

Erkunden von unbekanntem Terrain mit einem autonomen Modellfahrzeug

Bachelor-Thesis eingereicht von
Sebastian Ehmes
am 30. September 2016



Fachgebiet Echtzeitssysteme
Elektrotechnik und
Informationstechnik (FB18)
Zweitmitglied Informatik (FB20)
Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt
www.es.tu-darmstadt.de

Gutachter: Prof. Dr. rer. nat. A. Schürr
Betreuer: Géza Kulcsár

ES-B-0115

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis selbstständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die ich aus fremden Quellen direkt oder indirekt übernommen habe, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Ich erkläre mich damit einverstanden, dass die Arbeit auch durch das Fachgebiet Echtzeitsysteme der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 30. September 2016

(Sebastian Ehmes)



Zusammenfassung

Die vorliegende Bachelorthesis präsentiert ein Verfahren zur autonomen Erkundung auf einem mobilen Roboter mit Ackermann-Lenkung und stellt das Resultat als ROS-Paket automap [Ehm16a] zur Verfügung. Dabei wird ein Überblick über verschiedene Erkundungsstrategien gewährt. Es werden zwei Strategien, die den frontier based approach verwenden, implementiert und an die kinematischen Eigenschaften eines autoähnlichen Roboters angepasst. Dafür wird eine auf OpenCV basierende neue Methode zur Erkennung von Frontiers sowie ein auf dem Path-Transform-Algorithmus basierender globaler Wegplaner präsentiert. Es wird eine in ROS integrierte Simulation entwickelt, in der beide Strategien evaluiert werden und die im ROS-Paket simulation [Ehm16b] veröffentlicht ist. Am Ende der Thesis wird der besser geeignete Algorithmus auf dem, in der Arbeit präsentierten mobilen Roboter getestet und die Ergebnisse evaluiert.



Inhaltsverzeichnis

1	Einführung	1
1.1	Problemstellung und Zielsetzung	2
1.2	Aufbau der Arbeit	3
2	Grundlagen	5
2.1	Übersicht mobiler Robotertypen	5
2.2	Vorwärtsskinematik der Ackermann Lenkung	7
2.3	Einführung in das Robot Operating System (ROS)	9
2.4	Repräsentation der Umwelt auf einem autonomen Fahrzeug	10
2.5	Aus OpenCV verwendete Computer-Vision-Algorithmen	13
2.6	Etablierte Erkundungsstrategien	16
2.7	Wegplanung auf mobilen Robotern	20
2.8	Entfernungssensoren	22
3	Umsetzung	25
3.1	Beschreibung des Modellautos	25
3.1.1	Verwendete Hardware und Sensoren	25
3.1.2	Verwendete Software	27
3.2	Simulation	29
3.2.1	Modellierung des Fahrzeugs	30
3.2.2	Simulation von Entfernungssensoren	30
3.2.3	Simulation in ROS	32
3.3	Frontier-Erkennung	32
3.4	Umsetzung des Path-Transform-Algorithmus	35
3.5	Implementierung eines Exploration-Planers	39
3.5.1	Umsetzung einer einfachen Erkundungsstrategie	41
3.5.2	Umsetzung der Next-Best-View-Strategie	41
3.5.3	Der ROS-Node automap	44
4	Evaluation und Diskussion	47
4.1	Simulation der Erkundungsstrategien	47
4.1.1	Auswertung der Simulationsergebnisse	48
4.1.2	Auswirkungen von verschiedenen Parametern	53
4.2	Test der NBV-Strategie auf dem Modellauto	55
5	Verwandte Arbeiten	59
6	Fazit	63



Abbildungsverzeichnis

2.1	Freiheitsgrade mobiler Roboter	5
2.2	Beispiele verschiedener Roboterbauweisen [SK08]	6
2.3	Skizze zur Herleitung der Ackermann-Kinematik	7
2.4	Beispiel für ein occupancy grid [SK08]	11
2.5	Beispiel für eine line map [SK08]	12
2.6	Beispiel für eine topologische Karte [SK08]	13
2.7	OpenCV: Binary Threshold [BK08]	14
2.8	OpenCV: Erosion und Dilatation [BK08]	14
2.9	OpenCV: Canny-Algorithmus [BK08]	15
2.10	Frontiers und Zentroiden [Yam97]	17
2.11	NBV: Frontiers und pot. Positionen [GBL02]	18
2.12	NBV: pot. Informationsgewinn und nächste beste Position [GBL02]	19
2.13	Veranschaulichung der Distance- und Obstacle-Transformationen [WP07]	
	21
3.1	Bild des Modellautos	25
3.2	Anordnung der einzelnen Komponenten im Modellauto	26
3.3	Sensoren am Auto	26
3.4	Ablauf der Frontier-Detektion	33
3.5	Frontier-Detektion: occupancy grid	33
3.6	Frontier-Detektion: Schritt 1	34
3.7	Frontier-Detektion: Schritt 2	34
3.8	Frontier-Detektion: Schritt 3 - Resultat	35
3.9	Path-Transform: occupancy grid	35
3.10	Path-Transform: Distance-Transformation und Obstacle-Transformation	37
3.11	Path-Transform: Path-Transformation	38
3.12	Path-Transform: Pfade	38
3.13	Exploration-Planer: Ablaufdiagramm	39
3.14	ROS-Node automap: Ablaufdiagramm	45
4.1	Testkarten Teil 1	47
4.2	Testkarten Teil 2	48
4.3	Vergleich zwischen NBV- und einfacher Strategie	50
4.4	Probleme bei der NBV-Strategie	51
4.5	Vergleich der zurückgelegten Wege	52
4.6	Veränderung des Parameters alpha der NBV-Strategie	54
4.7	Problem des Pendelns	55
4.8	Testraum Teil 1	56
4.9	Testraum Teil 2	56
4.10	Erkundung mit dem Modellauto Teil 1	57
4.11	Erkundung mit dem Modellauto Teil 2	57
4.12	Erkundung mit dem Modellauto Teil 3	58



Tabellenverzeichnis

4.1	Simulationsergebnisse der NBV-Strategie für feste Parameter	49
4.2	Simulationsergebnisse der einfachen Strategie für feste Parameter	49
4.3	Simulationsergebnisse mit menschlicher Steuerung	51
4.4	Gemittelte Simulationsergebnisse	52
4.5	Simulationsergebnisse der NBV-Strategie für variierte Parameter	53
4.6	Simulationsergebnisse der einfachen Strategie für variierte Parameter . . .	54



1 Einführung

In den letzten Jahren ist die autonome Fortbewegung von Robotern, insbesondere von autoähnlichen mobilen Robotern, immer mehr zum Thema geworden und beschäftigt zunehmend sowohl universitäre, als auch kommerzielle Forschungseinrichtungen gleichermaßen. Besondere Aufmerksamkeit erfahren bei diesem Themenkomplex vor allem die Wegplanungsalgorithmen die, mit Hilfe einer Karte der Umgebung und der entsprechenden Sensorik am Fahrzeug dafür sorgen, dass eine kollisionsfreie und möglichst effiziente Fahrt von Punkt A nach Punkt B gewährleistet ist. Oft werden solche Karten bereits vorher mit Hilfe von Personen erstellt, indem etwa ein ferngesteuertes Fahrzeug mit Laserscanner zur Erkundung eingesetzt, oder ein vorhandener Gebäudegrundriss für so einen Einsatz entsprechend bearbeitet wird. Dabei stellt sich die Frage welche Maßnahmen getroffen werden können wenn, wegen mangelnder Infrastruktur, oder guter Abschirmung keine Verbindung zum mobilen Roboter hergestellt werden kann, oder diese abreißt. Zusätzlich kann selbst bei einer bereits vorhandenen Karte des Einsatzgebietes nicht immer gewährleistet sein, dass diese noch die Wirklichkeit abbildet, beispielsweise nach einem Katastrophenfall in einem teilweise eingestürzten Gebäude. Im Zuge eines Projektseminars im Fachgebiet Echtzeitsysteme an der TU-Darmstadt wurde den Studenten die Gelegenheit geboten, sich in die Thematik des autonomen Fahrens tiefergehend einzuarbeiten und verschiedene Ansätze zu entwickeln oder etablierte Ansätze auszuprobieren. Hierbei ergab sich ganz konkret die Fragestellung nach der geeigneten Methode, welche die Umgebung auf dem Roboter abbilden kann, um so kollisionsfreie Trajektorien für die eingesetzten mobilen Roboter zu planen. Als beste Methode zur Erstellung einer Karte erwies sich das Fernsteuern des Fahrzeugs durch das Gebäude, bei gleichzeitigem Einsatz eines SLAM-Algorithmus (Simultaneously Mapping And Learning). Damit konnte mit Hilfe einer Kinect-Kamera, die einen Laserscanner ersetzt, eine Karte aufgebaut und die Position des Roboters darin ermittelt werden. Trotz der guten Ergebnisse, war das Verfahren sehr aufwändig. Es wurde überlegt, ob es möglich ist diesen Prozess zu automatisieren. Das führt letztlich zur Frage was zu tun ist, wenn keine menschliche Assistenz beim Kartographieren zur Verfügung steht. Diese Bachelorarbeit beschäftigt sich deshalb mit der Problemstellung, eine Karte für die mobile Roboternavigation möglichst autonom zu erstellen.

1.1 Problemstellung und Zielsetzung

Die Aufgabe zur Entwicklung einer autonomen Erkundung birgt einige konkrete Probleme, die es zu lösen gilt.

- Es muss geklärt werden, auf welche Art die erkundete Umwelt auf dem Roboter repräsentiert wird.
- Dem mobilen Roboter muss es möglich sein, bereits erkundetes Territorium von unbekanntem unterscheiden zu können und die Richtung dorthin zu ermitteln.
- Der Algorithmus, der die autonome Erkundung leitet, benötigt ein Kriterium, nach dem ein möglicher nächster Erkundungsschritt ausgewählt wird.
- Als Letztes muss klar sein, wann der Roboter die Erkundung beendet, also wann ein Gebiet hinreichend gut erkundet ist.

Da das Problem der autonomen Erkundung kein gänzlich unerforschtes ist, werden zuerst bereits etablierte Erkundungsstrategien betrachtet. Zusätzlich spielen gängige Wegplanungsalgorithmen eine Rolle bei der Auswahl des Kartentyps.

Das Ziel dieser Arbeit ist es, ein Verfahren zur autonomen Erkundung basierend auf bereits etablierten Strategien für einen mobilen Roboter zu entwickeln. Dabei werden diese Strategien so modifiziert, dass sie den kinematischen Eigenschaften eines autoähnlichen Roboters Rechnung tragen und darauf einsatzfähig sind. Zusätzlich wird ein mit OpenCV entwickeltes, neues Verfahren zur Erkennung von Grenzen zwischen bekanntem und unbekanntem Territorium auf einer Karte vorgestellt. Da der theoretische Nachweis für ein optimales Verhalten eines autonomen Erkundungsmodus nur schwer bis gar nicht zu erbringen ist, werden zwei gängige Verfahren erläutert, implementiert und gegeneinander getestet. Diese Tests werden im Rahmen einer eigens entwickelten Simulation durchgeführt.

Der Algorithmus, der sich nach den Tests als am geeignetsten erweist, wird dann auf einem Roboter in der Realität innerhalb eines Gebäudes getestet. Dabei wird im Gegensatz zu den Plattformen, auf denen die etablierten Strategien demonstriert wurden, beim Roboter in dieser Arbeit kein light detection and ranging Sensor (LIDAR) eingesetzt, sondern eine Kinect Kamera in der Version 2. Da es sich bei dem mobilen Roboter um ein Fahrzeug aus dem Projektseminar Echtzeitsysteme handelt und im Zuge dieses Seminars ROS als Metabetriebssystem des Autos benutzt wurde, soll die im Rahmen dieser Arbeit entstehende Software den ROS-Standards und Schnittstellen genügen, um später von anderen Studenten weiterverwendet werden zu können.

Es wird folglich ein ROS-kompatibles Paket entstehen, das einen autoähnlichen, mobilen Roboter mit passender Sensorik in die Lage versetzt, autonom ein abgeschlossenes und näherungsweise ebenes Gebiet vollständig zu erkunden. Die daraus resultierende Karte bildet die erkundete Umgebung in einer 2D Repräsentation ab und wird von den meisten ROS-typischen Planungsalgorithmen verwendet werden können.

1.2 Aufbau der Arbeit

In Kapitel 2 werden die wichtigsten Grundlagen erläutert, insbesondere die Darstellung der Umwelt auf einer Karte, autonome Wegplanung und bekannte Strategien zur autonomen Erkundung.

Kapitel 3 befasst sich hauptsächlich mit der Umsetzung. Hier wird zunächst die Hardware des Modellautos und die darauf befindliche Software mit ihren verschiedenen verwendeten Bibliotheken und Paketen erläutert. Außerdem wird eine Simulation erarbeitet, auf der verschiedene Ansätze zur autonomen Erkundung getestet werden können. Der in Kapitel 2 beschriebene Wegplanungsalgorithmus und die beiden autonomen Erkundungsansätze werden implementiert. Am Ende des Kapitels wird dargelegt, wie die implementierten Erkundungsmodi in ROS integriert werden.

Im Kapitel 4, der Evaluation und Diskussion, werden die zwei Ansätze in verschiedenen Szenarien simuliert und die Ergebnisse miteinander verglichen. Als weiterer Vergleich werden die selben Szenarien von Menschen absolviert, um einen greifbaren Anhaltspunkt für die einzelnen Ergebnisse zu bekommen. Die hieraus resultierende beste Strategie wird anschließend auf dem Auto getestet und mögliche Probleme und Begrenzungen, die sich aus dem Test ergeben, werden erörtert.

Kapitel 5 betrachtet verwandte Arbeiten, die sich auf die selben Grundlagen stützen wie diese Thesis und vergleicht etwaige Gemeinsamkeiten und Unterschiede.

Im Abschlusskapitel werden die Ergebnisse dieser Arbeit rekapituliert und mögliche zukünftige Verbesserungen in Aussicht gestellt.



2 Grundlagen

In den Grundlagen werden verschiedene mobile Robotertypen betrachtet und die besonderen Eigenschaften des hier verwendeten Typs beleuchtet. Es werden die Eigenheiten von ROS genannt und wie sich diese auf die Umsetzung auswirken. Im darauf folgenden Abschnitt wird erläutert, welche Eigenschaften verschiedene Kartentypen haben, auf denen Wegplanungsalgorithmen arbeiten. Es werden einige Bildverarbeitungsalgorithmen im Kontext der Open-Source-Software OpenCV erläutert, da diese in der Umsetzung häufig zur Anwendung kommen werden. Von zentraler Bedeutung ist allerdings die Betrachtung von Erkundungsstrategien, die in fremden Arbeiten bereits erarbeitet wurden, in diese Arbeit eingehen sollen und in der Umsetzung modifiziert werden. Danach wird der für diese Arbeit am besten passende Wegplanungsalgorithmus ermittelt. Letztlich werden verschiedene Techniken zur Entfernungsmessung erläutert und die in dieser Arbeit verwendete Methode vorgestellt.

2.1 Übersicht mobiler Robotertypen

Da inzwischen viele verschiedene mobile Robotertypen entstanden sind und in den verwandten Arbeiten verschiedene Typen von mobilen Plattformen verwendet werden, sind im Folgenden die gängigsten Arten mit ihren Besonderheiten aufgezählt, wobei ausschließlich radgetriebene Roboter genannt werden. Fliegende, schwimmende oder laufende Roboter werden nicht betrachtet.

Die gängigsten radgetriebenen Typen lassen sich vom kinematischen Verhalten her in zwei große Gruppen unterteilen: holonome und nicht-holonom mobile Roboter. Das

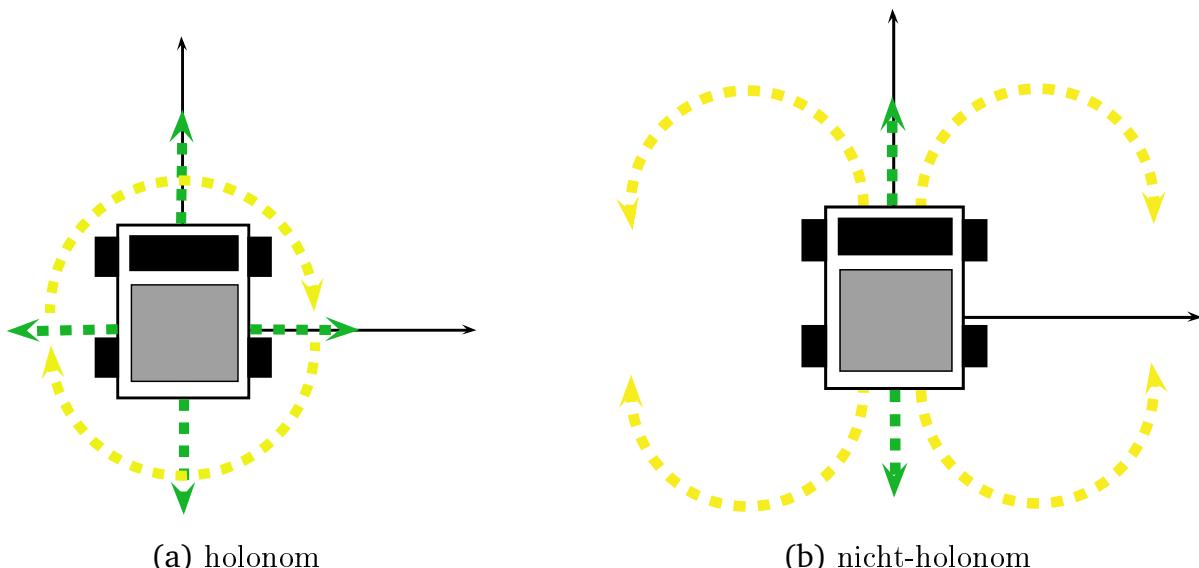


Abb. 2.1: *Freiheitsgrade mobiler Roboter*

Buch "Handbook of Robotics" [SK08] beschreibt im Kapitel 17 "Wheeled Robots" die mathematischen Definitionen und Besonderheiten der beiden Typen sehr ausführlich.

Zusammengefasst bedeutet holonom, dass das Kinematikmodell des Roboters keiner Zwangsbedingung unterworfen ist. Umgekehrt heißt das für nicht-holonom Fahrzeuge, dass sie konstruktionsbedingt nicht alle Bewegungen und die erlaubten nur auf ganz bestimmte Art und Weise ausführen können.

Wie man in Abbildung 2.1 a) sieht, kann der holonome Roboter, neben der Vorwärts-, Seitwärts und Rückwärtsbewegung, auch eine Drehung um die Z-Achse durchführen. Damit können problemlos alle Punkte in einer Ebene erreicht werden. Diese Tatsache erleichtert in erster Linie die Wegplanung erheblich, da die resultierenden Trajektorien keine Rücksicht auf besondere kinematische Eigenheiten, wie beispielsweise den minimalen Kurvenradius des Roboters und Orientierung am Zielpunkt, nehmen müssen. Außerdem kann sich ein Fahrzeug dieser Art nicht in ausweglose Situationen bringen. Diese sind Ausgangspositionen, die durch vorherige Manöver entstanden sind und von denen ein Zielpunkt nicht mehr, oder nur durch komplizierte Rangiermanöver erreichbar ist. Abbildung 2.2 a) zeigt eine mögliche Radkonstruktion, mit Hilfe derer eine holonome Kinematik realisiert werden kann.

Nicht-holonom mobile Plattformen sind die am häufigsten verwendeten radgetrieben

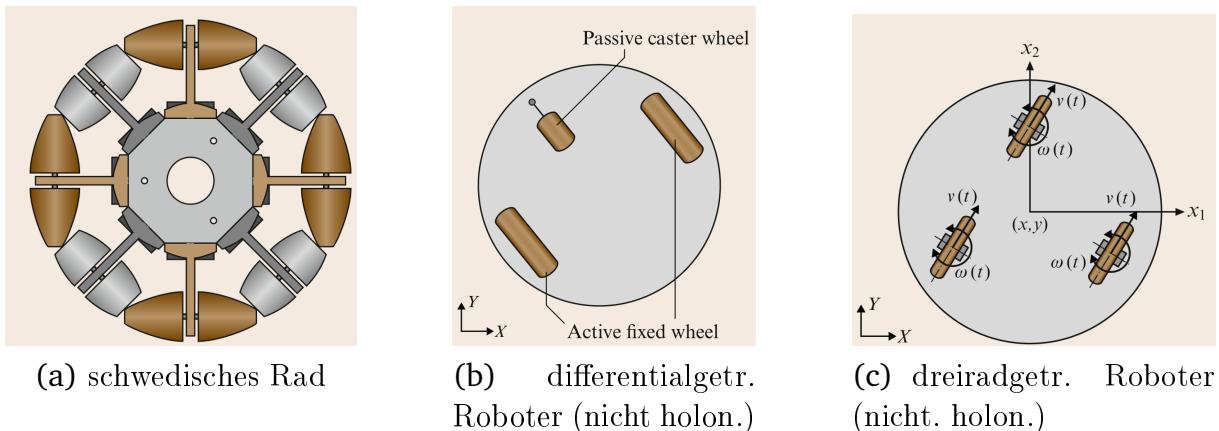


Abb. 2.2: Beispiele verschiedener Roboterbauweisen [SK08]

Roboter. Dabei sind die nicht-holonomen kinematischen Einschränkungen stark von der Bauweise des Fahrzeugs abhängig. In Abbildung 2.2 sind zwei nicht-holonom Fahrzeuge abgebildet, das aus Skizze b) ist ein differentialgetriebenes mit zwei Antriebsräder und einem frei drehbaren Stützrad und das aus Skizze c) hat drei drehbare Antriebsräder. Beide Typen können sich auf der Stelle drehen, aber nur der Typ in c) kann auch seitwärts fahren. Beim differentialgetriebenen mobilen Roboter wird das Maß der Richtungsänderung, dem Namen nach, durch den Geschwindigkeitsunterschied der beiden Antriebsräder bestimmt. Diese Abhängigkeit von Orientierung und Geschwindigkeits-differential ist ein Beispiel für eine nicht-holonom Zwangsbedingung. Der dreirädrige Roboter hat diese Beschränkung nicht, da alle Räder dem Antrieb dienen und frei gesteuert werden können. Dennoch muss dieser Typ bestimmten Beschränkungen gehorchen, wenn alle Räder ohne Schlupf rollen sollen. In der Abbildung 2.1 b) kann man die verschiedenen Freiheitsgrade eines autoähnlichen Roboters sehen. Dabei fällt auf, dass dieser im Gegensatz zu vielen anderen mobilen Robotern nicht in der Lage ist, sich

auf der Stelle zu drehen oder seitwärts zu fahren. Dieser Typ Fahrzeug ist mit einer sogenannten Ackermann-Lenkung ausgestattet und erlaubt nur Fahrbahnen auf Kreisbahnen um ein momentanes Rotationszentrum oder tangential dazu. Die beiden vorher genannten Robotertypen können dieses Verhalten aber imitieren, um beispielsweise glattere Trajektorien zu fahren, was unnötiges Abstoppen, Drehen und Beschleunigen vermeiden kann.

Im nächsten Abschnitt wird der kinematische Sachverhalt der Ackermann-Lenkung sehr ausführlich gezeigt, da der in dieser Arbeit verwendete mobile Roboter diesem Typ entspricht und die Vorwärtskinematik besonders in der Positionsbestimmung Anwendung findet.

2.2 Vorwärtskinematik der Ackermann Lenkung

Im Folgenden wird in der Herleitung gezeigt, wie man die Position und Orientierung dieses Fahrzeugtyps in der X-Y-Ebene, aus dem Lenkeinschlag α und der zurückgelegten Strecke S , berechnen kann.

Für die Herleitung gilt, dass α der Lenkeinschlag bezogen auf die Radnormale und

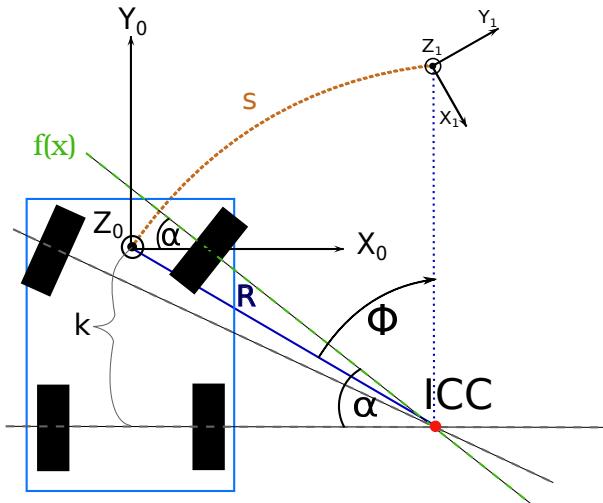


Abb. 2.3: Skizze zur Herleitung der Ackermann-Kinematik

Φ der gefahrene Kreisbogen, im Bogenmaß, um das ICC ist. Das **ICC** ist das sogenannte **instantaneous center of curvature**, das momentane Rotationszentrum eines Fahrzeuges, das sich auf einer Kreisbahn bewegt. Die Herleitung beginnt mit der Geradengleichung der grün gefärbten Geraden aus Abbildung 2.3:

$$f(x) = (x - x_0)m + t_0 \quad \text{mit } t_0 = -k \quad (1)$$

Dabei ist k der Betrag des Abstandes der beiden Achsen. Setzt man den Koordinatenursprung in die Vorderachse, ergibt sich als Verschiebung auf der Y-Achse $-k$.

Es folgt aus der Geraden, die entlang der Radnormalen des rechten Vorderrades verläuft:

$$ICC_x = \frac{-k}{\tan(\alpha)} \text{ und } ICC_y = -k \Rightarrow \vec{ICC} = -\left[\begin{array}{c} \frac{k}{\tan(\alpha)} \\ k \end{array} \right] \quad (2)$$

ICC_x und ICC_y sind die Koordinaten des ICC aus Sicht des Koordinatenursprungs. Daher gilt für den Radius R der Kreisbahn, auf dem sich ein Fahrzeug mit Ackermann-Lenkung bewegt:

$$R = |\vec{ICC}| = \sqrt{\left(\frac{k}{\tan(\alpha)} \right)^2 + k^2} \quad (3)$$

Damit ergibt sich für den Kreisbogen im Bogenmaß:

$$\Phi = \frac{S}{R} = \frac{S}{\sqrt{\left(\frac{k}{\tan(\alpha)} \right)^2 + k^2}} = \Theta \quad (4)$$

In Gleichung 4 ist S die gefahrene Strecke auf der Kreisbahn. Zusätzlich stellt sich nach Überlegung heraus, dass Φ nicht nur der gefahrene Kreisbogen um das ICC ist, sondern ebenfalls der Rotation des ursprünglichen Koordinatensystems 0, um die Z-Achse mit dem Winkel Θ , hin zum nächsten Koordinatensystem 1 entspricht. Daher berechnen sich die Koordinaten in der x-y-Ebene wie folgt:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} R \cdot \cos(\Phi + \alpha) - \frac{k}{\tan(\alpha)} \\ R \cdot \sin(\Phi + \alpha) - k \end{bmatrix} \quad (5)$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} R \cdot \cos(\pi - \Phi + \alpha) - \frac{k}{\tan(\alpha)} \\ R \cdot \sin(\pi - \Phi + \alpha) - k \end{bmatrix} \quad (6)$$

Da das Fahrzeug je nach Vorzeichen des Lenkeinschlags den Kreisbogen in unterschiedlicher Richtung durchquert und sich damit der Startpunkt verlagert, muss man an dieser Stelle eine Fallunterscheidung durchführen. Ist der Lenkwinkel positiv, lenkt das Auto nach links, so wird der Kreis in mathematisch positivem Sinne durchfahren und die Gleichung 5 gilt.

Lenkt man nach rechts, ist der Lenkwinkel negativ, so wird der Kreis in mathematisch negativem Sinne, mit dem Startpunkt π durchquert. Es gilt deswegen Gleichung 6.

Bei $\alpha=0$ entsteht ein mathematischer Grenzfall, ein Kreis mit unendlich großem Radius, was in der Praxis bedeutet, dass man geradeaus fahren möchte. Dieser Fall muss in der Anwendung abgefangen werden und produziert eine einfache Translation ohne rotatorischen Anteil.

In Folge dessen kann man die homogene Transformationsmatrix T der Vorwärtsskinematik aufstellen, mit der man Punkte vom lokalen in das globale Koordinatensystem transformieren kann und umgekehrt.

$${}^0T_i = \text{trans}(x, y, 0) \cdot \text{rot}(z, \Theta) = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(\Theta) & -\sin(\Theta) & 0 & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (7)$$

$$\Rightarrow {}^0T_i = \begin{pmatrix} \cos(\Theta) & -\sin(\Theta) & 0 & x \\ \sin(\Theta) & \cos(\Theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \text{mit } \Theta = \begin{cases} -\Theta, & \text{wenn } \alpha < 0 \\ \Theta, & \text{wenn } \alpha > 0 \end{cases}$$

Die vierte Spalte der i-ten Matrix entspricht dem letzten Translationsvektor des Fahrzeugs in der X-Y-Ebene. Spalte eins und zwei bilden die Einheitsvektoren des i-ten Koordinatensystems. Mit Hilfe des Arcus-Tangens mit Sektorinformation lässt sich, aus den Einträgen der jeweiligen Spalten, die Orientierung des Fahrzeugs in der Ebene im Bezug zum 0-ten Koordinatensystem als Yaw-Winkel, dem Rotationswinkel um die Z-Achse, berechnen.

