst die günstigsten Felder des Suchraums exploriert. Je teurer das Betreten eines Feldes und dessen vermutete weitere Kosten sind, desto später wird es betrachtet. Daher kann es passieren, dass durch eine nicht optimale Heuristik zuerst einige später nicht am Pfad beteiligte Felder durchsucht werden, da diese anfangs günstiger erscheinen.

Daher wurde die Wahrscheinlichkeit für das Auftreten einer Zeitüberschreitung in Abhängigkeit der Feldkosten des Pfades aus der $\langle x, y\rangle$-$A^\star$-Suche untersucht. Um unabhängig von der Länge einer Trajektorie zu sein, wurden die durchschnittlichen Kosten eines Feldes betrachtet. In Abbildung 6.18 sieht man ein Histogramm, welches die Wahrscheinlichkeit für eine Zeitüberschreitung bei der Suche im $\langle x, y, \theta, v, \omega\rangle$-Raum in Abhängigkeit der durchschnittlichen Kosten eines Feldes auf dem optimalen Pfad aus der $\langle x, y\rangle$-$A^\star$-Suche zeigt. Die Daten stammen aus einer simulierten Roboterfahrt während der $11\,228$ Berechnungen ausgeführt wurden und 103 Zeitüberschreitungen auftraten.

Dieses Experiment zeigt deutlich, dass Zeitüberschreitungen bei wachsenden Feldkosten entlang eines Pfades wahrscheinlicher werden. Daher macht es Sinn, bei der Bestimmung der Größe des *Channel* in Abschnitt 4.2.5, die zuvor bestimmten durchschnittlichen Kosten der Felder des Pfades aus der $\langle x, y\rangle$-$A^\star$-Suche zu berücksichtigen. Falls die Kosten einen bestimmen Wert überschreiten, kann der *Channel* verkleinert werden, um so den Aufwand für die Suche zu reduzieren und Zeitüberschreitungen unwahrscheinlicher werden zu lassen.

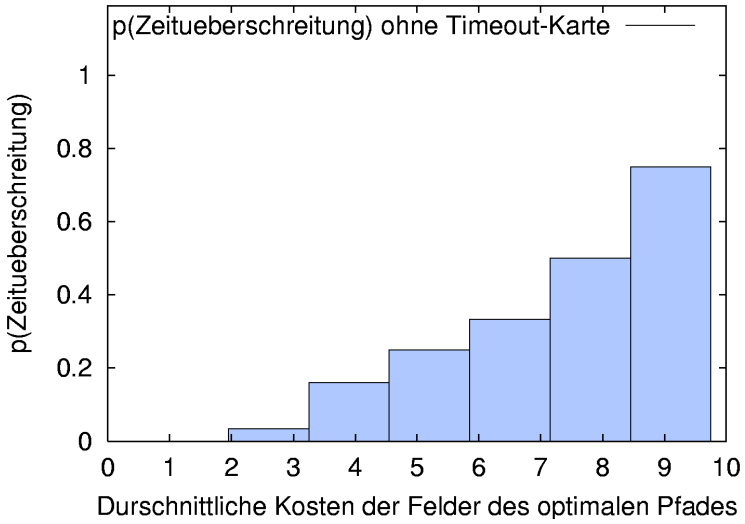

Abbildung 6.18: Dieses Histogramm zeigt die Wahrscheinlichkeit für eine Zeitüberschreitung bei der Suche im $\langle x, y, \theta, v, \omega \rangle$-Raum in Abhängigkeit der durchschnittlichen Kosten eines Feldes auf dem optimalen Pfad aus der $\langle x, y \rangle$-$A^\star$-Suche.

6.6. Die Größe des *Channel* und der Vergleich mit der optimalen Lösung

In den nun folgenden Versuchen haben wir den Einfluss der Größe des *Channel* auf die Fahrzeit untersucht. Dazu wurden verschiedene Versuche durchgeführt, die jeweils einen unterschiedlich langen und breiten *Channel* verwendeten. Die meisten dieser Experimente mussten offline durchgeführt werden, da die derzeitig zur Verfügung stehende Rechenleistung nicht ausreichte, um das Suchproblem innerhalb des gegeben Zeitfensters von $250\,ms$ zu lösen.

