Institut für Parallele und Verteilte Systeme

Abteilung Bildverstehen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3640

Erweiterte autonome Exploration in unbekannten Umgebungen

Michael Eberspächer

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. habil. Paul Levi
Betreuer:	Dr. rer. nat. Oliver Zweigle
begonnen am:	01.10.2013
beendet am:	02.04.2014
CR-Klassifikation:	I.1.2, I.2.8, I.2.9, I.4.3

Abstract

Navigation ist ein Begriff, der heutzutage in verschiedenen Bereichen der Robotik und der autonomen Fahrzeuge weit verbreitet ist und ständig an Bedeutung gewinnt. Logistik auf der ganzen Welt ist von einer fehlerfreien und effektiven Navigation abhängig, um Kosten – monetär und zeitlich – so gering wie möglich zu halten. Fasst man den Begriff der Navigation nun aber weiter und nimmt zusätzlich die Aspekte Selbstlokalisierung und Exploration hinzu, so entsteht ein breites Spektrum an Möglichkeiten, welche über die bloße Navigation weit hinausreichen und einen Agenten in vielerlei Hinsicht unabhängig vom Eingreifen des Menschen autonom die Welt entdecken lässt. Zwar gibt es auf diesem Gebiet schon diverse Ansätze, dennoch soll mit vorliegender Arbeit ein robustes Framework entstehen, welches in die Thematik der *frontiers* einführt. Ergänzend zu vielen anderen Arbeiten auf diesem Gebiet kommt ein Filtermechanismus zum Einsatz. Außerdem werden die extrahierten *frontiers* qualitativ bewertet und eine nutzenbasierte Zielsuche vollzogen.

Inhaltsverzeichnis

1. Einleitung

Exploration unbekannter Gebiete stellt eines der grundlegenden Probleme in der mobilen Robotik dar. Weit verbreitetet und viel diskutiert auf diesem Gebiet ist der grenzzellenbasierte Ansatz von Yamauchi [YAM97, YAM98]. Yamauchi gilt als Vorreiter der autonomen Exploration und sein Konzept spiegelt sich in vielen Arbeiten wieder. Basierend auf seiner Idee entstand in den letzten Jahren eine Vielzahl an Lösungen für konkretisierte Probleme. Stochastische [FO05], auf Leistung optimierte [KK12] oder topologiebasierte Erweiterungen [KLS+05] seien eine Auswahl davon.
Anhand von Abb 1.1 lässt sich die Komplexität dieses Problems und die Verknüpfung der verschiedenen Teilaspekte erkennen.
Während der Schwerpunkt der Navigation in einer effizienten Pfadplanung zu suchen ist, so findet sich der Schwerpunkt der Exploration in der Berechnung eines nächsten Zieles, welches bezogen auf die Aufgabenstellung den Informationsgewinn maximiert. So macht es zum Beispiel einen Unterschied im Sinne einer möglichst weiträumig aufgedeckten Karte zu explorieren oder das Finden und Markieren radioaktiver Gebiete als Aufgabenstellung vorzugeben. Hierbei wird im ersten Fall zum Beispiel ein Lasersensor benötig und das Fahren Richtung u-Space gewinnbringend sein, wohingegen im zweiten Fall erweiterte Sensorik von Nöten ist und Gebiete erhöhter Strahlung angefahren werden. Leider gilt es festzustellen, dass ein optimaler Pfad mit maximal gewinnbringenden Zielen ähnlich dem NP-harten *Traveling Salesman Problem* ist. Einige viel versprechende Ansätze nehmen daher Abstand von einem einzelnen Roboter und ersetzen diesen durch eine Vielzahl von Robotern. Grundlegende Vorteile hierbei sind unter anderem eine erhöhte Ausfalltoleranz einzelner Systeme, die Möglichkeit entfernte Kartenabschnitte gleichzeitig erkunden zu können und eine dadurch verbundene erhöhte Explorationsgeschwindigkeit. Nachteilig sind jedoch der Bedarf einer Koordinationsstrategie, komplexe Kommunikation und das Verwalten einer verteilt erstellten Karte. Beispielhaft sei hier der Bedarf nach einer Strategie erwähnt, welche gewährleistet, dass die verwendeten Roboter unterschiedliche Zielpunkte erkunden [BMF+00].

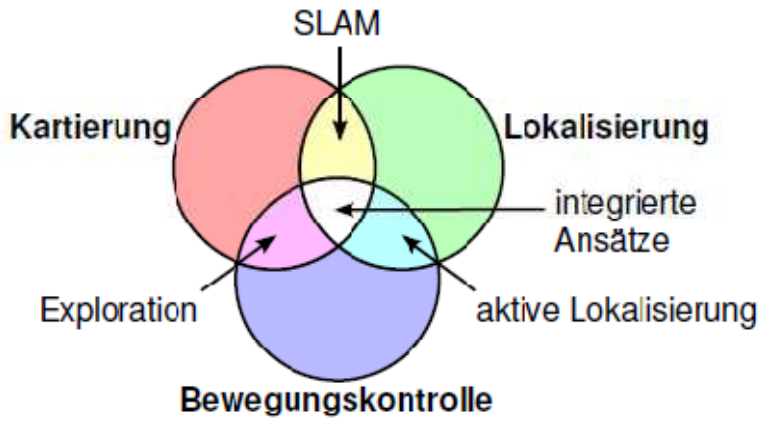


Abb 1.1: Teilaspekte der mobilen Robotik [JOH07]

Anlass zu weiteren Untersuchungen liefert die Frage nach einer geeigneten Repräsentation der zu erstellenden Karte. Viele Ansätze verwenden eine zellbasierte Darstellung und speichern relevante Informationen direkt in der betroffenen Zelle ab.

Neben einigen anderen Varianten bietet das ROS-Framework eine ideale Arbeitsstütze, um die Aspekte der Navigation und der damit verbundenen Sensorik ohne großen Aufwand in die eigenen Projekte einbinden zu können. Ausgehend von diesem Framework besteht nun die Möglichkeit bereits existierende Stacks weiter zu optimieren, oder eigene Stacks darauf aufbauend zu entwerfen, um spezifische Aufgaben zu lösen. Sowohl die Lokalisierung, die Pfadplanung als auch die Aggregation verschiedener Sensordaten bereits zur Verfügung stellend, fehlt dem ROS-Framework bis dato aber eine Abstraktionsschicht, welche sich den vorhandenen Funktionalitäten bedient, und einen Stack bildet, welcher es dem Agenten ermöglicht, in unbekannten Gebieten selbständig und möglichst effizient eben diese zu erkunden.
Basierend auf den Stacks *slam*, *gmapping* und *sb_navigation* werden *frontiers* eingeführt, um über diese möglichst interessante und informative Punkte auf der bereits explorierten Karte zu finden. Neben der Erläuterung theoretischer Hintergründe und einer Reihe von Abwägungen unterschiedlicher Ansätze liegt das Hauptaugenmerk auf dem Finden von *frontiers* und deren qualitativen Bewertung.
Da der Anspruch in Echtzeit zu arbeiten besteht, werden sämtliche Komponenten auf deren Laufzeitverhalten untersucht und gegebenenfalls so lange optimiert, bis ein nachweislich zufriedenstellendes Ergebnis zu Tage tritt.

Von den theoretischen Grundlagen über eine simulationsgestützte Entwicklung bis hin zu Tests in realer Umgebung deckt diese Arbeit den vollständigen Entwicklungszyklus ab und stellt zu guter letzt den Stack *frontier_navigation* zur Verfügung.

Für dieses Projekt wurde die Distribution ROS FUERTE TURTLE [ROS] unter anderem mit den Paketen gmapping und sb_navigation verwendet. Für die Simulation kamen die Pakete stageros, rxgraph und rviz zum Einsatz.

1.1. Aufgabenbeschreibung

Die Aufgabe dieser Diplomarbeit besteht aus dem Entwurf und der Implementierung eines Algorithmus zur autonomen Exploration eines autonomen Roboters in unbekannten Umgebungen. Bestehende Ansätze in diesem Bereich sind meist in Bezug auf Wegplanung und Laufzeit nicht optimal. Ziel dieser Arbeit soll es sein unabhängig vom eingesetzten Lokalisierungssystem und dem Hardwareaufbau des Roboters ein allgemeines Framework zur autonomen Exploration zu entwerfen. Dabei ist besonders eine Optimierung notwendig, die doppelte Wege sowie sogenannte „Stuck“ Situationen ausschließt und gleichzeitig die Zykluszeiten eines typischen Robotersystems (30ms) einhält.
Dazu sind in dieser Arbeit zunächst vorhandene Ansätze zu untersuchen. Daraufhin soll basierend auf ROS eine Komponente entwickelt werden, die sich dynamisch in das Framework einfügt und den Roboter selbständig optimiert nach bestimmten Vorgaben explorieren lässt.

1.2. Aufbau und Struktur

Kapitel 2 stellt bereits geleistete Arbeiten auf dem Gebiet der autonomen Exploration vor und gibt in Kürze einen Einblick in die für die Arbeit relevante Terminologie. In Kapitel 3 wird anhand von [YAM97, FO05, KK12] thematisch in *frontiers* und *frontierRegions* eingeführt. Es wird gezeigt wie diese berechnet werden und wie anhand eines Filtermechanismus die zu erwartenden Ergebnisse verbessert werden können. Kapitel 4 setzt sich basierend auf den extrahierten *frontierRegions* mit der für die Exploration wichtigen Zielsuche auseinander, legt den vollständigen Explorationszyklus dar und stellt fallback Strategien vor. Eine Ausarbeitung der experimentellen Ergebnisse findet in Kapitel 5 statt. Kapitel 6 fasst die Arbeit zusammen und Kapitel 7 bietet einen Ausblick in mögliche Anpassungen sowie Erweiterungen. Desweiteren wird anhand aktueller Diskussionen und Umsetzungen zum Thema der autonomen Exploration die Verwendbarkeit veranschaulicht.

Der Autor zeigt nach und nach wiederholt die Gedanken und theoretischen Grundlagen eines Ansatzes auf, versucht diesen zu validieren, erörtert die Vor- und Nachteile und entwickelt daraus entweder einen besseren Ansatz oder fährt mit einer weiteren Problematik fort.

2. Grundlagen

Im Folgenden wird ein ausführlicher Exkurs über die aktuell verfügbaren Technologien stattfinden und diese anhand von ausgewählten Veröffentlichungen in Kürze beschrieben. Desweiteren wird die für das Verständnis dieser Arbeit wichtige Terminologie erörtert und bestehende Einschränkungen und Rahmenbedingungen skizziert.

2.1. State of the Art

Das Reportoire bereits vorhandener Lektüre und Technik zu dieser Thematik ist bereits sehr erschöpfend. Einige für diese Arbeit grundlegenden Artikel werden im Verlauf dieses Kapitels vorgestellt. Es wird versucht die Unterschiede zur vorliegenden Arbeit an ausgewählten Stellen herauszuarbeiten, um den Beitrag dieser Arbeit abschätzen zu können.

Yamauchi [YAM97] führt die Begrifflichket der *frontiers* beziehungsweise *frontierRegions* ein und definiert diese auf einem zellbasiertem Gitter. Die einzelnen Zellen werden hierfür in die drei Gruppen frei – belegt – unbekannt unterteilt. Die Einteilung in die jeweilige Gruppe ist Abhängig von der Belegungswahrscheinlichkeit der jeweiligen Zelle und entsprechend definierten Grenzwerten. Viele der im Weiteren vorgestellten Techniken verwenden diesen Ansatz in mehr oder weniger abgeänderter Form. Yamauchi extrahiert hierfür im ersten Schritt *frontiers* (explorierte Zellen, die an unbekanntes Gebiet grenzen) und fasst zusammenhängende *frontiers* mittels Kantendetektion zu *frontierRegions* zusammen. **Die am nächsten gelegene *frontierRegion*** wird als nächstes Ziel definiert und unterstützt durch ein kollisionsvermeidendes Verfahren angefahren. Beim Erreichen wird ein 360° Scan vollzogen, die *frontierRegion* als erledigt markiert und die nächste Iteration eingeleitet. Unter anderem in einem Flur mit angrenzendem Büro wurde dieses Verfahren bereits erfolgreich getestet.
Aufbauend darauf stellt Yamauchi in einer weiteren Publikation [YAM98] ein Verfahren vor, welches es in den Grundzügen ermöglicht mit mehreren Robotern gleichzeitig zu explorieren. Er erweitert seinen ersten Ansatz soweit, dass jeder Roboter ein *localGrid* erstellt, sobald eine *frontierRegion* erreicht ist. Dieses *localGrid* wird in das robotereigene *globalGrid* integriert und an die anderen Teilnehmer versendet. Da das Hauptaugenmerk auf der Erprobung einer dezentralisierten Strategie mit minimalen Kommunikationskosten lag, gibt es hier keine weiteren Absprachen unter den teilnehmenden Robotern. **So kann es zu dem Fall kommen, dass sie sich gegenseitig behindern oder die gleiche *frontierRegion* als Ziel auswählen.** Unter anderem in einer Innenanwendung konnte dieses Verfahren bereits bestätigt werden.
In [BMF+00] wird diese Thematik weiter erörtert und ein Verfahren vorgestellt, mit Hilfe dessen die Exploration effektiv auf die Anzahl der verwendeten Roboter verteilt werden kann. Hierfür wird eine Kostenfunktion definiert, welche proportional abhängig zum Belegungswert einer Zelle deren Traversierungskosten ermittelt. Ein Pfadplaner kann nun ausgehend von den einzelnen Traversierungskosten einen optimalen Pfad von A nach B finden. $V_{x,y}^{i}$ beschreibt hierbei die Kosten für Roboter i um nach <x,y> zu gelangen. In einem weiteren Schritt wird versucht die Sichtweite der Roboter an den zuvor bereits berechneten *frontiers* grob abzuschätzen. $U_{x,y}$ beschreibt den daraus entstehenden Nutzen jeder *frontier* <x,y>. $V_{x,y}^{i}$ und $U_{x,y}$ werden nun gegeneinander abgewogen und die optimale Kombination Roboter – *frontier*

gesucht. Der Nutzen jeder Zelle im Sichtbereich der gewählten *frontier* wird reduziert und ein Ziel für den nächsten Roboter gesucht. Durch die Reduktion des Nutzens wird gewährleistet, dass zwei Roboter jeweils unterschiedliche Ziele anfahren. **Jedoch wird außer dem Sichtbereich keine weitere qualitative Auswertung von *frontiers* vorgenommen**. Experimente zeigen, dass mehrere koordinierte Roboter einen Vorteil gegenüber einem einzelnen Roboter bezogen auf die Explorationsgeschwindigkeit haben.

[MSW01] stellt die Frage, in welchem Kontext welche Informationsquellen von Bedeutung sind. Zum Beispiel benötigt ein explorierender Roboter, welcher nach Wasser in Eisform sucht, unter anderem Temperaturinformationen, um im Sinne seiner Aufgabe explorieren zu können. Die Autoren beschäftigen sich im Weiteren mit der konkreten Aufgabe Traversionskarten zu erstellen, die im späteren Verlauf unter anderem von verschiedenen Robotern verwendet werden können. Hauptaugenmerk liegt hierbei auf einer sicheren, zuverlässigen und effizienten Navigation durch das erfasste Gebiet. Um solch eine Traversionskarte zu erstellen schlagen die Autoren vor, jeder Zelle einen Attributvektor $A(a_r, a_h, a_t, a_{ch}, a_{ct})^T$ und einen Nutzenvektor $G(g_f, g_c, g_r)^T$ zuzuordnen. Der Attributvektor A beinhaltet somit Informationen über die Erreichbarkeit, Höhe und Traversierbarkeit, sowie über die Zuverlässigkeit der Höhe und Traversierbarkeit. Durch den Nutzenvektor G lässt sich eine Aussage über *frontiers* und die Anpassung von a_{ct} und a_r machen. In simulierter binärer (traversierbar/nicht traversierbar) Umgebung konnten erfolgreich Traversionskarten erstellt werden.

[KLS+05] untersucht im Gegenzug zu [MSW01] keine Traversions-, sondern topologiebasierte Karten. Das dafür verwendete Verfahren läuft in sechs Schritten ab: Scan – Glättung der Rohdaten – Extraktion von Kanten – Bestimmung von vier Raumwänden – Festlegung eines Raumes um den Roboter – Bestimmung von Öffnungen und Verdeckungen. Ein großer Vorteil dieses Verfahrens liegt darin, dass aufgrund der verwendeten Kartenrepräsentation Türen und Durchgänge denkbar einfach erfasst werden können. Nachteilig ist jedoch, dass die Repräsentation ausschließlich anhand von rechtwinkligen Räumen erfolgt. In einer simulierten Umgebung ohne Odometriefehler konnte dieses Verfahren **durch manuelles Navigieren** bereits erfolgreich angewendet werden.

Eine weitere Technik zur autonomen Exploration wird in [GKC03] vorgestellt. Im Gegenzug zu den meisten anderen Verfahren besteht die Aufgabe der Exploration hierbei nicht ausschließlich darin das Vorkommen von u-Space zu verringern, sondern zu großen Teilen darin, die Informationssicherheit entdeckter Hindernisse zu erhöhen. Begründet wird dies durch den Umstand, dass Fehler in der Odometrie zum Beispiel durch Objekterkennung entfernt werden können. Sind diese Objekte aber ungenau lokalisiert, so wird ein Fehler durch einen anderen ersetzt. Darauf aufbauend versuchen die Autoren einen nächsten Zielpunkt so zu wählen, dass zum einen u-Space aufgedeckt, zum anderen aber explizit die Qualität im Umkreis von Hindernissen erhöht wird. Um dies zu erreichen wird das *inverse sensor model* beschrieben und angewandt.