In der Simulation und auf dem Roboter wird mit Hilfe der beschriebenen Technik die Position nach jedem Zeitschritt bestimmt. Da die Simulation im Robot Operating System eingebettet ist, wird ROS im nächsten Abschnitt kurz erläutert.

2.3 Einführung in das Robot Operating System (ROS)

Das **Robot Operating System** (ROS) ist laut der Onlinerefenz [ROS14b] ein open-source Metabetriebssystem für Roboter. Es stellt die Dienste bereit, die normalerweise von einem Betriebssystem erwartet werden: Hardwareabstraktion, low-level Gerätesteuerung, bereits vorimplementierte oft benutzte Funktionen, Kommunikation zwischen Prozessen und Paketverarbeitung. Zusätzlich werden Werkzeuge und Bibliotheken zur Verfügung gestellt, die das Entwickeln und gleichzeitige Betreiben auf mehreren verteilten Geräten unterstützen.

Der ROS Laufzeitgraph ist ein peer-to-peer Netzwerk von Prozessen (**Nodes**), die mit Hilfe der ROS-Kommunikationsinfrastruktur verbunden sind. Diese Nodes, im weiteren Verlauf teilweise als Knoten bezeichnet, können miteinander asynchron über sogenannte **Topics** kommunizieren, indem sie entweder abonnieren (subscribe), um Nachrichten von anderen Knoten zu erhalten, oder auf ihnen veröffentlichen (publish), um Nachrichten zu senden. Auf einen Topic können beliebig viele Knoten zugreifen, so kann ein Knoten bei Bedarf auf einem Topic zugleich veröffentlichen und abonnieren. Wenn im Ausarbeitungsteil der Arbeit davon die Rede ist, dass Nachrichten oder Daten zu bestimmten Knoten gesendet werden, so ist diese Abonnenten-/Veröffentlicher-Mechanik gemeint. Außerdem wird eine synchrone Methode der Prozesskommunikation über sogenannte **Services** angeboten. Diese Art zu Kommunizieren ist im Gegensatz zur Node-/Topicdynamik nicht zum Datenaustausch zwischen vielen Prozessen gedacht, sondern explizit zwischen genau zwei Knoten. Dabei sendet ein Knoten Daten und wartet auf die Bestätigung des Anderen und umgekehrt. Es existieren noch andere Arten der Kommunikation zwischen Prozessen, die in der Referenz [ROS14a] erläutert sind.

Wichtig ist, dass ROS naturgemäß kein echtzeitfähiges Framework ist, obwohl es laut Onlinereferenz mit Modifikationen möglich ist, ROS in ein Echtzeitprojekt einzubetten. Was ROS für diese Arbeit und für die Teilnehmer des Projektseminars Echtzeitsysteme, aus dem diese Thesis hervorgeht, so interessant macht, ist die große Zahl an Open-Source-Softwarebibliotheken. Diese sind dank der einheitlichen Schnittstellen sehr einfach in eigene Projekte zu integrieren und erfahren wegen der aktiven Beteiligung der Community am ROS-Projekt, oft noch lange Wartung und Verbesserung. Außerdem sind in ROS bereits viele Standardprobleme der Robotik gut gelöst, wie etwa Selbstlokalisierung, Wegplanung und SLAM-Verfahren. So hat man die freie Entscheidung, ob man ein Problem selbst neu lösen möchte oder eine etablierte Methode verwendet beziehungsweise verbessert, was die softwareseitige Entwicklungsphase eines Roboters vereinfacht und beschleunigt.

Bei ROS sind occupancy grids als Darstellung der Umgebung, also als Karte, besonders beliebt und werden unter anderem im nächsten Abschnitt näher betrachtet.

2.4 Repräsentation der Umwelt auf einem autonomen Fahrzeug

Mit Repräsentation der Umwelt ist konkret eine Art Karte gemeint, die ein mobiler Roboter intern verwendet, um sich in seiner Umgebung einzuordnen, darauf kollisionsfreie Wege zu planen und nach Bedarf besondere Punkte von Interesse auf ihr zu vermerken. Im Folgenden wird geklärt, welche Kartentypen existieren, was für Vor- und Nachteile sie haben und welcher Typ in dieser Arbeit verwendet wird.

Im "Handbook of Robotics" [SK08] werden in Kapitel 36 "World Modeling" die Kartentypen in zwei Kategorien eingeteilt: einerseits Karten für Innenräume bzw. geordnete Strukturen, andererseits Karten für Außenumgebungen bzw. ungeordnete Strukturen. Da im Rahmen der Arbeit ausschließlich abgeschlossene Innenraumumgebungen betrachtet werden, sind nur die gängigsten Typen aufgezählt.

Occupancy grid Karten sind ein sehr beliebter Ansatz, um die Umwelt in einem diskreten Raster darzustellen. Jeder Zelle im Raster wird eine Wahrscheinlichkeit zugeordnet, dass sich darin ein Hindernis befindet. Es wird für jede Zelle angenommen, dass die Aufenthaltswahrscheinlichkeit eines Hindernisses unabhängig von der Nachbarzelle ist. Das entspricht eigentlich nicht der Wirklichkeit, denn erfahrungsgemäß liegen einzelne Hindernisse oft wegen ihrer Form oder Größe in mehreren Zellen gleichzeitig, je nach definierter Zellengröße. Damit müsste die Wahrscheinlichkeit für Hindernisse von Zellen, die um eine besetzte Zelle angeordnet sind, eigentlich eher steigen, als davon unabhängig zu sein. Doch diese Annahme erlaubt es, Zellen isoliert zu betrachten und einzeln zu berechnen, was Komplexität und Rechenaufwand vermindert.

Sei p_{prior} die Aufenthaltswahrscheinlichkeit vor der Messung, p_{occ} die Wahrscheinlichkeit eines Hindernisses und p_{free} die Wahrscheinlichkeit für Freiraum, wobei bei einem Laserscan oder Ähnlichem folgende Relation gilt:

$$0 \leq p_{\text{free}} < p_{\text{prior}} < p_{\text{occ}} \leq 1 \quad (8)$$

Weiterhin ist m_l die aktuell betrachtete Zelle, x_t der Aufenthalt des Roboters und z_t die Entfernungsmeßung eines strahlenartigen Entfernungsmeßers, zum Zeitpunkt t. Die

Größe r steht für die Auflösung des occupancy grids in Metern pro Pixel. Die Hindernis-aufenthaltswahrscheinlichkeit einer Zelle berechnet sich, aus den gegebenen Größen, wie in Gleichung 9.

$$p(m_l|x_t, z_t) = \begin{cases} p_{\text{prior}}, & z_t = \text{max. Messdistanz} \\ p_{\text{prior}}, & m_l \text{ nicht in Messung von } z_t \\ p_{\text{occ}}, & |z_t - \text{dist}(x_t, m_l)| < r/2 \\ p_{\text{free}}, & z_t \geq \text{dist}(x_t, m_l) \end{cases} \quad (9)$$

Nach einem Kartierungsvorgang kann eine Karte dann so aussehen wie in Abbildung 2.4. Hier erkennt man, dass in der Praxis häufig nicht mit Aufenthaltswahrscheinlichkeiten in Zellen gearbeitet wird, sondern dass eine Zelle wohl definierte Zustände annehmen kann, sie ist also entweder besetzt (schwarz), unbesetzt (weiß) oder unbekannt (grau). Der große Vorteil, die Umwelt auf diese Art darzustellen liegt darin, dass



Abb. 2.4: Beispiel für ein occupancy grid [SK08]

zwischen bekannten und unbekannten Bereichen unterschieden werden kann, was insbesondere nützlich für Erkundungsalgorithmen ist. Außerdem muss man für occupancy grids keine Merkmale wie Kanten, Polygonzüge oder andere Features vordefinieren, die zuerst erkannt und dann eingefügt werden müssen. Zusätzlich erlaubt dieser Typ Karte eine konstante Zugriffszeit auf jede Zelle, wovon vor allem Wegplanungsalgorithmen profitieren.

Ein Nachteil dieses Kartentyps sind die Diskretisierungsfehler, da Zellen nur endlich klein sein können und damit Hindernisse größer oder kleiner erscheinen können, als sie tatsächlich sind. Möchte man dem entgegenwirken, so muss die Auflösung und damit die Anzahl der Zellen erhöht werden, was den zweiten Nachteil andeutet, der große Speicherbedarf dieser Karten. Dieser wächst quadratisch, bei einer Verdopplung der Auflösung vervierfacht sich der Speicherbedarf. Dieser Punkt ist allerdings in der heutigen Zeit nur noch begrenzt von Bedeutung.

Line Maps sind eine Alternative zu den occupancy grids. Bei diesen wird nicht versucht, die Umwelt zu diskretisieren, um sie in einem Raster abzulegen, sondern den

eingehenden Messdaten der Entfernungssensoren werden Linien zugeordnet, die dann abgespeichert werden. Abbildung 2.5 zeigt das Resultat einer solchen Zuordnung. Die

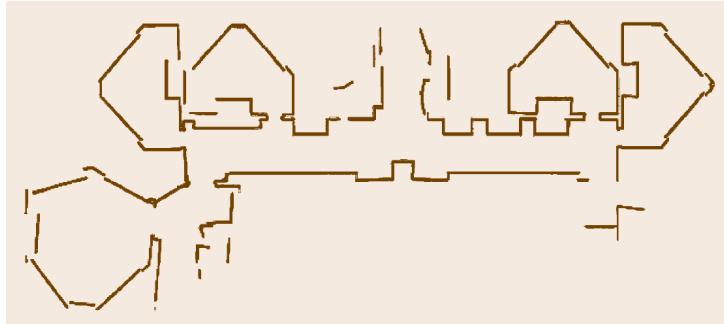


Abb. 2.5: Beispiel für eine line map [SK08]

Punkte können in der Regel mit Hilfe der least-squares-Methode nicht in jedem Szenario gut zugeordnet werden, da die Wirklichkeit meistens aus sehr vielen Liniensegmenten zusammengesetzt ist und nicht nur aus einem Einzigen. Oft wird deswegen unter anderem mit einem split-and-merge-Algorithmus gearbeitet. Die Idee hierbei ist, die Punktmenge rekursiv so zu unterteilen, dass Punkte einem einzelnen Liniensegment besser zugeordnet werden können. Der Algorithmus berechnet zu Beginn eine Linie, die am besten durch die Punktmenge zu legen ist. Danach wird der Punkt mit der maximalen Distanz zur Linie ermittelt. Liegt diese Distanz unter einem vorher festgelegten Wert, so terminiert der Algorithmus und gibt die Linie aus. Andernfalls wird die Punktmenge an diesem am weitesten entfernten Punkt geteilt und der Algorithmus beginnt dann rekursiv mit den beiden Punktmengen von vorne.

Ein Vorteil, die Welt auf diese Art abzubilden, ist dass line maps deutlich weniger speicherintensiv sind als occupancy grids und damit in der Größe besser skalieren. Zusätzlich sind sie akkurate, da keine Diskretisierungsfehler wie beim Rastern auftreten. Allerdings gibt es keine Möglichkeit festzustellen, wo sich erkundeter und unerkundeter Freiraum befindet, da explizit nur der Aufenthaltsort von Hindernissen aufgezeichnet wird. Damit eignet sich diese Kartenart ohne Modifikationen nicht zum autonomen Erkunden.

Eine weitere Alternative stellen die **topologischen Karten** dar, bei denen die Umwelt als ein zusammenhängender Graph abgebildet wird. Hierbei werden nicht die Hindernisse vermerkt, sondern die Wege innerhalb der Freiräume als Kanten und die Kreuzungen solcher Wege als Knoten. Es werden keine geometrischen Strukturen der Umwelt abgelegt, wie es bei den oberen beiden Varianten der Fall ist. Solche Graphen werden wie ein Voronoi-Diagramm dargestellt, worin eine Kante eine Menge von Punkten abbildet, die äquidistant zu ein oder mehreren Hindernissen sind. An den Knoten dieses Diagramms treffen diese Kanten aufeinander. Diese Art Karte, wie in Abbildung 2.6 exemplarisch gezeigt, vereinfacht die Wegplanung enorm, da nur ein kurzes Stück Weg zum Voronoi-Graphen durch den Freiraum geplant werden muss. Anschließend kann der Pfad auf dem Graphen, beispielsweise mit einem Dijkstra-Algorithmus, bis in die Nähe des Ziels gefunden werden. Von dort aus muss dann nur ein kurzes Stück vom Graphen durch

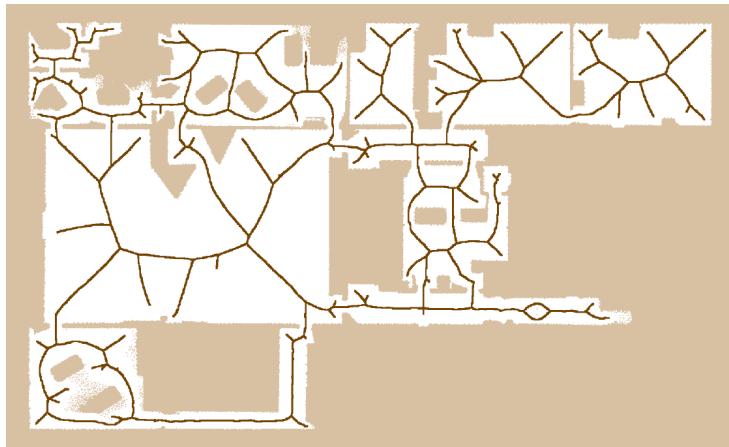


Abb. 2.6: Beispiel für eine topologische Karte [SK08]

den Freiraum zum Ziel berechnet werden.

Diese Arbeit verwendet occupancy grids, nicht nur wegen der genannten Vorteile, sondern vor allem wegen der vielen bereits gut bekannten Wegplanungsalgorithmen und Erkunungsstrategien, die auf diesen Rasterkarten arbeiten.

Bevor Planer und die Ansätze zur autonomen Erkundung näher betrachtet werden, wird zunächst OpenCV und seine wichtigsten Funktionen erläutert. Mit dieser Open-Source-Software wird in der Umsetzung häufig auf den entstandenen occupancy grids gearbeitet, um beispielsweise unbekannten Raum zu finden.

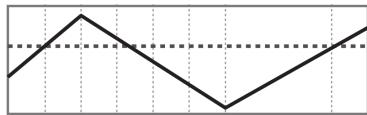
2.5 Aus OpenCV verwendete Computer-Vision-Algorithmen

Die **Open Source Computer Vision Library** - OpenCV ist eine kostenlose Bibliothek, die mehrere hundert Computer-Vision-Algorithmen für C/C++ und Python beinhaltet. Laut der Onlinerefenz [Ope16d] ist OpenCV modular aufgebaut, sodass im Softwarepaket viele einzelne Bibliotheken mit den verschiedenen Funktionalitäten enthalten sind. Im Folgenden werden die im Rahmen der Arbeit verwendeten Funktionen aufgezählt und kurz erläutert, damit es im Ausarbeitungsteil leichter nachzuvollziehen ist, aus welchem Grund bestimmte Funktionen verwendet werden und was diese bewirken.

Der Datentyp **Mat** [Ope16a] wird bei den meisten Operationen verwendet und dient in der Regel dem Speichern von Bilddaten, wobei jeder Bildpunkt eines Graustufenbildes einem 8-Bit unsigned integer (= unsigned char) in einem 2D-Array entspricht. Bei einem RGB-Bild hat jeder Pixel drei 8-Bit Zahlen als Farbinformation, die im Array hinterlegt sind. In der Dokumentation dieser Klasse wird beschrieben, wie diese Arrays intern aufgebaut sind. Neben Bildinformationen können diese Matrix-Typen auch Gleitkommazahlen enthalten und zum Verarbeiten von komplexen Matritzenkalkulationen benutzt werden. Alle gängigen Matrix- und Vektoroperationen sind auf diesen Datentypen implementiert.

Der **Threshold-Operator** [Ope16c] ist eine einfache, aber sehr nützliche Funktion. Hier-

bei wird eine Eingangsmatrix elementweise durchlaufen. Bei jedem Element wird überprüft, ob der gespeicherte Wert dem vorher festgelegten Threshold-Kriterium entspricht. Anschließend wird je nach Kriterium ein Wert an die selbe Stelle in der Ausgangsmatrix



(a) Funktionswert und Threshold-Level

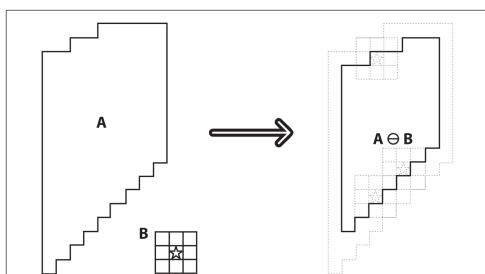


(b) Funktionswert nach einem Binary Threshold

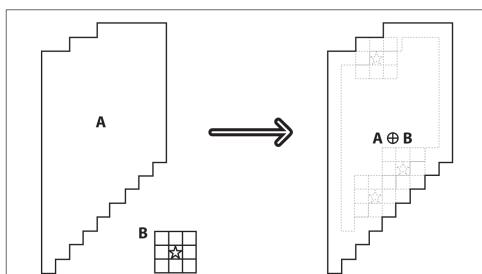
Abb. 2.7: OpenCV: Binary Threshold [BK08]

hinterlegt. Als Beispiel für ein Kriterium sei der klassische Schwellwertentscheider genannt. Hierbei wird, wenn der Wert auf oder über dem Schwellwert liegt, das Maximum des Wertebereiches zurückgegeben. Liegt der Wert unter dem definierten Schwellwert, so wird das Minimum des Wertebereichs zurückgeliefert, wie in Abbildung 2.7 illustriert. Dieser Operator findet in Unterkapitel 3.3 häufig Anwendung, um unerwünschte Farbübergänge die durch andere Prozeduren entstanden sind zu filtern. In OpenCV gibt es darüber hinaus noch andere vorimplementierte Threshold-Kriteria.

Die **Erode-/Dilate**-Funktionen [Ope16c] setzen jeweils die klassische Bilderosion bzw. die Bilddilatation um. Bei der Erosion werden in einem Grauwertbild in Abbildung 2.8 a) dunkle Bereiche im Bild vergrößert und helle Bereich verkleinert. Die Dilatation ar-



(a) Erosion von A mit Kernel B



(b) Dilatation von A mit Kernel B

Abb. 2.8: OpenCV: Erosion und Dilatation [BK08]

beitet umgekehrt, sie verkleinert dunkle Bildbereiche und vergrößert helle Bereiche in Abbildung 2.8 b). Erode bzw. Dilate werden in der Umsetzung häufig dazu verwendet, um Linien zu verbreitern und damit nicht zugeordnete, in der Nähe liegende freie Pixel in die Linien einzubeziehen.

Der **Sobel**-Operator [Ope16c] liefert die erste, zweite oder dritte Ableitung der Intensität, einer gegebenen Bildmatrix eines Graustufenbildes, in x- oder y-Richtung oder x-y-Richtung. Dies geschieht mit Hilfe einer Faltungsmatrix, die über das Bild geschoben wird. Das Resultat zeigt an wo sich der Helligkeitswert eines Bildes am stärksten ändert. Diese Information ist besonders nützlich wenn man nach Kanten in einem Bild

sucht und findet Anwendung im Canny-Algorithmus.

Die **Canny**-Funktion [Ope16c] ist eine der vielen bereitgestellten Methoden zur Kanten detektierung in Bildern mit OpenCV. Hierbei wird eine Mat-Variable übergeben, die die Bildinformationen trägt, in denen nach Kanten gesucht werden soll. Mit Hilfe des Canny-Algorithmus [Can86] werden Bildkanten ausfindig gemacht und in einer Ausgabematrix markiert. Das Ergebnis ist ein Graustufenbild mit den gleichen Dimensionen des Ursprungsbildes, wobei die Kanten als weiße Linien auf schwarzem Hintergrund eingetragen sind.

Der Canny-Algorithmus arbeitet nur auf Graustufenbildern und teilt sich in 5 aufeinanderfolgende Verarbeitungsschritte auf:

- Zuerst wird mit Hilfe eines Gausschen Filters das Bild geglättet und damit Rauschen entfernt.
- Anschließend werden die Helligkeitsgradienten des Bildes ausfindig gemacht. Dies geschieht in der Regel mit dem Sobel-Filter, der die erste Ableitung jeweils in x- und in y-Richtung bildet. Mit Hilfe des $\text{atan2}(y,x)$ können nun die Richtungen der Kanten als Winkel berechnet werden.
- In diesem Schritt werden die absoluten Kantenstärken berechnet, dabei werden pixelweise die Beträge der Ableitungen in x- und y-Richtung addiert.
- Im vierten Schritt werden dann alle potentiellen Kanten mit einer Thresholding-Operation hervorgehoben.
- Im letzten Teil des Prozesses werden die einzelnen Pixel mit Hilfe der Hysteresetechnik den entsprechenden Kanten zugeordnet. Dabei wird eine Kante von beiden Seiten abgetastet und Pixel die sich innerhalb zweier Schwellwerte befinden, der Kante zugeordnet.

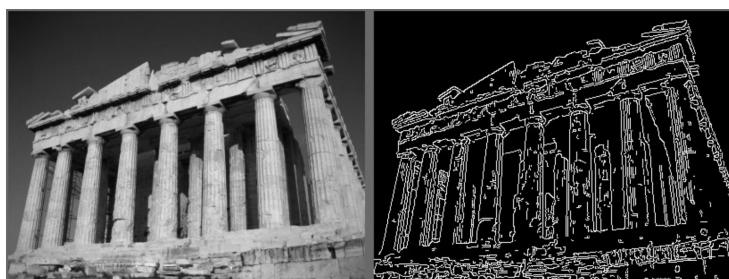


Abb. 2.9: OpenCV: Canny-Algorithmus [BK08]

In der Abbildung 2.9 kann man erkennen wie die Farbübergänge als Kanten detektiert wurden. Das macht die Canny-Funktion zu einem wichtigen Werkzeug in Unterkapitel 3.3, da dort Grenzen zwischen Grauwertübergängen erkannt werden müssen.

Die **findContours**-Funktion [Ope16c] ist mit Hilfe des Algorithmus von Suzuki et al. [SA83] in der Lage, zusammenhängende Linien in einem Bild zu erkennen und die

Anfangs- und Endpunkte sowie deren Wendepunkte in einem Ausgabearray zu hinterlegen. Das ist ideal, um aus abstrakten Datenmengen wie Bildern oder den in Unterkapitel 2.4 erläuterten occupancy grids, konkret verarbeitbare Punkte zu erhalten. Damit der Algorithmus optimal arbeiten kann, muss das zu untersuchende Bild in monochromer Farbe vorliegen und die Linien sollten bereits durch einen Kantenerkennungsalgorithmus vorgezeichnet sein. Das heißt, dass `findContours` immer in Kombination mit der Canny-Funktion verwendet werden sollte um ein optimales Ergebnis zu erzielen.

Der **LineIterator** [Ope16b] stellt eine Art Wrapper-Klasse für den Bresenham-Algorithmus zur Linieninterpolation dar. Zu Anfang wird dem Konstruktor eine Bildmatrix übergeben, sowie Anfangs- und Endpunkt der zu interpolierenden Linie. Danach kann auf das instanzierte Objekt dieser Klasse einfach ein Inkrement- oder Dekrementoperator verwendet werden, um sich den nächsten oder vorherigen Punkt auf der interpolierten Line, zwischen den gegebenen Anfangs- und Endpunkten, berechnen zu lassen. Das Objekt verhindert hierbei auch, dass über die Bildgrenzen hinaus gearbeitet wird, um etwaigen Speicherfehlzugriffen vorzubeugen. Besondere Anwendung findet der LineIterator immer, wenn in dieser Arbeit ein Bild entlang einer Linie abgetastet werden muss. Das ist besonders beim Simulieren von Entfernungssensoren in Unterabschnitt 3.2.2 der Fall.

Nachdem die Werkzeuge aufgezählt sind, mit denen in dieser Arbeit auf occupancy grids gearbeitet wird, werden im Folgenden Strategien zur autonomen Erkundung näher betrachtet.

2.6 Etablierte Erkundungsstrategien

An der Problematik der autonomen Erkundung mit Hilfe von mobilen Roboterplattformen wird bereits seit einiger Zeit geforscht. Dabei ist eine große Zahl an Lösungsansätzen entstanden, die Lidoris in "State Estimation, Planning and Behavior" [Lid11] in die folgenden vier Klassen aufteilt:

- Bei der **pattern-based search** Methode wird ein vorher definiertes Gebiet mit Hilfe eines statischen Bewegungsplanes erkundet. Der Bewegungsablauf kann beispielsweise rasterförmig oder spiralförmig sein und ist dabei unabhängig von neuen Erkenntnissen während der Erkundung. Dabei werden Hindernisse entweder als bekannt betrachtet oder dynamisch umfahren.
- Der **semi-autonomous** Ansatz benötigt den aktiven Eingriff eines Menschen, um zu entscheiden, in welche Richtung sich ein mobiler Roboter als nächstes auf seiner Erkundungsfahrt bewegen soll. Dabei wird nur in die Zielvorgabe eingegriffen, die Wegplanung und das Ausweichen von Hindernissen geschieht autonom.
- Der **maximize-terrian-knowledge** Ansatz verfolgt die Idee Gebiete anzusteuern, die zuvor noch unerkannt geblieben sind. Hierbei wird die Welt in erkundete und unerkannte Bereiche aufgeteilt. Da die Grenzen zwischen diesen Gebieten potentielle Ziele des Roboters darstellen, wird dieser Ansatz in den meisten wissenschaftlichen Arbeiten, die sich mit mobiler autonomer Erkundung befassen, oft

frontier-based approach genannt. Die beschriebenen Grenzen werden daher in dieser Arbeit auch als **Frontiers** bezeichnet.

- Der letzte Erkundungstyp ist der sogenannte **information-based autonomous exploration** Ansatz, der eine ähnliche Strategie wie der frontier-based approach verfolgt. Allerdings wird neben dem potentiellen Informationsgewinn auch bewertet, ob und wie sehr die neue Information über die Umwelt, die Qualität der Karte und der Positionsbestimmung verbessern kann.

Im Rahmen dieser Thesis wird ein frontier-based approach verwendet, da zu diesem Ansatz bereits einige etablierte Methoden erarbeitet wurden. Die besten Vertreter dieser Erkundungsstrategie bilden den State-of-the-Art im Bereich autonome Erkundung.

Der frontier-based approach wurde erstmals in der Arbeit "A Frontier-Based Approach for Autonomous Exploration" von Brian Yamauchi [Yam97] beschrieben. Darin wird ein occupancy grid als Grundlage einer Karte verwendet, worin die Umwelt in Raster diskretisiert wird und einzelne Zellen entweder den Zustand *besetzt*, *frei* oder *unbekannt* haben können. Damit wird eine Unterscheidung in bekannte und unbekannte Bereiche möglich, die in Abbildung 2.10 a) weiß bzw. grau gepunktet gekennzeichnet sind. Während der Erkundungsfahrt werden die Grenzen zwischen diesen Bereichen erkannt und deren **Zentroiden** berechnet.

Die Zentroiden in Abbildung 2.10 c) sind dabei die geometrischen Schwerpunkte von Polylinien und berechnen sich nach folgendem Schema:

$$\vec{c}(F) = \left\{ \frac{\sum_{x_i \in F} x_i}{n} \vec{e}_x + \frac{\sum_{y_i \in F} y_i}{n} \vec{e}_y \right\} \quad (10)$$

Hierbei ist F die Menge aller Punkte in einer Polylinie und n die Anzahl dieser Punkte. Diese berechneten Zentroiden sind während der Erkundung die Zielvorgabe für den Globalplaner des mobilen Roboters.

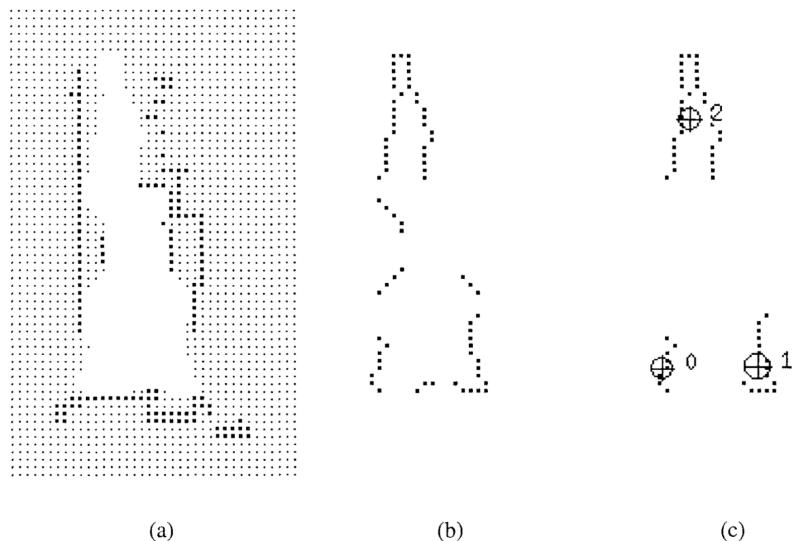


Abb. 2.10: *Frontiers und Zentroiden* [Yam97]

Die Erkennung der Grenzen erfolgt in Yamauchis Arbeit zellenweise. Das heißt, dass jede Zelle dahingehend überprüft wird, ob sie sich an einem Übergang zwischen bekannt und unbekannt befindet und ob sie an eine andere Zelle angrenzt, die bereits einer Grenze zugewiesen wurde. Erfüllt sie beide Bedingungen, so wird diese Zelle der bereits angelegten Grenze zugeordnet, erfüllt sie nur die erste Bedingung, so wird eine neue Grenze angelegt. Ist die Frontier-Detektion beendet, kann das Ergebnis so aussehen wie in Abbildung 2.10 b), wobei a) das ursprüngliche occupancy grid darstellt. Bei der Erkundung wird immer die nächstgelegene Grenze angefahren, durch die das Fahrzeug hindurch passen würde. Sind keine Grenzen dieser Art mehr auf der Karte erkennbar, so terminiert der Algorithmus und ein Gebiet gilt als erkundet.