Zusätzlich haben wir auch Vergleiche mit der theoretisch optimalen Lösung angestellt, gegeben die modellierten physikalischen Eigenschaften des Roboters und die Diskretisierung der Welt (siehe Tabelle 4.1). Zur Berechnung des Optimums wird keine Einschränkung des vollen Konfigurationsraums vorgenommen, d.h. der *Channel* erstreckt sich über den gesamten Raum. Zusätzlich besitzt keiner der verwendeten Algorithmen eine zeitliche Beschränkung.

Neben der optimalen Trajektorie wurden in den Experimenten Pfade untersucht, die mit unterschiedlichen Parametern berechnet wurden. Der *Channel* hatte eine Länge von $1.5\,m$, $2.5\,m$ und $5\,m$ mit jeweils einer Breite von $70\,cm$, $90\,cm$ und $1.10\,m$. Bei diesen Versuchen musste ein Roboter verschiedene Zielpunkte hintereinander anfahren. Da die Trajektorien offline berechnet werden mussten, sind die Ergebnisse jeweils reproduzierbar. Kleine Unsicherheiten und

Störungen, beispielsweise in der Position des Roboters, oder zeitkritische Aktionen, wie sie bei realen Fahrten auftreten, existierten daher bei diesem Versuch nicht. Die im Folgenden präsentierten Ergebnisse wurden mit den in Tabelle 6.1 dargestellten kinematischen Einschränkungen ermittelt.

In den Abbildungen 6.19 bis 6.21 sind die absoluten Fahrzeiten dieser Versuche gegeneinander aufgetragen und zusätzlich der relative Mehraufwand gegenüber der optimalen Lösung.

Man sieht deutlich, dass die Güte der Lösung mit der Größe des *Channel* wächst und, wie Abbildung 6.21 zeigt, bereits ab einer Länge von $5\,m$ und einer Breite von $90\,cm$ mit der optimalen Lösung identisch ist. Dies bedeutet, dass die optimale Trajektorie relativ dicht am Pfad der zweidimensionalen Suche liegt, wir also eine gute Einschränkung des vollen Konfigurationsraums vorgenommen haben. Auf einem Standard-Laptop ist ein *Channel* dieser Größe jedoch nicht klein genug, um die Suche unterhalb der $250\,ms$-Grenze auszuführen. Allerdings wird die Leistung der Computer in den nächsten ein bis zwei Jahren vermutlich ausreichend sein, um diese Berechnungen in dem gegebenen Zeitfenster ausführen zu können und einen Roboter so auf dem optimalen Pfad zu seinem Zielpunkt zu führen.

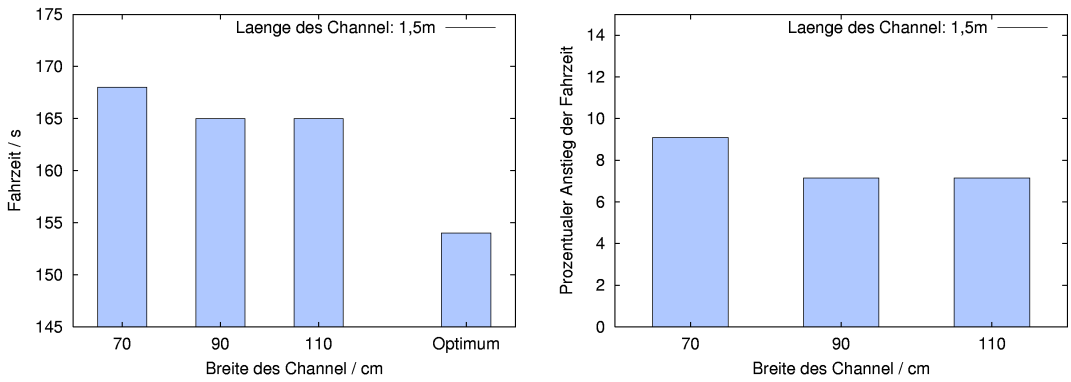

Abbildung 6.19: Vergleich der Fahrzeiten berechneter Trajektorien bei einer *Channel*-Länge von $1.5\,m$ und einer Breite von $70\,cm$, $90\,cm$ und $1.1\,m$, sowie mit der optimalen Lösung. Links sind die absoluten Fahrzeiten der einzelnen Versuche zu sehen und rechts der relative Mehraufwand gegenüber dem Optimum.