Ein zufallsbasierter Ansatz der Exploration wird in [FO05] vorgestellt. Hierfür wird die Datenstruktur *sensor-based random tree (srt)* vorgestellt. Jeder Knoten dieses Baumes enthält eine kollisionsfreie Konfiguration q und eine dazugehörige Beschreibung der *local save region S(q)*, wie sie durch die Sensorik erfasst wurde. Die Erweiterung des Baumes erfolgt durch eine zufällig ausgewählte Explorations-Richtung, so dass unter anderem $q^{next} \in S(q^{curr})$. Es werden die Varianten *SRT-Ball* und *SRT-Star* definiert. *SRT* unterscheidet in

den Grundzügen nicht zwischen f-/o-/u-Space und kann laut den Autoren in großen Umgebungen zu einer ineffektiven Exploration führen. Um diesem entgegenzutreten werden zusätzlich die auf *frontiers* basierenden Version *FB_SRT-Ball* und *FB_SRT-Star* vorgestellt, um die Exploration in Richtung unerschlossener Bereiche zu lenken. Diese werden im Hauptteil genauer betrachtet. Experimente zeigen, dass beide *FB_* Varianten gegenüber den rein zufälligen bei gleicher Anzahl Iterationen eine deutlich höhere Explorationsquote aufweisen.
[KK12] stellt zwei Varianten vor, die im Gegenzug zu vielen anderen Ansätzen **die Suche nach *frontiers* nicht global, sondern in einer lokalen Umgebung** um den Roboter herum durchführen. Ziel ist es die Rechenzeit deutlich zu reduzieren. *Wavefront Frontier Detection* sucht hierbei alle *frontierCells* auf der Karte, verwirft aber alle bereits besuchten Zellen und nutzt nur diejenigen, welche in in f-Space liegen. *Fast Frontier Detection* jedoch ist eine Variante, die von der globalen Karte völlig entkapselt funktioniert. Der grundlegende Gedanke besteht darin, die Suche nach *frontiers* nur in den neu erzeugten Sensordaten zu vollziehen. Es ist leicht einzusehen, dass dies deutlich schneller geschieht, als andere Varianten, die auf der ganzen Karte nach *frontiers* suchen. Hierfür werden die gelieferten Sensordaten zuerst nach deren Winkel sortiert um anschließend die äußersten Punkte pro Scanlinie miteinander zu verbinden. Entlang dieser entstandenen Contour werden nun *frontiers* extrahiert. Im Weiteren wird ein Verfahren vorgestellt, welches es ermöglicht ***frontiers* für spätere Zwecke zu speichern und bereits besuchte *frontiers* in folgenden Iterationen zu ignorieren.** Experimente zeigen, dass *FFD* deutlich schneller als *WFD* und *WFD* wiederum deutlich schneller als bis dato aktuelle andere *frontier* extrahierende Algorithmen arbeitet.
Weitere Strategien zur Exploration von Gebäuden werden in [GL02] vorgestellt. Diese Arbeit liefert einen Einblick in die Thematik und beschreibt unter anderem detailliert, wie extrahierte Kartenmodelle aggregiert werden. Für die Exploration werden desweiteren beispielsweise *free-curves* verwendet und es wird die Möglichkeit diskutiert den nächsten Zielpunkt zufällig auszuwählen.
[LMB+02] trägt das Explorationsproblem vom Inneren eines Gebäudes in die Außenwelt und entwickelt im Gegenzug zu den weiter oben beschriebenen Techniken eine Lösung, die auf visueller Erfassung der Umgebung basiert. Es wird anhand von Bildern eine digitale Höhenkarte erstellt und diese für Traversierungszwecke verwendet. Außerdem wird die Selbstlokalisierung anhand von markanten Punkten in der Landschaft, sowie anhand von Panoramabildern erörtert. Der erhöhten Anforderung in der Außenanwendung zum Trotz konnte experimentell die Verwendbarkeit der vorgestellten Techniken überprüft und bestätigt werden. Unter anderem gelang es dem verwendeten Roboter LAMA einen zwei Meter hohen Hügel zu erklimmen.
Ein bereits praktisch umgesetztes und erfolgreich erprobtes Verfahren findet sich in [TTW+04]. Kernaspekt dieser Arbeit ist es, einen Roboter zu entwickeln, welcher die Fähigkiet besitzt unter Tage gelegene Minen autonom zu erkunden. Der verwendete Roboter Groundhog ist symmetrisch aufgebaut und kann somit in beide Richtungen explorieren ohne wenden zu müssen. Das vorgestellte Verfahren verwendet wie viele anderen eine Kombination aus Laserscans uns SLAM, um eine Umgebungskarte zu erstellen. Nächste Zielpunkte werden hierbei aber nicht durch das Auswerten von *frontierCells* bestimmt. Vielmehr wird in einer Enfernung von fünf Metern vor dem Roboter eine *goalArea* definiert

und diese solange iterativ vergrößert, bis ein A* einen Pfad dorthin planen kann. Es wird im Gegenzug zu den meisten anderen Techniken zusätzlich eine Strategie entwickelt, **welche es dem Roboter ermöglicht zu explorieren, selbst wenn aktuell keine geeigneten Ziele gefunden werden**. Experimente in unterschiedlichen Minen lieferten positive Ergebnise, zeigten aber auch physikalische Grenzen auf. So sind zum Beispiel viele verlassene Minen überflutet und somit für Groundhog nicht explorierbar.
Eine weitere Dimension der autonomen Exploration liefert [SCJ+05]. Bereits vorgestellte und bekannte Verfahren werden in dieser Arbeit erweitert, um selbst einen fliegenden Roboter autonom navigieren zu lassen. Hierbei wird initial ein Pfad von der aktuellen Position bis hin zum Ziel berechnet. Ähnlich wie in weiter oben beschriebenen Verfahren kommt spezifische Sensorik zum Einsatz, welche ein lokales Modell der Karte erstellt. Basierend darauf passt ein *model predictive control* Algorithmus den Pfad mit jeder Iteration an und gewährleistet einen kollisionsfreien Flug. Schwerpunkt liegt hierbei auf der Passierbarkeit von Hindernissen. Kann sich ein bodenbetriebener Roboter eine Kollision mit einem Hinderniss leisten, so resultiert eine Kollision in der Luft in der Regel in einem Absturz. Desweiteren wird ein Verfahren vorgestellt, welches es ermöglicht Hindernisse zu ignorieren, welche für das Flugobjekt keine Gefahr bedeuten. Fallende Blätter seien ein Beispiel hierfür. In einem ungefähr 50m auf 50m großen mit Hindernissen bestückten Gebiet wurde dieses Verfahren bereits mit einem 3,5m langem Fluggerät erfolgreich getestet.

Zusammenfassend wurden bereits viele Verfahren entwickelt und getestet. Doch ist jedes dieser Verfahren einzig für die definierte Aufgabe zu gebrauchen. Ein Erkundungsroboter für das Innere von Gebäuden wird aufgrund der unterschiedlichen Gegebenheiten im offenen Gelände voraussichtlich wenig zufriedenstellende Ergebnisse liefern. Selbiges gilt vice versa. Desweiteren fanden bis dato keine großflächigen Tests statt.

Im Gegenzug zu den meisten vorgestellten Veröffentlichungen wird in dieser Arbeit detailliert die zugrunde liegende Algorithmik beschrieben und evaluiert, sowie ein effektiver Filtermechanismus entwickelt, um qualitativ hochwertigere *frontierRegions* extrahieren zu können. Desweiteren werden Ziele weder zufällig, noch nach dem in [YAM97] vorgestellten Prinzip gesucht, sondern in optimierter Form.

2.2. Terminologie

In den meisten Veröffentlichungen wird der Begriff *frontier* für den Grenzbereich zwischen u-Space und f-Space verwendet. Um während dieser Ausarbeitung besser unterscheiden zu können wird eine etwas detailliertere Begrifflichkeit verwendet. Mit *frontierCell* wird eine einzelne dieser Grenzzellen in indizierter Version beschrieben. *frontierPoint* beschreibt analog dazu einen Grenzpunkt in kartesischen Koordinaten. Ein Verbund zusammenhängender *frontierCells (frontierPoints)* bildet eine *frontierRegion.* Die Bezeichnung *frontier* wird in dieser Arbeit als Oberbegriff für vorangehende Bezeichnungen verwendet.
Desweiteren wird auf eine deutsche Übersetzung von Fachbegriffen, Algorithmen-bezeichnungen, Variablen und an andereren Stellen verzichtet.

2.3. Rahmenbedinungen

Die in dieser Arbeit beschriebenen Mechanismen beruhen stets auf der Annahme, dass sich der Roboter auf einer Ebene befindet und dass die verwendete Sensorik ein ausreichend genaues Abbild der Umgebung erfassen kann.

Desweiteren haben die auf den Abbildungen zu sehenden Zellgitter eine Zellgröße von 1m x 1m.

3. Verfahren zur *frontier* Detektion

Grenzzellen sind für eine autonome Exploration von großem Interesse, da sich dahinter potentiell interessantes Gebiet verbirgt. Der erste Schritt in Richtung autonomer Exploration ist nun also das Finden von einfachen *frontierCells*. Darauf aufbauend werden in weiteren Schritten die Nachbarschaftsbeziehungen berechnet, um daraus wiederum Aussagen über Verbünde von *frontierCells* treffen zu können. Der Autor geht hierbei davon aus, dass ein Verbund von zusammenhängenden *frontierCells* einen höheren Informationsgehalt gewährleistet, als eine einzelne *frontierCell*.
In diesem Kapitel werden Verfahren vorgestellt, die es ermöglichen *frontierCells* aus einer zellbasierten Karte zu extrahieren. Weiterführend wird gezeigt, wie anhand der zuvor gefundenen *frontierCells* letztendlich *frontierRegions* berechnet werden können. Im weiteren Verlauf wird auf das Problem ungenauer Sensordaten eingegangen und versucht dieses anhand eines angepassten Filtermechanismus zu lösen. Jeder vorgestellte Ansatz wird auf seine Verwendbarkeit in einem Echtzeitsystem hin untersucht.

3.1. Definitionen

frontierCell:
Zelle, die in f-Space liegt und an u-Space angrenzt.

$$\forall i \in map \mid map(i) = F\_SPACE \wedge \exists j \in neighbours(i) \mid map(j) = U\_SPACE \Rightarrow i \in frontierCells \quad \text{eq i}$$

oder

$$\forall i \in map \mid map(i) = U\_SPACE \wedge \exists j \in neighbours(i) \mid map(j) = F\_SPACE \Rightarrow i \in frontierCells \quad \text{eq ii}$$

frontierRegion:
Verbund von *frontierCells*, die eine Zusammenhangskomponente anhand ihrer Nachbarschaftsrelationen bilden.

$$G \subseteq frontierCells = frontierRegion_k \Leftrightarrow \forall i \exists j : j \in neighbours(i) \mid i, j \in G \quad \text{eq iii}$$

3.2. Konvertierung von linear-Space nach cartesian-Space und vice versa

Die Konvertierung von indizierten Punkten zu kartesischen Koordinaten ist aufgrund der Flexibilität der Karte nicht trivial und soll im Folgenden in wenigen Sätzen vorab erörtert werden.

cellToPoint:

Herfür werden zuerst die entsprechende Reihe und Spalte im Gitter berechnet, danach die Auflösung und Verschiebung mit einbezogen und letztendlich der Mittelpunkt der Zelle berechnet.

$$col = index \bmod with \quad \text{eq iv}$$

$$row = \frac{index}{width} \quad \text{eq v}$$

$$x = col \cdot res + 0{,}5 \cdot res + x\_org = res \cdot (col + 0{,}5) + x\_org \quad \text{eq vi}$$

$$y = row \cdot res + 0{,}5 \cdot res + y\_org = res \cdot (row + 0{,}5) + y\_org \quad \text{eq vii}$$

pointToCell:

Um willkürliche Punkte verarbeiten zu können, muss zuerst das Zentrum der aktuellen Zelle berechnet werden. Dies geschieht, indem entlang beider Achsen minimiert und dann zentriert wird.

$$x\_center = res \cdot \left(floor\left(\frac{point.x}{res}\right) + 0{,}5\right) \quad \text{eq viii}$$

$$y\_center = res \cdot \left(floor\left(\frac{point.y}{res}\right) + 0{,}5\right) \quad \text{eq ix}$$

Ausgehend vom Zellzentrum findet sich die indizierte Version des Punktes, indem eq iv in eq vi und eq v in eq vii eingesetzt und schließlich eq vi und eq vii nach index umgeformt und addiert werden.

$$index = \left(\frac{y\_center - y\_org}{res} - 0{,}5\right) \cdot width + \frac{x\_center - x\_org}{res} - 0{,}5 \quad \text{eq x}$$

3.3. Finden von *frontierCells* mit Hilfe von occupancy grids

Ein erster trivialer Ansatz besteht darin, über den gewünschten Suchbereich zu iterieren und für jede gefundene f-Space Zelle alle ihrer u-Space Nachbarn zu *frontierCells* hinzuzufügen. Nun gilt es aber festzustellen, dass hierbei duplizierte Einträge entstehen. Hat zum Beispiel die f-Space Zelle i den u-Space Nachbarn i+w, so hat auch die f-Space Zelle i+1 den u-Space Nachbarn i+w, wodurch die Zelle i+w zweimal zu *frontierCells* hinzugefügt wird. Dies ist wenig praktikabel, da entweder ein aufwendiges Entfernen von Duplikaten (eq xix**)** durchgeführt werden müsste, oder aber eine höhere Laufzeit in weiteren Verarbeitungsschritten in Kauf genommen wird.
Darauf aufbauend umgehen die Definitionen eq i und eq ii dieses Problem, da nur die aktuell betrachtete Zelle zu *frontierCells* hinzugefügt werden kann. Ein weiterer positiver Nebeneffekt entsteht durch die Feststellung, dass im Gegenzug zur ersten Variante nicht zwingend alle acht Nachbarn betrachtet werden müssen, sondern nur so viele, bis eine u-Space Zelle gefunden wird. In den meisten Fällen wird diese Verbesserung nicht von Belang sein, da die meisten f-Space Zellen von f-Space umgeben sein werden. Dennoch kann selbst eine Optimierung in kleinem Maßstab bezogen auf sehr viele Iterationen durchaus auf einen deutlichen Laufzeit Gewinn hinauslaufen. Letztendlich steht nun ein mit *frontierCells* gefüllter Vektor – folgend mit *frontierIdxs* bezeichnet – zur Verfügung. Aus diesem können nun wahllos Zellen für die Exploration herangezogen werden, doch haben diese einzelnen Zellen alle den gleichen Informationsgehalt (*frontierCell*) und tragen zu einer effektiven Exploration in ihrer Rohform nur beschränkt bei. Um den Informationsgehalt zu erhöhen ist unter anderem die gegenseitige Lage der *frontierCells* interessant. Hat eine *frontierCell* zum Beispiel keine Nachbarn ist sie deutlich uninteressanter, als eine *frontierCell* mit mehreren Nachbaren, da der zu erkundende u-Space dadurch größer und potentiell interessanter wird.

3.4. 3x3 Nachbarschaftsrelationen

Nachbarschaftsrelationen lassen sich bekanntermaßen auf einfache Art und Weise durch Adjazenzmatritzen ausdrücken. Eine herkömmliche Adjazenzmatrix hat hierbei n x n Einträge, wobei n die Anzahl der verwendeten Knoten (hier *frontierCells*) ist. Beachtet man aber den Umstand, dass eine Zelle nur acht Nachbarn haben kann und eine herkömmliche Adjazenzmatrix somit im Großen aus Nullen bestehen würde, bedarf es eines etwas anderen Datenformates um die Nachbarschaftsrelationen effizient speichern zu können. Ein geschachtelter Vektor bietet hierfür das optimale Werkzeug: $vector < vector < type > >$.
Es wird über *frontierIdxs* iteriert und im Selbigen nach Nachbarzellen gesucht. Dadurch entsteht der geschachtelte Vektor *adjacencyMatrixOfFrontiers*, dessen Größe nun nicht mehr durch $O(n^2)$, sondern durch $O(8n)$ abgeschätzt werden kann, und somit linear in Abhängigkeit von $|frontierIdxs|$ ist. Die Laufzeit des Vorgangs an sich ist durch $O(8n^2)$ beschränkt, kann durch eine einfache Annahme aber deutlich verbessert werden.

Annahme: *frontierIdxs* ist sortiert

Beweis: Erfolgt die Suche nach *frontierCells* in einem Strikt einzuhaltendem rechteckigen Muster, so ergibt sich *Code 3.1:*

```
startIndex = center − r_x − w × r_y
for i = 0 to 2r_y do
    for j = 0 to 2r_x do
        index = startIndex + j + i × w
        ...
```

Code 3.1: Finden von *frontierCells* in Rechteck

$$index_n = startIndex + j' + i' \times w$$
$$index_{n+1} = startIndex + (j' + 1) + i' \times w$$
$$\vee$$
$$index_n = startIndex + j' + i' \times w \mid j' = 2r_x$$
$$index_{n+1} = startIndex + 0 + (i' + 1) \times w$$

Zu zeigen ist: $index_n < index_{n+1}$

$$n = 0$$
$$startIndex < startIndex + 1$$

$$\cancel{startIndex} + j' + \cancel{i' \times w} < \cancel{startIndex} + (j' + 1) + \cancel{i' \times w}$$
$$j' < j' + 1$$

$$\cancel{startIndex} + j' + i' \times w < \cancel{startIndex} + 0 + (i' + 1) \times w \mid w = 2r_x + 1$$
$$2r_x + i' \times (2r_x + 1) < (i' + 1) \times (2r_x + 1)$$
$$\cancel{2r_x} + \cancel{2r_x i'} + \cancel{i'} < \cancel{2r_x i'} + \cancel{i'} + \cancel{2r_x} + 1$$
$$0 < 1$$

$q.e.d.$

Folgende Eigenschaften lassen sich daraus ableiten:

$$\exists i \mid frontierIdxs[i] = frontierIdxs[i + 1] - 1$$
$$\Rightarrow right(frontierIdxs[i]) = frontierIdxs[i + 1]$$

$$\exists i \mid frontierIdxs[i] = frontierIdxs[i - 1] + 1$$
$$\Rightarrow left(frontierIdxs[i]) = frontierIdxs[i - 1]$$

$$\forall j > i: frontierIdxs[j] \neq \{bottom(i);\ leftBottom(i);\ rightBottom(i)\}$$
$$\forall j < i: frontierIdxs[j] \neq \{top(i);\ leftTop(i);\ rightTop(i)\}$$

Daraus ergibt sich eine durch $O(n(n-1) \times 3 + 2n) = O(3n^2 - n)$ beschränkte Laufzeit.