Die zweite Strategie, die dieser Arbeit als Grundlage dient, basiert ebenfalls auf dem frontier-based approach. Allerdings wird bei der **Next-Best-View**-Technik (NBV), die erstmals von González-Baños in "Navigation Strategies for Exploring Indoor Environments" [GBL02] dargelegt wird, ein größerer Fokus auf die Auswahl der Grenzen und des Blickwinkels auf diese gelegt. Das hat zum Ziel, Grenzen mit potentiellem maximalen Informationsgewinn zu favorisieren und eine optimale Sensorposition zu gewährleisten, um die Anzahl der benötigten Manöver während einer Erkundung zu verringern. Dies ist insbesondere für mobile Roboter mit Ackermann-Lenkung interessant, da Wendemanöver einerseits kompliziert sind, dadurch viel Rechenaufwand benötigen, und andererseits zusätzliche gefahrene Strecke produzieren, was die Batterien belastet. Der Ablauf der NBV-Technik nach Eingang neuer Karteninformationen gestaltet sich wie folgt:

Zuerst werden, wie in Yamauchis Ansatz beschrieben, die Grenzen zwischen bekanntem und unbekanntem Gebiet ermittelt (siehe Abbildung 2.11 a)).

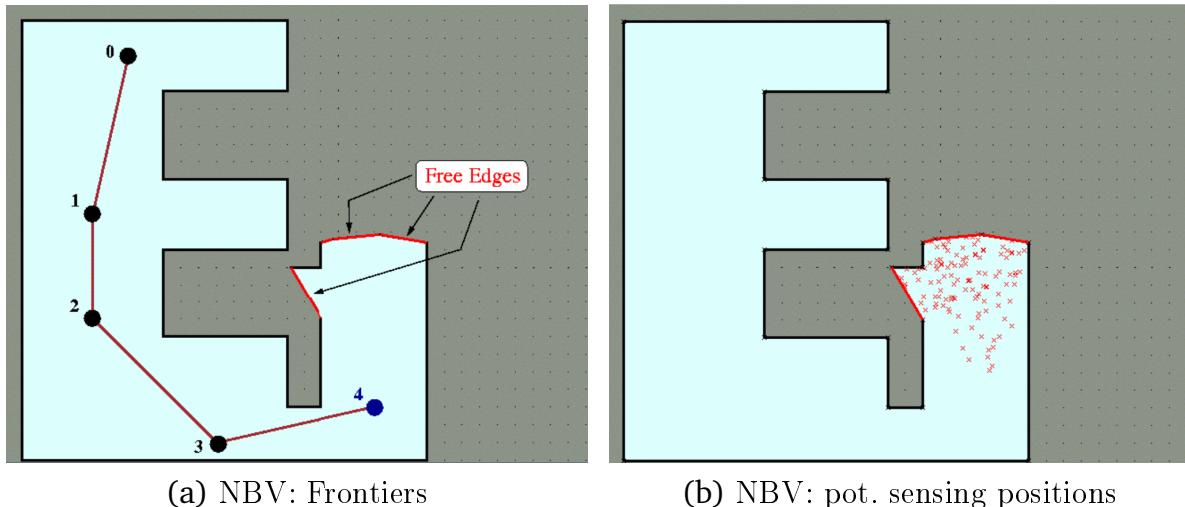


Abb. 2.11: *NBV: Frontiers und pot. Positionen* [GBL02]

Als Nächstes werden in der Nähe der einzelnen Grenzen, konkret innerhalb der maximalen Sensorreichweite, zufällig Punkte q , sogenannte **sampling points**, im freien Raum erzeugt, wie in Abbildung 2.11 b) illustriert ist.

Im dritten Schritt wird die Länge der Grenzen berechnet. Es werden alle Punkte verworfen, von denen aus ihre zugehörige Grenze nicht vollständig in das Sichtfeld, das

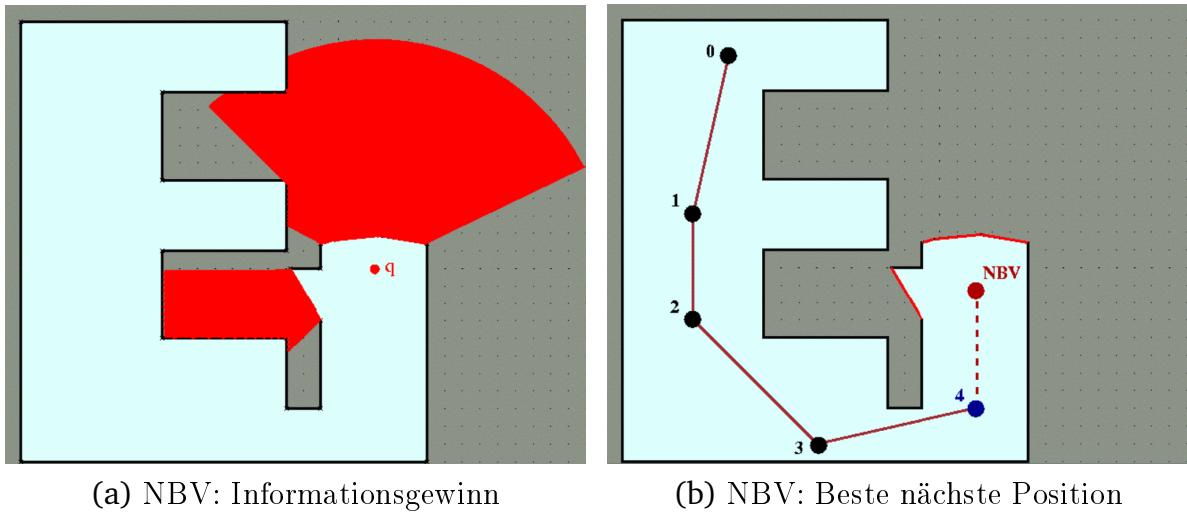


Abb. 2.12: *NBV: pot. Informationsgewinn und nächste beste Position [GBL02]*

sogenannte **field of view**, des Roboters passen würde.

In Schritt vier werden Pfade $\mathcal{L}_{\text{path}}$ zu allen übrig gebliebenen Punkten berechnet. Es werden alle Punkte verworfen die nicht erreichbar sind.

Im fünften Schritt wird an den erreichbaren Punkten aus dem vorherigen Schritt, die potentielle zusätzlich gewonnene freie Fläche $A(q)$ berechnet, die sich durch eine Messung an diesem Punkt ergeben würde. In Abbildung 2.12 a) ist das anhand eines der möglichen Punkte dargestellt.

Im letzten Schritt wird die nächste beste Messposition q_{k+1} durch den Punkt q aus der Menge aller potentiellen Messpositionen Q bestimmt, der in Gleichung 11 die Funktion $g(q) \quad \forall q \in Q$ maximiert. Für die gezeigten Skizzen ist der beste Punkt in Abbildung 2.12 b) dargestellt.

$$g(q) = A(q)e^{-\lambda L(q, q_k)} \quad (11)$$

Die Konstante λ ist hierbei das Maß, wie sehr die Wegstrecke $L(q, q_k)$ von der Position q_k des Roboters zur potentiellen Messposition q als Bewertungskriterium eingeht. Wird λ auf Null gesetzt, zählt ausschließlich der Informationsgewinn.

Der NBV-Algorithmus terminiert wie Yamauchis Ansatz erst dann, wenn keine passierbaren Grenzen mehr auf dem occupancy grid zu finden sind.

In dieser Arbeit wird die frontier-based approach Strategie Yamauchis im Folgenden als naiver oder einfacher Ansatz bezeichnet und dient später in Kapitel 4 als Vergleich für die komplexe Next-Best-View-Strategie.

Da für alle Strategien eine Wegplanung zu den gefundenen Frontiers benötigt wird, um ihre Erreichbarkeit zu überprüfen und ihre räumliche Nähe zum Roboter zu bewerten, wird im nächsten Abschnitt dieses Problem adressiert und der in dieser Thesis verwendete Planer vorgestellt.

2.7 Wegplanung auf mobilen Robotern

Das Thema der kollisionsfreien Trajektorienplanung für mobile Roboter ist trotz vieler guter Lösungsansätze nach wie vor Teil aktueller Forschung. Es gibt keine universelle Lösung dieses Problems welche auf alle Situationen gleich gut anwendbar ist. Vielmehr sind viele verschiedene Lösungen entstanden, die abhängig vom Robotertyp, der Umwelt und anderer Anforderungen zum Einsatz kommen. Es hat sich bewährt, dieses komplexe Problem in einen lokalen und einen globalen Kontext aufzuteilen.

Der Globalplaner befasst sich mit dem globalen Kontext. Hierbei wird versucht, mit verschiedenen Algorithmen, meistens auf einer Rasterkarte, also einem occupancy grid, einen kollisionsfreien Pfad zu planen. Dabei wird oft noch keine Rücksicht auf die besonderen kinematischen Eigenschaften des Roboters genommen, um die Komplexität der Aufgabe zu verringern. So können anwendungsbedingt verschiedene Globalplaner gleichzeitig verwendet werden.

Im lokalen Kontext wird vor allem den kinematischen Eigenschaften der mobilen Plattform Rechnung getragen, da beispielsweise ein autoähnliches Fahrzeug nicht die gleichen Manöver durchführen kann wie ein kettengetriebenes Fahrzeug. Zusätzlich werden auf dieser Ebene oft die Ausweichmanöver für dynamisch auftretende Hindernisse geplant. Diese Art von Hindernissen, wie beispielsweise Menschen, bewegliche Gegenstände oder andere Roboter, sind nicht auf der Karte verzeichnet und werden deswegen nicht beim globalen Plan berücksichtigt. Der hier eingesetzte Time-Elastic-Band-Planer von Rösmann et al. [RFW⁺12, RFW⁺13] wird in Unterabschnitt 3.1.2 kurz erklärt und dessen Besonderheiten erläutert.

Bei den meisten modernen Globalplanern wird versucht, den Roboter abhängig von seiner räumlichen Ausdehnung, in ausreichendem Abstand an Hindernissen vorbei zu führen. Das soll zum einen verhindern, dass der lokale Planer vor ausweglose Situationen gestellt wird, was einen neuen Globalplan erfordern würde. Zum anderen werden so Wege ausgeschlossen, die der Roboter nicht befahren könnte, weil er zu groß für bestimmte Passagen ist.

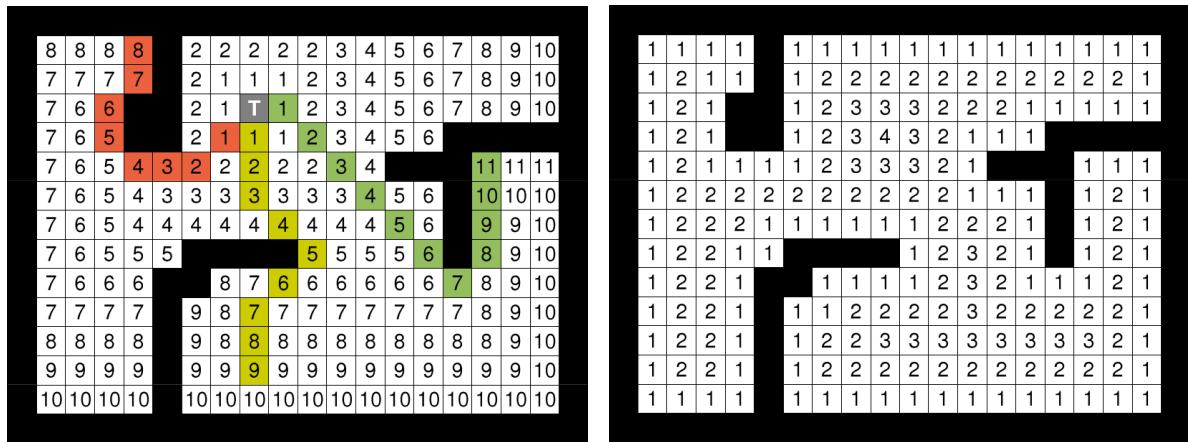
Um so eine Verhaltensweise zu modellieren, wird nicht etwa der Roboter bei den Berechnungen als Polygon angenommen, das wäre zu rechenaufwändig, dieser wird weiterhin als punktförmig betrachtet, sondern die Hindernisse werden vergrößert. Man kann die Hindernisse direkt um einen bestimmten Faktor ausdehnen, was aber dazu führen kann, dass bestimmte Wege virtuell nicht passierbar werden, obwohl sie es für den mobilen Roboter in der Realität wären. Aus diesem Grund wird gerne die Potentialfeldmethode verwendet. Hierbei werden der Roboter und die Wand als gleichnamig geladene Teilchen betrachtet, die sich abstoßen. Damit wird ein exponentiell ansteigendes Gefahrenpotential zu Hindernissen beschrieben, was bei der Wegplanung in Form von Kosten beachtet wird. Dabei gibt es letale Kosten, also Zellen, bei deren Durchquerung Schaden am Roboter entsteht, bis hin zu keinen Kosten im freien Raum. Nach der Ausdehnung der Hindernisse oder Berechnung des Potentialfeldes kann ein A*- oder ein Dijkstra-Algorithmus verwendet werden, um den kürzesten Weg zum Ziel zu finden. Diese Algorithmen eignen sich besonders gut, da das Raster automatisch einen Graphen aufspannt, bei dem die Gitterkreuzungen Knoten und die Gitterlinien Kanten darstellen.

In dieser Arbeit kommt jedoch ein etwas anderer Ansatz zum Einsatz. Wie in Unter-

Kapitel 2.6 erläutert, müssen Grenzbereiche zwischen bekanntem und unbekanntem Gebiet auf einer Rasterkarte unter anderem dahingehend bewertet werden, ob und wie sie vom Fahrzeug aus erreichbar sind. Da das je nach Situation denkbar viele Grenzen sein können, müssen entsprechend viele Wege geplant werden. Mit den oben genannten Verfahren wäre dies viel zu aufwändig, da jeder Weg von neuem geplant werden müsste. Zusätzlich können bei potentialbasierten Methoden lokale Minima entstehen, die es ohne Modifikationen unmöglich machen können, einen Pfad zum Ziel zu finden. Daher wird hier der **Path-Transform-Algorithmus** von Alexander Zelinsky [Zel94] angewendet, der für dieses Problem eine geschickte Lösung anbietet und in drei Schritten abläuft:

Zuerst wird eine Distanztransformation (**Distance-Transformation**) nach Jarvis et al. [JB86] auf einem Raster mit den Dimensionen, die der Karte entsprechen, durchgeführt. Hierbei werden Elementweise die Kosten berechnet, um von einer beliebigen freien Zelle zur Zielzelle zu gelangen. Die Kosten zwischen zwei Zellen c_i und c_j , zwischen denen sich kein Hindernis befindet, können beispielsweise über die Manhattanmetrik oder die euklidische Norm berechnet werden. Danach kann man bereits den kürzesten Weg, ohne besondere Berücksichtigung von Hindernissen, von einer freien Zelle zum Ziel durch den steilsten abfallenden Gradienten finden. Das kann man in Abbildung 2.13 a) erkennen, wobei T das Ziel darstellt.

Ab dem zweiten Schritt beginnt der Ansatz von Zelinksy. Hier wird eine Hindernis-



(a) Distance-Transformation zu Punkt T

(b) Obstacle-Transformation

Abb. 2.13: Veranschaulichung der Distance- und Obstacle-Transformationen [WP07]

transformation (**Obstacle-Transformation**) berechnet. Dabei wird für jede freie Zelle die Distanz zum nächsten Hindernis berechnet. Das Ergebnis wird wie im oberen Schritt in einem separaten Raster, wie in Abbildung 2.13 b), festgehalten.

Im dritten Schritt wird dem Umstand Rechnung getragen, dass nach dem ersten Schritt die Pfade zwar optimal hinsichtlich ihrer Länge sind, aber noch sehr nahe an Hindernissen vorbeiführen können. Dazu wird die Pfadtransformation (**Path-Transformation**) durchgeführt. Hierbei werden für jede Zelle c die Kosten auf dem kürzesten Weg zum

Ziel c_g zu gelangen gemäß folgender Gleichung berechnet:

$$\phi(c, c_g) = \min_{C \in P_c^{cg}} \left(l(C) + \alpha \sum_{c_i \in C} c_{\text{danger}}(c_i) \right) \quad (12)$$

P_c^{cg} ist dabei die Menge aller Pfade von c nach c_g , C ist dabei ein Pfad aus der Menge P_c^{cg} und $l(C)$ ist die Länge eines Pfades. Die Konstante α gewichtet die Summe aller Kosten aus Schritt 2 entlang des Pfades C .

Nach der Pfadtransformation kann der kürzeste sichere Pfad ermittelt werden, indem man immer der Richtung des steilsten Abfalls folgt.

Diese Art Wege zu planen wird deswegen in dieser Arbeit verwendet, da sie den Vorteil hat, dass die Pfadtransformation nur durchgeführt werden muss, wenn sich entweder die Karte geändert hat oder das Fahrzeug bewegt wurde. Ist das nicht geschehen, so können Wege von einem Startpunkt zu beliebig vielen Zielen, je mit Komplexität $O(n)$, gefunden werden.

Im nächsten Abschnitt der Grundlagen werden einige Typen von Entfernungssensoren und der in dieser Arbeit verwendete aufgezählt, mit denen occupancy grids erstellt werden können.

2.8 Entfernungssensoren

Um eine Karte der Umgebung aufzubauen, benötigt ein mobiler Roboter geeignete Sensoren. Hierfür werden in der Regel Entfernungssensoren verschiedener Art eingesetzt. Das "Handbook of Robotics" [SK08] teilt die Sensortypen in fünf verschiedene Klassen ein: Stereo Vision-, Laser-Based-, Time-of-Flight-, Modulation- und Triangulation Range Sensors.

Bei Entfernungsmessung durch **Stereo-Vision** werden Bilder der selben Szene von zwei nebeneinander angebrachten Kameras aufgenommen. Danach kommt das Konzept der Triangulation zum Einsatz, dabei wird das selbe Objekt, z.B. ein Pixel, in beiden Bildern identifiziert, so kann man mit den zwei Kameras ein Dreieck aufspannen. Da der Abstand der Kameras zueinander bekannt ist, kann die Entfernung des Objekts zu den Kameras bestimmt werden. Die größte Herausforderung dieser Technik ist es, zuverlässige Treffer für Pixel in beiden Bildern zu finden, die sich auf den selben Punkt einer Szene beziehen.

Laser-Based Range Sensors ist der Oberbegriff für alle entfernungsmeßenden Sensoren auf Laserbasis und sind oft die Grundlage für die unteren drei Sensortypen. Sie bieten sich besonders an, da Laser selbst in kompakter Bauweise hohe Intensitäten erzeugen können. Zusätzlich können Laser sehr gut fokussiert werden und fremde interferierende Frequenzen lassen sich leicht herausfiltern, da Laser inhärent monochrom sind. Anzumerken ist, dass alle laserbasierten Entfernungsmeßgeräte, wenn auch weniger präzise, mit beliebigen anderen Lichtquellen funktionieren würden. Die nächsten drei Sensortypen sind häufige Vertreter dieser Sensorenart.

Time-of-Flight Range Sensors messen wie lange ein Lichtimpuls benötigt, um von der Lichtquelle, über eine reflektierende Fläche wieder zurück zum Detektor zu kommen. Multipliziert man diese Reisezeit mit der Lichtgeschwindigkeit, erhält man die zurückgelegte Distanz, was dem Wirkprinzip eines Radars sehr ähnlich ist. Deshalb werden diese Sensoren in Verbindung mit einem Laser laser radar - LADAR oder light detection and ranging - LIDAR Sensoren genannt. Die Genauigkeit dieser Sensoren hängt von der minimalen zeitlichen Auflösung des Systems ab, was den minimalen Abstand und die kleinste auflösbare Struktur bestimmt. Die maximale Reichweite hängt zum größten Teil von der maximalen Intensität bzw. Sensorempfindlichkeit ab. Die typischen Vertreter dieser Art, die regelmäßig auf mobilen Robotern eingesetzt werden, haben eine Reichweite zwischen 10 und 100 Metern bei einer Genauigkeit zwischen 5 und 10 Millimetern.

Modulation Range Sensors senden ein kontinuierliches Lichtsignal aus, das entweder frequenz- oder amplitudenmoduliert ist. Wird das Lichtsignal nun reflektiert und anschließend detektiert, so kann man über die Modulationsänderung Rückschlüsse auf die zurückgelegte Distanz ziehen. Diese Sensoren können Reichweiten von 20-40 Metern bei einer Auflösung von 5mm abdecken.

Triangulation Range Sensors basieren auf dem gleichen Prinzip wie die Stereosensoren. Hierbei ist die Position und der Abstrahlwinkel der Laserlichtquelle und des Detektors bekannt. Mit Hilfe einer einfachen Triangulation lässt sich so berechnen, an welchem Punkt im Raum der Laserstrahl reflektiert wurde. Dabei hängt die Präzision hauptsächlich davon ab, wie gut die Maße des Messaufbaus bekannt sind. Diese Sensoren tendieren wegen ihres Wirkprinzips dazu, nicht besonders kompakt zu sein.

Die laserbasierten Sensoren können alle nur Punktstrecken messen, daher muss der Sensor um die Z-Achse geschwenkt werden, um planare Entfernungsinformationen erhalten zu können. Dabei wird in festen Winkelabständen eine Messung durchgeführt. Um dreidimensionale Entfernungsinformationen wahrzunehmen, muss der Sensor zusätzlich um eine weitere Raumachse geschwenkt werden. Die Winkelauflösung hängt dann im Wesentlichen davon ab, wie viele Impulse der Sensor senden, empfangen und verarbeiten kann.

Im Rahmen dieser Arbeit wird ein Time-of-Flight Range Sensor in Form einer Kinect-Kamera in der Version 2 eingesetzt, die mit einer Infrarotlichtquelle arbeitet. Dabei werden die reflektierten Strahlen in den einzelnen Pixeln des Sensorchips registriert und mit Hilfe der Laufzeit die Entfernung zum Reflektionsursprung, also dem Hindernis, berechnet. Die Reichweite beträgt etwa 10 Meter, das Sichtfeld 70° in horizontaler und 60° in vertikaler Ausrichtung. Die Auflösung des Tiefenbildes beträgt etwa 7 Pixel/Grad in horizontaler Richtung. Damit sind die Kenndaten zwar geringer als bei laserbasierten Sensoren, dafür ist dieser Sensortyp für einen Bruchteil, etwa einem Zehntel der Kosten, verfügbar.



3 Umsetzung

Im Kapitel Umsetzung wird die mobile Plattform, auf der die autonome Erkundung stattfindet, beschrieben. Außerdem werden alle Entwicklungsschritte wie dem Entwickeln der Simulation und dem Implementieren der einzelnen Ansätze, bis zur Integration auf dem Roboter beschrieben.

3.1 Beschreibung des Modellautos

Bei dem Modellauto, zu sehen in der Abbildung 3.1, handelt es sich um eine Eigenanfertigung, die für das Projektseminar Echtzeitsysteme an der TU-Darmstadt entwickelt wurde. Das Chassis mit der Lenkung, dem Antriebsstrang und der Motorregelung wurde einem fertigen Modellbausatz der Firma Tamiya entnommen. Darauf ist eine Holzplatte montiert, die der Befestigung der Steuerelektronik, des Akkus, der Kinect, der übrigen Sensoren und eines Mainboards dient. Geschützt wird die Elektronik mit seitlich angebrachten Aluminiumwänden und einem Schaumstoffpolster als Stoßstange. Über dem Mainboard ist eine weitere Platte angebracht, die zusätzliche Elektronik und Sensoren trägt.

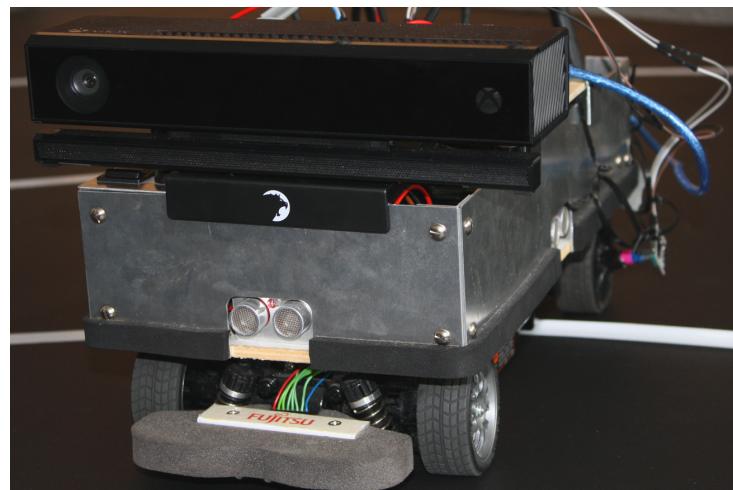


Abb. 3.1: *Bild des Modellautos*

3.1.1 Verwendete Hardware und Sensoren

Zur Ansteuerung der Lenkung und des Antriebs sowie dem Empfangen von Messwerten der Ultraschallsensoren und eines Hall-Sensors wird ein 16-Bit Fujitsu Mikrocontroller der Serie MB96300 verwendet. Dieser kommuniziert via UART über ein USB-Kabel, mit dem darüber verbauten Mainboard. Dabei handelt es sich um ein Mitac PD10BI MT Thin Mini-ITX DN2800, mit einem Intel Quadcore-Prozessor und einer Intel HD Graphics Onboard-Grafikkarte. Als Hauptspeicher sind 8 GB DDR3-1600 RAM in Form eines einzelnen Moduls verbaut. Zusätzlich bietet das Mainboard einen 1Gbit/s Ethernet-Anschluss, VGA- und HDMI-Anschlüsse, USB 3.0/2.0, SATA-Ports und einen Intel Dual Band Wireless AC 7260 Netzwerkadapter, der an zwei externe WLAN-Antennen angeschlossen ist. Über einen der integrierten PCI-Express Ports ist eine 60GB große

SSD-Festplatte von Kingston verbaut. Die Stromversorgung wird durch einen 3200 mAh fassenden Li-Fe Akku gewährleistet. In Abbildung 3.2 sind die Positionen der genannten Bauteile dargestellt.