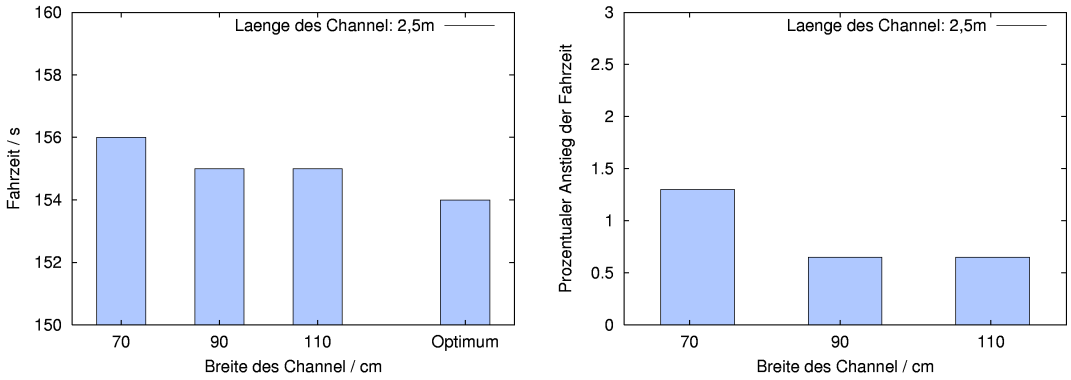

Abbildung 6.20: Entsprechende Ergebnisse mit einer *Channel*-Länge von $2.5\ m$.

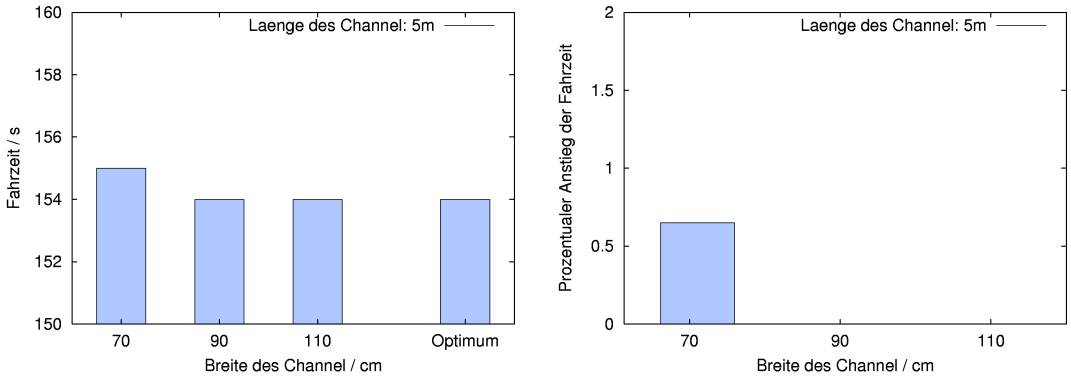

Abbildung 6.21: Vergleich der Fahrzeiten berechneter Trajektorien bei einer *Channel*-Länge von $5\ m$ und einer Breite von $70\ cm$, $90\ cm$ und $1.1\ m$ sowie mit der optimalen Lösung. Oben sind die absoluten Fahrzeiten der einzelnen Versuche zu sehen und unten der relative Mehraufwand gegenüber dem Optimum. Hier zeigt sich, dass ab einer Länge von $5\ m$ und einer Breite von $90\ cm$ die Lösung identisch mit der optimalen Lösung ist.