3.5. Extraktion relevanter *frontierRegions*

Der durch *adjacencyMatrixOfFrontiers* vorliegende Datensatz hat bereits einen deutlich erhöhten Informationsgehalt als die bloßen *frontierCells* an sich, da über deren Nachbarschaftsbeziehungen Aussagen getroffen werden können. Noch mehr Information steckt aber hinter einer größeren Menge zusammenhängenden *frontierCells*. Ziel dieses Abschnitts ist es die gefundenen *frontierCells* in *frontierRegions* zu gruppieren und im optimalen Fall scharfe Kanten zwischen f-Space und u-Space zu extrahieren.
Mit Hilfe der Graphentheorie [RUO13] und einem auf dieses Problem zugeschnittenem weiter entwickeltem Algorithmus zur Detektion von Zusammenhangskomponenten steht nach einer rekursiven Suche ein geschachtelter Vektor *fronterRegions* zur Verfügung, welcher alle im Suchbereich entdeckten *frontierRegions* enthält.
Wie in Abb 3.2 zu sehen steht in der ersten Spalte von *adjacencyMatrixOfFrontiers* jeweils der Startknoten und in den folgenden Spalten die dazugehörigen Nachbarknoten. Nun wird nacheinander jeder Knoten n_i besucht. Ist n_x der aktuelle Knoten, so werden alle seine Nachbarknoten n_j und wiederum deren Nachbarknoten n_k besucht. Bereits besuchte Knoten werden markiert, wodurch ein Abbruchkriterium für die Rekursion entsteht. Mit jeder abgebrochenen Rekursion wird eine *frontierRegion* berechnet.

3.6. Datenmodelle

Abb 3.1 zeigt einen Ausschnitt aus einer Karte und deren Legende. Basierend auf der Karte wurde in den vorangehenden Kapiteln gezeigt, welche Datensätze auf welche Art zu extrahieren sind.

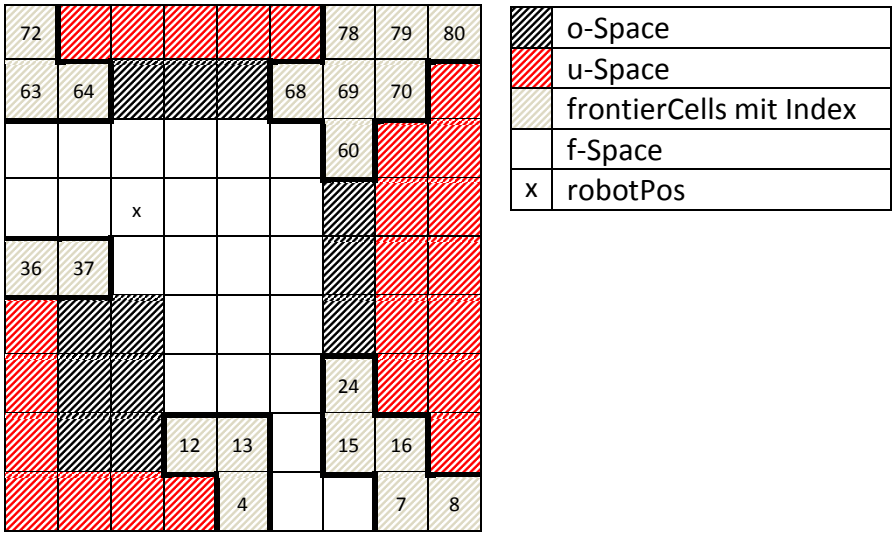


Abb 3.1: Schematische Darstellung eines o*ccupancy grid*

Abb 3.2 liefert einen Überblick über die verwendeten Datenstrukturen und zeigt zudem das Resultat der vorangehenden Algorithmen. Die dick gedruckten Ziffern dienen der Nummerierung innerhalt der Rekursion.

```
vector <unsigned int> frontierIdxs
vector <vector <unsigned int>> adjacencyMatrixOfFrontiers
vector <vector <unsigned int>> frontierRegions
```

frontierIdxs = {4, 7, 8, 12, 13, 15, 16, 24, 36, 37, 60, 63, 64, 68, 69, 70, 72, 78, 79, 80}
adjacencyMatrixOfFrontiers = {{4, 12, 13}, {7, 8, 15, 16}, {8, 7, 16}, {…}, {80, 70, 79}}
frontierRegions = {{**1**, 4, 12, 13}, {**2**, 7, 8, 15, 16, 24}, {**3**, 36, 37}, {**4**, 60, 68, 69, 70, 78, 79, 80}, {**5**, 63, 64, 72}}

Abb 3.2: Anfallende Datenstrukturen und Datensätze bei der Suche nach *frontierRegions*

3.7. Erste Laufzeitbetrachtung

Im Verlauf dieses Unterkapitels werden die einzelnen Schritte der *frontier* Detektion hinsichtlich Laufzeit und Speicherverbrauch untersucht. Das diskutierte Ergebnis wird als Vergleichsbasis für folgende Kapitel und Ansätze dienen.

Finden von *frontierCells*

Jede f-Space Zelle x innerhalb eines Radius r wird bezogen auf die Gleichgungen eq i oder eq ii untersucht. Daraus ergibt sich eine von r abhängige quadratische Laufzeit. Jedoch spielt die Häufigkeitsverteilung zwischen u-/f-/o-Space eine Rolle. So steigt die Laufzeit mit erhöhter Wahrscheinlichkeit von f-Space, da ausgehend von x bis zu acht Nachbarn besucht werden müssen.

$$O((1-p)\cdot d^2 + p\cdot 8\cdot d^2) \mid d = 2r \wedge \mathrm{p} = \mathrm{P}(\mathrm{f\_Space}) \qquad \text{eq xi}$$

Nachbarschaftsrelationen

Wie weiter oben schon beschrieben lässt sich der Speicherverbrauch für die entstehende Adjazenzmatrix unter der Bedingung, dass eine Zelle maximal acht Nachbarn haben kann, durch eq xii abschätzen. Desweiteren ist die Laufzeit durch eq xiii beschränkt.

$$O(8n), \qquad n = |frontierCells| \qquad \text{eq xii}$$

$$O(3n^2 - n), \qquad n = |frontierCells| \qquad \text{eq xiii}$$

Extraktion von *frontierRegions*

Während der Rekursion wird jeder Knoten genau einmal besucht. Gewährleistet wird dies durch ein entsprechend gesetztes Flag. Jeder Knoten besucht dessen Nachbarknoten, welche wiederum ihre Nachbarknoten besuchen, usw. Dadurch entsteht der Bedarf effizient in der Adjazenzmatrix A (nx8) nach Knoten suchen zu können. Finden sich die Knotennummern in der ersten Spalte, so kann im einfachsten Fall in $O(n)$ Schritten die gesuchte Knotennummer gefunden werden. Von maximal acht Nachbarknoten pro Knoten ausgehend ergibt sich somit eq xiv.

$$O(8n^2), \qquad n = |frontierCells| \qquad \text{eq xiv}$$

Es ist offensichtlich, dass das Suchen nach Knoten in A auf diese Weise wenig effizient ist. Besser ist es die Knoten zusätzlich in einer Hashmap zu speichern, um dadurch in $O(1)$ suchen zu können. Der zusätzliche Speicherverbrauch ist hierbei vernachlässigbar. Eq xv gibt die verbesserte Laufzeit an.

$$O(8n), \qquad n = |frontierCells| \qquad \text{eq xv}$$

Die Gleichungen eq xi, eq xiii und eq xv betrachtend entsteht für die Detektion von *frontierRegions* eine durch eq xvi beschränkte Laufzeit.

$$8d^2 + 3d^4 - d^2 + 8d^2 \qquad \text{eq xvi}$$

Bedingt durch den Umstand, dass für einen direkten Vergleich $n = d^2$ gesetzt wurde, gilt diese Laufzeit als sehr grob und nie erreichbar. Das Tatsächliche Verhältnis zwischen d^2 und n lässt sich durch Tabelle 3.1 erschließen. Demzufolge lässt sich der quartische Teilterm ignorieren und eine quadratisch beschränkte Laufzeit folgern.

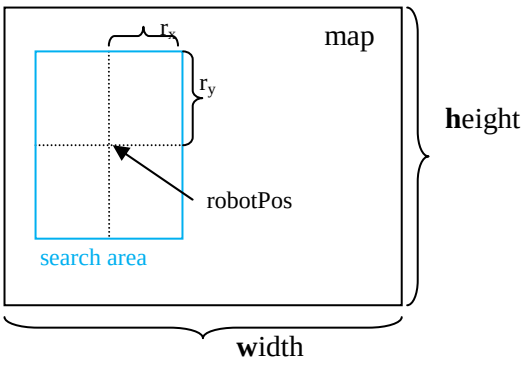


Abb 3.3: Schematische Darstellung einer Karte mit Suchgebiet

3.8. Sonnenstrahleffekt

Ein Nebeneffekt der Sensorik besteht darin, dass nicht mit beliebig genauer Auflösung abgetastet werden kann. In Abb 3.5 zu erkennen, wurde die Karte um den Roboter herum detailliert erfasst. Entfernt man sich aber vom Roboter, so stellt man fest, dass sich Rauschen entwickelt, welches bildlich ausgedrückt in Strahlen vom Zentrum wegläuft und immer dichter wird. Als Resultat daraus entsteht durch weiter oben beschriebenes Verfahren jeweils die obere Bildfolge von Abb 3.6 und Abb 3.7. Es ist leicht einzusehen, dass ein Großteil der hierbei entdeckten *frontierRegions* keine sinnvollen Navigationsziele produzieren kann, da es sich um verrauschte Daten handelt. Desweiteren ist die Anzahl der *frontierCells* in verrauschten Bereichen deutlich höher als in rauschfreien Bereichen, wodurch die Laufzeit dort zunimmt. [FO05] schlägt als Lösung hierfür vor einen kollisionsfreien Bereich in Form eines Balls oder eines Sterns um den Roboter herum zu finden. [KK12] wählt für dieses Problem einen anderen Ansatz und arbeitet direkt auf den neu erzeugten Sensordaten. Im Zuge dieser Arbeit soll ein Verfahren entstehen, welches die erfasste Karte innerhalb eines bestimmten Radius vorverarbeitet um Rauschen zu entfernen, beziehungsweise zu minimieren. Abb 3.4 zeigt ein mögliches Ergebnis dieser Vorverarbeitung. Abb 3.6 und Abb 3.7 zeigen in jeweils der unteren Bildfolge bessere extrahierte *frontierRegions*.

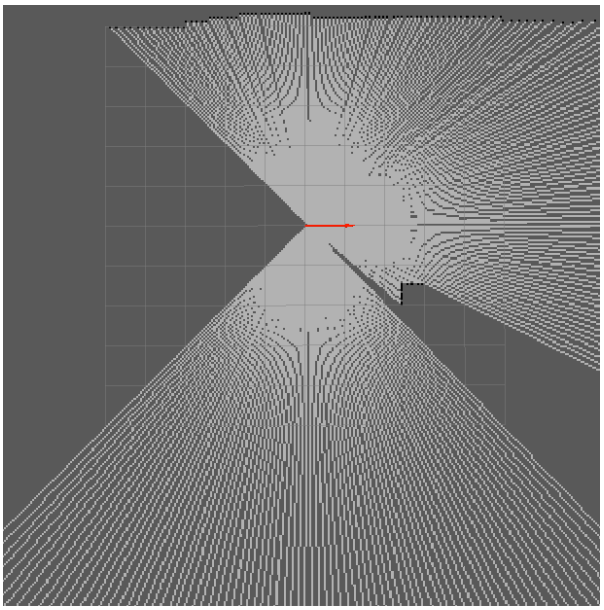


Abb 3.5: Ungefilterte Karte

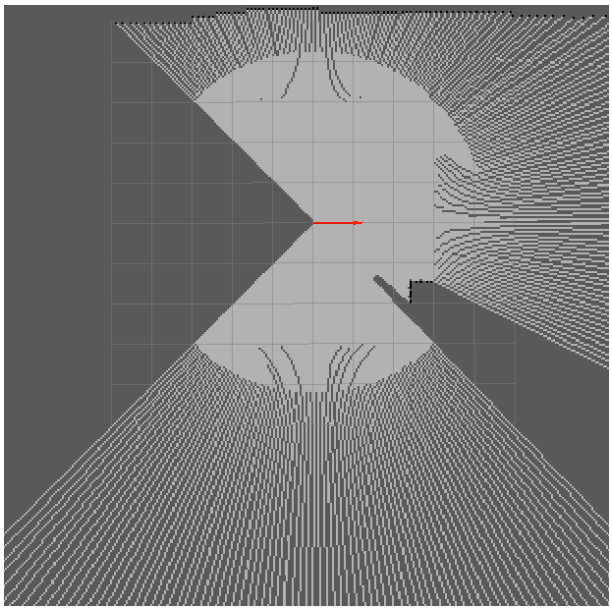
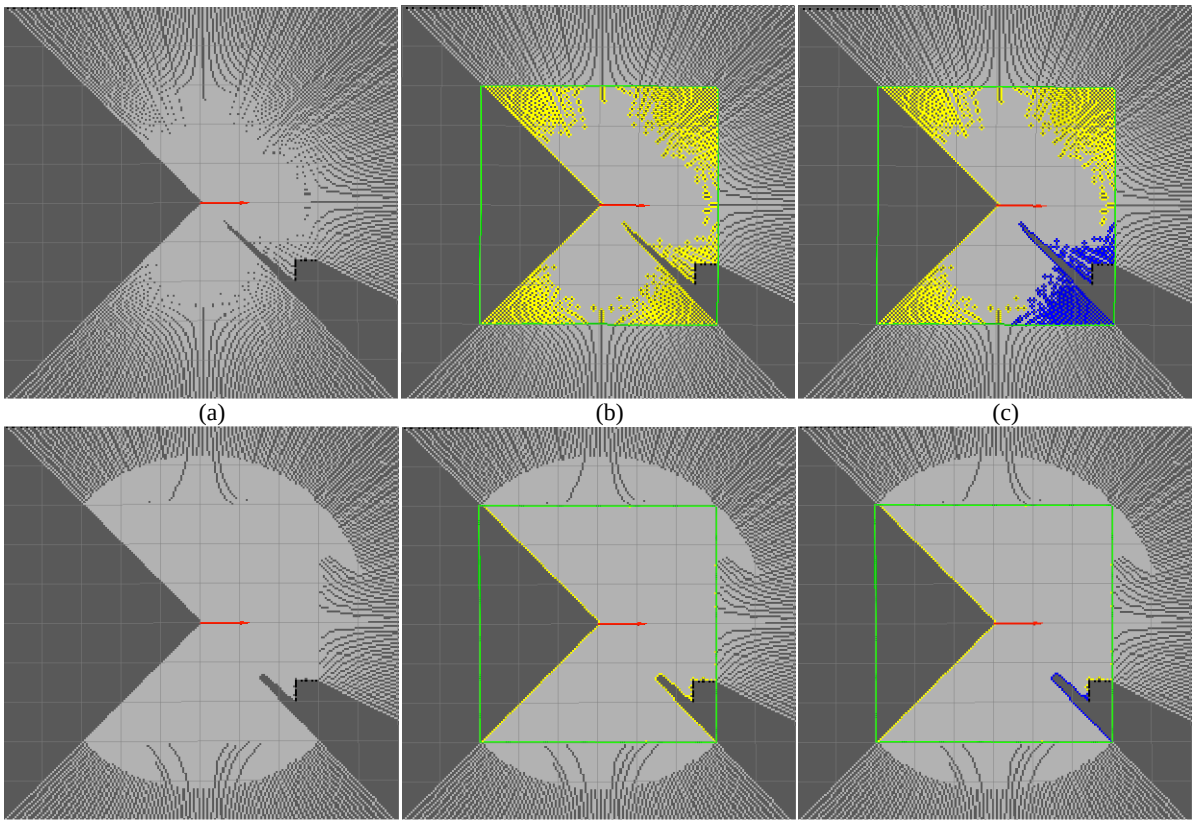


Abb 3.4: Gefilterte Karte

	ungefiltert		gefiltert	
	r=3 (14400 Zellen)	r=6 (57600 Zellen)	r=3	r=6
frontierCells	2790	14722	194	701
frontierRegions	18	13	14	7

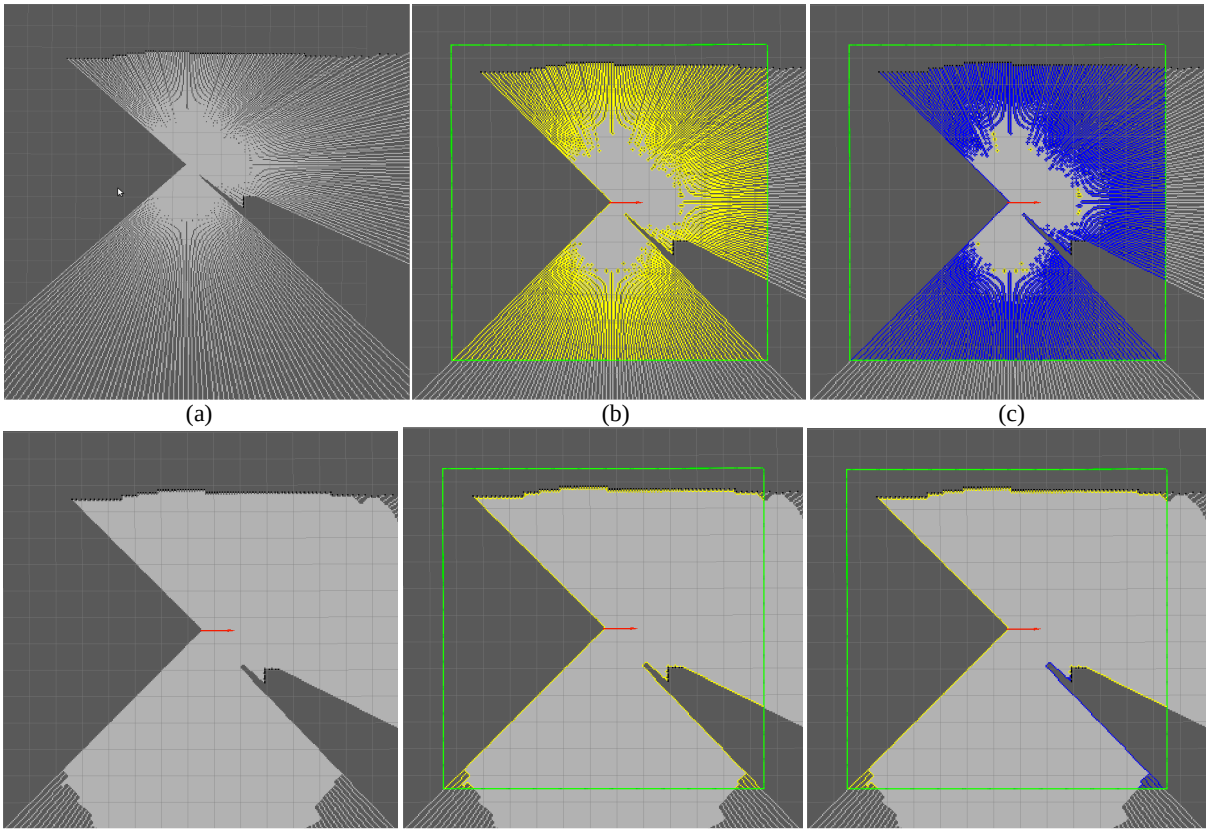
Tabelle 3.1: Vergleich ungefiltert/gefiltert

Tabelle 3.1 stellt Abb 3.6 und Abb 3.7 in Zahlen dar.