Bei den verbauten Sensoren handelt es sich unter anderem um drei Ultraschallsenso-

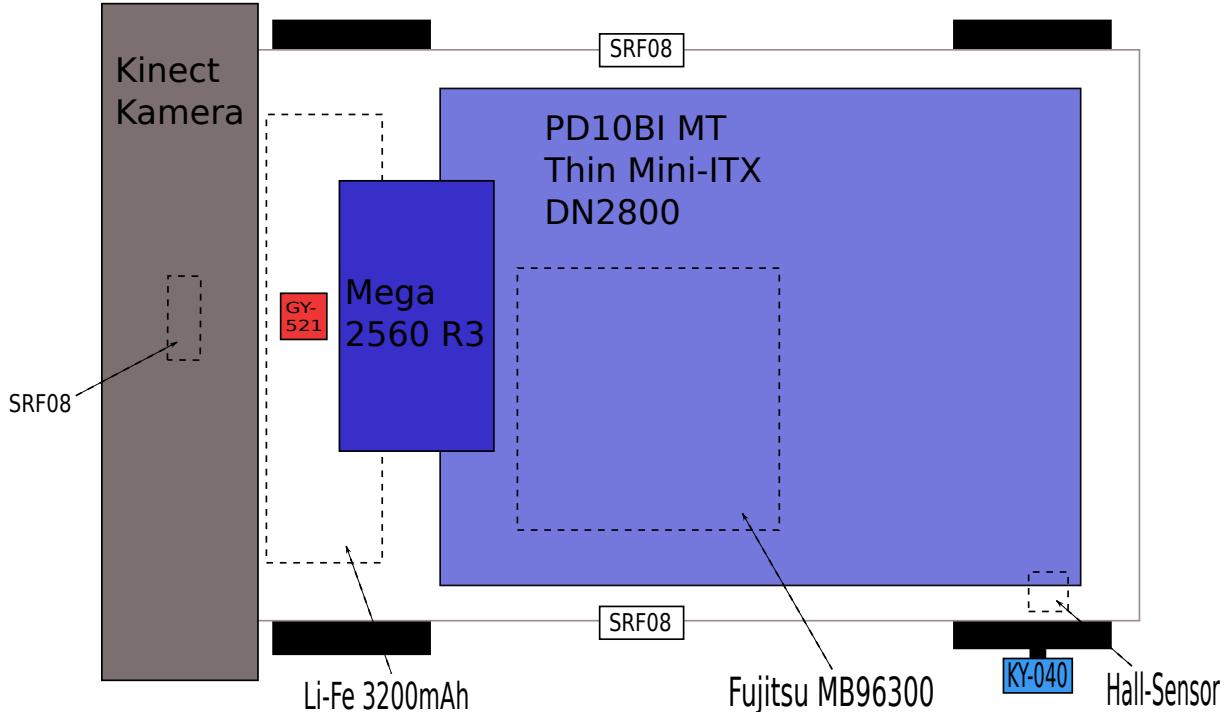
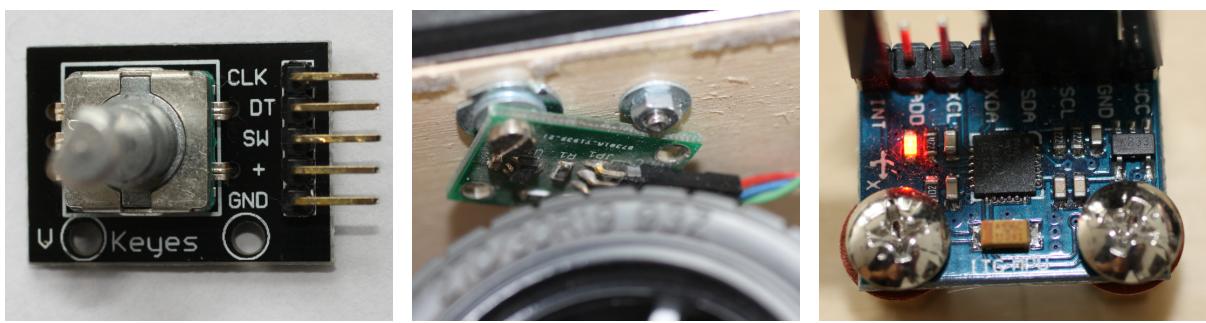


Abb. 3.2: Anordnung der einzelnen Komponenten im Modellauto

ren vom Typ SRF08, um Hindernisse im Nahbereich schnell, aber ungenau erkennen zu können. Diese sind in Fahrtrichtung links und nach rechts ausgerichtet, wie man in Abbildung 3.1 an den Aussparungen in den Seitenplatten erkennen kann. Außerdem be-



(a) KY-040

(b) Hall-Sensor

(c) Gyro./Acc. GY-521

Abb. 3.3: Sensoren am Auto

findet sich ein Hall-Sensor (Abbildung 3.3 b.) über dem Hinterrad, der mit Hilfe von im Rad verbauten Permanentmagneten die Radumdrehungen messen kann. Daraus lässt sich der zurückgelegte Weg ermitteln. Am Rad befindet sich seitlich ein zusätzlicher optischer Inkrementaldrehgeber vom Modell KY-040 (Abbildung 3.3 a.), der eine feinere Rotationsauflösung besitzt, aber nur für niedrige Geschwindigkeiten, etwa $0.3 \frac{m}{s}$

genutzt werden kann. Die minimal auflösbare Wegstrecke lässt sich mit Gleichung 13 bestimmen.

$$s_{\min} = \frac{r_{\text{Rad}} 2\pi}{n_{\text{PulseProRadumdrehung}}} \quad (13)$$

Beim KY-040 ergibt sich durch 60 messbare Pulse pro Radumdrehung eine Auflösung von 3,4 mm. Beim Hall-Sensor basierten Drehgeber ergeben sich mit 8 Pulsen pro Radumdrehung 2,5 cm als minimal messbare Wegstrecke. Um die Orientierung des Modellautos zu bestimmen, ist ein Gyro-Sensor, wie in Abbildung 3.3 c) zu sehen, vom Typ GY-521 am Fahrzeug angebracht, der wie der KY-040 Drehgeber mit einem MEGA2560 R3 Mikrocontroller der Firma Funduino verbunden ist. Dieser kommuniziert wie der Fujitsu-Mikrocontroller über UART und einem USB-Kabel mit dem Mainboard. Die für diese Arbeit essentiellen Tiefeninformationen des Raumes werden von einer Microsoft Kinect Version 2 geliefert. Diese ist ein Time-Of-Flight Sensor und stellt auf Kosten der Auflösung eine leichte, kostengünstige Alternative zu einem klassischen LIDAR dar.

3.1.2 Verwendete Software

Als Betriebssystem kommt auf dem Roboter Lubuntu zum Einsatz, was auf Ubuntu 14.04LTE basiert. Durch die Verwendung von ROS erhält man Zugriff auf eine Vielzahl kostenloser Open-Source-Pakete, in denen häufig vorkommende Probleme oft sehr gut gelöst sind. Dieser Umstand erleichtert den Entwicklungsprozess für einen autonomen Roboter enorm. Im Folgenden sind alle wichtigen Pakete und deren Funktionen genannt, die zum Einsatz kommen.

Die Kommunikation mit dem Mikrocontroller, der die Motoren und die Ultraschall-sensoren ansteuert, wird mit einem ROS-Node, der im Rahmen des Projektseminars Echtzeitsysteme bereitgestellt wurde, bewerkstelligt. Besagter Node, im Folgenden als **car_handler** Node bezeichnet, empfängt die Sensordaten vom Mikrocontroller und veröffentlicht diese als Topic. Umgekehrt können Aktuatorenbefehle über einen ROS-Topic an car_handler gesendet werden, der anschließend diese Befehle interpretiert und an den Mikrocontroller zur Ausführung weitergibt. Zusätzlich wurde im Verlauf des Projektseminars von der Arbeitsgruppe Apollo13 [EA16] ein weiterer Node entwickelt, der mit Hilfe des Gyrosensors, den Daten des Drehgebers und den Formeln aus Unter-kapitel 2.2 berechnet, wo sich der Roboter nach jedem Zeitschritt in Weltkoordinaten befindet. Es wird damit eine Odometrie bereitgestellt, die allerdings nicht weiter gefil-tert wird und damit wegen des Sensordrifts des Gyrosensors mit der Zeit an Genauigkeit einbüßt.

Das ROS-Paket **gmapping** [Ger07] passt das Gmapping Programm von OpenSlam [Ope07] an ROS an und stellt einen SLAM-Algorithmus zum Kartographieren der Um-welt mit einem mobilen Roboter zur Verfügung. Hierbei wird ein ROS-Node betrieben, der die Position des Roboters und die Entfernungsdaten eines LIDARs oder ähnlichem entgegen nimmt und die gewonnenen Daten auf einem occupancy grid einordnet. Da-bei wird versucht in regelmäßigen zuvor bestimmten Zeitschritten, mit Hilfe eines Scan-Matching-Algorithmus der anhand des Laserscans die Position des Roboters schätzt, eine

neue, um die neuen Entfernungsdaten ergänzte, Karte zu erstellen. Die Qualität dieser Schätzung hängt stark vom Typ des Entfernungsmessers ab. Kann die Position nicht gut geschätzt werden, weil die Scans zu wenige Informationen tragen, wird zusätzlich die von der Odometrie gelieferte Position verwendet, um die Scandaten auf der Karte zu platzieren. Die neu entstandene Karte wird danach von einem Partikelfilter als Partikel betrachtet. Jedem Partikel wird vom Filter eine Punktzahl zugeordnet, die sich aus der Unsicherheit bei der Positionsschätzung und der Anordnung der Scan-Daten ergibt. Fällt diese Punktzahl im Verlauf einer Erkundung unter einen Schwellwert oder wird die maximal erlaubte Anzahl an Partikeln überschritten, so wird das Partikel verworfen und ein neues auf Basis der vorhandenen geschätzt. Der Partikelfilter ermittelt aus der Menge der Partikel den wahrscheinlichsten Aufenthaltsort des Roboters und das dazu passende occupancy grid. OpenSlam empfiehlt LIDARs mit sehr großer Reichweite, um die besten Ergebnisse zu erzielen. Bei der Verwendung einer Kinect und unter hohen Winkelgeschwindigkeiten, kann es wegen des geringen Sichtfelds und der beschränkten Reichweite, häufig passieren, dass keine Position geschätzt werden kann und deshalb nur die Odometrie bei der Einordnung der Scan-Daten verwendet wird. Daher muss vermieden werden, dass das Fahrzeug zu hohe Drehraten erfährt und eine Schätzung der Position anhand der Entfernungsdaten unmöglich macht. Denn nur mit Hilfe des Partikelfilters und der daraus gewonnenen geschätzten Position, kann die zuvor mit Hilfe der Odometrie (siehe Unterabschnitt 3.2.1) ermittelte Position korrigiert werden.

Damit die Kinect Kamera unter ROS funktioniert, wurde an der Universität Bremen das Paket **iai_kinect2** [Wie15] von Wiedemeyer et al. entwickelt, das den Treiber libfreenect2 der Gruppe OpenKinect [Ope] für Linux benutzt. Wenn iai_kinect2 aktiv ist, werden mit der gewünschten Frequenz nur Daten auf den Topics gepublished, die aktive Subscriber haben. Will man, wie in dieser Arbeit ausschließlich die Tiefeninformation in Form eines Tiefenbildes in einer bestimmten Auflösung, so wird ausschließlich dieser Topic aktiv mit Daten versorgt. Das erzeugt weniger Last für das verarbeitende System, da nur Berechnungen für die angeforderten Daten ausgeführt werden und spart Bandbreite im Netzwerk. Besonders ist zu erwähnen, dass in diesem Paket die Beschleunigung durch Grafikkarten entweder mit OpenCl, OpenGL oder CUDA unterstützt wird. Damit kann, falls ein Grafikchip verbaut ist, was bei dem Roboter in dieser Arbeit der Fall ist, die Bildverarbeitung beschleunigt und der Prozessor entlastet werden. Damit man in ROS die für gmapping benötigten 2D Tiefeninformationen, die einem typischen LIDAR-Scan entsprechen, erhält, wird das Paket **pointcloud to laserscan** [Bov] eingesetzt. Hierbei werden die 3D Tiefeninformationen von iai_kinect2 empfangen und sowohl oberhalb als auch unterhalb einer definierten Höhe, am besten der der Kamera, abgeschnitten. Damit wird aus einer dreidimensionalen eine zweidimensionale Punktwolke, mit der alle Anwendungen versorgt werden können die einen Laserscan oder etwas Ähnliches benötigen.

Das letzte Paket ist ein lokaler Planer. Wie in den Grundlagen kurz genannt, kommt hier der **Time-Elastic-Band-Planer** (TEB) von Rösmann et al. [RFW⁺12, RFW⁺13] zum Einsatz. Darin wird das Problem der Inverskinematik gelöst, also das Herausfin-

den von Lenkeinstellung und zurückzulegender Strecke, um einen bestimmten Punkt in der Ebene zu erreichen. Dabei werden Hindernisse vermieden, die der globale Planer noch nicht eingeplant hat, ebenso wird Rücksicht auf die kinematischen Eigenschaften des mobilen Roboters genommen. Das Finden einer sicheren Trajektorie wird beim TEB-Planer als Optimierungsproblem aufgefasst und mit Hilfe verschiedener Optimierungsverfahren gelöst. Es werden immer verschiedene mögliche Fahrstrategien verfolgt und dynamisch die Beste für eine bestimmte Situation ausgewählt. Damit wird vermieden, dass für aufeinander folgende Zielpunkte ungünstige Ausgangspositionen geschaffen werden, da lokale Planer den globalen Plan nicht vollständig kennen. Der Begriff Time-Elastic-Band bezieht sich hierbei auf die interne Repräsentation der Wegpunkte eines Pfades. Diese werden nicht nur durch ihre verschiedenen Konfigurationen in Relation gesetzt, sondern auch durch die benötigte Zeit verschiedene Konfigurationen zu erreichen. Das elastische ist dabei der Grad der Optimierung, es kann also das Optimum zweier benachbarter Konfigurationen zu Gunsten einer oder mehrerer Anderer vermindert werden, um die Erreichbarkeit des Ziels zu gewährleisten. Damit das Paket arbeiten kann, benötigt es einen globalen Plan zum gewünschten Ziel, sowie die Eigenschaften des Fahrzeugs. Mit Eigenschaften ist hier die maximale bzw. minimale Geschwindigkeit, minimaler Wendekreis und Radstand gemeint. Für das dynamische Umfahren von Hindernissen werden zusätzlich noch Entfernungssensoren benötigt.

3.2 Simulation

Würde man den Lösungsansatz direkt auf dem Modellauto entwickeln, so könnte man sich zwar besser auf die physikalischen Eigenschaften und Limitationen einstellen, aber der zeitliche Aufwand beim Testen wäre zu groß. Schließlich muss neben dem Komplizieren und Starten der Anwendung auch das Auto zum Einsatzort gebracht und dessen Verhalten beobachtet werden. Erfahrungsgemäß können so kleinste Fehler in den Überlegungen einige Stunden Fehlersuche am Fahrzeug mit sich bringen, vor allem weil manche Fehler erst nach einigen Minuten Laufzeit erkennbar werden. Um den Lösungsansatz effektiv und effizient testen zu können, bietet es sich an eine Simulation zu schreiben, die das kinematische Verhalten eines Fahrzeuges mit Ackermann-Lenkung wiedergibt und die entsprechenden Entfernungssensoren zur Kartographierung darstellt. Hierbei werden dynamische Effekte wie Beschleunigung, Reibung, Momente und Kräfte, die in der Wirklichkeit auftreten, außer Acht gelassen. Zum einen sind diese Effekte zum Testen des Algorithmus unwichtig, zum anderen erfordert eine physikalisch korrekte Simulation deutlich mehr Ressourcen, die möglicherweise an anderer Stelle benötigt werden. Zusätzlich kann man das Auto in der Simulation schneller fahren lassen, als es in der Realität sicher wäre, da man keine Beschädigung durch Kollision mit Hindernissen befürchten muss. So kann man Zeit beim Testen sparen und unerwartetes Verhalten schneller entdecken. Es finden außer den Paketen, die die Funktionalität der Kinect und des Roboters betreffen, alle anderen in Unterabschnitt 3.1.2 genannten ROS-Pakete Anwendung.

3.2.1 Modellierung des Fahrzeugs

Der Fokus bei der Modellierung liegt auf der Berechnung der Position und Orientierung im Raum, nachdem bestimmte Steuereingaben eingegangen sind. Es handelt sich folglich um die Odometrie.

In der klassischen Auslegung berechnet die Odometrie nur mit Hilfe der Steuergrößen, Lenkeinschlag und zurückgelegter Strecke, die Positions – und Orientierungsänderung des Fahrzeugs im Raum. Hierbei können in der Realität Unsicherheiten auftreten, die durch endliche Einstellgeschwindigkeit des Lenkwinkels und endlicher Genauigkeit der Geschwindigkeiten - bzw. Streckenmessung entstehen. Das heißt, die Odometrie wird mit wachsender Strecke immer ungenauer. Da aber die Position in einer Simulation absolut präzise nach jedem Zeitschritt ermittelt werden muss, um etwa die simulierten Entfernungssensoren oder den SLAM-Algorithmus mit Positionsdaten zu beliefern, sind in diesem Fall alle Lenkwinkel und Geschwindigkeiten exakt so, wie vorgegeben. Das heißt, es wird kein Sensorrauschen simuliert, die Odometrie ist damit präzise und weist nicht wie sonst Unsicherheiten auf.

Da das Fahrzeug in der Simulation etwa die Fahreigenschaften des Modellautos aufweisen soll, wird für die Positionsberechnung die Kinematik der Ackermann-Lenkung zu Grunde gelegt. Hiermit kann wie in Unterkapitel 2.2 die Position nach jedem Zeitschritt berechnet werden. Die benötigten Steuereingaben können im Rahmen dieser Arbeit entweder Lenkwinkel in Radianen und Geschwindigkeit in $\frac{m}{s}$, oder Lenkstellgröße $a \in [-50, 50] \subseteq \mathbb{Z}$ und Geschwindigkeitsstellgröße $s \in [-10, 10] \subseteq \mathbb{Z}$ sein. Um die Fahreigenschaften des Modellautos so gut wie möglich zu imitieren, wurde eine Tabelle angelegt, in der real gemessene Lenkwinkel bzw. Geschwindigkeiten den Steuergrößen zugeordnet sind. So kann das Vorwärtskinematikmodell mit annähernd den gleichen Lenkwinkeln versorgt werden, wie sie im realen Betrieb auftreten können.

Im Hintergrund wird die Distanz des gesamten zurückgelegten Weges gespeichert, um später die Effizienz eines Algorithmus zu bestimmen.

3.2.2 Simulation von Entfernungssensoren

Da die Entfernungssensoren für einen Algorithmus, der sich mit der Kartografierung des Raumes beschäftigt, unerlässlich sind, müssen auch diese in die Simulation mit einbezogen werden. Der einfachste Weg dies zu bewerkstelligen, wäre über eine ROS-Bag, also eine Datei, in der alle Sensormesswerte während einer Erkundungsfahrt hinterlegt sind. Diese kann man wie einen Film abspielen und die Simulation mit einem Laserscan versorgen. Allerdings muss sich das Modell dann auf genau derselben Trajektorie bewegen wie das physikalische Gegenstück, sonst stimmen die Tiefeninformationen nicht mehr mit den Positionsdaten der Simulation überein und die Visualisierung ergibt keinen Sinn mehr. Das schränkt die Modularität der Simulation enorm ein, denn man kann nur auf bereits gefahrenen Wegen simulieren, was natürlich die Simulation eines Algorithmus zur autonomen Erkundung unmöglich macht. Aus diesem Grund wird der Laserscan in dieser Arbeit auf Grundlage einer Karte, die als Bilddatei vorliegt, erstellt.

Die Idee ist, dass man ausgehend von der Position und Orientierung des Modells auf der Karte anfängt, diese innerhalb des Sichtfeldes des virtuellen Sensors abzutasten, bis

man auf ein Hindernis (schwarzer Pixel) stößt oder die maximale Reichweite des Scans überschritten ist. Ein solcher Sehstrahl enthält entweder die Information, dass sich ein Hindernis im Sichtfeld befindet und wie weit es entfernt ist, oder, dass sich kein Hindernis im Sichtfeld befindet und der Raum frei ist. Im Algorithmus 1 ist umrissen, wie sich der Ablauf des Abtastalgorithmus gestaltet.

Die Karte, also das occupancy grid und dessen Auflösung (mapResolution), gilt als gegeben. Weiterhin ist das zu simulierende Sichtfeld (fieldOfView), die maximale Reichweite (sensorRange) und die Auflösung des Sichtfelds (angularResolution) des Sensors gegeben. Entscheidend bei der Funktionalität ist die Verwendung des in Unterkapitel 2.5

Algorithmus 1 Entfernungssensor

```

1: function GETRANGEINFO(robotPosition, yaw)
2:   numOfAngles  $\leftarrow \frac{\text{fieldOfView}}{\text{angularResolution}}$ 
3:   rangeArray  $\leftarrow \emptyset$ 
4:   currentAngle  $\leftarrow \text{yaw} - \frac{\text{fieldOfView}}{2}$ 
5:   rayLength  $\leftarrow \frac{\text{sensorRange}}{\text{mapResolution}}$ 
6:   i  $\leftarrow 0$ 
7:   for i < numOfAngles do
8:     endPoint.x  $\leftarrow \text{robotPosition}.x + \text{rayLength} \cdot \cos(-\text{angle})$ 
9:     endPoint.y  $\leftarrow \text{robotPosition}.x + \text{rayLength} \cdot \sin(-\text{angle})$ 
10:    LineIterator  $\leftarrow \text{INITITERATOR}(\text{occupancyGrid}, \text{robotPosition}, \text{endPoint})$ 
11:    currentPosition  $\leftarrow \text{LineIterator}.pos$ 
12:    for LineIterator.pos  $\neq \text{LineIterator}.maxPos + 1$  do
13:      currentPosition  $\leftarrow \text{LineIterator}.pos$ 
14:      if occupancyGrid.at(currentPosition) == obstacleColor then
15:        break
16:      LineIterator.increment
17:      scannedRange  $\leftarrow \|\text{robotPosition} - \text{currentPosition}\| \cdot \text{mapResolution}$ 
18:      rangeArray.push(scannedRange)
19:      i  $\leftarrow i + 1$ 
20:   return rangeArray

```

genannten LineIterators. Dieser interpoliert sehr effizient Punkte auf einer Linie, die in Algorithmus 1 durch den Anfangspunkt robotPosition und einen Endpunkt endPoint definiert wird. Der Punkt robotPosition ist durch die Position des Roboters gegeben, Punkt endPoint wird durch die maximal mögliche Sichtweite bestimmt. Mit den durch LineIterator interpolierten Punkten wird an den Koordinaten der Punkte überprüft, ob die Karte an dieser Stelle schwarze Pixel (obstacleColor) besitzt, also Hindernisse im Weg des Sehstrahls sind. Damit wird anschließend das wahre Ende des Sehstrahls bestimmt, daraus die maximal freie Distanz berechnet und als Entfernung in einem Array festgehalten. Bei einem Laserscan ergeben sich so $n = \frac{\text{Sichtfeld}}{\text{Winkelauflösung}}$ Linien, die abgetastet werden müssen und damit entsprechend viele Entfernungswerte.

Ultraschallsensoren stellen hier eine Besonderheit dar, da diese pro Sensor nur genau einen Entfernungswert liefern. Um dem Umstand Rechnung zu tragen, werden für die-

sen Sensortyp ebenfalls n Linien im Datenblatt definierten Öffnungswinkel abgetastet und der kleinste Entfernungswert ausgegeben.

3.2.3 Simulation in ROS

Die Simulation des kinematischen Modells, der Kinect als Laserscanner und der Ultraschallsensoren verteilen sich auf drei ROS-Nodes und sind im ROS-Paket simulation [Ehm16b] zusammengefasst.

Im Node **simulation_control** wird das in Unterabschnitt 3.2.1 beschriebene Modell eingebettet. Dabei wird das kinematische Modell alle 10ms mit der aktuell eingestellten Geschwindigkeit und dem Lenkwinkel versorgt. Daraus kann im nächsten Zeitschritt die zurückgelegte Strecke und mit Hilfe der Vorwärtsskinematik die neue Position im Raum sowie die neue Orientierung berechnet werden. Diese Informationen werden dann unter dem Topic `odom` für alle anderen Nodes veröffentlicht.

Der Node **simulation_laserscan** implementiert die in Unterabschnitt 3.2.2 beschriebene Technik zur Simulation eines Laserscans. Hierbei wird beim Starten des Nodes zuerst die abzutastende Karte mit ihren Metainformationen, wie Auflösung und Ursprung, geladen. Danach wird alle 33ms die Position des Roboters von ROS abgefragt und eine Abtastung mit Hilfe der Karte und des LineIterators durchgeführt. Die resultierenden Entfernungsdaten werden als LaserScan-Nachricht auf dem Topic `scan` veröffentlicht.

Der Node **simulaton_usscan** funktioniert wie `simulation_laserscan`, nur werden in diesem anstelle eines Laserscans alle drei Ultraschallsensoren simuliert. Zuerst wird ebenfalls eine Kopie der Karte geladen und dann alle 33ms an der Position des Roboters eine Abtastung durchgeführt. Bei der Abtastung wird wie in Unterabschnitt 3.2.2 beschrieben, nur der kleinste Entfernungswert aus dem Array mit den Scandaten entnommen und zurückgegeben. Die resultierenden Werte werden dann als Range-Nachricht auf den Topics `front-/left-/right_us_scan` veröffentlicht.

Zusätzlich zu diesen drei Nodes, die die Simulation umsetzen, gibt es noch einen Node **simulation_control**, der eine GUI implementiert. Mit Hilfe der GUI wird der Node `simulation_control` gesteuert. Hierin werden die Geschwindigkeit, der Lenkwinkel und später die Erkundungsstrategie eingestellt. Zusätzlich werden mit diesem Node Eingabesignale der Tastatur eingefangen, falls der Benutzer den virtuellen Roboter direkt steuern möchte.

Mit einer fertigen Simulationsumgebung kann die im Folgenden dargelegte Methode zur autonomen Erkundung getestet werden. Dafür wird im nächsten Kapitel zuerst eine Methode entwickelt um Frontiers zu erkennen.

3.3 Frontier-Erkennung

Bei einer Erkundungsstrategie, die dem frontier-based Approach folgt, sind die Frontiers, also Grenzen zwischen bekanntem und unbekanntem Gebiet, auf einem

occupancy-grid von zentraler Bedeutung. Es muss daher gewährleistet sein, dass diese Grenzen auch bei verrauschten Karten noch erkannt werden. Gerade bei occupancy grids kann das vorkommen, da Punkte, je nach SLAM-Verfahren, nicht immer Linien zugeordnet und als diese abgebildet werden. Deshalb werden in dieser Arbeit die in den Grundlagen beschriebenen Computer Vision Verfahren verwendet, da diese üblicherweise dazu ausgelegt sind, mit verrauschem Bildmaterial zu arbeiten. Außerdem sind diese gut getestet und auf Performanz ausgelegt.

Der Prozess, um solche Frontiers zu erkennen und der in dieser Arbeit komplett neu entwickelt wurde, gliedert sich in drei Schritte die sukzessive aufeinander aufbauen und in Abbildung 3.4 dargestellt werden.

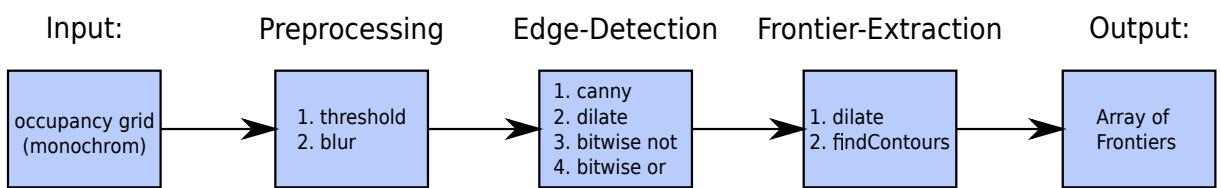


Abb. 3.4: Das Ablaufdiagramm zeigt die einzelnen Stationen der Detektion mit den benutzten Funktionen.

Damit der Algorithmus anfangen kann, darf das verwendete occupancy grid nur drei diskrete Zustände aufweisen. Ein Eintrag im Raster muss entweder frei, besetzt oder unbekannt sein, im Folgenden als weiß, schwarz und grau bezeichnet. Zusätzlich muss das occupancy grid im Mat Datentyp als monochromes Bild vorliegen, auf dem die von OpenCV bereitgestellten Algorithmen arbeiten können.

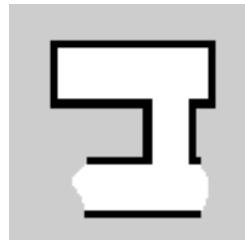


Abb. 3.5: Frontier-Detektion: occupancy grid

Im ersten Schritt werden auf dem in Abbildung 3.5 gezeigten Bild zwei Schwellwertoperationen mit Hilfe des Threshold-Operators aus Unterkapitel 2.5 durchgeführt. Hierbei liefert die erste Operation das Bild in Abbildung 3.6 b), bei dem alle Grauwerte, die größer als 0 (Schwarz) sind, auf 255 (Weiß) gesetzt werden, damit bleiben nur die Hindernisse im Bild übrig. In der zweiten Schwellwertoperation werden alle Grauwerte, die kleiner als 255 (Weiß) sind, auf 0 (Schwarz) gesetzt, damit entsteht das Bild in Abbildung 3.6 a), bei dem unbekannte Bereiche mit Hindernissen verschmelzen. Zuletzt wird eine blur-Operation auf das Bild in Abbildung 3.6 a) angewendet, was Rauschen aus den weißen, also bekannten Bereichen, entfernt. Das kann entstehen, wenn die Auflösung des Entfernungssensors nicht hoch genug ist und dadurch die Messstrahlen nicht nah genug beieinander liegen. So entsteht beim Mappingprozess keine homogene

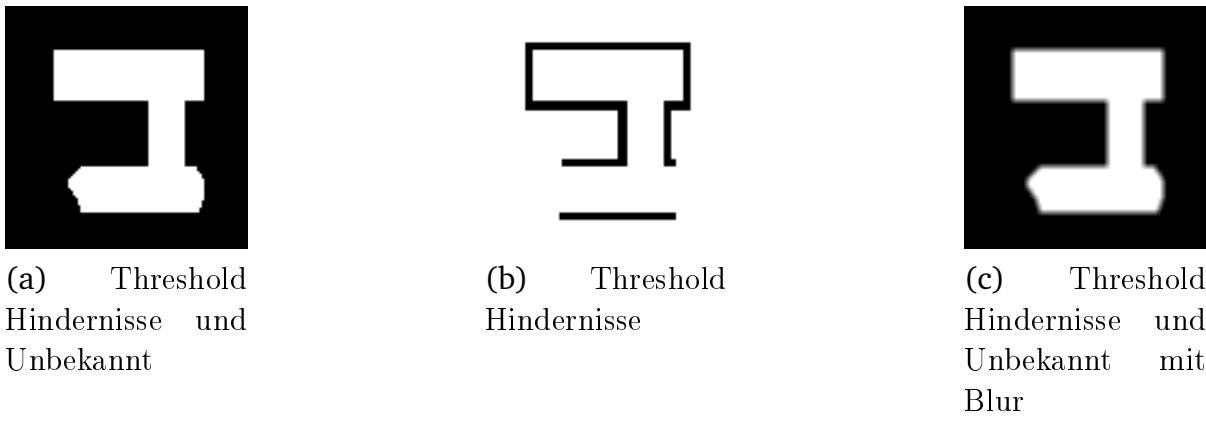


Abb. 3.6: *Frontier-Detektion: Schritt 1*

weiße Fläche, sondern es verbleiben vereinzelt graue Flecken. Abbildung 3.6 c) zeigt das Resultat der blur-Operation.

In Schritt zwei werden nun auf Abbildung 3.6 b) und Abbildung 3.6 c) mit der Canny-Funktion ein Algorithmus zur Kantenerkennung angewendet. Damit werden die vorher noch unzusammenhängenden Punktmengen, die die Grenzen ausgemacht haben, zu durchgängigen Linien. Das erzeugt zwei neue Bilder. In Abbildung 3.7 b) sind ausschließlich Hindernisse enthalten und bei Abbildung 3.7 a) Hindernisse und Grenzen. Eine Anwendung des Dilate-Operators auf Abbildung 3.7 b), dehnt Hindernisse aus und verbindet nicht zugeordnete, noch freie Punkte. Anschließend wird dieses Bild invertiert und als Maske (Abbildung 3.7 c.) benutzt, um elementweise eine or-Operation mit Abbildung 3.7 a) und einem völlig schwarzen Bild durchzuführen. Das Resultat ist das Bild in Abbildung 3.8 a), das ausschließlich Grenzen, also die gesuchten Frontiers, enthält.