7. Zusammenfassung

In dieser Arbeit haben wir einen Ansatz zur zielgerichteten und sensorbasierten Kollisionsvermeidung für mobile Roboter vorgestellt. Im Gegensatz zu anderen gängigen Verfahren plant unser System im $\langle x, y, \theta, v, \omega \rangle$-Konfigurationsraum und berücksichtigt daher auch die dynamischen Eigenschaften des Roboters, wie das Brems- und Beschleunigungsverhalten. Der Vorteil dieses Vorgehens ist eine vorausschauende Planung im Raum der Geschwindigkeiten des Roboters, wodurch dieser beispielsweise vor engen Abzweigungen seine Fahrt verlangsamt, um sicher abbiegen zu können. Des weiteren ist unser Ansatz durch das Verwenden der $A^\star$-Suche frei von lokalen Minima.

Unser Algorithmus beinhaltet verschiedene Vorgehensweisen, um den Suchraum geeignet einzuschränken und dadurch mit der Komplexität des beschriebenen Suchproblems umgehen zu können. Dadurch ist unser Verfahren in Echtzeit auf einem Standard-Laptop ausführbar. Es wurde auf verschiedenen Roboterplattformen mit verschiedenen kinematischen Eigenschaften getestet. In allen Fällen wurden kollisionsfreie Trajektorien berechnet und ausgeführt, auch wenn unerwartete und sich bewegende Hindernisse den zuvor berechneten Pfad des Roboters blockierten.

Zusätzlich kann sich das System an die zur Verfügung gestellte Rechenleistung der Hardware adaptieren und wird dadurch mit den Rechnern der nächsten Jahre noch bessere Ergebnisse erzielen können als heute. Doch auch mit aktuellen Computersystemen liegen die benötigten Fahrzeiten des Roboters in Experimenten schon relativ nah an der optimalen Lösung, d.h. die Trajektorie des Roboters ist kollisionsfrei und die Fahrzeit kann nur noch im einstelligen Prozentbereich gegenüber dem Optimum verbessert werden.

Zusätzlich wurden Vergleiche mit dem weit verbreiteten *Dynamic Window Approach* in Verbindung mit einem Planungsmodul gezogen. Dabei zeigte sich ein deutlicher Vorteil unseres Verfahrens, sobald das Durchfahren enger Passagen wie Türen nötig war. Beim Abfahren gerader Strecken erzielten beide Ansätze identische Ergebnisse.

Wie in diversen Experimenten gezeigt, haben wir mit dieser Arbeit ein praxistaugliches Verfahren zur Steuerung autonomer Roboter vorgestellt. Auch in realen und von Menschen bevölkerten Umgebungen realisiert unser Ansatz eine effektive Pfadplanung und navigiert einen mobilen Roboter kollisionsfrei zum gewünschten Zielpunkt.

7.1. Ausblick

An dieser Stelle möchten wir einige Punkte unserer Arbeit erwähnen, die man noch verbessern könnte oder um die sich unser derzeitiger Ansatz erweitern ließe.

Einer der zentralen Punkte unseres Verfahrens ist eine gute Einschränkung des vollen Konfigurationsraums, um mit der Größe des Suchraums umgehen zu können. Derzeit wird bei der Adaption der Länge des *Channel* auf die *Timeout-Karte*, die durchschnittlichen Feldkosten und die Information über eine vorhergehende Zeitüberschreitung zurückgegriffen. Eventuell könnte man hier noch weitere Kriterien finden, anhand derer der Aufwand der Suche im Vorhinein besser abschätzbar wird. Bei ausreichend vorhandener Rechenzeit könnte man dann die Diskretisierungen, beispielsweise der Translationsgeschwindigkeit, verfeinern und das Verfahren so weiter optimieren.

Falls ein Roboter aufgrund eines blockierten Pfades sein Ziel nicht erreichen kann, wird nach einer gewissen Zeit des Wartens die Planung abgebrochen. Hier lassen sich zusätzlich andere Strategien in *Newplanner* integrieren, so dass der Roboter - je nach Situation - unterschiedlich auf ein solches Ereignis reagieren kann. Dies könnte, je nach Art der Aufgabe, beispielsweise das spätere Anfahren des Zielpunktes sein. Da unsere Software den Abbruch der Planung allerdings über Nachrichten anderen Modulen mitteilt, kann diese Aufgabe auch durch eine externe Komponente, d.h. ohne Veränderung von *Newplanner*, realisiert werden.