(a) (b) (c)

Abb 3.6: Suche nach *frontierRegions* ohne und mit Filter (r=3)
(a) (un)gefilterte Karte; (b) *frontierRegions* in gelb mit Suchradius; (c) zusätzlich qualitativ beste *frontierRegion* in blau.



(a) (b) (c)

Abb 3.7: Suche nach *frontierRegions* ohne und mit Filter (r=6)
(a) (un)gefilterte Karte; (b) *frontierRegions* in gelb mit Suchradius; (c) zusätzlich qualitativ beste *frontierRegion* in blau.

In beiden Abbildungen werden zuerst die ungefilterte und danach die gefilterte Version gezeigt.

3.9. Vorfiltern von Kartenausschnitten

Resultierend aus dem Sonnenstrahleffekt und den damit verbundenen schlechten Ergebnissen bei der Suche nach *frontierRegions* sowie der Problematik beim Navigieren, ist es empfehlenswert die Karte im verwendeten Suchradius durch einen Filter zu glätten. Hierunter ist unter zuvor definierten Umständen das Ersetzen von u-Space Zellen durch f-Space Zellen zu verstehen. Im folgenden Verlauf werden Fälle herausgearbeitet, bei denen ein Glätten sinnvoll ist. Anhand dieser Fälle wird eine Metrik definiert, die es algorithmisch möglich macht, diese Fälle zu bearbeiten. Darauf aufbauend wird ein Ansatz schrittweise an diese Problematik angepasst, verbessert und mit den Schritten zuvor verglichen. Letztendlich erfolgen eine ganzheitliche Betrachtung der Laufzeit und eine ausgewählte Sammlung von Simulations- und realen Daten.

3.9.1. Betrachtung der zu glättenden Fälle

Folgende Betrachtungen gehen nur von dem Fall aus, dass u-Space direkt an f-Space grenzt. Es wird also keine Aussage über das Einwirken von o-Space gemacht. Daraus ergeben sich für eine u-Space Zelle neun Möglichkeiten der Anordnung mit f-Space. Dies drückt sich durch die aufsummierten Nachbarschaftswerte aus und wird im weiteren Verlauf als Metrik zur Bestimmung von zu glättenden Zellen verwendet.

Nachbarschaftssume =

- 0:

Im einfachsten Fall ist eine einzelne u-Space Zelle von f-Space umgeben und stellt ein Analogon zum Rauschen in der Bildverarbeitung dar. Zu sehen in Abb 3.8.

- -1 und -2:

Desweiteren äußert sich die zellbasierte Struktur eines Sonnenstrahls in den einfacheren Fällen wie in Abb 3.9 zu sehen.

- -3 und -4:

Praxistest haben durchweg positive Resultate ergeben, solange die zu glättenden Zellen Nachbarschaftssummen im Intervall $I = [-3; 0]$ aufweisen. Erweitert man das Intervall auf $I = [-4; 0]$, so ist festzustellen, dass in vielen Fällen fälschlicherweise zu weit in den u-Space hinein gefiltert wird, wodurch an manchen Stellen blasenartige Auswüchse entstehen. Eine Veranschaulichung beider Intervallgrößen ist in Abb 3.12 zu sehen.

- -5; -6; -7; -8:

Fast vollständig von u-Space eingeschlossene u-Space Zellen befinden sich ohne Einschränkung in u-Space. Siehe Abb 3.11.

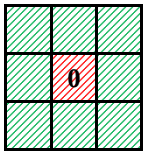


Abb 3.8: Nachbarschaftssumme = 0

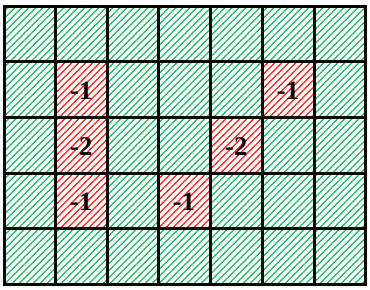


Abb 3.9: Nachbarschaftssumme = {-1; -2}

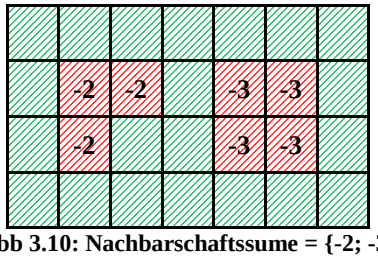


Abb 3.10: Nachbarschaftssume = {-2; -3}

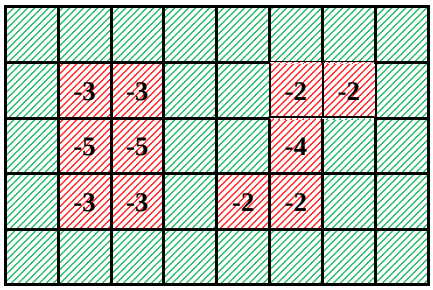
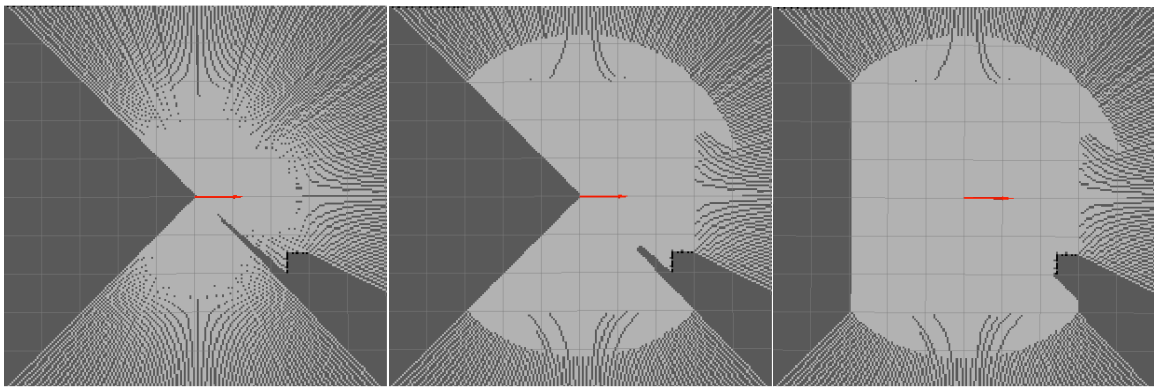
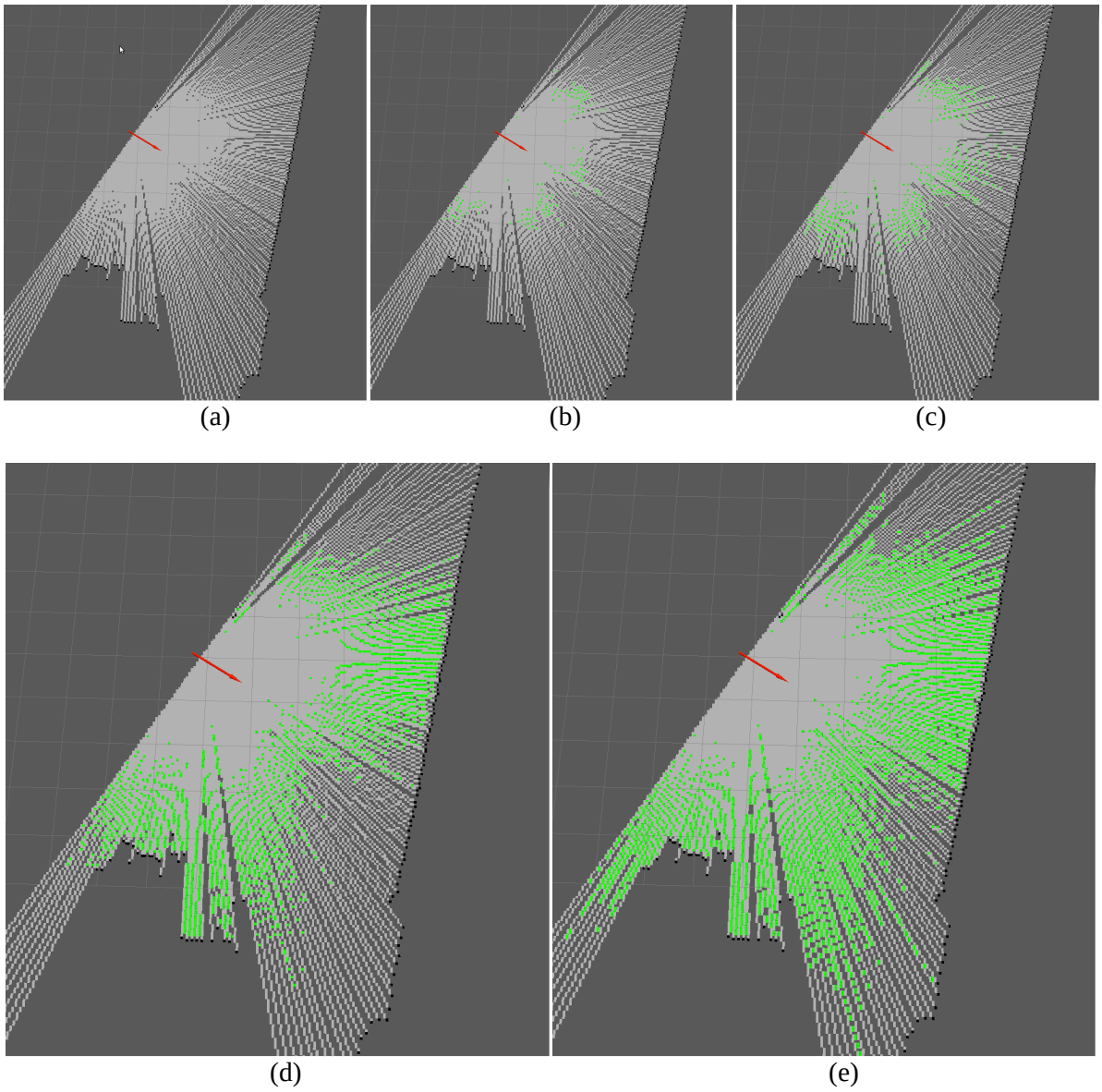


Abb 3.11: Nachbarschaftssumme = {-2; -3; -4; -5}



(a) (b) (c)

Abb 3.12: Gefilterte Karte mit verschiedenen NSIs
(a) ungefilterte Karte; (b) gefilterte Karte mit NSI = [-3;0]; (c) gefilterte Karte mit NSI = [-4;0]



(a) (b) (c)

(d) (e)

Abb 3.13: Karte mit Filterkandidaten anhand verschiedener NSIs

(a) ungefilterte Karte; (b) u-Space Zellen, deren Nachbarn alle in f-Space liegen; (c) zuzüglich alle u-Space Zellen, deren Nachbarschaftssumme = -1 ist; (d) hinzukommen alle u-Space Zellen, deren Nachbarschafts-summe = -2 ist; (e) für Verfahren als am besten eingestuftes NSI = [-3; 0].

3.9.2. Beschreibung des Verfahrens

Angelehnt an das Prinzip eines 3x3 Tiefpassfilters wird im Folgenden ein modifizierter Filter vorgestellt. Grundlegend hierbei ist, dass ein primitiver Tiefpassfilter über einen angegebenen Bereich iteriert, zu jeder Zelle deren Nachbarschaftswerte (gewichtet) zu einer Nachbarschaftssumme addiert, diese normiert und schließlich die Zelle im Zentrum entsprechend des Ergebnisses angleicht. Im vorliegenden Fall sind aber nur diejenigen Zellen von Bedeutung, welche in u-Space liegen und somit potentiell verrauscht sein könnten. Ein für dieses Problem angepasster Filter überspringt somit alle o-Space und f-Space Zellen und wertet ausschließlich u-Space Zellen aus. [KK12] Pro ignorierter Zelle werden dadurch acht Nachschlageoperationen, sieben Additionen, eine Auswertung der berechneten Summe und ein Schreibvorgang eingespart.
Wie in Zeile L3 zu sehen ist wird genau dies angewendet, indem nur diejenigen Zellen genauer betrachtet, die zum einen in u-Space liegen und zum anderen zusätzlich eine Nachbarschaftssumme innerhalb des Intervalls $I = [-3; 0]$ aufweisen

Anfänglich ist davon auszugehen, dass alle Zellen innerhalb des Suchradius potentielle Kandidaten für eine Glättung sind. (L1) Die *while*-Schleife vorerst einmal außen vor lassend wird nun über alle gesammelten Zellen iteriert. Wie in Zeile L4 zu sehen, werden die verrauschten Zellen geglättet, indem sie in den f-Space integriert werden. Bedingt durch diesen Vorgang ändern sich die Nachbarschaftssummen aller benachbarten Zellen. Dadurch entstehen wiederum weitere potentielle Glättungskandidaten, die durch die aktuelle Iteration des Filters aber übergangen werden. Genau diese werden nun für die nächste Iteration vorgesehen (*while* – L2) und bis dahin zwischengespeichert (*temp* – L5). Wie in Abb 3.14 zu sehen führt dies nun aber unweigerlich zu duplizierten Einträgen. Diese müssen vor der nächsten Iteration entfernt oder während der aktuellen Iteration ignoriert werden, um das Terminieren des Algorithmus zu gewährleisten.

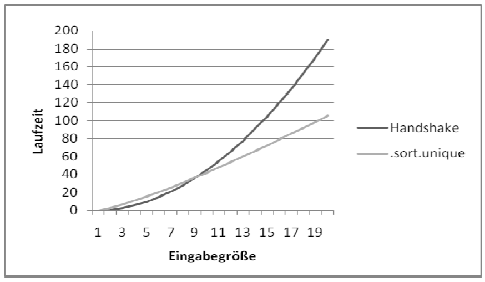


Diagramm 3.1: Handshake vs .sort.unique

Entfernen duplizierter Einträge nach Iteration

Hierfür lässt sich wenig performant ein handshake Algorithmus direkt auf der Liste der Indizes anwenden. Deutlich besser ist es jedoch die Indizes zuerst zu sortieren um danach die Liste linear nach Duplikaten untersuchen zu können.

Ignorieren duplizierter Einträge während Iteration

Um bereits vorgemerkte Indizes nicht mehrfach in *temp* zu speichern, bedarf es eines zusätzlichen Zwischenspeichers. Dies kann zum Beispiel in Form eines booleschen Arrays *arr* erreicht werden, dessen Felder den Indizes der Karte entsprechen. Wird Index *j* in *temp* aufgenommen folgt darauf *arr[j] = true*. Am Ende jeder Iteration muss *arr* somit zurückgesetzt werden.

Beide Varianten sind schematisch in Abb 3.14 dargestellt.