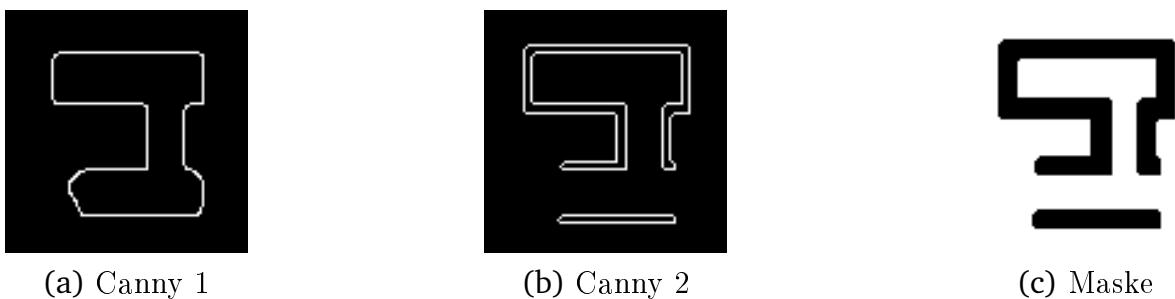


Abb. 3.7: *Frontier-Detektion: Schritt 2*

Die letzte Phase beginnt indem der Dilate-Operator auf Abbildung 3.8 a) angewendet wird, um Frontiers, die nur durch wenige Pixel, also Rauschen, voneinander getrennt sind, zu verbinden. Als Letztes werden die Grenzen mit der `findContours`-Funktion extrahiert und als ein Array von Punkten, die die Stützstellen einer Frontier bilden, in einem Ausgabearray abgelegt. Die Punktmenge, die aus der letzten Operation resultiert, ist in Abbildung 3.8 b) dargestellt und rot markiert.

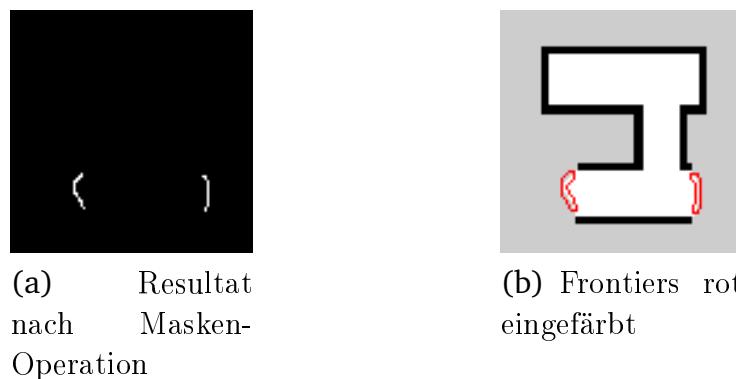


Abb. 3.8: *Frontier-Detektion: Schritt 3 - Resultat*

Die detektierten und als Array vorliegenden Frontiers können dann im Exploration-Planer verwendet werden, um damit die verschiedenen Erkundungsstrategien zu versorgen.

3.4 Umsetzung des Path-Transform-Algorithmus

Eine weitere wichtige Grundlage für eine autonome Erkundung, ist einerseits die Fähigkeit, kollisionsfreie Wege zu Frontiers zu planen, um dorthin zu fahren und andererseits mit Hilfe dieser Wege zu bewerten, welche Grenze zuerst angefahren werden soll.

Zum Funktionieren benötigt die hier implementierte Variante des Path-Transform Algorithmus die selben Voraussetzungen wie die Frontier-Erkennung. Das heißt, die zu untersuchende Karte muss ein occupancy grid sein, das als Mat Datentyp in monochromer Farbe vorliegt. Ebenso darf die Karte pro Zelle nur genau einen der drei Zustände

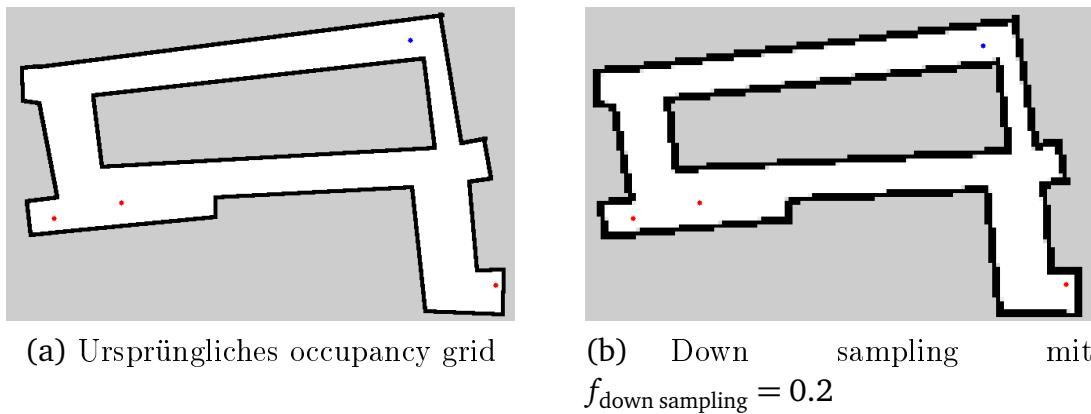


Abb. 3.9: *Path-Transform: occupancy grid*

enthalten: frei, unbekannt, besetzt. Zusätzlich dazu muss die Auflösung der Karte bekannt sein, also wie viele Meter in Länge und Breite ein Pixel auf der Karte in der Realität darstellt. Darüber hinaus muss, um kollisionsfrei planen zu können, die Länge und Breite des Roboters und die minimale erlaubte Distanz zu Hindernissen gegeben

sein. Im Folgenden werden gemäß dem Unterkapitel 2.7 die Implementierung der darin genannten Teilschritte dargelegt. Die gezeigte Karte in Abbildung 3.9 a) dient als Grundlage um zu zeigen, wie der Planer in jedem Schritt arbeitet. Der Startpunkt ist blau und die Zielpunkte sind rot markiert.

Der Algorithmus startet mit der Vorgabe eines Startpunktes, der sich auf der Karte befinden muss. Zu Beginn wird geprüft, ob die Karte nicht leer ist und ob sich die Koordinaten des Roboters auf dem occupancy grid befinden. Ist beides gewährleistet, so wird die Grundfläche des Roboters auf der Karte als frei deklariert, logischerweise kann sich dort kein Hindernis befinden. Da die folgenden Prozesse sehr rechaufwändig sind, wird die gegebene Karte um einen frei wählbaren Faktor $f_{\text{down sampling}}$ verkleinert und der Startpunkt entsprechend skaliert. Da hierbei Unschärfen entstehen können, werden mit einer Schwellwertoperation alle Grauwerte, die dunkler als der Grauwert für unbekannt sind, auf schwarz gesetzt. Damit werden die Hindernisse wieder geschärft. Nach der Anwendung der Operationen auf Abbildung 3.9 a), zeigt Abbildung 3.9 b) das Resultat.

Algorithmus 2 Distance-Transformation

```

1: distMat ← INF(occupancyGrid.rows, occupancyGrid.cols)
2: distMat.at(robotPose) ← 0
3: tempMat ← ZEROS(occupancyGrid.rows, occupancyGrid.cols)
4: error ← L2NORM(distMat, tempMat)
5: while error > 0 do
6:   tempMat ← distMat
7:   i ← 0
8:   for i < occupancyGrid.rows do
9:     j ← 0
10:    for j < occupancyGrid.cols do
11:      if occupancyGrid.at(j, i) > unknownColor then
12:        distMat.at(j, i) ← APPLYFWDS SCANWINDOW(distMat, j, i)
13:    i ← occupancyGrid.rows
14:    for i ≥ 0 do
15:      j ← occupancyGrid.cols
16:      for j ≥ 0 do
17:        if occupancyGrid.at(j, i) > unknownColor then
18:          distMat.at(j, i) ← APPLBYWD SCANWINDOW(distMat, j, i)
19: error ← L2NORM(distMat, tempMat)

```

Ist die Vorverarbeitung abgeschlossen, wird zuerst die Distanztransformation, wie im Algorithmus 2 beschrieben, durchgeführt. Die Funktionsaufrufe im Algorithmus 2 in den Zeilen 12 und 18 benutzen die in Gleichung 14 aufgeführten Schemata, welche sich aus der euklidischen Distanz von der Mitte c zu den jeweiligen Außenpunkten ergeben.

$$\text{ForwardWindow : } \begin{pmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & c & - \\ - & - & - \end{pmatrix}, \quad \text{BackwardWindow : } \begin{pmatrix} - & - & - \\ - & c & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{pmatrix} \quad (14)$$

Dabei wird an der jeweils betrachteten Position $p_{j,i}$ das entsprechende Fenster auf die distMat-Matrix gelegt und der Wert des Fensters auf den Wert der Matrix addiert. Ist einer der Werte kleiner als der Wert c an der Stelle $p_{j,i}$, so wird dieser Wert zurückgegeben, anderenfalls wird c zurückgegeben.

Der Algorithmus terminiert, wenn kein Eintrag in der distMat-Matrix mehr minimiert werden kann, und somit alle kürzesten Wege zum Startpunkt gefunden worden sind. Auf der Abbildung 3.10 a) kann man erkennen, wie ein Grauwertgradient entstanden

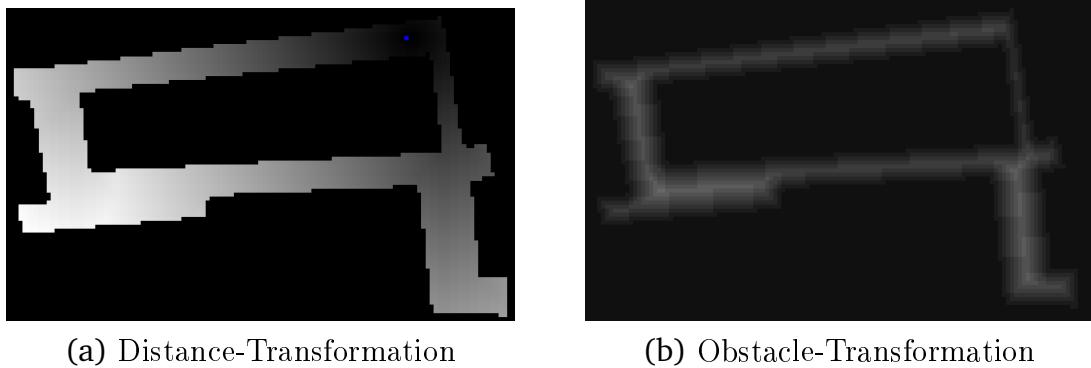


Abb. 3.10: *Path-Transform: Distance-Transformation und Obstacle-Transformation*

ist, der von jedem Punkt im Freiraum aus in Richtung des kürzesten Weges zum Ziel zeigt.

Als nächstes wird eine obstacle-Transformation durchgeführt. Diese sorgt dafür, dass nicht die Distanzen von einem Punkt zu allen berechnet werden, sondern von allen Punkten im Freiraum die Distanz zum nächsten Hindernis. Das lässt sich mit der distance-Transform von oben durchführen, es benötigt dafür nur eine Modifikation im Algorithmus 2 in Zeile 2. Hier wird nicht der Startpunkt auf Distanz 0 gesetzt, sondern alle Punkte, die ein Hindernis auf dem occupancy grid darstellen, erhalten den Wert 0. Danach läuft der Algorithmus genauso ab und terminiert mit der gleichen Abbruchbedingung. Die Abbildung 3.10 b) zeigt, wie der entstandene Grauwertgradient an jedem Punkt im Freiraum zum nächsten Hindernis zeigt.

Algorithmus 3 Path-Transformation

```

1:  $i \leftarrow 0$ 
2: for  $i < \text{occupancyGrid.rows}$  do
3:    $j \leftarrow 0$ 
4:   for  $j < \text{occupancyGrid.cols}$  do
5:     if  $\text{occupancyGrid.at}(j, i) > \text{unknownColor}$  then
6:        $c \leftarrow \text{obsMat.at}(j, i)$ 
7:       if  $c \leq d_{\min, \text{obstacle dist}}$  then
8:          $\text{cost} \leftarrow \alpha_{\text{obstacle awareness}} \cdot (d_{\min, \text{obstacle dist}} - \text{obsMat.at}(j, i))^3$ 
9:       else
10:         $\text{cost} \leftarrow 0$ 
11:         $\text{pathMat.at}(j, i) \leftarrow \text{distMat.at}(j, i) + \text{cost}$ 

```

In diesem Schritt werden die Distance- und die Obstacle-Transformation in der Path-Transformation vereint. Dabei gibt die Konstante $d_{\min, \text{obstacle dist}}$ an, wie weit ein Pfad mindestens von einem Hindernis entfernt sein muss und $\alpha_{\text{obstacle awareness}}$ legt fest, wie groß die Toleranz des Planers bei der Einhaltung der ersten Konstante ist. Wie man in Algorithmus 3 erkennen kann, hält sich diese Variante des Algorithmus nicht genau an die von Zelinsky, da hier nicht für jede Zelle im Freiraum der Weg zum Startpunkt hinsichtlich Distanz und Kosten optimiert wird. Es wird lediglich die vorgeschlagene Kostenfunktion benutzt, um jedem Element in der pathMat-Matrix einem durch Hinderniskosten modifizierten Distanzwert zuzuordnen. Damit können zwar lokale Mini-

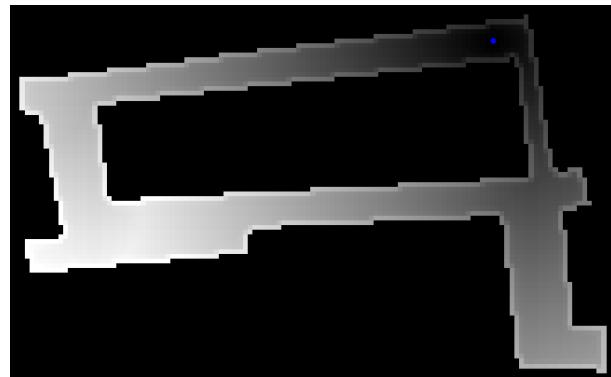
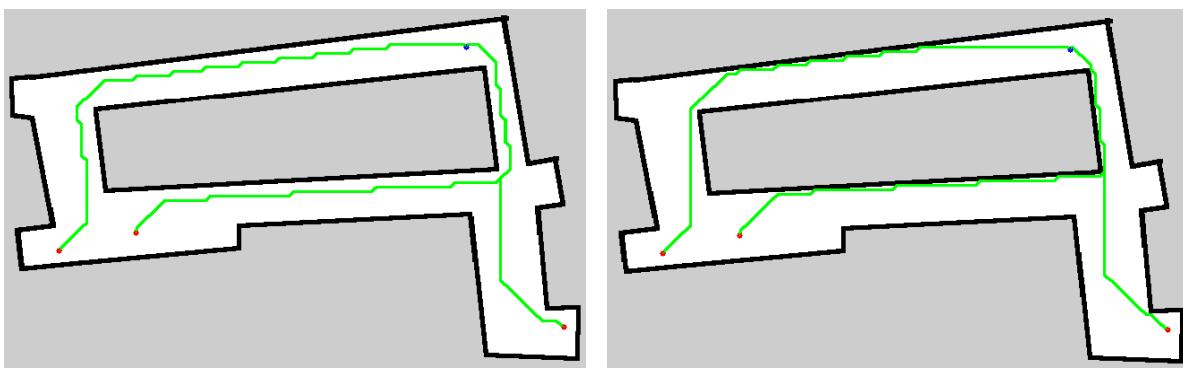


Abb. 3.11: *Path-Transform: Path-Transformation mit $\alpha = 0.7$ und $d = 0.7m$*

ma entstehen und die gleichen Probleme auftauchen, wie bei potentialbasierten Wegplanern, allerdings erfordert diese Variante deutlich weniger Rechenaufwand als die oben genannte Optimierung. Indem man die $\alpha_{\text{obstacle awareness}}$ -Konstante, die gewichtet, wie sehr Hindernisse in die Wegplanung eingehen, nicht zu groß wählt, liefert der hier präsentierte Algorithmus brauchbare Ergebnisse. Die Abbildung 3.11 zeigt den Grauwertgradient, der in jedem Punkt des Freiraums in Richtung des kürzesten Pfades zum Ziel zeigt, unter Berücksichtigung der Hindernisse.



(a) Pfade mit Mindestabstand zu Hindernissen

(b) Pfade ohne Mindestabstand zu Hindernissen

Abb. 3.12: *Path-Transform: Pfade*

In der letzten Phase wird das angestrebte Ziel als Beginn des Algorithmus gesetzt. Von

dort aus folgt man in jedem Schritt dem steilsten Abfall, also dem Gradienten, der in Richtung des kürzesten Pfades zeigt, und legt den Wegpunkt in einer Liste ab. Findet sich kein Punkt mehr entlang des steilsten Abfalls und ist der Punkt, auf dem man sich aktuell befindet, nicht der Startpunkt aus den vorherigen Schritten, so ist das Ziel nicht erreichbar. Ist der aktuelle Punkt der Startpunkt, so wird die Suche nach dem steilsten Abfall beendet, die Liste invertiert und ausgegeben. Damit erhält man sehr schnell einen Pfad vom gegebenen Start zu einem beliebigen erreichbaren Ziel. In der Abbildung 3.12 b) kann man geplante Wege ohne Berücksichtigung der Hindernisse sehen, in der Abbildung 3.12 a) mit Berücksichtigung von Hindernissen und einem Mindestabstand zu diesen. Die Pfade im Bild a) sind hierbei sehr viel einfacher von einem Fahrzeug zu befahren, da nicht zu nah an Hindernissen vorbei gefahren werden muss.

Mit dem, in Unterkapitel 3.3, gezeigten Verfahren können zuverlässig Frontiers auf einem occupancy grid erkannt werden. Zudem steht ein Wegplaner zur Verfügung, um zu überprüfen, ob diese erreicht werden können und wenn ja, wie weit sie entfernt sind. Damit kann im nächsten Abschnitt ein Exploration-Planer entwickelt werden.

3.5 Implementierung eines Exploration-Planers

Der hier vorgestellte Exploration-Planer nimmt die Informationen der Methoden aus den beiden vorangegangenen Abschnitten, wertet sie mit Hilfe einer der beiden Strategien aus und schlägt den daraus besten nächsten Punkt auf der Karte vor, der erkundet werden soll. Dabei wird nicht nur der Zielpunkt vorgegeben, sondern auch ein kollisionsfreier Weg dorthin geplant.

Der Ablauf des Planers stellt sich am besten mit Hilfe des in der Abbildung 3.13 gezeigten Ablaufdiagramms dar. Es beginnt mit der Übergabe der aktuellen Position des

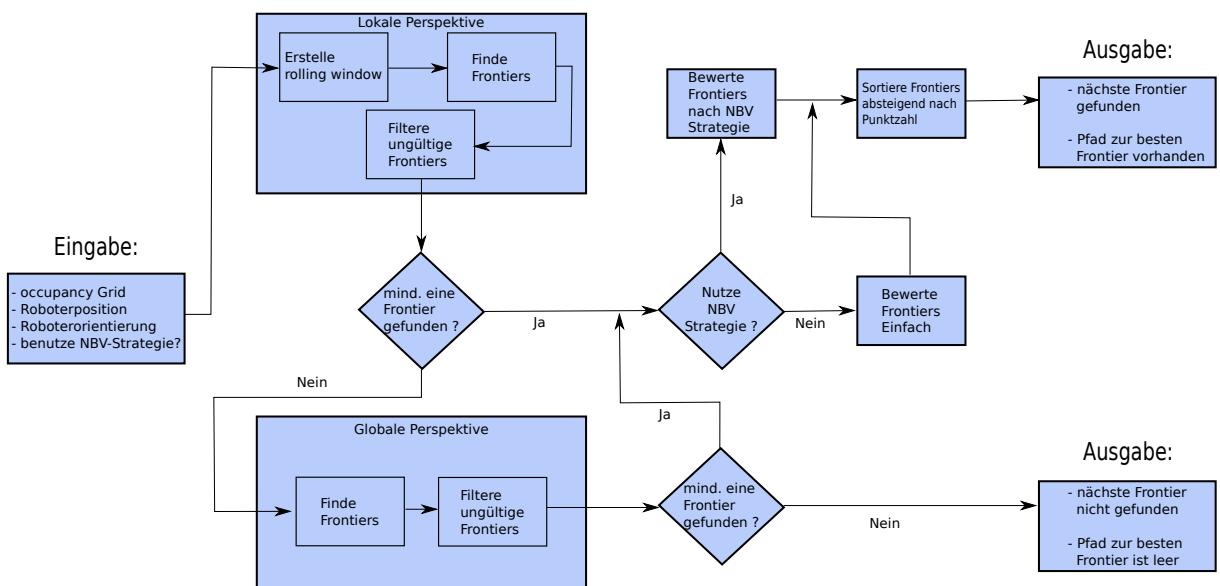


Abb. 3.13: *Exploration-Planer: Ablaufdiagramm*

Roboters auf dem occupancy grid und dem occupancy grid selbst, das in Form eines Mat-Datentyps in monochromer Farbe vorliegen muss. Bei der Karte gelten die gleichen Voraussetzungen wie bei Unterkapitel 3.3 und Unterkapitel 3.4, es dürfen pro Raster nur einer der drei Zustände frei, besetzt und unbekannt gelten.

Da sich die meisten freien Grenzen erfahrungsgemäß in der unmittelbaren Umgebung des mobilen Roboters befinden, wird zunächst in einem Teilausschnitt der Karte nach Frontiers gesucht. Das geschieht mit Hilfe eines sogenannten **rolling windows**, ein rechteckiger Ausschnitt der Karte, der sich mit dem Roboter als Zentrum bewegt. Dieses Fenster sollte mindestens doppelt so lang und doppelt so breit sein wie die Sichtweite des Roboters, sonst würden neu entdeckte Grenzen nie im lokalen Umfeld auftauchen. Die Größe des rolling windows wird im Algorithmus durch die Konstante $d_{\text{rolling window}}$ festgelegt und gibt die Länge einer Seite des quadratischen Fensters an. Werden lokal keine Frontiers entdeckt, so wird die ganze Karte im globalen Kontext, nach diesen durchsucht. Finden sich dabei auch keine Grenzen mehr, so wird ein Flag ausgegeben, dass kein nächster Pfad mehr zur Verfügung steht und das abgeschlossene Gebiet als hinreichend erkundet gilt. Werden im lokalen oder globalen Kontext Grenzen entdeckt, so geschieht in beiden Verfahren der gleiche Ablauf.

Nach dem erfolgreichen Suchen nach Frontiers mit dem in Unterkapitel 3.3 beschriebenen Verfahren wird überprüft, ob diese gültig sind. Gültige Grenzen müssen mindestens so groß sein, dass der mobile Roboter sie in seiner Gänze durchfahren könnte. Genauer bedeutet das, dass der Durchmesser des minimal einschließenden Kreises einer Frontier größer sein muss, als die Diagonale der Robotergrundfläche, wobei der minimal einschließende Kreis der Kreis ist, in den alle Punkte, die auf einer Frontier liegen, gerade noch passen.

Anschließend wird der Zentroid jeder gültigen Grenze und ihre **Normale** bestimmt. Die Normale einer Frontier ist in dieser Arbeit die Richtung, von der aus der Grauwertgradient am steilsten abfällt. Den Grauwertgradienten erhält man mit dem in Unterkapitel 2.5 beschriebenen Sobel-Operator. Mit dieser Normalen erhält man sozusagen die Blickrichtung ins Unbekannte.

Im nächsten Schritt wird, mit Hilfe des in Unterkapitel 3.4 beschriebenen Path-Transform-Planers, ein Pfad zu jeder gültigen Grenze, zum zugehörigen Zentroiden geplant. Die Frontiers, die nicht erreichbar sind, werden verworfen und die geplanten Pfade den übrigen Frontiers für die spätere Verwendung zugeordnet.

Im vorletzten Schritt werden die Grenzen mit einer der beiden unten genannten Strategien bewertet. Das heißt, den Frontiers wird eine Punktzahl zugeordnet. Dabei gilt: je höher desto besser. Über ein Flag kann von außen gesteuert werden, welche Strategie zum tragen kommt, wobei es möglich ist dieses nach jedem Durchlauf zu ändern.

Schließlich wird die Liste von Frontiers in absteigender Reihenfolge sortiert. Ist diese Liste nach allen Schritten nicht leer, so wird ein Flag ausgegeben, dass ein nächster Pfad zur Verfügung steht und das Gebiet noch weiter erkundet werden kann.

In den nachfolgenden Unterabschnitten werden die beiden Bewertungsverfahren näher erläutert, die maßgeblich die Erkundungsstrategie definieren.

3.5.1 Umsetzung einer einfachen Erkundungsstrategie

Bei der einfachen Erkundungsstrategie handelt es sich um eine modifizierte Version von Yamauchis Ansatz aus Unterkapitel 2.6, wobei immer die nähste erreichbare Frontier als beste bewertet wird. Das ist für holonome und nicht-holonome mobile Roboter, die sich auf der Stelle drehen können, natürlich ein einleuchtendes Kriterium. Betrachtet man jedoch Fahrzeuge mit Ackermann-Lenkung, so kann es passieren, dass eine Grenze direkt hinter dem Fahrzeug wegen ihrer Nähe besser bewertet wird, als eine direkt vor dem Fahrzeug liegende. Das Wendemanöver, um die vermeindlich besser bewertete Frontier zu erreichen, ist dann zeitlich und aus Sicht der zurückgelegten Wegstrecke oft viel aufwändiger, als eine weiter entfernte, aber viel leichter erreichbare Frontier anzufahren.

Es wird zunächst die Distanz d vom Roboter zum Zentroiden der Frontier anhand des vorher der Frontier zugordneten geplanten Pfades ermittelt und anschließend der Winkelunterschied $\delta\Theta$ zwischen der Normalen der Frontier und der Blickrichtung, dem Yaw-Winkel des Roboters, berechnet. Das Ergebnis der Distanzberechnung wird mit dem Faktor $\alpha_{\text{simple}, \text{distance weight}}$ und der Winkelunterschied mit $\beta_{\text{simple}, \text{angle weight}}$ gewichtet.

$$f(d) = e^{-d\alpha_{\text{simple}, \text{distance weight}}} , \quad g(\delta\Theta) = e^{\frac{2\pi - |\delta\Theta|}{\pi}\beta_{\text{simple}, \text{angle weight}}} \quad (15)$$

$$\text{score}(d, \delta\Theta) = f(d) \cdot g(\delta\Theta) \quad (16)$$

Bei Gleichung 16 geht das Ergebnis der Funktion $g(\delta\Theta)$ mit wachsendem Winkelunterschied gegen 1 und ausschließlich die Funktion der Distanz $f(d)$ wirkt. Im Umkehrschluss bedeutet das: Geht die Winkeländerung gegen 0, so wird der Faktor Winkelunterschied maßgeblich und die Funktion der Distanz $f(d)$ verliert außer in Extremfällen ihre Aussagekraft. Durch die Gewichtungsfaktoren kann bestimmt werden, wie sehr Frontiers, die in Blickrichtung liegen, gegenüber näheren Frontiers vorgezogen werden.

Diese Art der Bewertung zieht nicht in Betracht, ob es sich im Sinne des Informationsgewinns lohnt, zu einer Frontier zu fahren. Dafür ist der einfache Ansatz relativ ressourcenschonend, da keine komplexen Berechnungen stattfinden.

Der Rückgabewert dieses Algorithmus ist nur die Bewertung, da Zentroid und Blickwinkel der Grenze bereits vorher festgelegt wurden.

3.5.2 Umsetzung der Next-Best-View-Strategie

Um dem Umstand des potentiellen Informationsgewinns bei einer neu zu entdeckenden Kante Rechnung zu tragen, wird als Bewertungsalternative eine Version der Next-Best-View-Strategie vorgestellt. Da die ursprünglichen Entwickler dieser Methode González-Baños et al. [GBL02], wie Yamauchi [Yam97], einen Roboter verwendet haben, der in der Lage war, sich auf der Stelle zu drehen, wird diese Strategie ebenfalls modifiziert. Zu Beginn wird der Zentroid und die Richtung der Normale einer aktuell betrachteten Frontier benötigt, die in Algorithmus 4 und Algorithmus 5 als centroid und yaw bezeichnet werden. Die Konstante n_{points} bezeichnet die Anzahl der sampling points, accuracy die Anzahl der zu überprüfenden Blickwinkel und $\delta_{\text{resolution}}$ die Auflösung des

Algorithmus 4 NBV: Berechnung einer Frontier-Punktzahl Teil1

```
1: yaw ← frontier.yaw
2: centroid ← frontier.centroid
3: bestScore ← 0
4: bestYaw ← yaw
5: bestPoint = centroid
6: fit ← 0
7: fittingFrontierFound ← false
8: fittingAngle ← yaw
9: fittingPoint ← centroid
10: sensingPositions ← GETNRANDOMPOINTS(centroid, npoints, dreach, occupancyGrid)
11: i ← 0
12: numOfAngles ←  $\frac{\pi}{\text{accuracy}}$ 
```

simulierten field of view des Sensors, was die Genauigkeit bei der Ermittlung des Informationsgewinns beeinflusst. Field of view (FOV) bezeichnet das Sichtfeld des Entfernungsmessers und d_{reach} dessen Reichweite.