Was unser Ansatz in der aktuellen Version nicht berücksichtigt, ist der Umgang mit Unsicherheiten, beispielsweise in der Lokalisierung des Roboters. Somit ist das Verfahren vom Wissen über die aktuelle Position des Roboters innerhalb seiner Umgebung von einem Lokalisierungsmodul abhängig. Bei einer massiven Fehllokalisierung des Roboters ist eine zuverlässige Fahrtplanung und Kollisionsvermeidung mit unserem Ansatz derzeit nicht möglich. Von Fox et al. existiert ein Ansatz der innerhalb des *Dynamic Window Approach* eine Kollisionsvermeidung mit Unsicherheiten erlaubt [FOX et al. 1998]. So kann beispielsweise auch bei einer Unsicherheit in der Lokalisierung, bedingt durch eine teilweise symmetrische Umgebung, eine kollisionsfreie Fahrt in vielen Fällen gewährleistet werden. Grundlegende Ideen zum Umgang mit Unsicherheiten existieren auch bei Schmidt und Azarm [SCHMIDT und AZARM 1992]. Man könnte unseren Ansatz in der Weise erweitern, dass auch in einigen dieser Fälle eine sichere und kollisionsfreie Fahrt des Roboters gewährleistet werden kann. Hierzu könnte man ähnlich wie bei Fox et al. bei Unsicherheiten in der Lokalisierung zusätzliche Hindernisse in die Umgebungskarte eintragen.

Ein weiteres Problem, das bei mobilen Robotern in der realen Welt auftritt, ist eine Unsicherheit im Rahmen der Befehlsausführung. Falls der Roboter den Befehl erhält, sich beispielsweise um den Winkel θ zu drehen, kann es sein, dass er sich nur um $\theta - \epsilon$ dreht. Dies kann in Extremsituationen Auswirkungen auf die Fahrsicherheit des Roboters bzw. auf die Länge des geplanten Pfades haben. Bei diesem Problem handelt es sich um ein so genanntes *Markov Decision Problem* [SUTTON und BARTO 1998]. Es existieren Lösungen, um diese Unsicherheiten in der Planung zu berücksichtigen und die Fahrt des Roboters so robuster zu machen. Jedoch ist dies

nicht trivial, insbesondere die benötigte Rechenleistung steigt stark an.

Falls Unsicherheiten bei der Aktionsausführung und in der Lokalisierung des Roboters zusammenkommen, hat man es mit einem so genannten *Partially Observable Markov Decision Problem (POMDP)* [CASSANDRA 1998] zu tun. Die Handhabung eines *POMDP* ist recht komplex und vor allem extrem zeitaufwendig, so dass es schwierig werden dürfte, diese Aspekte in die Online-Planung zu integrieren [KAELBLING et al. 1998]. Das folgende Beispiel könnte allerdings als eine Motivation für die Berücksichtigung der Unsicherheiten dienen:

Man stelle sich eine Umgebung mit zwei benachbarten Türen vor, die jeweils in nahezu identisch aussehende Korridore münden. Eine dieser Türen führt zum gewünschten Zielpunkt, über die andere ist dieser nicht zu erreichen. Aufgrund der Ähnlichkeit beider Korridore kann ein Roboter nicht ohne weiteres unterscheiden, in welchem genau er sich befindet. Ein solches Szenario macht deutlich, wie wichtig es sein kann, die richtige Tür zu durchfahren. Die Wahrscheinlichkeit, die falsche Tür zu durchfahren, sollte möglichst gering gehalten werden.

Es gibt Näherungen für das Planen von Trajektorien unter Berücksichtigung der *POMDP* Eigenschaften. Dabei wird u.a. versucht die Unsicherheiten in einer einzigen zusätzlichen Dimension zusammenzufassen [ROY et al. 1999, ROY und THRUN 1999]. Mit Hilfe ähnlicher Ansätze wäre es eventuell auch in unserem Verfahren möglich mit Unsicherheiten umzugehen.