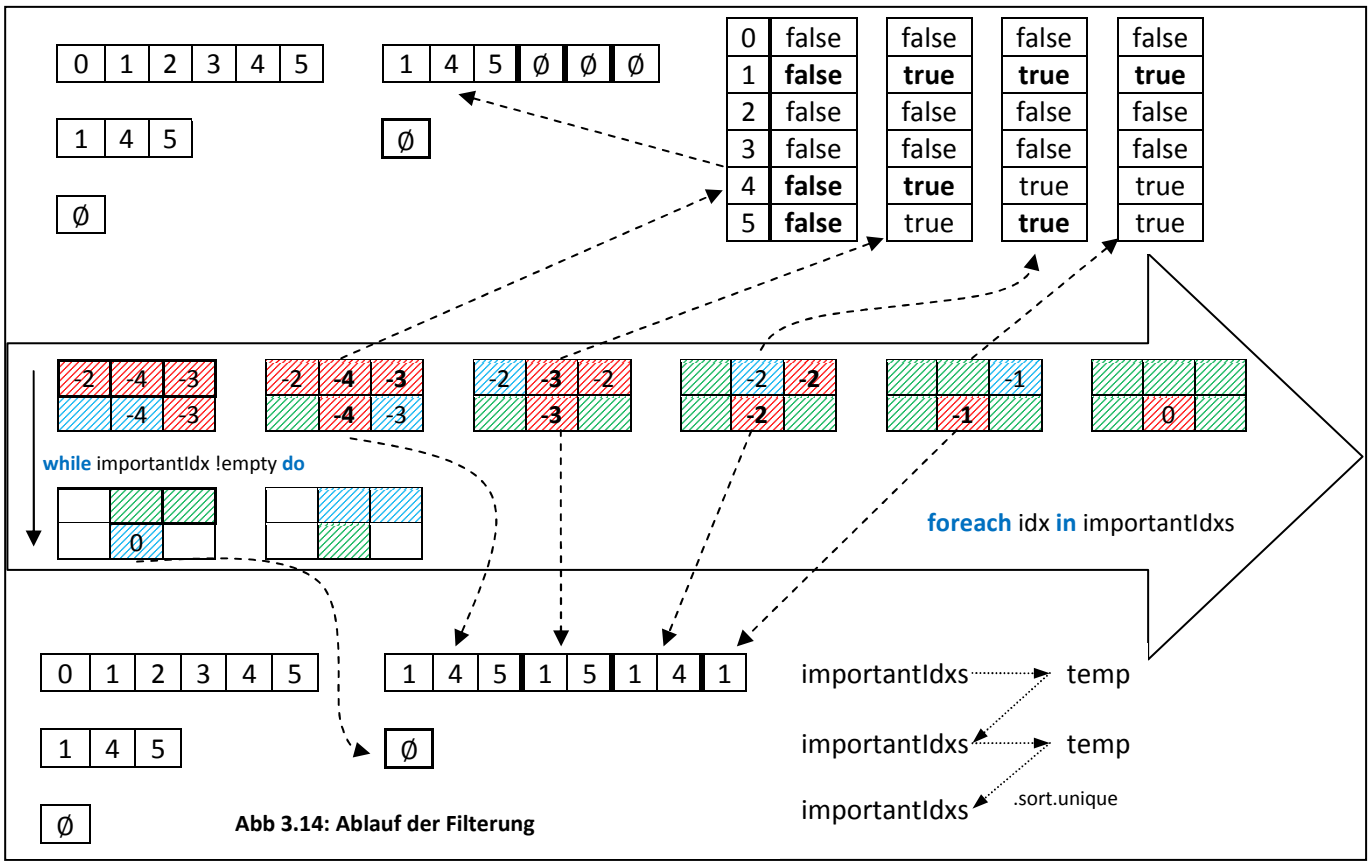


Abb 3.14: Ablauf der Filterung

$$rectangle = \{i \mid i_s + w \cdot r_{y'} \leq i \leq i_s + 2r_x + w \cdot r_{y'} \mid r_{y'} = 0 \dots 2r_y \ \wedge \ i \epsilon Map \ \wedge \ i_s = robotPos - r_x - wr_y\}$$

```
        // O((2r)²)
L1      vec_single importantIdxs = rectangle
2       vec_single temp
L2      while importantIdxs !empty do
4         foreach idx in importantIdxs do
L3          if map[idx] = U_SPACE ∧ Σ(neighbourValues(idx)) ∈ [−3; 0] then
L4            map[idx] = F_SPACE
L5            temp.push ({i | i ∈ neighbours(idx) ∧ map[i] = U_SPACE ∧ i ∈ Rectangle} )
8           fi
9         end
10        // O(n × log(n) + n) | n = temp. size
L6        importantIdxs = temp.sort.unique
12        temp.clear
13      end
```

Code 3.3: Entfernen duplizierter Einträge nach Iteration

```
L1      vec_single importantIdxs = rectangle
2       vec_single temp
3       bool flags[w × h] = {true, true, … , true}
L2      while importantIdxs !empty do
5         foreach idx in importantIdxs do
L3          if map[idx] = U_SPACE ∧ Σ(neighbourValues(idx)) ∈ [−3; 0] then
L4            map[idx] = F_SPACE
8             push = {i |i ∈ neighbours(idx) ∧ map[i] = U_SPACE ∧ i ∈ rectangle ∧ flags[i] = false}
L5            temp.push (push)
10            flags[push] = true
11          fi
12        end
L6        flags[temp] = false
14        temp.clear
15      end
```

Code 3.2: Ignorieren duplizierter Einträge während Iteration

3.9.3. Korrektheit und Terminierung

Die Betrachtung der Korrektheit und der Terminierung beruht auf folgenden Annahmen:

(1) U_SPACE= -1
(2) F_SPACE = 0
(3) O_SPACE= 100

Um die Beweisführung einfacher zu gestalten werden wie im Folgenden aufgelistet vier Fälle betrachtet.

Fall 1: Alle Zellen befinden sich in u-Space

$$\forall i \in importantIdxs : \sum\big(neighbourValues(i)\big) = -8$$
$$\Rightarrow \forall i \in importantIdxs : L3 = \text{ false}$$
$$\Rightarrow temp = \{\emptyset\}$$
$$\vdash \text{Terminierung von Fall 1}$$

Fall 2: Alle Zellen befinden sich in f-Space

$$\nexists i \in importantIdxs \mid map(i) = \text{ U\_SPACE}$$
$$\Rightarrow \forall i \in importantIdxs : L3 = \text{ false}$$
$$\Rightarrow temp = \{\emptyset\}$$
$$\vdash \text{Terminierung von Fall 2}$$

Fall 3: Alle Zellen befinden sich in o-Space

Siehe Fall 2

Fall 4: Gemischte Zellen

$$\forall i \in importantIdxs \mid map(i) = U\_SPACE \wedge neighbourValues(i) \in [-3;0] : \text{L3} = true$$
$$\Rightarrow |f\_Space_{neu}| > |f\_Space_{alt}|$$
$$\Rightarrow |u\_Space_{neu}| < |u\_Space_{alt}|$$
$$\Rightarrow |importantIdxs_{neu}| \leq |u\_Space_{neu}|$$
$$\Rightarrow |importantIdxs_{neu}| < |u\_Space_{alt}|$$
$$\Rightarrow \text{Algorithmus terminiert spätestens, wenn } |u\_Space_{neu}| = 0,$$

oder wenn:

$$\nexists i \in importantIdxs \mid map(i) = U\_SPACE \wedge neighbourValues(i) \in [-3;0]$$
$$\Rightarrow \forall i \in importantIdxs : \text{L3} = \text{ false}$$
$$\Rightarrow temp = \{\emptyset\}$$

$$\vdash \text{Terminierung von Fall 4}$$

3.9.4. Laufzeitbetrachtung

Im Folgenden werden die verschiedenen Ansätze bezogen auf Laufzeit und Speicherverbrauch analysiert und letztendlich gegeneinander abgewogen. Desweiteren findet eine ganzheitliche Abschätzung der Laufzeit der Glättung statt und es wird versucht eine tatsächliche Laufzeit anhand von Praxistests anzugeben.

Einfacher 3x3 Tiefpassfilter

Ein herkömmlicher Filter iteriert über einen gegebenen Ausschnitt und bildet für jede besuchte Zelle eine Nachbarschaftssumme aus deren direkten acht Nachbarn und der Zelle selbst. Die einzelnen Summanden sind je nach Anwendungsfall gewichtet. Um eine Normierung zu erreichen wird die Nachbarschaftssumme durch die Summe der Gewichte geteilt. Es entsteht somit eine von r abhängige quadratische Laufzeit pro Iteration.

$$O(s \cdot d^2), \qquad d = 2r \qquad \text{eq xvii}$$

Modifizierter 3x3 Tiefpassfilter

Wie weiter oben bereits erwähnt ist für das vorliegende Problem kein Auswerten aller Zellen notwendig. Vielmehr gilt es hier festzustellen, dass durch das überspringen irrelevanter Zellen Laufzeit eingespart werden kann. Abhängig von der Häufigkeitsverteilung von u-Space Zellen entsteht folgende Laufzeit pro Iteration.

$$O((1-p) \cdot d^2 + p \cdot s \cdot d^2) \mid d = 2r \wedge \mathrm{p} = \mathrm{P(u\_Space)} \qquad \text{eq xviii}$$

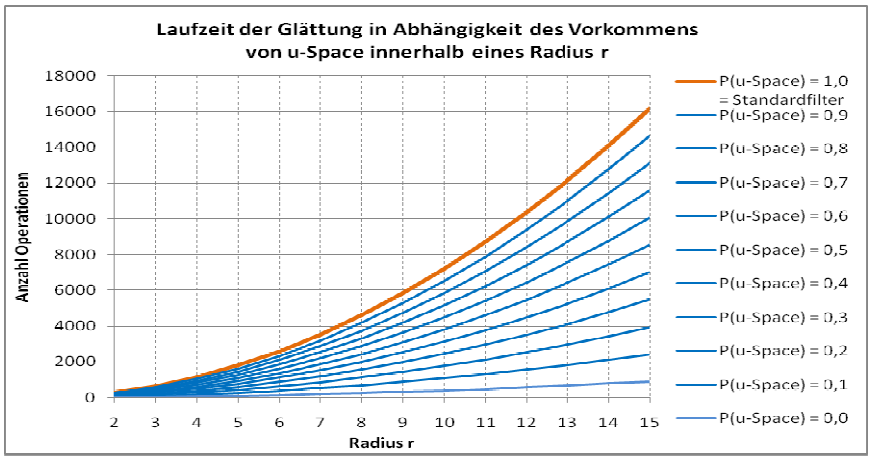


Diagramm 3.2: Laufzeiten der Glättung mit modifiziertem Filter

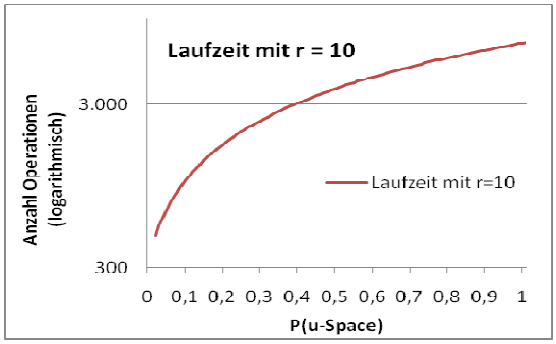


Diagramm 3.3: Laufzeiten der Glättung

Entfernen duplizierter Einträge nach Iteration (Variante a)

temp muss im ersten Schritt sortiert werden um im zweiten Schritt effizient Duplikate entfernen zu können. Es gilt zu beachten, dass die Laufzeit nicht mehr direkt von r abhängig gemacht werden kann, da *temp* ausschließlich für die nächste Iteration relevante Indizes enthält. Es gilt ohne Einschränkung, dass$|temp| < d^2$.

$$O(n \cdot \log(n) + n - 1), \qquad n = |temp| \qquad \text{eq xix}$$

Ignorieren duplizierter Einträge während Iteration (Variante b)

Das Vorhalten eines Speichers, um bereits verwendete Indizes behandeln zu können, generiert bei sehr großen Karten einen Speicherverbrauch, welcher quadratisch mit dem Radius wächst. Besteht die Karte zum Beispiel aus 4000x4000 Zellen, so müssen 16.000.000 boolesche Werte gespeichert werden. Dies resultiert in einer Speicherbelegung von ca. 8MB.

Davon ausgehend, dass maximal $|temp|$ Zellen für die nächste Iteration vorgesehen werden, lässt sich schließen, dass $c < |temp|$ duplizierte Einträge während der aktuellen Iteration verworfen werden. Setzt man für das Verwerfen x Operationen an, so entsteht eq xx. Diese Laufzeitbeschränkung ist wenig aussagekräftig und wird in den folgenden Abbildungen und Diagrammen anhand von Testläufen verdeutlicht.

$$O(x \cdot c), \qquad c < |temp| \qquad \text{eq xx}$$

Es ist leicht zu erkennen, dass der modifizierte Filter einen deutlichen Vorteil gegenüber dem einfachen Filter bietet. Abhängig von der Häufigkeitsverteilung der Zellen kann eine Laufzeit Optimierung von bis zu 90% erreicht werden (siehe Diagramm 3.4). Auch die sukzessive Einschränkung des Suchbereichs auf relevante Zellen ist leicht einzusehen. Hierbei gilt es aber den Gewinn durch weniger zu betrachtende Zellen mit der zusätzlichen Last durch die Handhabung der Duplikate gegeneinander abzuwägen. Versuche in echter Umgebung zeigen, dass die Anzahl der zu betrachtenden Zellen in den ersten Filteriterationen stets drastisch abnimmt (Abb 3.15). Daraus lässt sich schließen, dass der hinzugefügte Mehraufwand aufgrund einer geringen Anzahl an Zellen nicht ins Gewicht fällt, wohingegen aber das primitive Iterieren über den vollständigen Suchbereich eine wenig optimale Lösung darstellt.

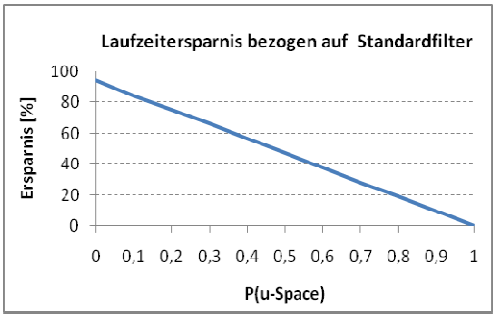
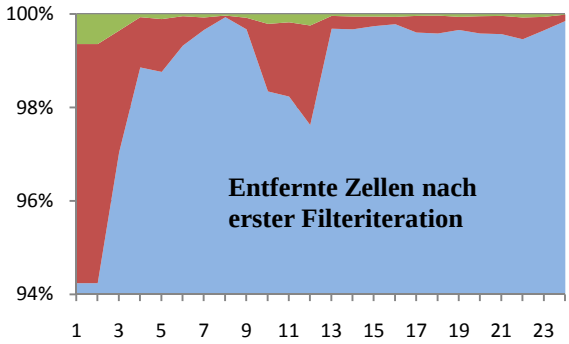
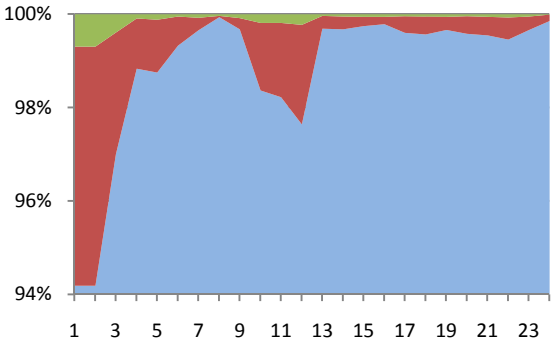


Diagramm 3.4: Optimierungspotential

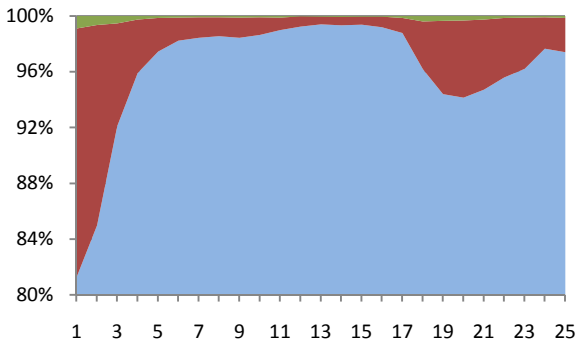
Im Folgenden werden anhand von in Versuchen mitgeschriebenen Daten die Varianten a und b gegeneinander abgewogen, um das passendere – beziehungsweise stabilere - Verfahren für die Exploration zu verwenden.



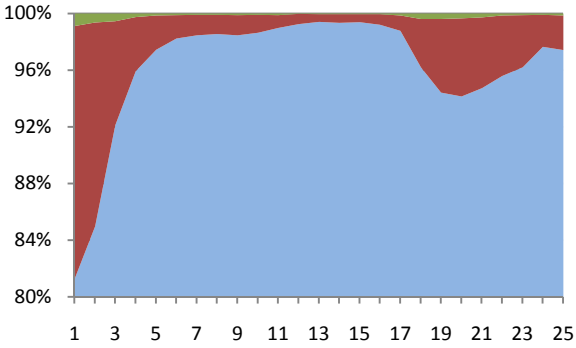
(a) **r=3**; 14.400 Zellen



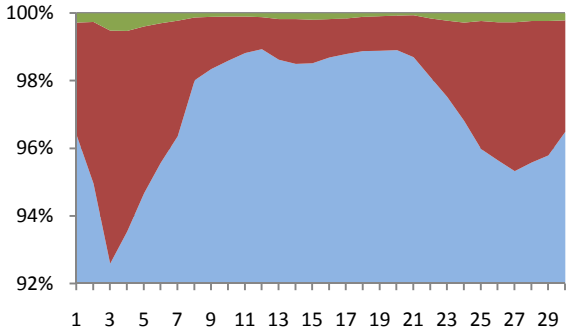
(e) **r=3**; 14.400 Zellen



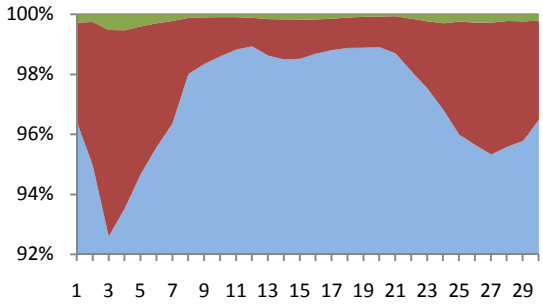
(b) **r=6**; 57.600 Zellen



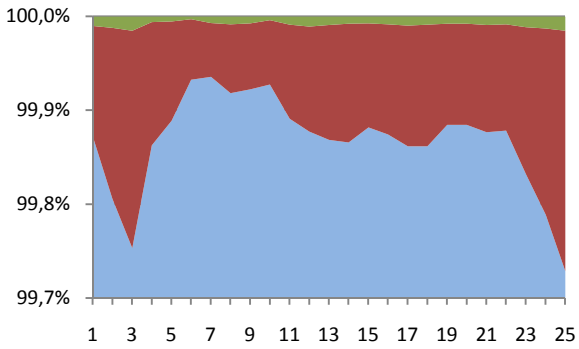
(f) **r=6**; 57.600 Zellen



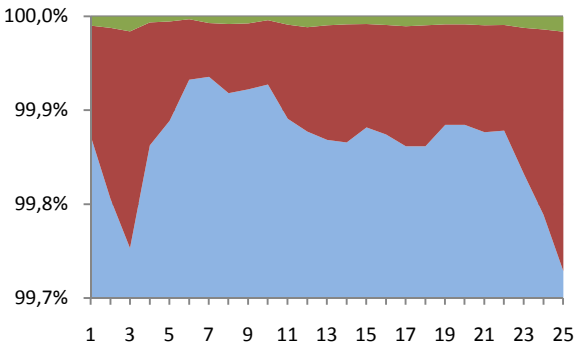
(c) **r=15**; 360.000 Zellen



(g) **r=15**; 360.000 Zellen



(d) **r=80**; 10.240.000 Zellen



(h) **r=80**; 10.240.000 Zellen

Abb 3.15: Zellenreduktion bei der Filterung
Die blaue/rote Fläche symbolisiert die verworfenen Zellen nach der ersten/zweiten Filteriteration. Grün stellt die verbliebenen Zellen vor der dritten Filteriteration dar. (a-d) Variante a; (e-h) Variante b.