Die Zeilen 1 bis 5 in Algorithmus 4 sind dabei die Variablen für den Fall, dass eine Frontier vollständig ins field of view, also das Sichtfeld, passt, wohingegen die Variablen in Zeile 6 bis 9 für den Fall vorgesehen sind, dass die Frontier nicht vollständig durch das Sichtfeld des Roboters erfasst wird. Dieses Problem wird ab Zeile 14 in Algorithmus 5 adressiert, dort wird die Frontier als die Beste angenommen, die die größte Überdeckung mit dem Sichtfeld hat.

Die Funktion *getNRandomPoints* in Zeile 10 erzeugt, wie in den Grundlagen beschrieben, n Punkte im freien Bereich, die in Sensorreichweite um den Zentroiden der Frontier liegen.

Mit der Funktion *isInView* aus Zeile 6 in Algorithmus 5 wird anhand des im Moment betrachteten sampling points p und dem dazugehörigen Blickwinkel das Verhältnis von Punkten einer Frontier, die im field of view liegen, zu denen, die nicht darin liegen, bestimmt. Dabei wird mit dem momentanen Standpunkt und den Randbegrenzungen des Sichtfelds ein Dreieck aufgespannt und gezählt, wie viele Punkte einer Frontier darin liegen. Damit werden, bis auf manche Extremfälle, die meisten Grenzen gut erfasst. Eine genauere Überprüfung wäre, aus Sicht der Laufzeit, zu aufwändig, da alle Eckpunkte des Sichtfeldes durch Abtasten bestimmt werden müssten, was in der Berechnung des Informationsgewinns noch einmal geschieht.

calcScore aus Zeile 8 und 21 in Algorithmus 5 ist die wichtigste Funktion des Algorithmus, darin wird der Informationsgewinn ermittelt und mit Hilfe des Pfades zur Grenze bewertet. Die Menge an Informationen, die mit einer Messung an einem Punkt mit einer bestimmten Blickrichtung gewonnen werden können, bestimmen sich wie folgt:

Zuerst wird ein Teilausschnitt der Karte erzeugt, in dessen Zentrum die Frontier liegt. Dieser Kartenausschnitt ist quadratisch und hat als Kantenlänge die doppelte Sichtweite. Von diesem Ausschnitt wird anschließend eine Kopie angelegt.

Auf der Kopie des Ausschnittes wird jetzt das Abtastverfahren aus Unterabschnitt 3.2.2 angewendet, allerdings mit dem Unterschied, dass hier keine Entfernungsinformationen gewonnen werden sollen, sondern dass jeder Sehstrahl, solange er auf kein Hindernis trifft, einen Punkt in der Kartenkopie weiß einfärbt. Dies geschieht über das komplette

Algorithmus 5 NBV: Berechnung einer Frontier-Punktzahl Teil2

```

1: for  $i < \text{sensingPosition.size}$  do
2:    $p \leftarrow \text{sensingPositions.at}(i)$ 
3:   angle  $\leftarrow \text{yaw} - \frac{FOV}{2}$ 
4:    $j \leftarrow 0$ 
5:   for  $j < \text{numOfAngles}$  do
6:     qualityOffFitInFOV  $\leftarrow \text{ISINVIEW}(\text{frontier}, p, \text{angle}, d_{\text{reach}}, FOV)$ 
7:     if  $\text{qualityOffFitInFOV} \geq 1.0$  then
8:       score  $\leftarrow \text{CALCSCORE}(p, \text{angle}, FOV, d_{\text{reach}}, \delta_{\text{resolution}}, \text{occupancyGrid})$ 
9:       if  $\text{score} > \text{bestScore}$  then
10:        bestScore  $\leftarrow \text{score}$ 
11:        bestYaw  $\leftarrow \text{angle}$ 
12:        bestPoint  $\leftarrow p$ 
13:        fittingFrontierFound  $\leftarrow true$ 
14:     else
15:       if  $\text{qualityOffFitInFOV} > \text{fit}$  then
16:         fit  $\leftarrow \text{qualityOffFitInFOV}$ 
17:         fittingAngle  $\leftarrow \text{angle}$ 
18:         fittingPoint  $\leftarrow p$ 
19:         angle  $\leftarrow \text{angle} + \text{accuracy}$ 
20:     if fittingFrontierFound! = true then
21:       bestScore  $\leftarrow \text{CALCSCORE}(\text{fittingPoint}, \text{fittingAngle}, FOV, d_{\text{reach}}, \delta_{\text{resolution}}, \text{occupancyGrid})$ 
22:       bestYaw  $\leftarrow \text{fittingAngle}$ 
23:       bestPoint  $\leftarrow \text{fittingPoint}$ 

```

field of view mit dem aktuell betrachteten Blickwinkel als Mitte.

Im letzten Schritt wird der Informationsgewinn mit Hilfe der L2-Norm ermittelt. Die Kopie des Bildausschnitts wird mit dem originalen Ausschnitt pixelweise verglichen und die L2-Norm mit Hilfe der aufgeführten Gleichung 17 berechnet:

$$\text{L2 - Norm} = \sqrt{\sum_{i=0}^N d(\text{color}_{i,a}, \text{color}_{i,b})^2} \quad (17)$$

Es wird vorausgesetzt, dass beide Bilder gleich viele Einträge haben. Die Indizes a und b stehen dabei für die verschiedenen Bilder, N ist die Anzahl aller Einträge im Bild. Die Funktion $d(\text{color}_a, \text{color}_b)$ stellt den Betrag der Differenz, der beiden Pixelgrauwerte dar. Damit erhält man ein gutes Maß, wie sehr sich ein Bild vom anderen unterscheidet. Je größer die Zahl dabei ist, desto mehr unbekannte Fläche kann aufgedeckt werden. Ist die potentiell aufdeckbare Fläche ermittelt, kann mit der nachfolgenden Gleichung die Punktzahl der Frontier errechnet werden, wobei hier angemerkt sei, dass die Gleichung 11 aus Unterkapitel 2.6 angepasst wurde, um Grenzen, die vor dem mobilen Roboter liegen, den Vorzug zu geben, was dem zur Einleitung dieses Unterkapitels genannten Problem, der eingeschränkten Bewegungsfreiheit der Ackermann-Lenkung

Rechnung tragen soll. Im Folgenden ist d wieder die Distanz, $\delta\Theta$ der Blickwinkelunterschied zum Roboter und n das Ergebnis der L2-Norm. Wie bei der einfachen Strategie wird die Distanz mit einem Faktor $\alpha_{\text{nbv,distance weight}}$ und der Winkelunterschied mit $\beta_{\text{nbv,angle weight}}$ gewichtet.

$$f(n, d) = ne^{-d\alpha_{\text{nbv,distance weight}}} \quad , \quad g(\delta\Theta) = e^{\frac{2\pi - |\delta\Theta|}{\pi}\beta_{\text{nbv,angle weight}}} \quad (18)$$

$$\text{score}(n, d, \delta\Theta) = f(n, d) \cdot g(\delta\Theta) \quad (19)$$

In Gleichung 19 geht das Ergebnis der Funktion $g(\delta\Theta)$ mit wachsendem Winkelunterschied gegen 1 und ausschließlich die Funktion der Distanz und des Informationsgewinns $f(n, d)$ wirkt. Im Umkehrschluss bedeutet das: Geht die Winkeländerung gegen 0, so wird der Winkelunterschied maßgeblich und die Funktion $f(n, d)$ wird außer in Extremfällen abgeschwächt. Die Konstante $\alpha_{\text{nbv,distance weight}}$ bestimmt, wie sehr Rücksicht auf die zurückgelegte Distanz genommen werden soll. Wird diese sehr groß gewählt, so wird es für den Algorithmus irrelevant, wie weit Grenzen entfernt liegen und es wird nur der mögliche Informationsgewinn in Betracht gezogen. Analog kann der Distanz mit einem $\alpha_{\text{nbv,distance weight}} < 1$ eine sehr große Bedeutung beigemessen werden. Es hat sich bei Versuchen bewährt, ein kleines $\alpha_{\text{nbv,distance weight}}$ zu nehmen, da n durch die L2-Norm sehr groß wird.

Am Ende des Algorithmus 5 wird entweder der Punkt und Blickwinkel für eine Frontier mit der besten Bewertung zurückgegeben oder der sampling point, bei dem die Frontier die größte Überlappung mit dem field of view des Roboters hat.

3.5.3 Der ROS-Node automap

Damit der Exploration-Planer auf dem Roboter und in der Simulation zum Einsatz kommen kann, muss dieser in einen ROS-Node eingebunden werden. Es wird im Folgenden eine Art Wrapper für ROS erstellt, der sich im gleichnamigen ROS-Paket automap [Ehm16a] befindet.

Zunächst müssen die folgenden zwei Voraussetzungen für den Planer erfüllt werden:

Erstens abonniert der Node die Topics map und map_metadata, die hier vom Paket gmapping bereitgestellt werden. Dabei liegen die occupancy grids, die mit ROS-Navigationsnachrichten über das Topic map versendet werden, immer als eindimensionales Arrays vor. Allerdings benötigen alle Komponenten des Exploration-Planers ein occupancy grid im Format einer $n \times m$ Mat-Matrix. Daher werden bei jeder neu eintreffenden Karte die hintereinander abgelegten Daten des eindimensionalen Arrays, in eine Mat-Matrix umgewandelt, die von OpenCV-Algorithmen verwendet werden kann. Die Dimensionen dieser Matrix bestimmen sich über die Nachrichten von map_metadata, darin sind Höhe und Breite des occupancy grid vermerkt.

$$\text{ocvMat}_{\text{height}-(i+1), j} = \text{gmap}_{i \cdot \text{width}+j} \quad (20)$$

Mit Gleichung 20 lassen sich die Elemente des Arrays von gmapping denen der Mat-Matrix zuordnen, wobei i der Index der Zeilen und j der Index der Spalten ist. Während

dieser Umrechnung wird außerdem sichergestellt, dass die resultierende Karte nur die bereits genannten drei Zustände, frei, unbekannt und besetzt haben kann. In diesem Fall werden die Grauwerte für Schwarz, Grau, und Weiß, also 0, 205 und 255, verwendet.

Zweitens muss eine Odometriebestimmung stattfinden, die über das Topic `odom` Nachrichten verschickt, damit die Position des Roboters von ROS erfragt werden kann. Die Odometrie wird bei der Simulation wie in Unterabschnitt 3.2.1 beschrieben vom Modell bereitgestellt. Auf dem Roboter wird diese Aufgabe vom in Unterabschnitt 3.1.2 beschriebenen Paket aus dem Projektseminar Echtzeitsysteme erfüllt. Die von ROS gelieferten Positionsdaten sind dann in kartesischen Koordinaten im globalen Kontext angegeben. Man spricht dabei von Weltkoordinaten. Damit sie vom Exploration-Planer verwendet werden können, werden diese mit der folgenden Gleichung von Weltkoordinaten in Bildkoordinaten umgerechnet.

$$\text{grid}_x = \frac{\text{global}_x - \text{origin}_x}{\text{resolution}}, \text{ grid}_y = \frac{\text{global}_y - \text{origin}_y}{\text{resolution}} \quad (21)$$

Origin ist hierbei der Ursprungspunkt der globalen Koordinaten auf dem occupancy grid und resolution die Auflösung eines Pixels pro Meter. Beide Informationen werden ebenfalls über das Topic `map_metadata` geliefert.

Wie man im Diagramm in Abbildung 3.14 erkennen kann, ist der Exploration-Planer

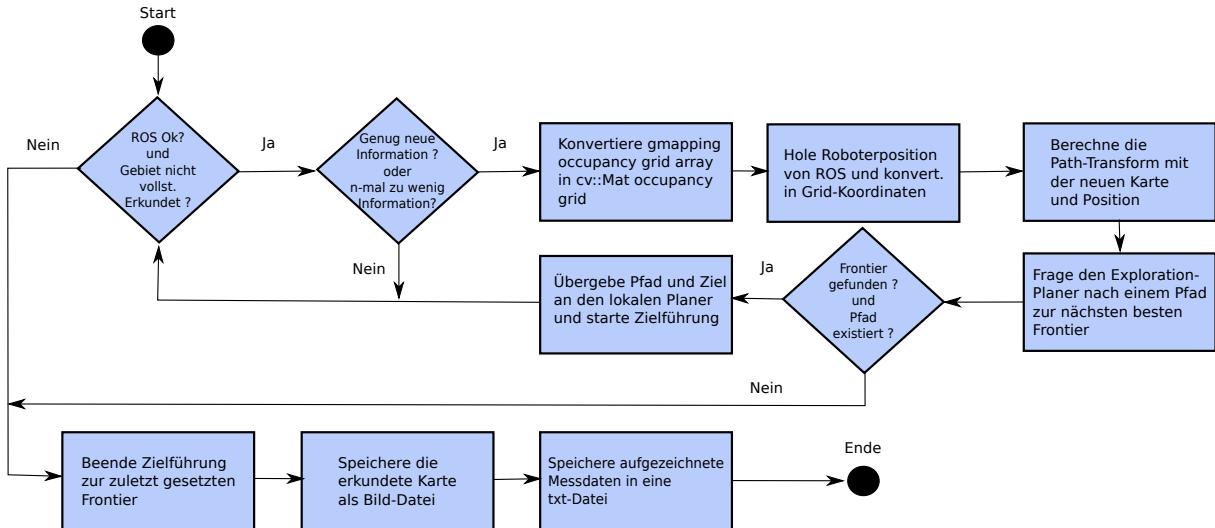


Abb. 3.14: ROS-Node `automap`: Ablaufdiagramm

in eine While-Schleife eingebettet. Das ist die Hauptschleife, die in jedem ROS-Node vorkommt. Ihre Abbruchkriterien können einerseits Signale vom System sein und andererseits ein Flag vom Exploration-Planer, wenn keine Frontiers mehr zu finden sind. Diese Schleife läuft mit einer einstellbaren Frequenz ab, das heißt, am Ende jedes Durchlaufs wird so lange gewartet, bis die Zeit, die durch die Wiederholrate eingestellt wurde, abgelaufen ist.

Die äußere Bedingung reguliert dabei, wie oft neue eingehende occupancy grids analysiert werden. In den meisten Fällen lohnt sich das nicht, da das Fahrzeug eine endliche

Geschwindigkeit besitzt und nach jedem Durchlauf der While-Schleife kaum neue Gebiete erkundet wurden. Die Regulierung findet statt, indem die neue eingehende Karte mit der alten über die L2-Norm verglichen wird. Ist die Punktzahl kleiner als die frei wählbare Konstante $\Delta_{\text{last grid}}$, so wird der Code in der Bedingung übersprungen. Das kann $n = n_{\text{retries}}$ -mal passieren, wobei n_{retries} ebenfalls eine frei wählbare Konstante ist. Danach wird eine Ausführung des Codes in der Bedingung forciert, um auch nach langsamem Fahrmanövern und wenig Informationsgewinn noch regelmäßig Analysen der Umgebung zu liefern.

Vor der Analyse durch den Exploration-Planer führt der Path-Transform-Planer seine Transformationen durch. Der Exploration-Planer erhält über einen Zeiger Zugriff auf das Path-Transform-Objekt und kann danach beliebig oft Wege auf den zuvor durchgeföhrten Transformationen planen lassen. Anschließend stellt der Exploration-Planer wie in Unterkapitel 3.5 beschrieben fest, welche Grenze als nächstes angefahren werden soll. Gibt es eine Frontier, so wird ein Pfad errechnet und eine Bestätigung ausgegeben, gibt es keine, so wird eine Verneinung ausgegeben und kein Pfad errechnet.

War der Exploration-Planer erfolgreich, so wird der berechnete Pfad vom Planer erfragt und als globaler Plan für den lokalen Planer vorgegeben. Dabei kann auch bei einer laufenden Zielführung ein neuer Plan übergeben werden, der local planner entscheidet, wie er am besten von einem Plan auf den anderen wechselt.

Ist das Flag für vollständige Erkundung vom Exploration-Planer gesetzt worden, so tritt die letzte Phase in Kraft. Hierbei wird die aktuelle Zielführung zu Ende gebracht und anschließend wird die erkundete Karte mit den Mapmetadaten gespeichert. Zuletzt wird der zurückgelegte Weg vom Modell oder der Fahrzeugkontrolle erfragt sowie die verstrichene Zeit während der Erkundung ermittelt und in einer Textdatei hinterlegt. Dies wird in Kapitel 4 dazu verwendet, um zu vergleichen, wie gut die Algorithmen gegeneinander abschneiden, bzw. wie sich Parameter auf die Erkundung auswirken.

4 Evaluation und Diskussion

In diesem Kapitel geht es darum, die in Unterkapitel 2.6 beschriebenen und in Unterkapitel 3.5 implementierten Erkundungsstrategien zu testen und zu evaluieren. Dafür werden die beiden Techniken zuerst mit Hilfe der in Unterkapitel 3.2 beschriebenen Simulation, in verschiedenen fiktiven Umgebungen getestet und Daten, wie beispielsweise zurückgelegter Weg und benötigte Zeit, gemessen. Im Anschluss werden die Messdaten der jeweiligen Strategien miteinander verglichen und wenn möglich die beste Strategie ermittelt. Um die Messwerte besser einordnen zu können, werden die Szenarien der Simulation auch von einem Menschen absolviert. Als letztes wird der Test auf dem echten Roboter evaluiert. Hierbei wird zuerst ein Raum mit Hindernissen, beispielsweise Tischbeine, mit Hilfe der besten Strategie erkundet. Danach wird das Ergebnis präsentiert und dabei auftretende Probleme betrachtet.

4.1 Simulation der Erkundungsstrategien

Um die verschiedenen Strategien zu simulieren, benötigt es zum einen die in Unterkapitel 3.2 beschriebene Simulation, die zeitgleich mit dem Node `automap` und dem `gmapping` Paket läuft. Die simulierten Sensoren erhalten Zugriff auf eine vollständige Karte, die dem Erkundungsalgorithmus und dem SLAM-Algorithmus nicht bekannt sind. Damit wird nur anhand der Sensorinformation, wie in der Wirklichkeit auch, eine neue Karte der Umgebung erstellt. Im Nachfolgenden sind die fünf Testkarten aufgeführt, auf denen die Simulation arbeiten soll.

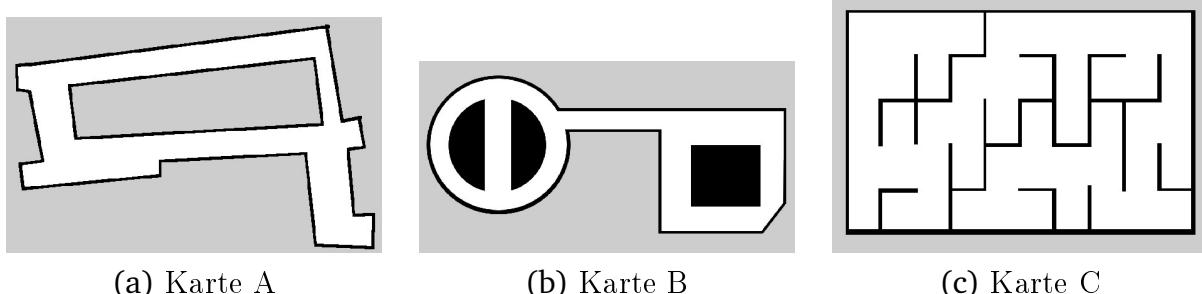


Abb. 4.1: *Testkarten Teil 1*

Karte A in Abbildung 4.1 ist die abstrahierte Version eines Teilstückes des Hans-Busch-Instituts der TU-Darmstadt. Karte B stellt eine fiktive Karte mit sehr großen Strukturen dar und soll die Funktionsfähigkeit der Algorithmen demonstrieren, wenn Teile einer Umgebung wegen ihrer Größe nicht voll vom Sichtfeld des Sensors erfasst werden können. Karte C soll zeigen, dass auch in kleinteiligen Strukturen gearbeitet werden kann. Die Karten D und E aus Abbildung 4.2 sind einer Büroumgebung ohne und mit Möblierung nachempfunden. Damit soll untersucht werden, wie sehr kleine Hindernisse wie Tisch- und Stuhlbeine Einfluss auf die autonome Erkundung haben. Die Abbildung 4.2 c) zeigt, wie die grafische Aufzeichnung des zurückgelegten Weges während der Erkundung aussieht, diese wird zusätzlich zur Aufzeichnung der Strecke durchgeführt. Um die von der Simulation bereitgestellte Position des Robotermodells nicht durch die

4.2 Testkarten

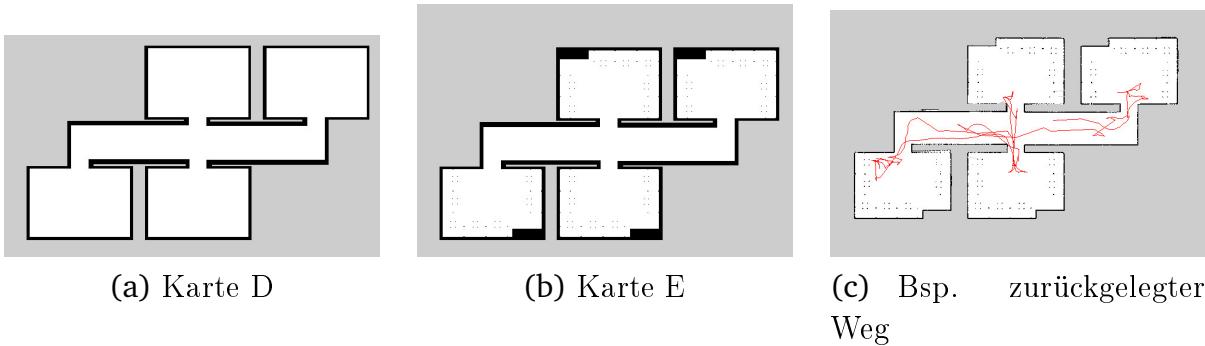


Abb. 4.2: *Testkarten Teil 2*

Positionsschätzung des in Unterabschnitt 3.1.2 beschriebenen gmapping Paketes zu verfälschen, wird diese Funktion deaktiviert. Die von den Entfernungssensoren erhaltene Information wird direkt in die Karte integriert, ohne die Korrektheit der Roboterposition in Frage zu stellen. Das produziert wegen der exakten Positionsbestimmung eine nahezu perfekte Karte und verhindert das Verfälschen der Messergebnisse durch nicht deterministische Nebeneffekte, die beim Anwenden von gmapping entstehen können. Die Simulation wird mit Hilfe von Rviz visualisiert. Dieses Werkzeug liegt ROS standardmäßig bei und stellt das occupancy grid, die Position des Roboters, geplante Trajektorien und die Entfernungssensoren dar. Diese Visualisierung wird mit einem screen capture Programm während eines Versuches aufgezeichnet, um nachher zum einen den erfolgreichen Versuch nachweisen zu können und zum anderen Verhaltensweisen der beiden Strategien zu analysieren.

4.1.1 Auswertung der Simulationsergebnisse

Vor der Diskussion der Simulationsergebnisse, werden die gemessenen Größen erläutert und es wird festgehalten, welche Werte die Parameter für die entsprechende Strategie annehmen.

Die Größe d_E beschreibt den gesamten zurückgelegten Weg während der Erkundung, analog dazu stellt t_E die gesamte verstrichene Zeit vom Start des Nodes automap bis zur Terminierung dar. Die Größen t_{PTP} , t_{EP} und t_{Node} beschreiben die über den gesamten Verlauf gemittelte Ausführungszeit des Path-Transform-Planers, des Exploration-Planers und der While-Schleife im ROS-Node. Es werden pro Karte fünf Durchläufe simuliert. Dies erzeugt sehr viele Messergebnisse, was die Diskussion unübersichtlich macht. Es werden daher nur die über alle Durchläufe gemittelten Ergebnisse und die Standardabweichung vom Mittelwert der Ergebnisse präsentiert. Die jeweilige Standardabweichung wird mit einem S nach der Bezeichnung angezeigt.

Für alle Durchläufe und beim Test mit menschlicher Steuerung, gilt eine maximale Geschwindigkeit des Roboters von $1 \frac{m}{s}$, sowohl vorwärts als auch rückwärts. Die in Unterkapitel 3.4 genannten Parameter downsampling-Faktor, minmale Entfernung zu Hindernissen und der alpha-Faktor für den Path-Transform-Planer, nehmen die Werte $f_{down\ sampling} = 0.2$, $d_{min,obstacle\ dist} = 0.4\ m$ und $\alpha_{obstacle\ awareness} = 1.0$ an.

Der Parameter des rolling window beträgt für beide Strategien $d_{rolling\ window} = 22\ m$ und

stellt in diesem Fall die Kantenlänge des quadratischen Fensters dar. Da sich der Roboter immer in der Mitte des Fensters befindet, ist mit diesen Ausmaßen gewährleistet, dass ein Scan mit der maximalen Reichweite von 10 m sich mit Toleranz noch im Fenster befindet.

Für die Parameter der einfachen Strategie (siehe Unterabschnitt 3.5.1) haben sich $\alpha_{\text{simple}, \text{distance weight}} = 0.13$ und $\beta_{\text{simple}, \text{angle weight}} = 1.96$ als gut funktionierende Werte herausgestellt.

Die NBV-Strategie (siehe Unterabschnitt 3.5.2) zeigte die besten Ergebnisse mit $n_{\text{points}} = 25$, $\alpha_{\text{nbv}, \text{distance weight}} = 0.12$ und $\beta_{\text{nbv}, \text{angle weight}} = 1.5$. Für das Sampling an den einzelnen generierten sampling points wird ein Sichtfeld $FOV = 70^\circ$, eine maximale Sensorreichweite $d_{\text{reach}} = 10 \text{ m}$ bei einer Winkelauflösung $\delta_{\text{resolution}} = 0.1^\circ$ angenommen. Das entspricht den Kenndaten einer Kinect Version 2.

Der Node automap wird mit einer Frequenz von 2 Hz betrieben. Nach $n_{\text{retries}} = 12$ oder beim überschreiten des Schwellwerts $\Delta_{\text{last grid}} = 2500$, wird ein Durchlauf des Exploration-Planers erzwungen. Der Schwellwert ist dabei ein zu überschreitender Wert, der aus dem Berechnen der L2-Norm vom occupancy grid der letzten Iteration mit dem neu eingegangen resultiert.

Für die Simulationen der NBV-Strategie mit fixen Parametern ergeben sich die in Tabelle 4.1 und für die einfache Strategie mit fixen Parametern die in Tabelle 4.2 präsentierten Ergebnisse.

Tabelle 4.1: *Simulationsergebnisse der NBV-Strategie für feste Parameter*

Karte	$d_E(m)$	$t_E(s)$	$t_{\text{PTP}}(s)$	$t_{\text{EP}}(s)$	$t_{\text{Node}}(s)$	$d_{\text{ES}}(m)$	$t_{\text{ES}}(s)$	$t_{\text{PTPS}}(s)$	$t_{\text{EPS}}(s)$	$t_{\text{Nodes}}(s)$
A	97,54	214,48	0,238451	0,452597	0,177684	9,54	33,33	0,005140	0,038041	0,013477
B	213,66	400,84	0,236562	0,460536	0,187192	16,22	40,42	0,002731	0,084241	0,027261
C	206,78	645,07	0,242448	0,973535	0,195135	20,74	14,54	0,004023	0,211105	0,029151
D	122,14	329,35	0,239729	0,532979	0,170962	8,20	68,09	0,006460	0,095286	0,019324
E	136,17	400,41	0,234205	0,722015	0,195121	25,65	81,39	0,004888	0,151157	0,040723

Tabelle 4.2: *Simulationsergebnisse der einfachen Strategie für feste Parameter*

Karte	$d_E(m)$	$t_E(s)$	$t_{\text{PTP}}(s)$	$t_{\text{EP}}(s)$	$t_{\text{Node}}(s)$	$d_{\text{ES}}(m)$	$t_{\text{ES}}(s)$	$t_{\text{PTPS}}(s)$	$t_{\text{EPS}}(s)$	$t_{\text{Nodes}}(s)$
A	139,87	303,11	0,244269	0,143596	0,102965	18,61	76,62	0,004542	0,026209	0,007022
B	243,61	479,40	0,247711	0,108964	0,103428	24,37	55,96	0,006168	0,028374	0,005626
C	224,57	710,79	0,240346	0,051902	0,073654	22,48	78,11	0,005803	0,015004	0,002056
D	113,26	292,93	0,242132	0,061977	0,089459	7,52	32,49	0,006748	0,026806	0,003745
E	110,02	302,22	0,244661	0,078544	0,091128	13,91	42,70	0,005565	0,032220	0,004076

Es ist zu erkennen, dass die NBV-Strategie in Karte A, B und C mit großem Abstand bessere Ergebnisse produziert. Das ist sowohl in Anbetracht der gefahrenen Strecke als auch der benötigten Zeit der Fall. Zusätzlich sind die Standardabweichungen der Erkundungszeit und des Weges in allen drei Fällen kleiner, was auf eine höhere Konsistenz hinweist. Wie erwartet fällt auf, dass die NBV-Strategie einen sehr viel höheren Rechenaufwand nach sich zieht, die Durchführungszeiten sind etwa 10 mal höher als bei der einfachen Strategie. Die Durchführungszeiten des Path-Transform-Planers sind

dagegen bei allen Karten fast konstant, unabhängig davon, wie groß die erkundete Karte ist. Diese Konstanz resultiert hauptsächlich daraus, dass das occupancy grid bereits mit 4000x4000 Pixel angenommen wird, egal wie groß die erkundete Fläche tatsächlich ist. Erst wenn die Ausmaße größer werden und das occupancy grid mit einer höheren Auflösung neu initialisiert werden muss, wird die Durchführungszeit des Planers massiv ansteigen.