A. Technische Daten

A.1. Der Roboter `Albert`

Typ: B21r von RWI

(a) Der Roboter `Albert`

Parameter	Wert
Anzahl Sonarsensoren	48
Anzahl Infrarotsensoren	24
Anzahl taktile Sensoren	56
CPU	Pentium
Translationsgeschwindigkeit	max. $90\,cm/s$
Rotationsgeschwindigkeit	max. $167\,^\circ/s$
Antrieb	4-Rad Synchro-Drive
Ladegewicht	$90\,kg$
Durchmesser	$52,5\,cm$
Höhe	$106\,cm$
Gewicht	$122,5\,kg$

(b) Die Daten

Abbildung A.1: Der mobile Roboter `Albert`. Sämtliche Daten sind Angaben des Herstellers.

A.2. Der Roboter Ludwig

Typ: Pioneer1 von ActiveMedia

(a) Der Roboter Ludwig

Parameter	Wert
Anzahl Sonarsensoren	8
CPU	Pentium Laptop
Translationsgeschwindigkeit*	max. ca. 60 cm/s
Rotationsgeschwindigkeit*	max. ca. 190° $/s$
Antrieb	Differentialantrieb
Länge	45 cm
Breite	36 cm
Höhe (mit Laserscanner)*	ca. 46 cm

(b) Die Daten

Abbildung A.2: Der mobile Roboter Ludwig. Die mit * gekennzeichneten Daten wurden selbst ermittelt, alle anderen Daten sind Angaben des Herstellers.

A.3. Der Laserscanner

Typ: PLS der Firma SICK

(a) Der Laserscanner PLS der Firma SICK

Parameter	Wert
Abmessungen	$155\,mm \times 185\,mm \times 156\,mm$
Ansprechzeit	$80\,ms$
Auflösung	$70\,mm$ (in $4\,m$ Entfernung)
Reichweite	max. $50\,m$
Öffnungswinkel	bis zu $180°$
Versorgungsspannung	$24\,V$
Gewicht	ca. $4.5\,kg$

(b) Die Daten

Abbildung A.3: Der Laserscanner PLS der Firma SICK. Daten sind Angaben des Herstellers.

Abbildungsverzeichnis

Index

Literaturverzeichnis

[ARRAS et al. 2002] ARRAS, K.O., J. PERSSON, N. TOMATIS und R. SIEGWART (2002). *Real-Time Obstacle Avoidance For Polygonal Robots With A Reduced Dynamic Window*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[BALCH und HYBINETTE 2000] BALCH, T. und M. HYBINETTE (2000). *Social potentials for scalable multi-robot formations*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[BENNEWITZ et al. 2002] BENNEWITZ, M., W. BURGARD und S. THRUN (2002). *Using EM to Learn Motion Behaviors of Persons with Mobile Robots*. In: *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[BORENSTEIN und KOREN 1991] BORENSTEIN, J. und Y. KOREN (1991). *The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots*. IEEE Transactions on Robotics and Automation, 7:278–288.

[BROCK und KHATIB 1999] BROCK, O. und O. KHATIB (1999). *High-Speed Navigation Using the Global Dynamic Window Approach*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[BURGARD et al. 1996] BURGARD, W., D. FOX, D. HENNIG und T. SCHMIDT (1996). *Estimating the Absolute Position of a Mobile Robot Using Position Probability Grids*. In: *Proc. of the Fourteenth National Conference on Artificial Intelligence*, S. 896–901.

[CANNY 1988] CANNY, J. F. (1988). *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA.

[CASSANDRA 1998] CASSANDRA, A. (1998). *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD Thesis, Brown University, Providence, RI.

[CHATILA 1982] CHATILA, R. (1982). *Path Planning and Environment Learning in a Mobile Robot System*. In: *Proc. of the European Conference on Artifical Intelligence*.

[CHOSET et al. 2000] CHOSET, H., E. ACAR, A. RIZZI und J. LUNTZ (2000). *Exact Cellular Decompositions in Terms of Critical Points of Morse Functions*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[CHOSET et al. 1997] CHOSET, H., B. MIRTICH und J. BURDICK (1997). *Sensor Based Planning for a Planar Rod Robot: Incremental Construction of the Planar Rod-HGVG*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[ELFES 1989] ELFES, A. (1989). *Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation*. PhD Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA.