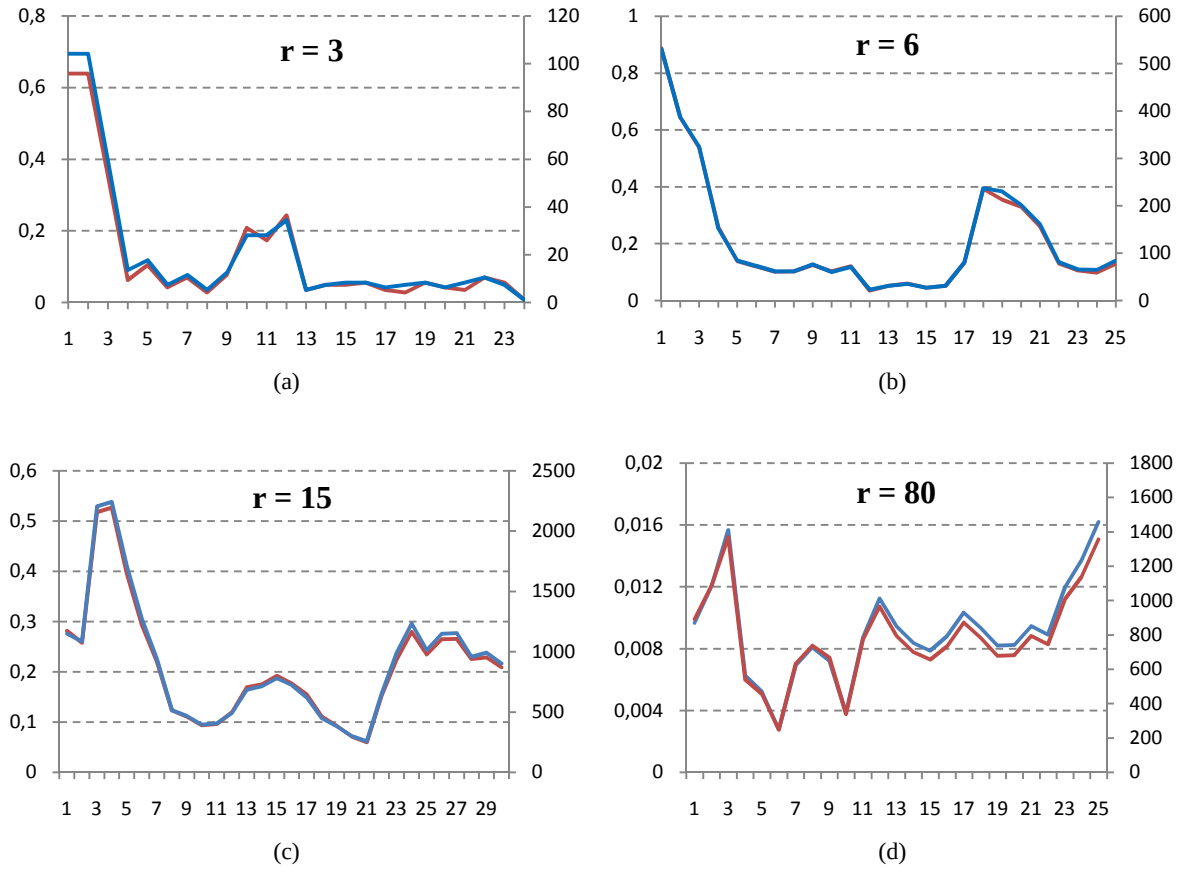


Abb 3.16: Zellenreduktion bei der Filterung
Zeigt den prozentualen Anteil der verbliebenen Zellen vor der dritten Filteriteration. Variante a ist rot, Variante b entsprechend blau. Die Linke Skale beschreibt den relativen Anteil in % und die rechte Seite die absolute Menge. Die waagrechte Skala steht für den Explorationszyklus. (a) r = 3; (b) r = 6; (c) r = 15; (d) r = 80.

Filteriteration	*importantIdxs*		Gefilterte Zellen		*potentials*		Duplikat Operationen	
Variante	a	b	a	b	a	b	a	b
1	57600	57600	10742	10742	16998	10793	**255871**	**118162**
2	10793	10793	344	357	721	510	**7565**	**3927**
3	511	510	285	298	613	443	**6288**	**3278**
4	440	443	265	273	573	273	**5822**	**3003**

Tabelle 3.2: Vergleich der Filtermechanismen mit r = 6

Filteriteration	*importantIdxs*		Gefilterte Zellen		*potentials*		Duplikat Operationen	
Variante	a	b	a	b	a	b	a	b
1	360000	360000	30071	30071	38162	26669	**618980**	**330781**
2	26669	26669	1326	1389	2650	1905	**32784**	**15279**
3	1865	1905	936	984	2119	1457	**25531**	**10824**
4	1418	1457	778	817	1765	1228	**20800**	**8987**

Tabelle 3.3: Vergleich der Filtermechanismen mit r = 15

Anhand von Abb 3.15 und Abb 3.16 lässt sich schließen, dass ein Großteil der initial als *importantIdxs* eingestuften Zellen bereits nach der ersten Filteriteration f_1 (zu sehen als blaue Fläche) verworfen ist. In der zweiten Filteriteration f_2 werden rot dargestellt zusätzlich weitere Zellen entfernt. Es ist zu erkennen, dass in den ersten beiden Filteriterationen jeweils ungleichmäßig viele Zellen entfernt werden. Nach f_2 jedoch ist die Anzahl der verbleibenden Zellen stets ähnlich (zu sehen als grüne Fläche). Dieses Verhalten lässt sich in weiteren Tests nachweisen.
Es ist somit klar, dass beide Filtervarianten den Bereich der relevanten Zellen bereits innerhalb der ersten beiden Filteriterationen deutlich einschränken. Tabelle 3.2 und Tabelle 3.3 zeigen jedoch, dass Filtervariante a aufgrund der Notwendigkeit des Sortierens nach jeder Filteriteration bereits hier doppelt so viele Ressourcen benötigt. Betrachtet man weitere Filteriterationen, so stellt man fest, dass sich die benötigten Operationen pro Filteriteration bei geringer werdender Zellenzahl annähern. Es zeichnet sich somit ab, dass Variante b mit zunehmendem Radius einen klaren Vorteil gegenüber Variante b bietet. Zusätzlich aufzuführen ist Tabelle 3.4. Hier ist zu erkennen, dass bei größer werdendem Radius die Anzahl der im Durchschnitt benötigten Filteriterationen ebenfalls für Variante b sprechen.

	Durchschnittliche Filteriterationen	
Radius [m]	Variante a	Variante b
3	**20,5**	**20,5**
6	**65,8**	67,64
15	108,7	**95,6**
80	98,72	**79,62**

Tabelle 3.4: Durchschnittliche Filteriterationen

4. Zielsuche und Exploration

Nachdem in den vorangehenden Kapiteln die technischen und algorithmischen Hintergründe diskutiert wurden, um eine Basis für weitere Schritte zu schaffen, wird in diesem Kapitel detailliert auf die Zielsuche und Exploration eingegangen. Im Gegenzug zu verschiedensten Ansätzen ([YAM97, YAM98, GL02]) wird hierfür weder die am nächsten gelegene *frontierCell*, noch ein zufälliges Ziel gewählt. Die im weiteren Verlauf vorgestellte Zielsuche stützt sich im wesentlichen auf zwei Komponenten. Die Komponente Qualität ([BMF+00]) gewährleistet, dass möglichst Interessante Ziele im Sinne einer Kartenaufdeckung berechnet werden. Die Komponente Sicherheit schließt im Gegenzug dazu bereits im Voraus potentiell gefährliche Ziele aus, um eine sichere Navigation des Roboters zu gewährleisten.
Im Verlauf dieses Kapitels wird gezeigt anhand welcher Kriterien *frontierRegions* für die qualitative Beurteilung eines Ziels herangezogen werden können. Darauf aufbauend werden ein Verfahren zur Berechnung möglichst gewinnbringender Ziele vorgestellt und relevante Sicherheitskriterien erörtert. Letztendlich wird der vollständige Explorationsablauf schematisch dargelegt und erläutert. Desweiteren werden mögliche Probleme identifiziert und deren Lösung anhand von fallback Strategien motiviert.

4.1. Qualitätskriterien

Die für diesen Ansatz verwendeten Kriterien dienen einer möglichst objektiven qualitativen Einschätzung der extrahierten *frontierRegions* bezogen auf die Position und Orientierung des Roboters. Um eine einheitliche Betrachtung zu gewährleisten, werden alle Qualitäten normalisiert. Umgesetzt wird dies durch lineare Interpolation. Der Autor geht von folgenden vier relevanten Kriterien aus

Connectivity (i):
Die *connectivity* einer *frontierRegion* f definiert sich durch die Anzahl der Nachbarschaftsrelationen innerhalb ihrer selbst. Im Idealfall besitzen alle *frontierCells* $\in f$ maximal zwei Nachbarn und bilden somit eine Linie (Abb 3.9).

$$\forall i \in frontierRegion : |neighbours(i)| \leq 2 \Leftrightarrow Linie \qquad \text{eq xxi}$$

Finden sich mehr als zwei Nachbarn pro *frontierCell*, so ist in den meisten Fällen davon ausgehen, dass ein Szenario ähnlich wie in Abb 3.10 vorzufinden ist. Eine Bewertung dieses Kriteriums findet statt, indem die tatsächlich vorhandenen Nachbarschaftsrelationen zur minimal möglichen Anzahl an Nachbarschaftsrelationen ins Verhältnis gesetzt werden.

$$connectivity = \frac{|edges|}{|f| \cdot 2 - 2} \qquad \text{eq xxii}$$

Größe (ii):
Je größer eine *frontierRegion*, desto mehr Information verbirgt sich dahinter. Es gilt hierbei aber zu beobachten, dass eine Relation zu *connectivity* hergestellt werden sollte. Eine geradlinige *frontierRegion* verbirgt bei gleicher Größe mehr Information als ein willkürlicher Punktehaufen, der durch Ungenauigkeiten in der Sensorik entstehen kann.

Distanz (iii):
Eine geringe Distanz zum nächsten Ziel ist bei der Exploration einer Karte wünschenswert. Begründet wird dies zum einen durch den Umstand, dass dadurch ein Zurückfahren vermieden wird und zum anderen dadurch, dass eine systematische Exploration des näheren Umfeldes gewährleistet wird. Zwei einfache Metriken ergeben sich durch eq xxiii und eq xxiv.

$$dist_j = \{\min, \max\}(dist(i, robotPos)) \mid i \in frontierRegion_j \qquad \text{eq xxiii}$$

oder

$$dist_j = \frac{1}{|frontierRegion_j|} \times \sum dist(i, robotPos) \mid i \in frontierRegion_j \qquad \text{eq xxiv}$$

Richtung (iv):
Ist eine möglichst geradlinige Exploration gewünscht, so kann dies durch eine entsprechende Gewichtung dieses Kriteriums erreicht werden. Die Richtung berechnet sich aus dem Winkel zwischen der Orientierung des Roboters und der Richtung, in der $frontierRegion_j$ zu finden ist.

Alle berechneten Qualitäten werden normiert und anhand ihrer Gewichte addiert.
Heuristisch und durch Tests in simulierter als auch in realer Umgebung lassen sich die Gewichte der einzelnen Kriterien abwägen. Dabei gilt es zu bedenken, dass unterschiedliche räumliche Gegebenheiten eine angepasste Gewichtung benötigen. So führt zum Beispiel ein hohes Gewicht für das Kriterium Richtung in einem Flur schnell zu Ergebnissen, wohingegen bei der Exploration kleiner Räume das Kriterium Distanz von Bedeutung sein wird.

4.2. Bestimmung eines Ziels

Mit der Wahl einer $frontierRegion_j$ lässt sich im nächsten Schritt ein dazugehöriges Ziel für die weitere Exploration finden. Hierbei gilt es festzustellen, dass sich ein unbekanntes Gebiet genau dann möglichst umfassend erfassen lässt, wenn die verwendete Sensorik direkt darauf gerichtet ist. Desweiteren ist zu beachten, dass auch alle potentiellen Ziele verschiedenen Qualitätskriterien unterliegen sollten.

Abstand zu o-Space (i):
Um Kollisionen zu vermeiden und Fehler in der Pfadplanung auszuschließen empfiehlt es sich Ziele, welche zu nahe an o-Space liegen, zu verwerfen.

Abstand zu u-Space (ii):
Da die Beschaffenheit des Gebietes hinter einer *frontierRegion* nicht vorhergesagt werden kann, ist es im Zuge der Sicherheit ratsam auch hier einen Sicherheitsabstand einzuhalten.

Abstand zu Roboter (iii):
Ein zu nahe am Roboter gewähltes Ziel resultiert zum einen in einer verlangsamten Exploration, da der Roboter schneller das Ziel erreicht und somit öfter ein neues gesucht werden muss. Zum anderen wird dadurch sicher gestellt, dass der Roboter in die *goalArea* einfährt und somit das Ereignis *goalAreaEntered* ausgelöst wird. Der Sicherheitsbereich um den Roboter muss demzufolge größer als der Auslöseradius um das letztendliche Ziel herum sein.

Ausgehend von einer bereits ausgewählten $frontierRegion_j$ F bedarf es eines Verfahrens, um ein darauf bezogenes Ziel zu finden. Einfach wäre es eine zufällige oder die nächst gelegene Zelle zu nehmen. Jedoch besteht die Absicht dieser Arbeit darin ein Verfahren zu entwerfen, welches eine möglichst effektive Zielwahl ermöglicht. Um dies zu gewährleisten, wird im Folgenden ein Ansatz vorgestellt, welcher entlang von F bildlich gesprochen ein Band von Zielen berechnet. Dieser sogenannten *potentialGoalArea* wird letztendlich ein Ziel entnommen, welches den Kirterien i, ii und iii stand hält.
Wie in Code 4.1 zu sehen wird hierfür Schrittweise zwischen je zwei nachfolgenden Zellen in F ein Vektor $\vec{v}$ aufgespannt. Orthogonal zu diesem wird ein Vektor $\vec{u}$ berechnet. Die Länge a von $\vec{u}$ berücksichtigt hierbei ii. Potentielle Explorationsziele entstehen nun zu beiden Seiten von F, wobei aber nur solche übernommen werden, die weder in o-Space, noch in u-Space liegen. Zusätzlich werden alle potentiellen Ziele aufgebläht, um bei der letztendlichen Auswahl eines Zieles noch flexibler zu sein. Somit entsteht ein Band aus überlappenden Kreisen entlang von F, welches die *potentialGoalArea* bildet. Abb 4.1 und Abb 4.2 zeigen das Ergebnis anhand einer Skizze und anhand von einer Visualisierung mit rviz.
In einem weiteren Schritt werden solange die einzelnen Zellen aus *potentialGoalArea* den Kriterien i, ii und iii unterworfen, bis eine qualitativ passende Zelle gefunden wird. Diese Zelle dient als nächster Explorationspunkt und gewährleistet bei guter Parametrisierung ein hochwertiges Ziel. Eine detaillierte Beschreibung des vollständigen Ablaufes inklusive Fehlerbetrachtung findet sich in 4.3 und 4.4, soll an dieser Stelle aber nicht relevant sein.

```
1       vector<point> potentialGoals
2       F = frontierRegion
3       for int i = 0; i < F.size(); i+=2 do
4         v⃗ = F_i F_{i+1}⃗
5         u⃗: u⃗ ⊥ v⃗ ∧ |u⃗| = a
6         goal_1 = F_i + u⃗
7         potentialGoalArea_1 = { p | d(p, goal_1) < b } if goal_1 ∉ (u_Space ∪ o_Space)
8         goal_2 = F_i − u⃗
9         potentialGoalArea_2 = { p | d(p, goal_2) < b} if goal_2 ∉ (u_Space ∪ o_Space)
10        potentialGoals.pushBack(potentialGoalArea_i)
11      end
```

Code 4.1: Berechnug der *potentialGoalArea*

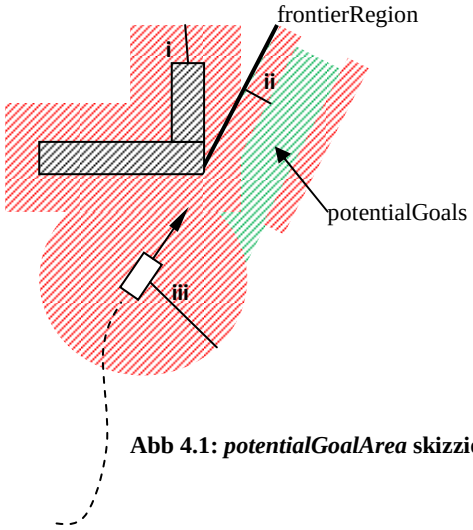


Abb 4.1: *potentialGoalArea* skizziert

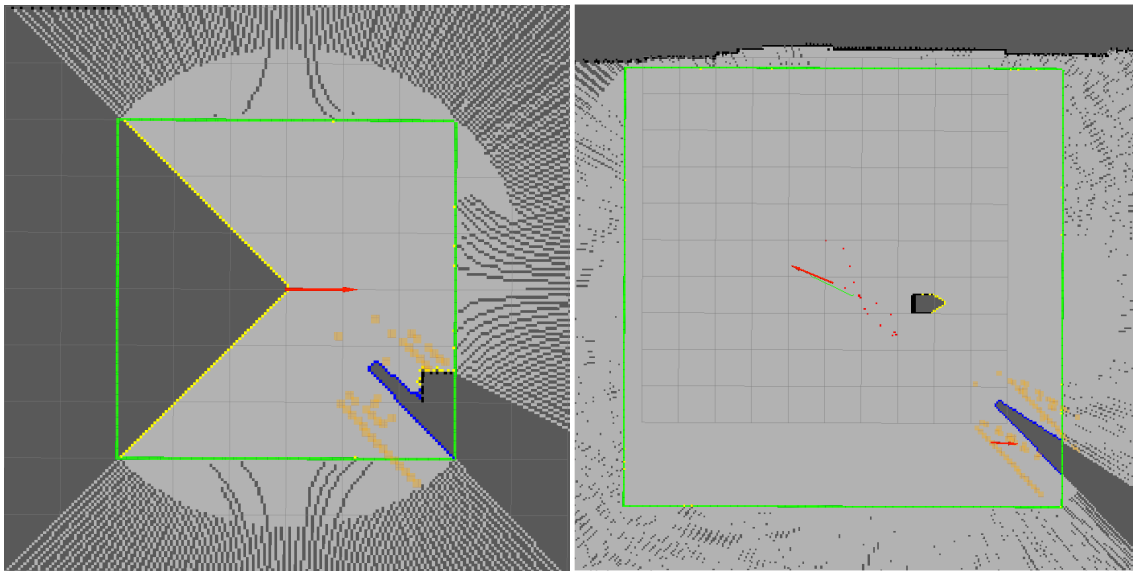


Abb 4.2: *potentielGoalArea* in rviz

(orange) *potentialGoalArea;* (blau) beste *frontierRegion*; (gelb) *frontierRegions*

Basierend auf einem ausgewählten Explorationsziel muss eine weitere Betrachtung der Art und Weise zu teil werden, wie sich der Roboter F nähert. Es werden hierfür zwei Varianten vorgestellt. Beide Varianten gehen im optimalen Fall von einer geradlinigen Repräsentation von F aus. Gilt diese Annahme nicht, so funktionieren beide Varianten dennoch, liefern aber ungenau Anfahrtsrichtungen. Desweiteren wird zum besseren Verständnis mit Richtungen in Form von Vektoren argumentiert. ROS verwendet hierfür einen Winkel. Die Umrechnung ist für die Erklärung beider Ansätze nicht relevant und wird somit übergangen.