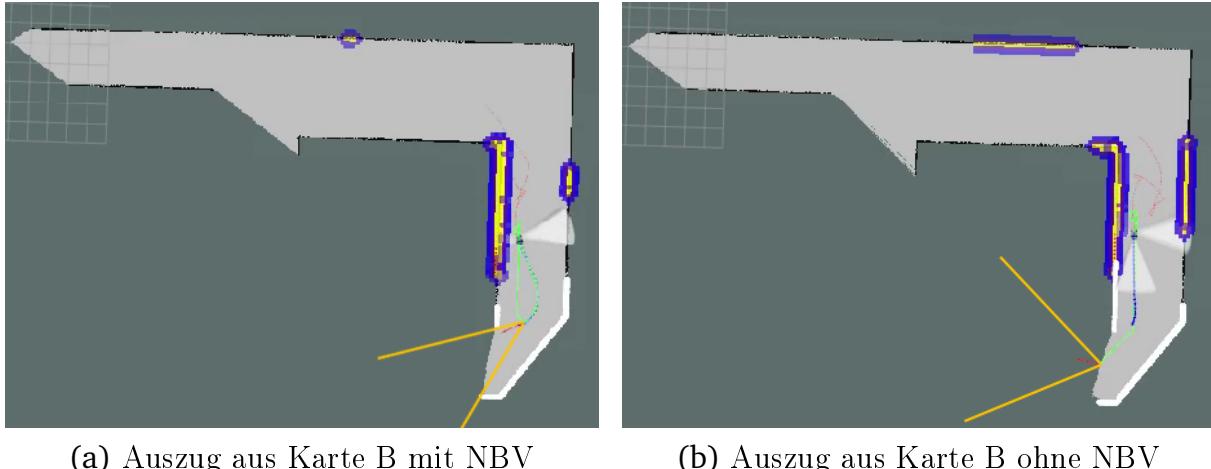
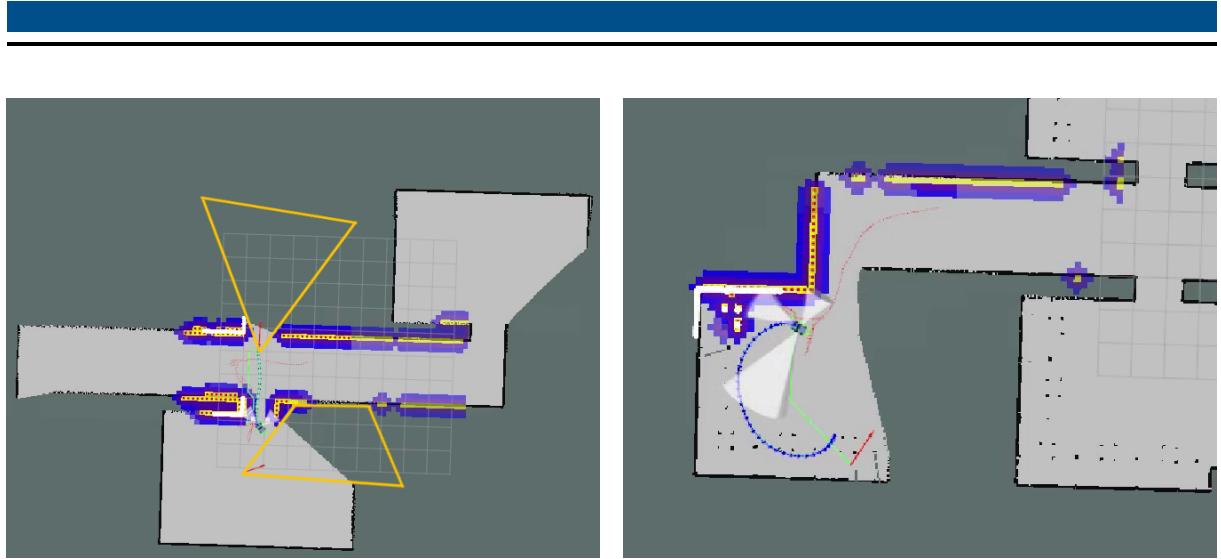


Abb. 4.3: *Vergleich zwischen NBV- und einfacher Strategie*

In Abbildung 4.3 kann man eine mögliche Ursache für die höhere Effizienz des NBV-Algorithmus erkennen. Wenn man die möglichen Blickwinkel, die mit orange eingefärbten Linien markiert sind, an den Zielpunkten für die jeweilige Strategie betrachtet, kann man bei der NBV-Strategie in a) sehen, dass die Ecke an dieser Stelle der Karte fast optimal erkundet wird, wohingegen die einfache Strategie in b) das Modellauto direkt auf die Grenze zum Unbekannten fahren lässt. Damit verbleiben rechts und links des Fahrzeuges unerkannte Gebiete, die im Anschluss durch Drehung des Fahrzeuges erkundet werden müssen, was Zeit kostet und zusätzlich gefahrene Strecke erzeugt. Eine weitere Ursache für die höhere Stabilität ist, dass bei der Bewertungsfunktion für die Qualität einer Frontier bei der NBV-Strategie, neben der Distanz zur Frontier und dem Blickwinkelunterschied von Roboter und Zielvorgabe, der potentielle Informationsgewinn betrachtet wird. Dadurch wird die Zielvorgabe mit NBV weniger opportunistisch, das Fahrzeug verfolgt ein zuletzt vorgegebenes Ziel länger und wird von näheren Frontiers mit weniger Informationsgewinn nicht abgelenkt. Bei der einfachen Strategie passiert es dabei häufig, dass ein spontaner Richtungswechsel erfolgt, weil sich beispielsweise eine neue Frontier durch Passieren eines abzweigenden Ganges aufgetan hat.

Umgekehrt scheint dieses Verhalten, das nahe und leicht zu erreichende Frontiers ungeachtet des potentiellen Informationsgewinns favorisiert, in den Karten D und E die richtige Strategie zu sein. Damit werden die von den Gängen abgezweigten Räume nacheinander erkundet. Unter Verwendung der NBV-Strategie hat die Bevorzugung von Frontiers, die in der Nähe von großen, unerkannten Bereichen liegen, dazu geführt, dass der simulierte Roboter zwischen den Räumen hin und her gesprungen ist. In Abbil-



(a) Auszug aus Karte D mit NBV

(b) Auszug aus Karte E mit NBV

Abb. 4.4: Probleme bei der NBV-Strategie

dung 4.4 a) zeigt sich, dass die Frontier mit dem orange markierten Informationsgewinn der deutlich näheren Frontier vorgezogen wird, was das zuvor beschriebene Verhalten des Springens provoziert. In Abbildung 4.4 b) kann man erkennen, dass die Zielvorgaben mit der NBV-Technik für den lokalen Planer oft anspruchsvoll sein können, gerade wenn Hindernisse im Raum sind. Die blauen Pfeilketten zeigen den Pfad des lokalen Planers, welcher sehr knapp an Hindernissen vorbeiführt, was zwangsläufig zu vielen Ausweichmanövern und damit zusätzlicher Laufzeit führt. Das erklärt auch, warum die NBV-Technik auf der Karte E zwar kaum mehr Weg erzeugt, aber viel mehr Laufzeit.

Beim Test mit menschlicher Kontrolle wurde pro Karte nur ein Durchlauf mit zwei Personen durchgeführt, da Menschen naturgemäß die Karten lernen und mit jedem Durchlauf besser werden. Die Ergebnisse dieser Simulation zeigt Tabelle 4.3 und werden zusammen mit den, über alle Karten gemittelten Ergebnisse der beiden Strategien, in Tabelle 4.4 noch einmal übersichtlich zusammengefasst. Zu den Ergebnissen, die mit

Tabelle 4.3: *Simulationsergebnisse mit menschlicher Steuerung*

Karte	$d_E(m)$	$t_E(s)$	$d_{ES}(m)$	$t_{ES}(s)$
A	118,80	307,66	15,63	100,92
B	268,07	586,24	8,79	186,06
C	251,34	1124,33	53,18	409,83
D	125,27	388,81	54,97	56,58
E	97,025	343,26	6,88	55,16

menschlicher Kontrolle entstanden sind, muss noch erwähnt werden, dass die Versuchspersonen die maximal erlaubte Geschwindigkeit selten erreicht haben, da die Eingabe über die Tastatur sich als weniger intuitiv erwiesen hat als gedacht. Hierbei dienten die Tasten W, S, A und D als Eingaben für Geschwindigkeit erhöhen, verringern, Lenkwinkel

Tabelle 4.4: *Gemittelte Simulationsergebnisse*

Strategie	d_ex	t_ex	t_pp	t_ep	t_ml	d_ex_stdv	t_ex_stdv	t_pp_stdv	t_ep_stdv	t_ml_stdv
NBV	155,26	398,03	0,238279	0,628332	0,185219	16,07	47,55	0,004649	0,115966	0,025987
Einfach	166,26	417,69	0,243824	0,088997	0,092127	17,38	57,18	0,005765	0,025723	0,004505
Mensch	172,10	550,06	-	-	-	27,89	161,71	-	-	-

erhöhen und verringern. Gerade bei der sehr kleinteiligen Karte C ist dieser Umstand stark aufgefallen, daher werden die Ergebnisse weniger als harte Bewertungsgrundlage, sondern als grober Vergleich für die beiden anderen Erkundungsstrategien verwendet.

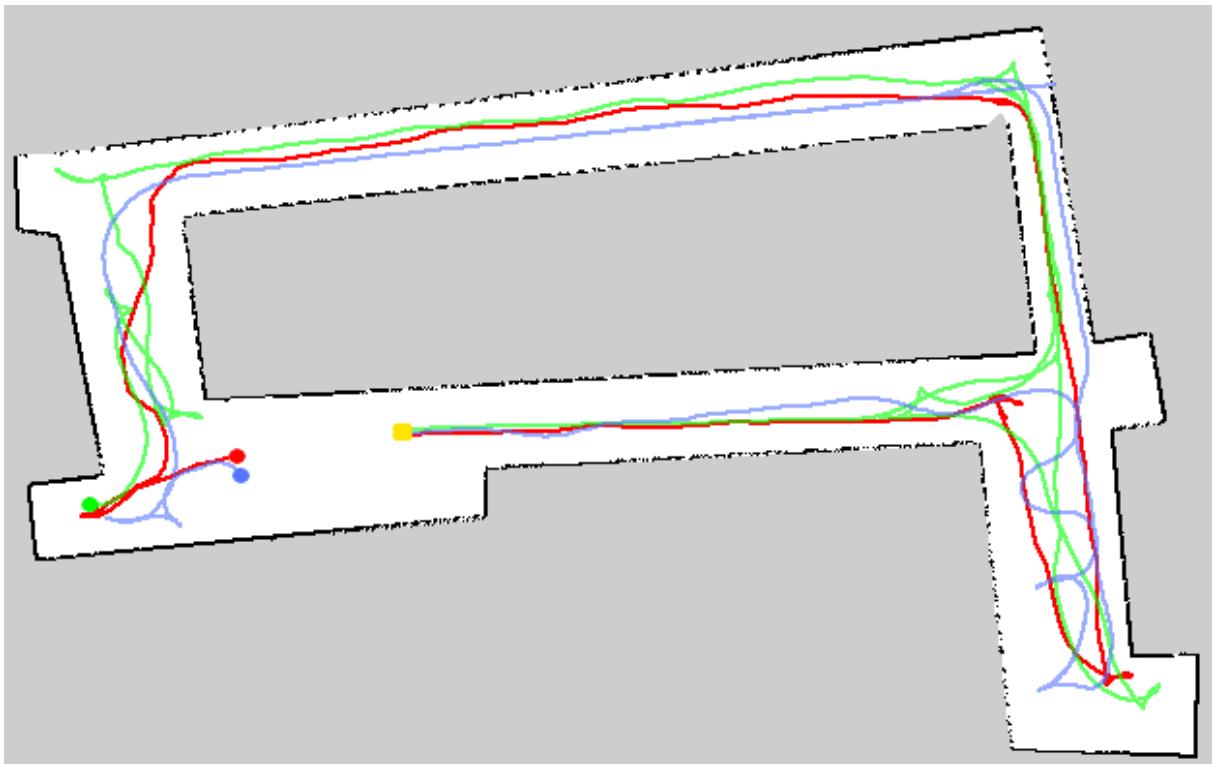


Abb. 4.5: *Vergleich der zurückgelegten Wege*

In Abbildung 4.5 sind die gefahrenen Wege unter Kontrolle der NBV-Strategie (Rot), der einfachen Strategie (Grün) und des Menschen (Blau) übereinander gelegt. Hierbei dient Karte A als exemplarisches Beispiel, da die Messwerte in allen drei Fällen nah beieinander liegen und sich so gut vergleichen lassen. Dabei fällt auf, dass unter Kontrolle der NBV-Strategie der sauberste Pfad entsteht, der dem menschlichen Pfad ähnlich ist. Im Vergleich dazu kann man bei der einfachen Strategie sehr viele Wendemanöver erkennen, die in längerer Laufzeit und mehr zurückgelegter Strecke resultieren.

Bei den Messwerten kann man auch sehen, dass beide Strategien in allen Karten bis auf E schneller und effizienter gearbeitet haben, als die menschlichen Versuchspersonen. Die längere Laufzeit bei der menschlichen Erkundung kann auf die zuvor erwähnte, weniger ergonomische Steuerung zurückzuführen sein. Zusätzlich kann durch den guten lokalen Planer, der eine konstant hohe Geschwindigkeit während der Erkundung ermöglicht hat, eine kurze Laufzeit bei den autonomen Modi erreicht werden.

Es zeigt sich, dass der NBV-Algorithmus, gemessen an den Standardabweichungen, der robusteste ist und auf den meisten Karten stabile Ergebnisse aufweist. Zudem sind die Zeiten oft die kleinsten und die Wegstrecken die kürzesten. Allerdings stellt sich auch heraus, dass eine einfache Herangehensweise in Szenarien wie Karte D und E, die Beste und in Anbetracht der generell sehr kleinen Ausführungszeiten t_{EP} , in manchen Anwendungsfällen eine gute Strategie ist. Für ein gutes Funktionieren der beiden autonomen Erkundungsmodi ist vor allem eine gute Einstellung der verschiedenen Parameter wichtig. Der Einfluss der wichtigsten Parameter wird im nächsten Abschnitt näher betrachtet.

4.1.2 Auswirkungen von verschiedenen Parametern

Zunächst wird betrachtet, welche der zu Beginn von Unterabschnitt 4.1.1 genannten Parameter tatsächlich Einfluss auf die verschiedenen Strategien haben. Die Größe des rolling windows kann eine Rolle beim Verhalten spielen, allerdings macht es wenig Sinn, das Fenster kleiner als die Sichtweite des Sensors zu dimensionieren, da sonst wichtige Frontiers verworfen werden. Umgekehrt ist es nicht sinnvoll, es sehr viel größer zu machen, da mit zunehmender Größe des Fensters auch die Ausführungszeit der Algorithmen steigt und die Funktion des rolling windows immer weniger Nutzen mit sich bringt. Ähnlich verhält es sich mit den Sampling-Parametern der NBV-Strategie, diese orientieren sich fest an dem zu simulierenden Sensor. Daher bleiben als entscheidende Parameter für das Verhalten der Erkundungsstrategien in beiden Fällen die α - und β -Faktoren. Diese bestimmen, wie sehr die Weglänge beziehungsweise der Blickrichtungsunterschied zwischen Zielpunkt und Roboter eine Rolle bei der Bewertung einer Frontier spielt. Auf dieser Bewertungsgrundlage wird die beste Frontier ausgewählt und als nächstes Ziel vorgegeben.

Um den Einfluss dieser Parameter zu untersuchen, werden auf Karte A pro eingestelltem Parameter und pro Erkundungsmodus zwei Durchläufe ausgeführt. Dabei wird immer ein Parameter auf einem Basiswert gehalten, der dem Wert der Simulationen aus Unterabschnitt 4.1.1 entspricht und nur der jeweils andere verändert. In Tabelle 4.5 und in Tabelle 4.6 sind die entsprechenden Messwerte für die NBV-Strategie beziehungsweise die einfache Strategie festgehalten. Die erste Zeile der jeweiligen Tabelle zeigt immer die Ergebnisse für die Basiswerte auf Karte A.

Tabelle 4.5: *Simulationsergebnisse der NBV-Strategie für variierte Parameter*

Parameter	$d_E(m)$	$t_E(s)$	$t_{PTP}(s)$	$t_{EP}(s)$	$t_{Node}(s)$	$d_{ES}(m)$	$t_{ES}(s)$	$t_{PTPS}(s)$	$t_{EPS}(s)$	$t_{NodeS}(s)$
Alpha: 0,12 Beta: 1,5	97,54	214,48	0,238451	0,452597	0,177684	9,54	33,33	0,005140	0,038041	0,013477
Alpha: 0,25 Beta: 1,5	109,59	250,59	0,236037	0,567953	0,185244	2,61	21,94	0,000591	0,065753	0,007481
Alpha: 0,5 Beta: 1,5	148,32	394,15	0,238285	0,577084	0,161668	1,36	38,69	0,008456	0,026409	0,005070
Alpha: 0,12 Beta: 0,8	102,59	232,78	0,229734	0,490408	0,175794	9,17	23,98	0,000669	0,093241	0,013847
Alpha: 0,12 Beta: 2,0	131,53	369,86	0,229276	0,538482	0,150518	20,38	0,73	0,001458	0,081650	0,016653

Tabelle 4.6: Simulationsergebnisse der einfachen Strategie für varierte Parameter

Parameter	$d_E(m)$	$t_E(s)$	$t_{PTP}(s)$	$t_{EP}(s)$	$t_{Node}(s)$	$d_{ES}(m)$	$t_{ES}(s)$	$t_{PTPS}(s)$	$t_{EPS}(s)$	$t_{Nodes}(s)$
Alpha: 0,13 Beta: 1,96	139,87	303,11	0,244269	0,143596	0,102965	18,61	76,62	0,004542	0,026209	0,007022
Alpha: 0,25 Beta: 1,96	101,49	264,30	0,237668	0,105767	0,097302	15,33	37,13	0,004658	0,050665	0,001048
Alpha: 0,5 Beta: 1,96	118,99	272,31	0,240102	0,136544	0,102366	24,06	46,88	0,010159	0,033643	0,000504
Alpha: 0,13 Beta: 0,8	91,72	243,74	0,236806	0,075979	0,091693	6,85	24,88	0,004547	0,028258	0,000977
Alpha: 0,13 Beta: 2,2	132,52	258,7	0,245012	0,105356	0,102005	18,78	17,83	0,004705	0,005765	0,000514

Die Messergebnisse zeigen, dass die Parameter der NBV-Strategie, zumindest für diese Karte, schon fast optimal waren. Umgekehrt wäre es bei der einfachen Strategie sinnvoll gewesen, entweder die Distanz zu den Frontiers stärker oder den Winkelunterschied geringer zu gewichten. Letzteres stellt die bessere Option dar, da die Standardabweichung für diese Parametereinstellung verhältnismäßig klein ausfällt und auf eine höhere Konsistenz bei den Erkundungsabläufen hindeutet. Bei der NBV-Strategie verkleinert sich

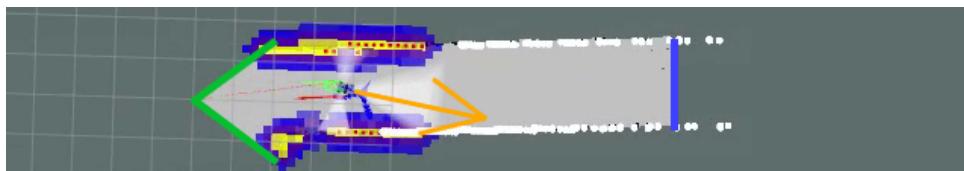


Abb. 4.6: Veränderung des Parameters alpha der NBV-Strategie

die Standardabweichung der Wegstrecke mit wachsendem α sehr stark, wobei die Wegstrecke und die Laufzeit steigen. Die starke Gewichtung der Wegstrecke führt zu einer schlechteren Effizienz, da nahen Frontiers der Vorzug gegenüber einfach zu erreichenden gegeben wird. Dieser Vorgang ist in Abbildung 4.6 gezeigt, hierbei stellt der gelbe Pfeil die Blickrichtung des Modellautos dar, die blau eingefärbte Frontier ist die einfacher zu erreichende und die grüne Frontier ist die vom Algorithmus bevorzugte. Diese Wendemanöver geschehen während der Simulation häufig an den gleichen Stellen auf der Karte und erklären die wachsende Konsistenz der Erkundungsabläufe und die höhere Laufzeit.

Umgekehrt wird die Standardabweichung mit wachsendem β immer größer, die Konsistenz der Abläufe immer schlechter. Das liegt daran, dass mit wachsendem Gewicht des Winkelunterschiedes zwischen den Zielen und der Roboterausrichtung die momentane Ausrichtung des Roboters während der Fahrt immer relevanter für die Bewertung wird. Es kann passieren, dass durch ein Lenkmanöver und die daraus resultierende Änderung der Ausrichtung eine Frontier attraktiv wird, die zwar sehr weit entfernt liegt, aber bei der Unterschied von Zielausrichtung und Roboterausrichtung sehr klein ist. Wird in Folge dessen eine neue Richtung eingeschlagen, um die jetzt besser bewertete Frontier anzusteuern, kann es passieren, dass die ursprünglich angesteuerte Frontier durch die erneute Richtungsänderung wieder besser bewertet wird. Damit entsteht ein Pendeln, das erst aufhört, wenn eine der beiden in Frage kommenden Frontiers nah genug ist

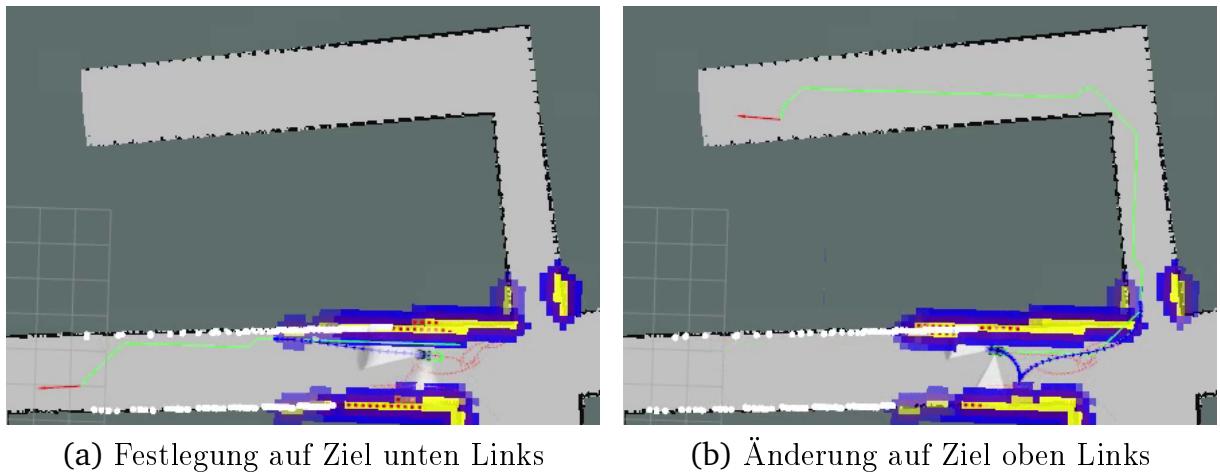


Abb. 4.7: *Problem des Pendels*

und das Gewicht des Winkelunterscheides keine Rolle mehr spielt. Die Abbildung 4.7 zeigt eine Situation, in der dieses Pendeln zwischen Frontiers auf Karte A mit $\beta = 2.0$ aufgetreten ist. Dieses Verhalten kann auch bei der einfachen Strategie auftreten und ist für die NBV-Strategie nur exemplarisch gezeigt. Bei den scheinbar optimalen Paramtern aus Unterabschnitt 4.1.1 ist dieses Problem, wenn auch selten, ebenfalls aufgetreten.

Es zeigt sich, dass durch Veränderung der Parameter eine deutliche Änderung des Verhaltens bei der Erkundung entsteht. Hierbei kann es im Falle der einfachen Strategie zu einer Verbesserung kommen oder bei der NBV-Strategie zu einer Verschlechterung, bis hin zur Unbrauchbarkeit. Da sich beim Auswerten der Messergebnisse die NBV-Technik, trotz ihrer Probleme, vom Verhalten und den Messwerten her als bessere Erkundungsstrategie herausgestellt hat, wird diese im nächsten Abschnitt unter realen Bedingungen auf dem Modellauto getestet.

4.2 Test der NBV-Strategie auf dem Modellauto

Der Test wird auf dem in Unterkapitel 3.1 vorgestellten Modellauto durchgeführt. Dabei kommen die im Unterabschnitt 3.1.2 beschriebenen Pakete zum Einsatz sowie der in dieser Arbeit entwickelte Node `automap`.

Als Vorbereitung für den Test wird das für die Simulationen ausgeschaltete Positionsschätzung des gmapping Paketes wieder aktiviert, da die durch Sensoren bestimmte Odometrie nicht gut genug ist für das korrekte Anordnen von Informationen auf einem occupancy grid. Die Positionsschätzung hilft, den Drift der Odometrie zu korrigieren. Das Problem ist, dass dieser Vorgang sehr viele Ressourcen in Anspruch nimmt. Es führt damit zu einer Überbelastung des Systems, das darauf mit Drosselung der anderen Nodes reagiert, was sich vor allem negativ auf den lokalen Planer auswirkt. Das äußert sich in schlechten Trajektorien, die teilweise nicht sicher oder ineffizient sind. Um dieses Problem zu umgehen, wird die Eigenschaft genutzt, dass ROS auf mehrere Systeme verteilbar ist. Konkret heißt das, dass der gmapping Node und die Visualisie-

rung auf einem anderen Rechner laufen, der sich im selben lokalen Netzwerk wie der mobile Roboter befindet. Auf dem Roboter wird neben den übrigen Nodes auch der ROS-Master betrieben, der die Kommunikation zwischen den einzelnen Nodes über das Netzwerk ermöglicht. Damit ist die Überlastung des Roboters beseitigt und der lokale Planer funktioniert wieder zuverlässig.

Zusätzlich hat sich herausgestellt, dass das Tiefenbild der Kinect v2 leicht rauscht. Das heißt, dass nach dem Abschneidevorgang durch das Paket `depthimage_to_laserscan` vereinzelt Entfernungswerte zufällig auftauchen, obwohl sich dort nichts befindet. Diese Fehlinformationen existieren nur sehr kurz, trotzdem kann es dazu führen, dass der SLAM-Algorithmus dieses Hindernis auf der Karte abbildet. Das verfälscht nicht nur die Karte, sondern macht die Erkundung fast unmöglich, da die große Anzahl an Hindernissen kaum fahrbare Trajektorien zulässt. Um dem zu begegnen, wird auf das Tiefenbild ein Tiefpassfilter in Form eines median blur Filters, aus der Image-Processing-Bibliothek von OpenCV [Ope16c], angewendet. Damit werden Punkte entfernt, die weit von der Mehrzahl der Punkte, dem Median der Punktmenge, entfernt liegen. Dies führt auch zu einer Glättung des Laserscans, was sich im Resultat in einer weniger verrauschten und weniger grobkörnigen Karte äußert.

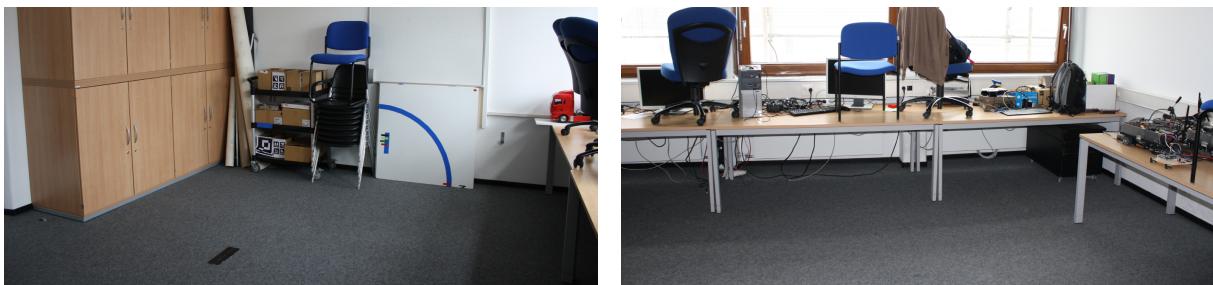


Abb. 4.8: *Testraum Teil 1*

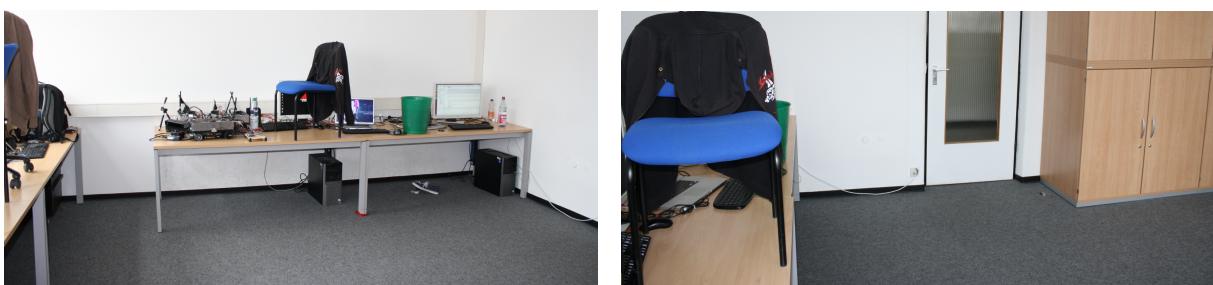


Abb. 4.9: *Testraum Teil 2*

Da die autonome Erkundung in dieser Arbeit ein abgeschlossenes Gebiet benötigt, um zu terminieren, ist das Testgebiet zum Kartographieren ein einzelner abgeschlossener Raum. Dieser ist in Abbildung 4.8 und in Abbildung 4.9 abgebildet, die Bilder geben von links nach rechts eine komplette Übersicht des Raums.