[FEDOR 1993] FEDOR, C. (1993). *TCX. An interprocess communication system for building robotic architectures. Programmer's guide to version 10.xx.*. Carnegie Mellon University, Pittsburgh, PA.

[FENG et al. 1994] FENG, L., J. BORENSTEIN und H. EVERETT (1994). *"Where am I?" Sensors and Methods for Autonomous Mobile Robot Positioning*. Technischer Bericht UM-MEAM-94-21, University of Michigan, Ann Arbor, MI.

[FOX et al. 1997] FOX, D., W. BURGARD und S. THRUN (1997). *The Dynamic Window Approach to Collision Avoidance*. IEEE Robotics & Automation Magazine, 4:23–33.

[FOX et al. 1998] FOX, D., W. BURGARD und S. THRUN (1998). *A Hybrid Collision Avoidance Method For Mobile Robots*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[FURCY und KÖNIG 2000] FURCY, D. und S. KÖNIG (2000). *Speeding up the Convergence of Real-Time Search*. In: *Proc. of the National Conference on Artificial Intelligence (AAAI)*.

[HÄHNEL et al. 1998] HÄHNEL, D., W. BURGARD und G. LAKEMEYER (1998). *GOLEX - Bridging the Gap between Logic (GOLOG) and a Real Robot*. In: *Proc. of the 22nd National German Conference on Artificial Intelligence*.

[HOLTE et al. 1996] HOLTE, R., M. PEREZ, R. ZIMMER und A. MACDONALD (1996). *Hierarchical A*: Searching Abstraction Hierarchies Efficiently*. In: *Pre-publication version of the National Conference on Artificial Intelligence (AAAI)*.

[HSU et al. 2000] HSU, D., R. KINDEL, J. LATOMBE und S. ROCK (2000). *Randomized kinodynamic motion planning with moving obstacles*. Workshop on the Algorithmic Foundations of Robotics.

[KAELBLING et al. 1998] KAELBLING, L., M. LITTMAN und A. CASSANDRA (1998). *Planning and acting in partially observable stochastic domains*. Artificial Intelligence, 101:99–134.

[KHATIB 1986] KHATIB, O. (1986). *Real-time obstacle avoidance for robot manipulator and mobile robots*. The International Journal of Robotics Research, 5:90–98.

[KÖNIG und LIKHACHEV 2002] KÖNIG, S. und M. LIKHACHEV (2002). *Improved Fast Replanning for Robot Navigation in Unknown Terrain*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[KO und SIMMONS 1998] KO, N. Y. und R. SIMMONS (1998). *The Lane-Curvature Method for Local Obstacle Avoidanc*. In: *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[KONOLIGE 2000] KONOLIGE, K. (2000). *A Gradient Method for Realtime Robot Control*. In: *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[LATOMBE 1991] LATOMBE, J. C. (1991). *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA. ISBN 0-7923-9206-X.

[MÍNQUEZ und L.MONTANO 2000] MÍNQUEZ, J. und L.MONTANO (2000). *Nearess Diagramm Navigation: A New Real Time Collision Avoidance Approach*. In: *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[MÍNQUEZ et al. 2002] MÍNQUEZ, J., L.MONTANO und J. SANTOS-VICTOR (2002). *Reactive Collision Avoidance for Non-holonomic Robots using the Ego-Kinematic Space*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[MÍNQUEZ et al. 2001] MÍNQUEZ, J., L.MONTANO, T. SIMEON und R. ALAMI (2001). *Global Nearess Diagramm Navigation*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[MORAVEC 1999] MORAVEC, H. (1999). *Robot – Mere Machine to Transcendent Mind*. Oxford University Press, NY.

[MORAVEC 1988] MORAVEC, H.P. (1988). *Sensor fusion in certainty grids for mobile robots*. AI Magazine, 9:61–77.

[NILSSON 1969] NILSSON, N.J. (1969). *A mobile automation: an application of artificial intelligence techniques*. In: *Proc. of the International Conference on Artificial Intelligence (IJCAI)*.

[O'DUNLAING und YAP 1982] O'DUNLAING, C. und C. YAP (1982). *A retraction method for planning the motion of a disc*. Journal of Algorithms, 6:104–111.