Variante 1.1:

Ausgehend von einer geradlinigen *frontierRegion* F wird F in Form eines Vektors $\vec{F}$ dargestellt. Es werden zwei Punkte a und b berechnet, wobei beide innerhalb von F liegen, a die minimale Distanz und b die maximale Distanz zum Roboter aufweist. Es ergibt sich somit $\vec{F} = \overrightarrow{ab}$. Im Weiteren wird ein Vektor $\vec{d}$ berechnet, welcher orthogonal auf $\vec{F}$ steht und die gewünschte Ausrichtung des Roboters am Zielpunkt angibt. Abhängig von der berechneten Trajektorie kann die Ist-Ausrichtung des Roboters am Zielpunkt von der Soll-Ausrichtung abweichen. In Abb 4.3 wird dies durch den Winkel α veranschaulicht.

Es gilt festzustellen, dass die relative Position des Roboters zu F keinen Einfluss auf die Richtung von $\vec{F}$ hat, da je zwei ungleiche Punkte auf einer Geraden stets linear voneinander abhängige Vektoren produzieren. a ≠ b wird durch die Umstände minimale/maximale Distanz gewährleistet, sobald $|F| > 1$.

Variante 1.2:

Selbige Vorgehensweise wie in Variante 1.1 mit dem Unterschied, dass $\alpha = \vec{d} \measuredangle \vec{F} \neq 90$.

Variante 2:

Analog zu Variante 1.1 wird b berechnet. Ausgehend vom Zielpunkt g wird $\vec{d} = \overrightarrow{bg}$ berechnet. Der Roboter ist somit entlang von F in Richtung des entfernten Punktes b ausgerichtet. Abb 4.4 skizziert diesen Ansatz.

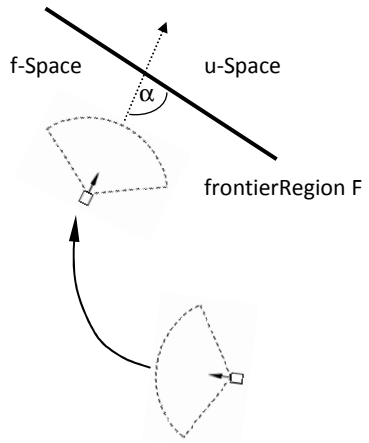


Abb 4.3: Anfahren von F in nahezu rechtem Winkel

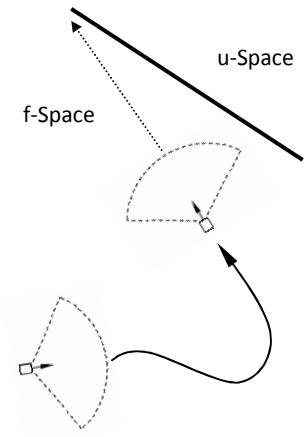


Abb 4.4: Anfahren von F mit Blick auf entfernte Ecke

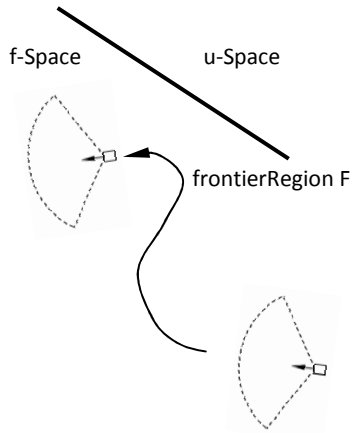


Abb 4.5: Ungültiges Anfahren von F

4.3. Ablauf

Ausgelöst wird jeder Explorationsschritt durch das Empfangen einer aktualisierten Karte über den Knoten gmapping. Ausgehend von der aktuellen Position des Roboters wird nun innerhalb eines initialen Radius zuerst die Karte gefiltert, um danach innerhalb des gefilterten Bereichs nach *frontierRegions* zu suchen. Wird keine *frontierRegion* gefunden, so wird der Radius erhöht und der Prozess von vorne gestartet. Andernfalls werden im nächsten Schritt alle *frontierRegions* hinsichtlich ihrer Qualität für die Exploration beurteilt und absteigend sortiert. Jede so gefundene und qualifizierte *frontierRegion* wird im folgenden anhand verschiedener Grenzwerte für eine Zielsuche zugelassen oder verworfen. Jede zugelassene *frontierRegion* wird in eine whitelist aufgenommen, um zu späteren Zeitpunkten darauf zurückgreifen zu können. Die erste zugelassene und somit die am qualitativsten wertvollste *frontierRegion* wird in diesem Explorationsschritt für die Zielsuche herangezogen. Anhand dieser *frontierRegion* werden nun potentielle Ziele gemäß 4.2. berechnet. Im letzten Schritt wird nun über alle potentiellen Ziele iteriert, um diese Qualitätskriterien zu unterwerfen. Das erste positive Ziel wird als nächstes Navigationsziel verwendet. Besteht kein potentielles Ziel diese Qualitätschecks wird mit der nächsten *frontierRegion* fortgefahren. In Abb 4.6 wird der Ablauf schematisch dargestellt.

4.4. Fallback Strategien

Es ist unumgänglich, dass es bei vorliegendem Prozess zu grenzwertigen Situationen kommen kann, die die Exploration erheblich verlangsamen oder sogar unterbrechen können. Um diesen Fällen entgegenzutreten wurden im Zuge der Arbeit fallback Strategien definiert. Im Folgenden werden diese Strategien vorgestellt und deren Existenz anhand von Beispielen begründet.

Strategien:

a) Suchen nach *frontierRegions* in *whitelist*
   Stellt die Möglichkeit zur Verfügung qualitativ wichtige, aber nicht erschlossene *frontierRegions* zwischen zu speichern und in bestimmten Situation eine für die aktuelle Situation passende heraus zu nehmen.

b) Suchen nach *frontierRegions* auf gesammter Karte
   Als ultimative Strategie konzipiert um bis dato unbekannte *frontierRegions* auf der gesammten Karte suchen zu können. Produziert deutlich erhöhte Laufzeitkosten, da sämtliche Operationen quadratisch mit dem veranschlagten Suchradius wachsen.

c) Aktuelles Ziel in *blacklist* aufnehmen und Erkundunsschritt neu starten
   Wird das aktuelle Ziel in eine *blacklist* aufgenommen, so wird dieses im nächsten Explorationsschritt ignoriert und ein dadurch möglicherweise verbundenes Problem beseitigt.

d) Annähern an Explorationsziel bevor neue Iteration gestartet wird
Strategie, welche abhängig von Vorgaben und Fehlerstatus angewandt werden kann. Algorithmisch umgesetzt wird dies durch die Pfadlänge d zwischen Startpunkt und Zielpunkt und durch die noch zu fahrende Strecke d'. Gesteuert durch einen Schwellenwert $0 \leq s \leq 1$ wird eine erneute Iteration zugelassen, sobald $\frac{d'}{d} \leq s$.

Probleme:

- Maximal zulässiger Radius erreicht
  Werden wiederholt innerhalb erhöhter Radien keine oder qualitativ unzureichende *frontierRegions* beziehungsweise Ziele gefunden, so kann dies unter anderem an folgenden Konstellationen liegen:

  i. Roboter befindet sich in bereits großflächig erkundetem Gebiet
  ii. Kriterien sind zu hart eingestellt
  iii. Karte ist vollständig erkundet (basierend auf der Annahme, dass eine maximale Kartengröße voreingestellt ist)
  iv. Roboter befindet sich in Gebiet mit sehr vielen kleinen Hindernissen

In allen vorliegenden Fällen wird zunächst Strategie a) angewendet. Findet sich hierdurch keine passende *frontierRegion*, wird Strategie b) gestartet. Liefert dieser Schritt immer noch keine *frontierRegions*, so bedarf es einer erweiterten Situationsanalyse und einer tiefer gehenden Fehlerbehandlung. Denkbar wäre eine an [TTW+04] angepasste Strategie.

- Roboter bewegt sich nicht
  Bewegt sich der Roboter nicht werden keine neuen Sensordaten produziert und somit durch das Ausbleiben einer aktualisierten Karte kein neuer Explorationszyklus gestartet. Wird als Lösung hierfür ein neuer Zyklus durch einen timeout ausgelöst und der Roboter bewegt sich immer noch nicht, läuft man Gefahr wiederholt in die gleiche Situation zu geraten, da auf einer unveränderten Karte mit hoher Wahrscheinlichkeit das selbe Ziel wiederholt berechnet wird und somit kein Fortschritt gewährleistet ist. Um nun zu verstehen, wie in dieser Situation vorzugehen ist, gilt es sich die Umstände dieses Falles klar zu machen.

  i. Roboter erreicht Zielgebiet bevor Karte aktualisiert wurde
  ii. Roboter fährt sich fest
  iii. Pfad enthält ungültige Wegpunkte

  Lösung:
  Das erste Problem lässt sich lösen, indem regelmäßig der Zustand des Pfadplaners ausgewertet wird. Wird innerhalb eines definierten Zeitfensters keine Karte empfangen und der Roboter befindet sich innerhalb der *goalRegion*, so darf in diesem Fall einmalig ein neuer Zyklus ausgelöst werden. Erst nachdem sich der Roboter bewegt hat, darf ein weiterer Zyklus zeitlich bedingt ausgelöst werden. Andernfalls

könnte wiederholt das gleiche Ziel berechnet werden. Löst dieses Vorgehen das Problem nicht, wird Strategie c) angewendet und ein neuer Zyklus gestartet.
Fährt sich der Roboter fest, so muss in der Regel manuell eingegriffen werden.
Der letzte Fall wird im übernächsten Punkt behandelt.

- Ziel wird von Pfadplaner verworfen
  Befindet sich ein berechnetes Ziel zu nahe an einem Hinderniss, so kann dieses vom Pfadplaner verworfen werden. Eine weitere Ursache besteht darin, dass der Pfadplaner eine Trajektorie berechnet, welche durch *u-Space* verläuft, und somit verworfen wird.

  Lösung:
  Tritt dieses Problem auf, kann durch Strategie c) schnell ein alternatives Ziel gefunden werden.

- Pfadplaner berechnet schlechten Pfad
  In seltenen, nicht nachstellbaren Fällen, konnte beobachtet werden, wie der Pfadplaner weder Querbeschleunigung noch Winkelbeschleunigung an die generierten Nachrichten anhängt. In weiterern seltenen Fällen konnte außerdem festgestellt werden, dass der Pfadplaner nur eine Winkelbeschleunigung berechnet, wodurch der Roboter sich im Kreis dreht. Dargestellt wird dies in eq xxv und eq xxvi.

$$\forall i \in path : i.linear.w = 0 \wedge i.angular.w = 0, \qquad w \in \{x, y, z\} \qquad \text{eq xxv}$$

$$\forall i \in path : i.linear.w = 0 \wedge i.angular.w \neq 0, \qquad w \in \{x, y, z\} \qquad \text{eq xxvi}$$

  Lösung:
  Da das Problem ursprünglich in einem von *frontier_navigation* unabhängigen Stack auftritt ist die Lösung dafür primär direkt dort zu suchen. Um jedoch die Exploration durch diesen Fall nicht abbrechen zu müssen, wird bei wiederholten nacheinander auftreten in beiden Fällen Strategie c) angewendet.

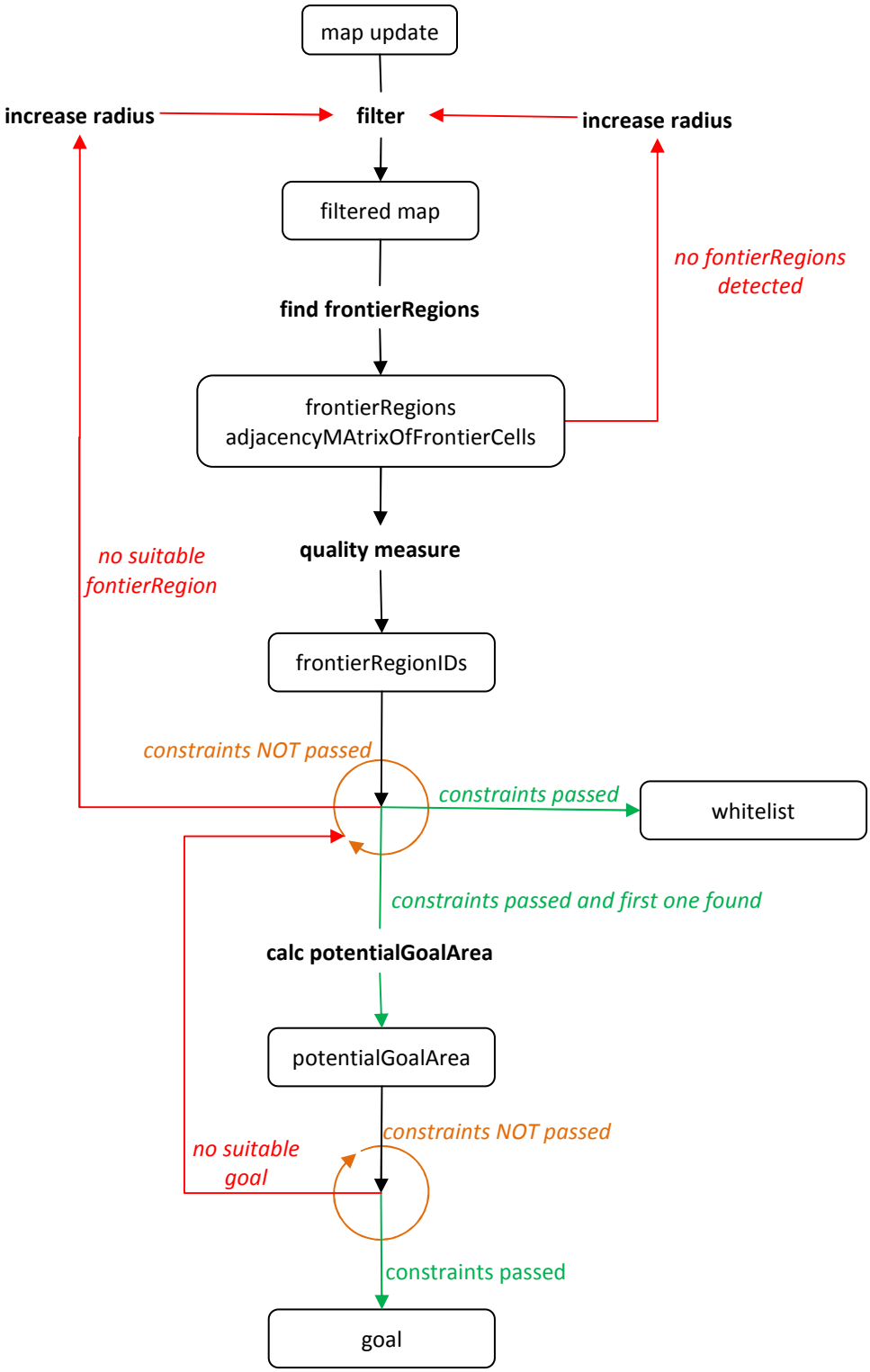


Abb 4.6: Schematischer Ablauf der Exploration

In rot dargestellt sind Fehlerfälle, die unter Umständen eine fallback Strategie benötigen. Grün gibt im Gegenzug den Standardfall an. In orange zu sehen sind die Auswahlzyklen für die beste *frontierRegion* und das beste Ziel. In den Kästen zu sehen sind die jeweiligen Datensätze nach der entsprechenden Operation.

5. Experimentelle Ergebnisse

In diesem Kapitel werden die gesammelten Ergebnisse und Erfahrungen anhand von Grafiken und Tabellen anschaulich dargestellt. Desweiteren wird ein Vergleich der angewandten Einstellungen gezogen.

Es werden ausschließlich folgende Rahmenparameter verwendet:

weightOfConnectivity:	3.0
weightOfSize:	2.0
weightOfDistance:	1.0
weightOfDirection:	4.0
Sensorreichweite:	20m
Samples:	250
Winkelabdeckung:	270°
Strategie 0:	Berechne neues Ziel während Fahrt zu aktuellem Ziel.
Strategie 1:	Berechne neues Ziel erst bei Erreichen des aktuellen Zielgebiets.

Die Rechenarbeit wurde von einem *Intel(R) Core(TM) i7 CPU Q740 @1,73GHz* mit 4GB Arbeitsspeicher übernommen.