Nachdem alle Vorbereitungen abgeschlossen sind, wird die Erkundung des Raumes durchgeführt, wobei die nachfolgende Bilderreihe den Status nach jedem größeren

Erkundungsschritt zeigt. In Abbildung 4.10 Schritt 1 kann man sehen, wie der Algo-

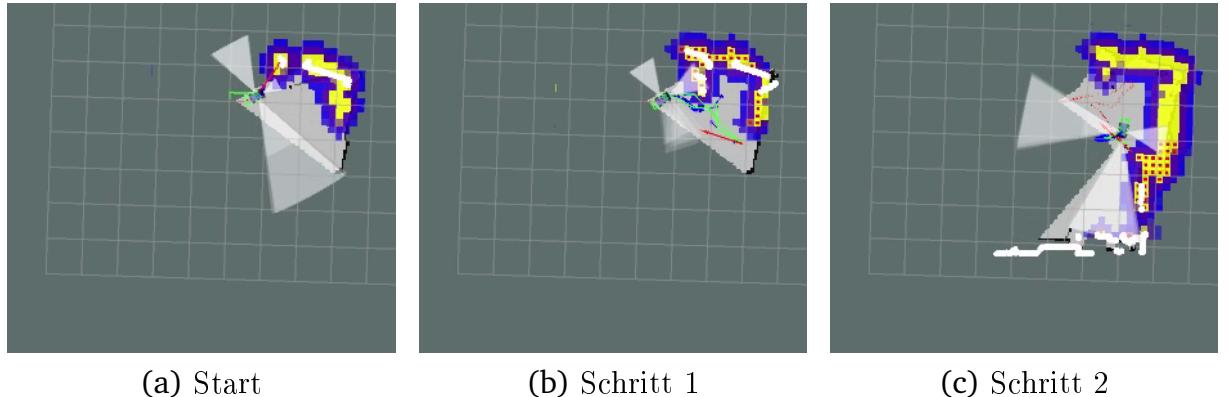


Abb. 4.10: *Erkundung mit dem Modellauto Teil 1*

rithmus festgestellt hat, dass in Blickrichtung nichts mehr zu erkunden ist und gibt eine Zielvorgabe für eine Erkundung in die entgegengesetzte Richtung. Diese neue Vorgabe

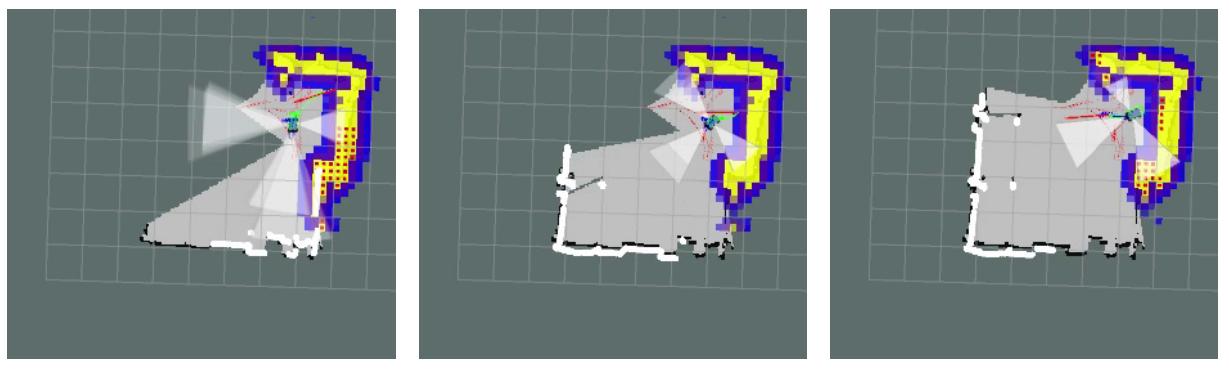
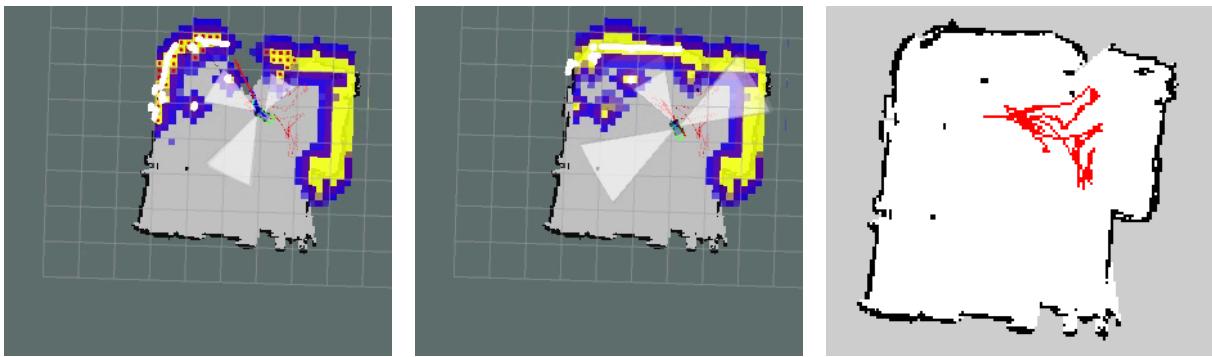


Abb. 4.11: *Erkundung mit dem Modellauto Teil 2*

setzt eine Drehung in Gang, die dazu führt, dass zwischen Schritt 2 in Abbildung 4.10 und Schritt 5 Abbildung 4.11 fast der komplette Raum erkundet wird. In Abbildung 4.12 Schritt 6 bleibt nur eine kleine Lücke zu schließen und die Erkundung ist Schritt 7 abgeschlossen. Die Abbildung 4.12 c) zeigt die fertige Karte mit dem zurückgelegten Pfad. Man erkennt, dass, wie in den Zwischenschritten beschrieben, im Endeffekt nur eine Drehung um 360 Grad stattgefunden hat.

In der Realität hat der Roboter große Probleme mit dem Bodenbelag, der in diesem Fall ein Teppichboden ist. Das führt dazu, dass die durch den lokalen Planer vorgegebenen Manöver wegen der langsam einlenkenden Vorderräder nur schlecht nachvollzogen werden können. Dadurch muss oft die Trajektorie korrigiert werden, was sich in den vielen Vorwärts - und Rückwärtsbewegungen äußert, die man auf dem abgebildeten Pfad sehen kann. Man kann zwar die einzelnen Merkmale des Raums gut erkennen, wie den Schrank rechts unten oder die einzelnen Tischbeine der Schreibtische, die sich als schwarze Punkte äußern, allerdings ist die Karte, wenn auch kaum bemerkbar leicht



(a) Schritt 6

(b) Schritt 7

(c) Resultat mit Pfad

Abb. 4.12: *Erkundung mit dem Modellauto Teil 3*

verzerrt. Das liegt daran, dass der Drift der Odometrie durch die Positionsschätzung des SLAM-Algorithmus nicht vollständig ausgeglichen werden kann. Damit werden Punkte nicht perfekt angeordnet, sondern immer nur so gut wie möglich oder die Information wird verworfen. Zusätzlich entsteht durch die Verteilung des Systems über das Netzwerk eine kleine Verzögerung bei der Korrektur der Odometrie. Dies äußert sich auch in der leichten Verzerrung der Karte. Insgesamt zeigt sich aber, dass der Algorithmus unter realen Bedingungen funktioniert und von sich aus terminiert.

Da bereits vor dieser Arbeit andere Arbeitsgruppen global erfolgreich, auf die eine oder andere Weise, autonome Erkundungsalgorithmen entwickelt haben, wird im folgenden Kapitel ein Blick auf ausgewählte Arbeiten geworfen. Dabei werden diese kurz zusammengefasst und erläutert, was in diesen Arbeiten anders, besser oder schlechter gemacht wurde.

5 Verwandte Arbeiten

Im folgenden Abschnitt werden einige ausgewählte Arbeiten genannt, die sich auf die gleichen Grundlagen stützen, deren Ergebnisse allerdings nicht Eingang in diese Arbeit gefunden haben. Sie dienen als Beispiel für den Einsatz von anderen Erkundungsstrategien oder mobilen Plattformen. Zusätzlich wird noch aufgezeigt, wo Unterschiede bestehen und Gemeinsamkeiten auftreten.

Allen voran wird an dieser Stelle die Arbeit "Hector Open Source Modules for Autonomous Mapping and Navigation with Rescue Robots" [KMG⁺13] von Kohlbrecher et al. genannt, die innerhalb der Arbeitsgruppe Team Hector an der TU-Darmstadt entstanden ist. Darin wird auf Basis von Wirth et al. [WP07] eine Methode zur Erkundung unbekanntem Terrains in drei Dimensionen beschrieben und am Schluss in Form eines ROS-Paketes zur Verfügung gestellt. Wie diese Arbeit, baut Team Hector auf einem frontier-based approach auf und plant die Trajektorien zu diesen Frontiers mit einem Path-Transform-Planer. Zusätzlich wurde noch eine alternative Strategie implementiert, die sich an der Erkundungsstrategie von Feuerwehrleuten orientiert. Hierbei werden zunächst die äußeren Begrenzungen eines Areals betrachtet. Hat sich eine abgeschlossene Außengrenze gebildet, so wird der grenzbasierte Ansatz verwendet, um den inneren Bereich zu erkunden. Während der Fahrt können mit Hilfe einer Infrarotkamera potentielle Opfer erkannt werden, die die Möglichkeit haben, dem mobilen Roboter Sprachnachrichten mit auf den Weg zu geben. Die Testplattform ist in ihrer Arbeit neben dem LIDAR mit einer Kinect-Kamera ausgestattet und in der Lage, 3D-Objekte bzw. Hindernisse zu erkennen und zu verarbeiten. Das Demonstrationsfahrzeug bei Team Hector ist wie in dieser Arbeit ein nicht-holonomer mobiler Roboter mit Ackermann-Lenkung. Seit dem Erscheinen dieses Paketes hat es sich im Rahmen der Robocup Rescue Challenge und darüber hinaus zum state-of-the-art der autonomen Erkundung entwickelt.

Das Paket von Kohlbrecher et al. kam im Zuge dieser Arbeit aus zwei Gründen nicht zum Einsatz. Erstens kommt im Hintergrund des Erkundungsalgorithmus ein von Team Hector selbst entwickeltes SLAM-Verfahren [KMKS11] zum Einsatz, das ausschließlich mit einer Scan-Matching-Technik arbeitet und die Daten der Positionsbestimmung, also der Odometrie, nicht betrücksichtigt. Das funktioniert mit einem LIDAR, wie bei der mobilen Plattform von Kohlbrecher verwendet, sehr gut, da durch das große Sichtfeld und der extremen Reichweite außreichende Überdeckung von Hindernissen gewährleistet ist. Das Modellfahrzeug in dieser Arbeit hat im Gegensatz dazu aus Kostengründen nur eine Kinect-Kamera als Lieferant für Tiefeninformation, die in einen Laserscan umgewandelt werden. Damit ergibt sich ein um ein Vielfaches kleineres Sichtfeld und eine verhältnismäßig geringe Sichtweite. Das führt zu den in Unterabschnitt 3.1.2 und Unterkapitel 4.2 beschriebenen Problemen mit der Positionsschätzung durch einen SLAM-Algorithmus. Somit funktioniert das proprietäre SLAM-Verfahren nicht und das Erkundungspaket kann nicht arbeiten, da das spezielle Kartenformat dieses SLAM-Paketes nicht vorliegt.

Der zweite Grund, ein eigenes Verfahren zu entwickeln war die Tatsache, dass Kohlbrecher et al., soweit ersichtlich, den Exploration-Transform-Algorithmus von Wirth et

al. verwendet haben, um den nächsten besten Schritt in der Erkundung zu ermitteln. Dieser stellte sich im Paper der Entwickler als sehr rechenaufwändig heraus. Es ist insgesamt wünschenswert, dass möglichst viele Pakete auf dem mobilen Roboter laufen und nur wenig über das Netzwerk ausgelagert werden muss.

Die Arbeit von Tovar et al. "Planning exploration strategies for simultaneous localization and mapping" [TGMC⁺06] beschäftigt sich ebenfalls mit einem grenzbasierten Ansatz. Darin wird auf die Lösung des Problems der optimalen Sensorpositionierung zur Maximierung des Informationsgewinns eingegangen. Dabei bedient man sich, wie in dieser Arbeit, bei den Überlegungen [GBL02] von González-Baños and Jean-Claude Latombe, die mit ihrer Next-Best-View-Technik einen guten Lösungsansatz liefern. Im Unterschied zu den meisten Veröffentlichungen, die sich mit Navigation von mobilen Plattformen beschäftigen, kommt bei Torvar et al. kein occupancy grid als Karte zum Einsatz, stattdessen wird dort eine feature-based map aufgebaut, die die Skalierungsprobleme einer Rasterkarte umgeht. Hierbei werden mit verschiedenen Verfahren Features aus den Daten des Laserscans extrahiert. Diese Features sind oft geometrische Primitive wie Dreiecke, Vierecke oder andere Polygone. In diesem Fall werden sogar nur Segmente, also Polygonzüge, verwendet. Außerdem werden noch sogenannte Landmarks benutzt, vorher definierte Objekte, beispielsweise ein Poster oder QR-Codes. Diese können erkannt und deren Position in der Karte vermerkt werden. Jedes mal, wenn der Roboter eine Landmark erkennt oder passiert, kann mit Hilfe der anderen bekannten Landmarks, durch eine least-squares-Methode oder ähnlichem, die Position des Roboters korrigiert werden. So ein Typ Karte wäre sicherlich interessant gewesen, da aber ROS als Framework gegeben war und dort so gut wie ausschließlich rasterbasierte Karten verwendet werden, hat man sich gegen eine feature-based map entschieden.

Zur genauen Umsetzung der NBV-Technik äußern sich Torvar und sein Team nur recht knapp, mit Hilfe eines Pseudocodes wird grob der Ablauf des Algorithmus gezeigt. Die Besonderheit bei dieser Implementierung ist, dass die Reihenfolge der anzufahrenden Frontiers nach jeder Iteration optimiert wird, um die Erkundungsfahrt so kurz wie möglich zu halten.

Getestet wurde die entwickelte Software auf einem nicht-holonomen differentialgetriebenen Roboter mit Stützrad, der mit einem LIDAR ausgestattet ist, sich selbstständig durch den Flur eines Gebäudes bewegt und dabei eine Karte aufbaut. Die im Paper abgebildeten Ergebnisse sind sehr gut, mit kurzen Erkundungswegen und nur wenigen Drehungen werden komplett Karten einer kleinen Büroetage geliefert. Allerdings benötigen die Berechnungen zur Wegoptimierung, nach Auskunft des Autors, mehrere Stunden, was gegen eine praxistaugliche Anwendung spricht.

Wettach und Berns vom Robotics Research Lab in Kaiserslautern haben in ihrer Arbeit "Dynamic Frontier Based Exploration with a Mobile Indoor Robot" [WB10] ebenfalls einen grenzbasierten Ansatz mit NBV-Technik zur Auswahl der besten Grenzen eingesetzt. Das Besondere hierbei ist neben dem eingesetzten Local Planner, der wie in dieser Arbeit ein Elastic-Band-Planer ist, auch der Typ der angelegten Karte. In diesem Fall wird eine topologische Karte verwendet, auf der Objekte und Hindernisse Knoten

darstellen und Distanzen dazwischen als Kanten modelliert sind, es wird folglich ein Graph aufgebaut. Getestet wird der Ansatz wie bei den meisten anderen Arbeiten auf einem nicht-holonomen Roboter mit Differentialantrieb und Stützrad, der mit LIDAR und zusätzlichen Ultraschallsensoren ausgestattet ist. Die Ultraschallsensoren dienen dem Zweck, Objekte die zu klein sind, um auf der Karte zu erscheinen, wahrzunehmen, um dann dynamisch ausweichen zu können, weil der globale Planer diese nicht berücksichtigt hat. Eine ähnliche Funktionalität bietet der Roboter, der im Rahmen dieser Thesis eingesetzt wird.

Auch Wettach und Berns Arbeit wurde als Grundlage nicht beachtet, da der Typ Karte mit ROS nicht vereinbar ist. Zusätzlich wird dort, wie bei Torvar et al., nicht auf die Problematik einer nicht-holonomen mobilen Plattform mit Ackerman-Lenkung eingegangen, was die Bewertung entdeckter Grenzen, wie in den Grundlagen erwähnt, zusätzlich erschwert.

Ungeachtet der teilweise beachtlichen Erfolge die in den verschiedenen Forschungsgruppen erzielt wurden, haben alle betrachteten Arbeiten, bis auf die Arbeit von Team Hector der TU-Darmstadt, das Fehlen von zur Verfügung stehenden Softwarebibliotheken gemeinsam. Es wird anhand der Testergebnisse gezeigt, dass der eingesetzte Algorithmus funktioniert, treibt aber allgemein nicht den Stand der Technik voran, da die meisten Probleme, die bei der Implementierung entstehen, immer wieder gelöst werden müssen. Damit fällt es auch schwer, Ergebnisse nachzuvollziehen, zu reproduzieren oder auf diesen aufzubauen.



6 Fazit

Ziel dieser Arbeit war es, ein Verfahren zur autonomen Erkundung eines abgeschlossenen Gebietes, basierend auf bereits etablierten Strategien, für einen mobilen Roboter zu entwickeln. Dazu mussten zuerst die geläufigsten Verfahren ermittelt werden. Bei der Recherche stellte sich heraus, dass die meisten gängigen Strategien einen frontier based approach verwenden, also versuchen, den bereits bekannten Raum im Verlauf der Erkundung immer weiter auszudehnen. Bei den beiden ausgewählten Strategien handelte es sich einerseits um den ersten entwickelten frontier based approach und andererseits um ein aufwändigeres Verfahren, dass in jedem Erkundungsschritt versucht, mit wenig Aufwand möglichst viel Information zu gewinnen.

Bei der Implementierung wurde zuerst ein Detektor entwickelt, der Frontiers auf einem occupancy grid erkennen und extrahieren kann. Zusätzlich wurde ein globaler Planer auf Basis des Path-Transform-Planners implementiert, der kollisionsfreie Trajektorien zu den erkannten Frontiers planen kann. In der Umsetzung der beiden Erkundungsstrategien wird unter anderem mit dem Ergebnis des Planers bewertet, ob sie erreichbar sind und wie nah sie sich am mobilen Roboter befinden. Die etablierten Strategien wurden so weiterentwickelt, dass Rücksicht auf die Eigenschaften der Ackermann-Lenkung genommen wird. Das wurde durch Einbeziehung des Unterschiedes zwischen Orientierung des Roboters und der Frontiers bewerkstelligt. Damit werden häufige Wendemanöver vermieden. Danach wurden die implementierten Verfahren zur autonomen Erkundung in ROS integriert.

In einer eigens für diese Arbeit entwickelten Simualtionsumgebung wurden beide Strategien in mehreren Szenarien getestet und gegeneinander evaluiert, um herauszufinden, welche die am besten geeignete ist. Als zusätzlichen Anhaltspunkt für ein gutes Verhalten bei der autonomen Erkundung wurden alle Szenarien von Versuchspersonen absolviert. Dabei stellte sich die aufwändigere Next-Best-View-Strategie als die geeignete heraus.

In Folge dessen wurde diese auf einem mobilen Roboter, der mit einer Kinect statt eines Laserscanners als Entfernungssensor ausgestattet ist, erfolgreich getestet.

Bei der Simulation, die für die Evaluation und Diskussion durchgeführt wurde, hat sich bereits herausgestellt, dass noch Verbesserungsbedarf beim Erkundungsalgorithmus besteht.

Die Funktion zur Gewichtung der Frontiers ist noch nicht ideal und führt dazu, dass von einander getrennte Räume nicht vollständig erkundet werden, bevor der nächste Raum angesteuert wird, oder in bestimmten Szenarios ein Pendeln zwischen Zielpunkten einsetzt. Um die Gewichtungsfunktion zu verbessern müssen noch zusätzliche Kriteria gefunden werden, was eine gute Frontier ausmacht.

Weiterhin wäre es sinnvoll, Frontiers nicht in jeder Iteration neu zu entdecken und alte Informationen zu verwerfen. Vielmehr müssten neue Informationen nach jeder Iteration in alte integriert werden. Damit könnte man Frontiers während der Laufzeit verfolgen und möglicherweise die ideale Abfahrreihenfolge ermitteln, um den gefahrenen Weg weiter zu minimieren.

Zusätzlich wäre eine eigene Umsetzung eines SLAM-Algorithmus auf Basis von OpenSLAM empfehlenswert. Das gmapping Paket erfüllt den Zweck des Kartieren zwar gut, aber bietet leider keine Schnittstellen für Feedback an. Denn oft wäre es gut zu wissen, wann neue Sensorinformationen in die Karte integriert wurden und es sich lohnt, einen neuen nächsten Erkundungsschritt zu planen. In dieser Arbeit wurde das Problem gelöst, indem in jeder Iteration des Algorithmus geprüft wurde, wie sehr sich die Karte im Vergleich zur letzten Iteration geändert hat. Dieser Workaround benötigt damit zusätzliche Ressourcen und wäre mit einer Schnittstelle zum SLAM-Algorithmus, die Feedback bietet, obsolet.

Beim Erkundungsversuch in der Realität sind einige Probleme aufgetreten, die weniger mit dem Algorithmus zu tun hatten, sondern viel mehr mit den eingehenden Daten der Sensorik. Es ist aufgefallen, dass Glastüren keine Hindernisse aus Sicht der Kinect darstellen. Da solche Türen in Büroumgebungen häufiger anzutreffen sind, wäre eine Möglichkeit zur Erkennung von Glasoberflächen und die damit einhergehende Modifikation des resultierenden Tiefenbildes, eine gute Verbesserung. Ähnlich verhält es sich mit nach unten führenden Treppen. Diese sind für einen aus einem Tiefenbild extrahierten Laserscan ebenfalls kein Hindernis, was in der Realität dazu führen kann, dass der Roboter durch einen Sturz ernsthaften Schaden nimmt. Durch ein Erkennen solcher Situationen könnte dann der Laserscan modifiziert werden, um diese Gefahren als Hindernisse abzubilden.

Zuletzt wäre es sinnvoll, die Bedienung zu erleichtern und damit das strenge Kriterium der Abgeschlossenheit des zu erkundenen Zielgebiets zu lockern. Dafür wäre einerseits eine Laufzeitbegrenzung denkbar, nach der die Erkundung abgebrochen wird und das bis dahin erkundete Areal speichert. Andererseits bietet sich auch eine Entfernungsbegrenzung zum Startpunkt an, über die hinaus keine weiteren Zielpunkte mehr gesetzt werden können.

Sieht man von den verbesserungswürdigen Punkten ab, wurde das Ziel dieser Arbeit erfüllt. Am Ende ist ein in ROS integrierter Erkundungsalgorithmus entstanden, der gezeigt hat, dass er in der Simulation effizient arbeitet und unter realen Bedingungen funktionieren kann. Nebenbei ist eine Simulation entwickelt worden, die nach dieser Arbeit von den Teilnehmern des Projektseminars Echtzeitsysteme dazu benutzt werden kann, ihre eigenen Projekte zu entwickeln und zu testen.

Literatur

- [BK08] BRADSKI, Gary ; KAEHLER, Adrian: *Learning OpenCV*. O'Reilly, 2008
- [Bov] BOVBEL, Paul: *pointcloud to laserscan*. http://wiki.ros.org/pointcloud_to_laserscan, Abruf: 23.08.2016
- [Can86] CANNY, John: A Computational Approach to Edge Detection. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (1986)
- [EA16] EHMES, Sebastian ; ACERO, Nicolas: *Projektseminar Echtzeitsysteme 2015/16 - Arbeitsgruppe Apollo13*. https://github.com/Nsteel/Apollo_13, 2016
- [Ehm16a] EHMES, Sebastian: *ROS-Paket automap zur autonomen Erkundung auf einem Roboter mit Ackermann-Lenkung*. <https://github.com/arg0n1s/automap>, 2016
- [Ehm16b] EHMES, Sebastian: *ROS-Paket simulation zur einfachen Simulation eines Roboters mit Ackerman-Lenkung*. <https://github.com/arg0n1s/simulation>, 2016
- [GBL02] GONZÁLEZ-BAÑOS, Héctor H. ; LATOMBE, Jean-Claude: Navigation Strategies for Exploring Indoor Environments. In: *The International Journal of Robotics Research* (2002)
- [Ger07] GERKEY, Brian P.: *slam gmapping: gmapping*. <http://wiki.ros.org/gmapping>. Version: 2007, Abruf: 23.08.2016
- [JB86] JARVIS, R. A. ; BYRNE, J.C.: Robot navigation: Touching, seeing an knowing. In: *Australian Conf. on Artificial Intelligence*, 1986
- [KMG⁺13] KOHLBRECHER, Stefan ; MEYER, Johannes ; GRABER, Thorsten ; PETERSEN, Karen ; KLINGAUF, Uwe ; STRYK, Oskar von: Hector Open Source Modules for Autonomous Mapping and Navigation with Rescue Robots / TU Darmstadt. 2013. – Forschungsbericht
- [KMKS11] KOHLBRECHER, Stefan ; MEYER, Johannes ; KLINGAUF, Uwe ; STRYK, Oskar von: A Flexible and Scalable SLAM System with Full 3D Motion Estimation. In: *IEEE International Symposium on Safety, Security and Rescue Robotics*, 2011
- [Lid11] LIDORIS, Georgeos: *State Estimation, Planning and Behavior Selection Under Uncertainty for Autonomous Robotic Exploration in Dynamic Environments*. Kassel : kassel university press GmbH, 2011
- [Ope] OPENKINECT: *OpenKinect - libfreenect*. https://openkinect.org/wiki/Main_Page, Abruf: 23.08.2016

-
- [Ope07] OPENSLAM: *OpenSlam - GMapping*. <http://openslam.org/gmapping.html>. Version: 2007, Abruf: 23.08.2016
- [Ope16a] OPENCV: *cv::Mat Class Reference*. http://docs.opencv.org/master/d3/d63/classcv_1_1Mat.html. Version: 2016, Abruf: 10.08.2016
- [Ope16b] OPENCV: *OpenCV - cv::LineIterator Class Reference*. http://docs.opencv.org/master/dc/dd2/classcv_1_1LineIterator.html. Version: 2016, Abruf: 10.08.2016
- [Ope16c] OPENCV: *OpenCV - image processing*. http://docs.opencv.org/master/d7/dbd/group__imgproc.html. Version: 2016, Abruf: 10.08.2016
- [Ope16d] OPENCV: *OpenCV - Introduction*. <http://docs.opencv.org/master/d1/dfb/intro.html>. Version: 2016, Abruf: 10.08.2016
- [RFW⁺12] RÖSMANN, C ; FEITEN, W ; WÖSCH, T ; HOFFMANN, F ; BERTRAM, T: Trajectory modification considering dynamic constraints of autonomous robots. In: *7th German Conference on Robotics*, 2012
- [RFW⁺13] RÖSMANN, C ; FEITEN, W ; WÖSCH, T ; HOFFMANN, F ; BERTRAM, T: Efficient trajectory optimization using a sparse model. In: *IEEE European Conference on Mobile Robots*, 2013
- [ROS14a] ROS: *ROS - Concepts*. <http://wiki.ros.org/ROS/Concepts>. Version: 2014, Abruf: 10.08.2016
- [ROS14b] ROS: *ROS - Introduction*. <http://wiki.ros.org/ROS/Introduction>. Version: 2014, Abruf: 10.08.2016
- [SA83] SUZUKI, Satoshi ; ABE, Keiichi: Topological structural analysis of digitized binary images by border following. In: *Computer Vision, Graphics and Image Processing* (1983)
- [SK08] SICILIANO, Bruno ; KHATIB, Oussama: *Handbook of Robotics*. Berlin and Heidelberg : Springer-Verlag, 2008
- [TGMC⁺06] TOVAR, Benjamín ; GÓMEZ, Lourdes M. ; MURRIETA-CID, Rafael ; ALENCASTRE-MIRANDA, Moisés ; MONROY, Raúl ; HUTCHINSON, Seth: Planning exploration strategies for simultaneous localization and mapping. In: *Robotics and Autonomous Systems* (2006)
- [WB10] WETTACH, Jens ; BERNS, Karsten: Dynamic Frontier Based Exploration with a Mobile Indoor Robot. In: *International Symposium on Robotics*, 2010
- [Wie15] WIEDEMAYER, Thiemo: *IAI Kinect2*. University Bremen : https://github.com/code-iai/iai_kinect2, 2014 – 2015. – Accessed June 12, 2015
- [WP07] WIRTH, Stephan ; PELLENZ, Johannes: Exploration Transform: A stable exploring algorithm for robots in rescue environments / University of Koblenz and Landau. 2007. – Forschungsbericht

-
-
- [Yam97] YAMAUCHI, Brian: A Frontier-Based Approach for Autonomous Exploration. In: *Computational Intelligence in Robotics and Automation*. Monterey, CA : IEEE, 1997, S. 146–151
- [Zel94] ZELINSKY, Alexander: Using path transforms to guide the search for find-path in 2D. In: *The International Journal of Robotics Research* (1994)