[OTTMANN und WIDMAYER 1996] OTTMANN, T. und P. WIDMAYER (1996). *Algorithmen und Datenstrukturen*. Spektrum Verlag, Heidelberg.

[RATERING und GINI 1993] RATERING, S. und M. GINI (1993). *Robot navigation in a known environment with unknown moving obstacles*. In: *IEEE Transactions on Robotics and Automation*.

[ROY et al. 1999] ROY, N., W. BURGARD, D. FOX und S. THRUN (1999). *Coastal Navigation: Mobile Robot Navigation with Uncertainty in Dynamic Environments*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[ROY und THRUN 1999] ROY, N. und S. THRUN (1999). *Coastal Navigation with Mobile Robots*. In: *Proc. of the Neural Information Processing Systems (NIPS)*.

[RUSSEL und NORVIG 1994] RUSSEL, S. und P. NORVIG (1994). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ.

[SCHLEGEL 1998] SCHLEGEL, C. (1998). *Fast Local Obstacle Avoidance under Kinematic and Dynamic Constraints for a Mobile Robot*. In: *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[SCHMIDT und AZARM 1992] SCHMIDT, K. und K. AZARM (1992). *Mobile Robot Navigation in a Dynamic World Using an Unsteady Diffusion Equation Strategy*. In: *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[SCHWARZ und SHARIR 1983] SCHWARZ, J. T. und M. SHARIR (1983). *On the 'Piano Movers' Problem: II. Gerneral Technique for Computing Topological Properties of Real Algebraic Manifolds*. Advances in Applied Mathematics, 4:298–351.

[SCHWARZ et al. 1987] SCHWARZ, J. T., M. SHARIR und J. HOPCROFT (1987). *Planning, Geometry and Complexity of Robot Motion*. Ablex, Norwood, NJ.

[SIMMONS 1996] SIMMONS, R. (1996). *The Curvature-Velocity Method for Local Obstacle Avoidance*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

[SIMMONS und JAMES 2001] SIMMONS, R. und D. JAMES (2001). *IPC. An interprocess communication system. Reference Manual to version 3.4*. Carnegie Mellon University, Pittsburgh, PA.

[SMITH und CHEESEMAN 1986] SMITH, R. und P. CHEESEMAN (1986). *On the Representation of end Estimation of Spatial Uncertency*. International Journal of Robotics Research (IJRR), 5:56–68.

[STENTZ 1995] STENTZ, A. (1995). *The Focussed D* Algorithm for Real-Time Replanning*. In: *Proceedings of the International Joint Conference on Artificial Intelligence*.

[SUTTON und BARTO 1998] SUTTON, R. S. und A. G. BARTO (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

[THRUN et al. 1998] THRUN, S., A. BÜCKEN, W. BURGARD, D. FOX, T. FRÖHLINGHAUS, D. HENNIG, T. HOFMANN, M. KRELL und T. SCHIMDT (1998). *Map Learning and High-Speed Navigation in RHINO*. In: KORTENKAMP, D., R. BONASSO und R. MURPHY, Hrsg.: *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press, Cambridge, MA.

[UDUPA 1977] UDUPA, S. (1977). *Collision Detection and Avoidance in Computer Controlled Manipulators*. PhD Thesis, Department of Electrical Engineering, California Institute of Technology, CA.

[ULRICH und BORENSTEIN 1998] ULRICH, I. und J. BORENSTEIN (1998). *VFH+: Reliable Obstacle Avoidance for Fast Mobile Robots*. In: *IEEE Transactions on Robotics and Automation*.

[ULRICH und BORENSTEIN 2000] ULRICH, I. und J. BORENSTEIN (2000). *VFH*: Local Obstacle Avoidance with Look-Ahead Verification*. In: *IEEE Transactions on Robotics and Automation*.

[XU und YANG 2002] XU, H. und S. YANG (2002). *Real-time Collision-free Motion Planning of Nonholonomic Robots using a Neural Dynamics Based Approach*. In: *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Außerdem erkläre ich, dass die Diplomarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

(Cyrill Stachniss)
Freiburg, den 8. November 2003