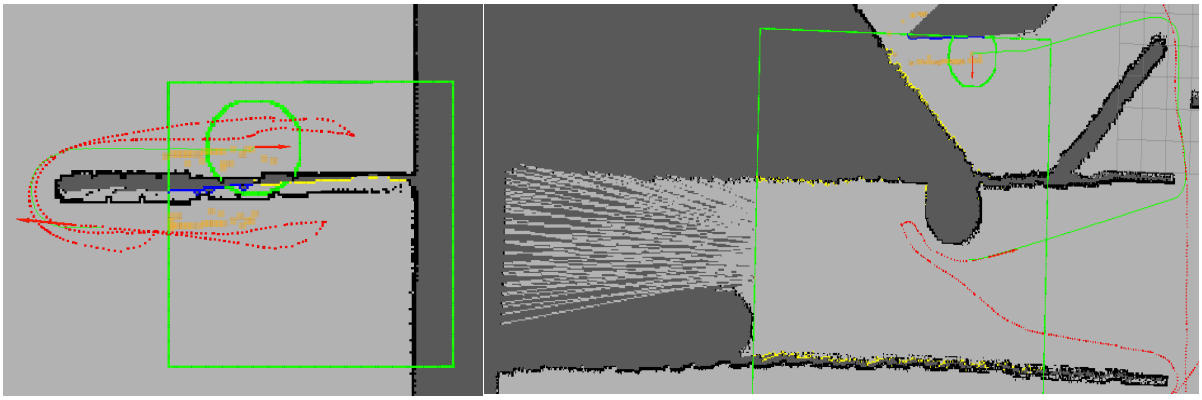
Die Anpassung der Radien und die verwendete Suchstrategie werden durch die Bezeichnung a_b_c_d beschrieben, wobei a für den Startradius, b für die inkrementelle Erhöhung des Radius, c für die Anzahl der Versuche und d für die verwendete Strategie steht.
Im Verlauf des Testens stellte sich heraus, dass das Erreichen eines Zielbereichs vor einer erneuten Zielberechnung einen Vorteil bezogen auf die Explorationsgeschwindigkeit gegenüber regelmäßiger Zielsuche während der Fahrt zum aktuellen Ziel bietet. Desweiteren lieferten Varianten mit kleineren Schrittgrößen bessere Ergebnisse als Varianten mit großer Schrittgröße. Genaueres hierzu lässt sich den folgenden Abbildungen und Tabellen entnehmen.

Wie in Abb 5.1 (a) dargestellt, können auch bei ansonsten effizienter Kalibrierung Situationen entstehen, die ein Problem für die Exploration darstellen. So ist in diesem Fall zu sehen, dass *frontierRegions* innerhalb eines eingeschlossenen Bereiches gefunden wurden. Der Roboter pendelt somit zwischen zwei Zielpunkten, um aus verschiedenen Richtungen einen Blick auf die *frontierRegion* werfen zu können. Durch pendeln entstehende duplizierte Ziele werden zwar in die dafür vorgesehene *blacklist* aufgenommen, wodurch sich dieses Problem nach einer gewissen Zeit selbstständig löst, doch benötigt dieser Vorgang im Vergleich zu einer fehlerfreien Exploration viel Zeit.

Desweitern stellte sich während des Testens heraus, dass die verwendeten fallback Strategien zwar in den meisten Fällen das gewünschte Lösungsverhalten bereit stellen. Es konnte aber auch festgestellt werden, dass in seltenen Fällen eine Kombination unterschiedlicher Probleme eine ausgereiftere Fehlerbehandlung voraussetzt. So kann ein *ROBOT_NOT_MOVING_TIMER* verschiedene Ursachen haben. Treten diese einzeln auf, kann jede Ursache gekapselt behandelt werden. Treten aber die Ursachen *GOAL_REJECTED* und *GOAL_AREA_ENTERED* zur gleichen Zeit auf, so sind detailliertere Fallunterscheidungen notwendig.

In einigen weiteren Fällen konnte festgestellt werden, dass Explorationsgebiete zuerst verlassen werden, um diese später wieder zu besuchen (siehe Abb 5.1 (b)). Im Sinne einer effizienten Exploration wäre eine vollständige Exploration des aktuellen Raumes zu bevorzugen. Dies würde aber ein topologisches Verständnis der Umgebung gemäß [KLS+05] voraussetzen.

Im Verlauf verschiedener Simulationen konnte beobachtet werden, dass mit zunehmend aufgedeckter Karte Wege mehrfach gefahren werden. Anders ausgedrückt nimmt der Anteil der explorierten Areale im Verhältnis zur zurückgelegten Strecke ab. Zu sehen ist dies in Abb 5.17. In weiteren Ausbaustufen des Projektes könnte eine Entdeckung dieses Verhaltens ein Indiz für eine ausreichend vollständig explorierte Karte sein.



(a) (b)

Abb 5.1: Fehlerfälle
(a) Pendeln zwischen zwei Zielen und eingeschlossene *frontierRegions*; (b) Raum wird verlassen um nächsten Raum zu explorieren. Resultiert in einer erneuten Exploration des ersten Raumes.

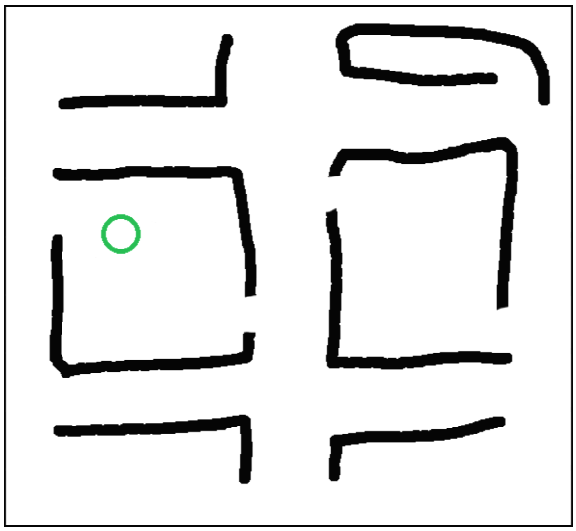
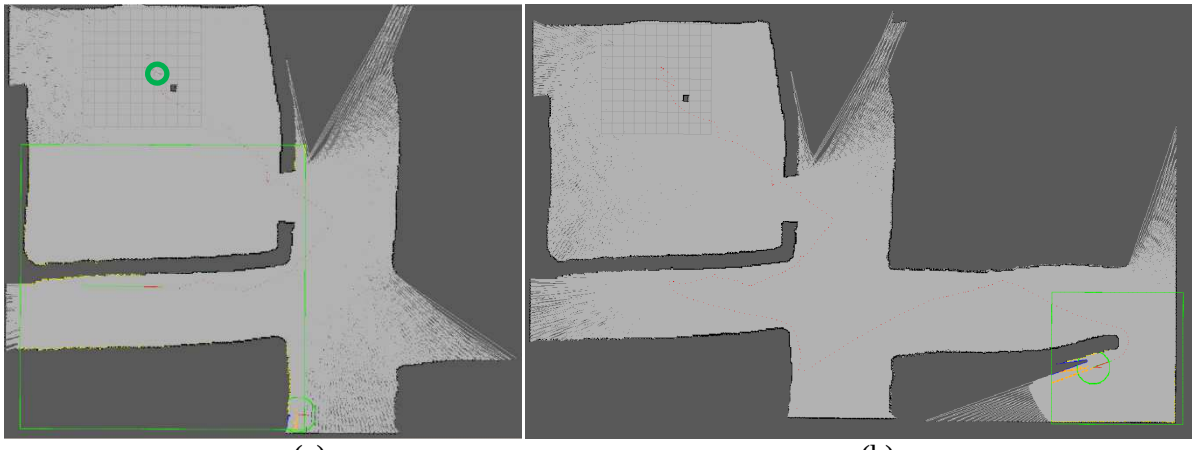
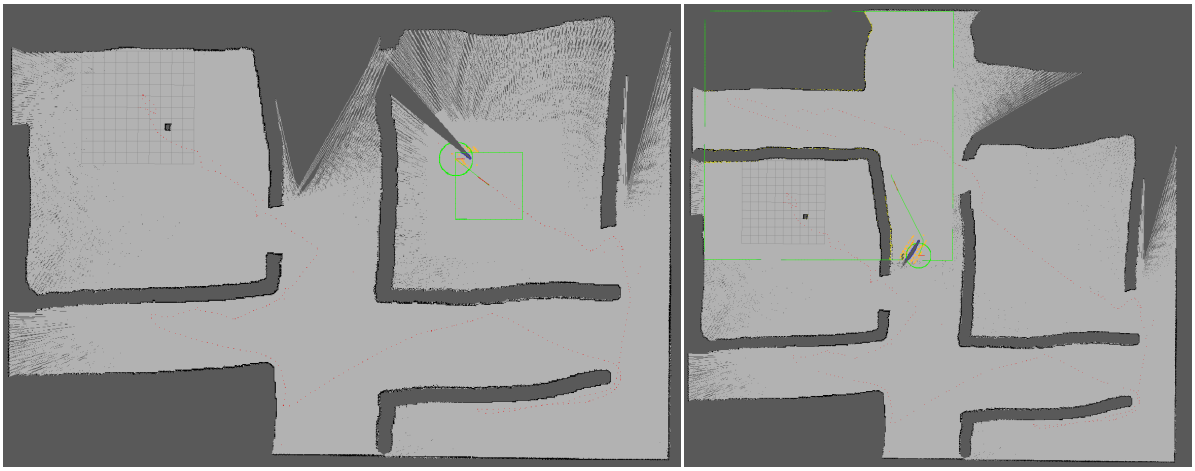


Abb 5.2: Simulierte Umgebung 1



(a) (b)

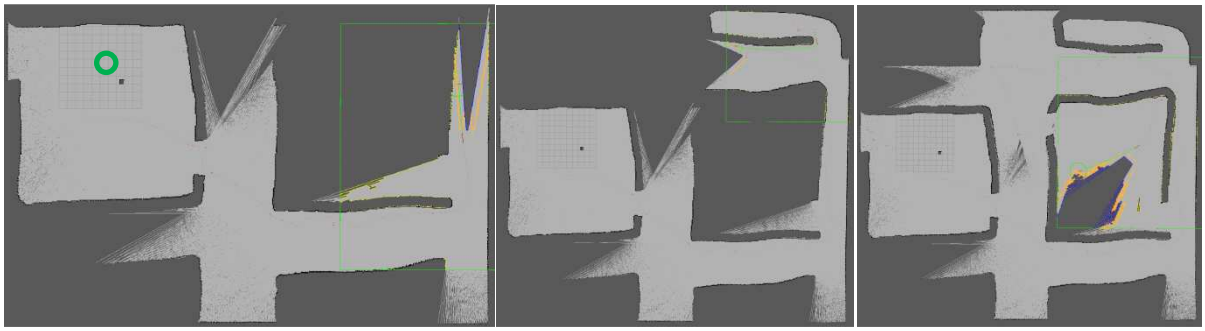


(c) (d)

Abb 5.3: Explorationsergebnis 3_3_5_0
(a) nach 10 min; (b) nach 20 min; (c) nach 30 min; (d) nach 40 min.

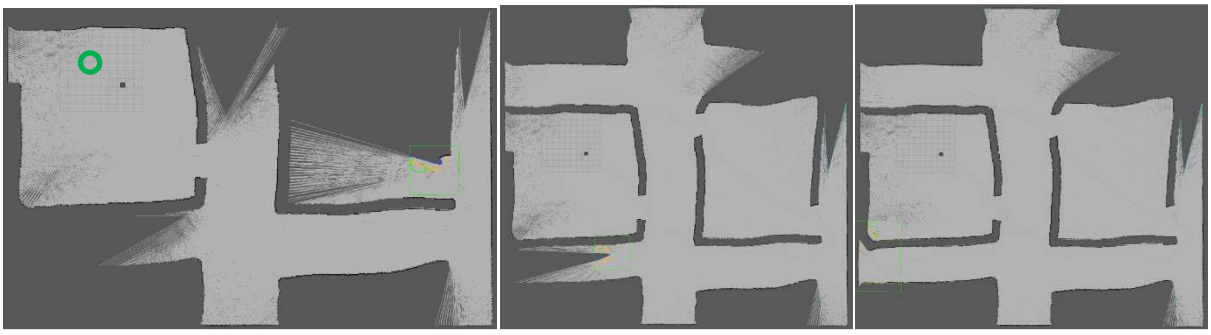
(a) (b) (c)

Abb 5.4: Explorationsergebnis 6_3_4_0
(a) nach 10 min; (b) nach 20 min; (c) nach 30 min.



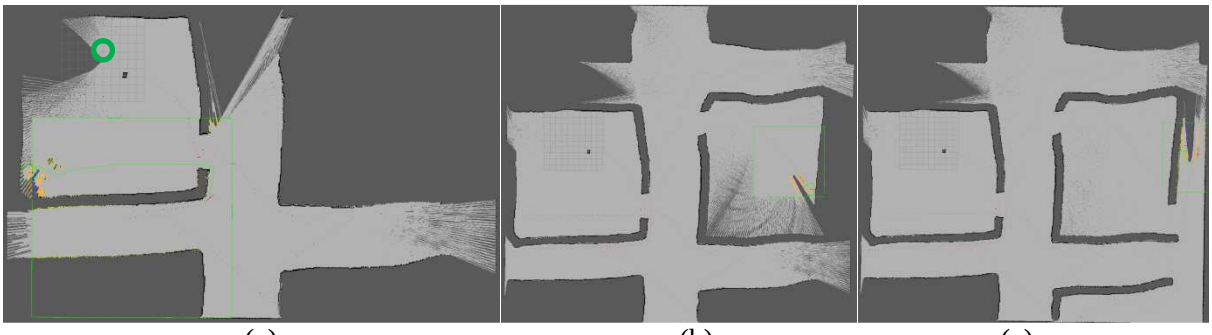
(a) (b) (c)

Abb 5.5: Explorationsergebnis 15_5_2_0
(a) nach 10 min; (b) nach 20 min; (c) nach 30 min.



(a) (b) (c)

Abb 5.6: Explorationsergebnis 3_3_5_1
(a) Nach 10 min; (b) nach 20 min; (c) nach 25 min.



(a) (b) (c)

Abb 5.7: Explorationsergebnis 6_3_4_1
(a) nach 10 min; (b) nach 20 min; (c) nach 30 min.

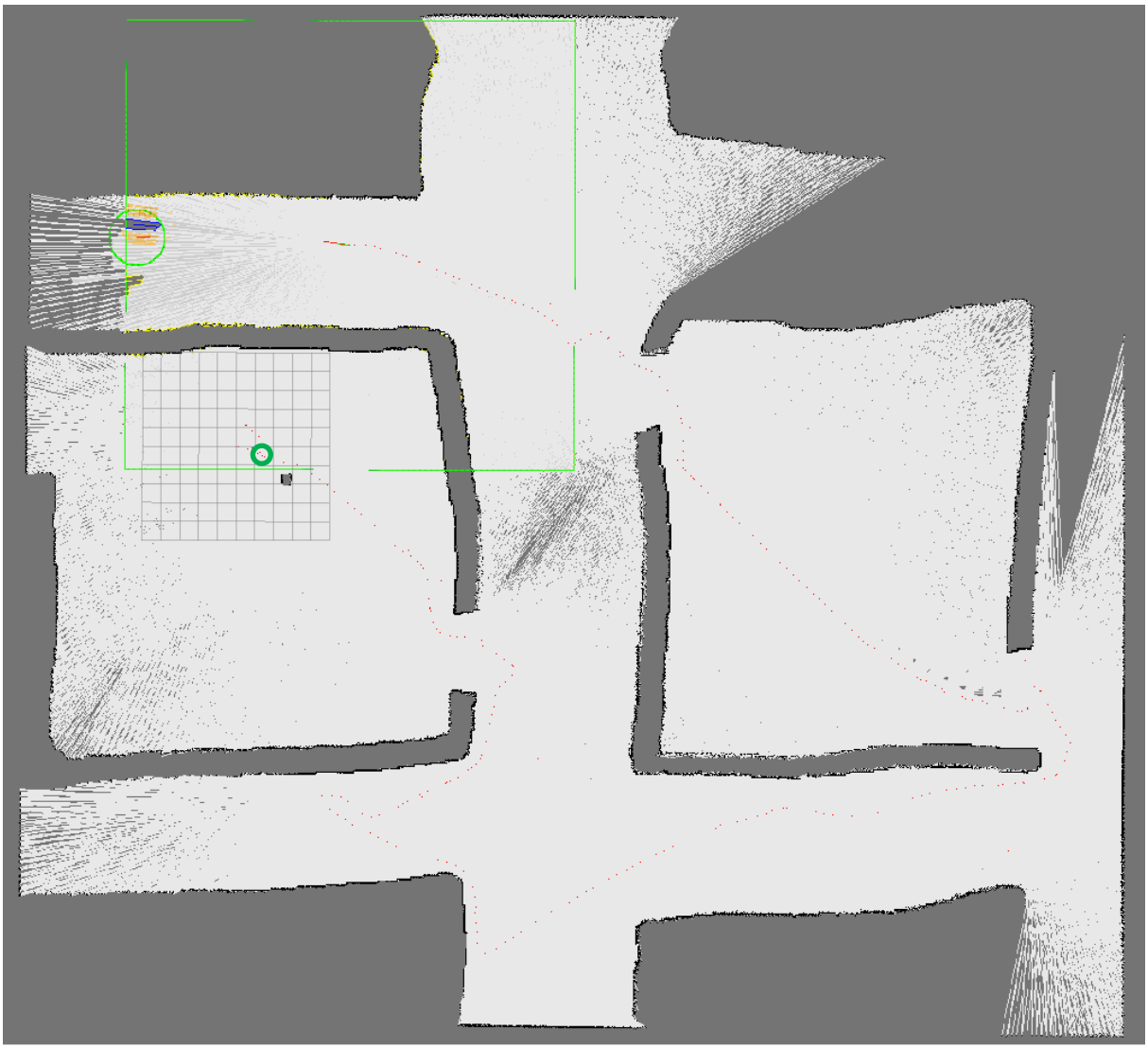


Abb 5.8: Explorationsergebnis 3_3_5_0 nach 30 min - zweiter Lauf

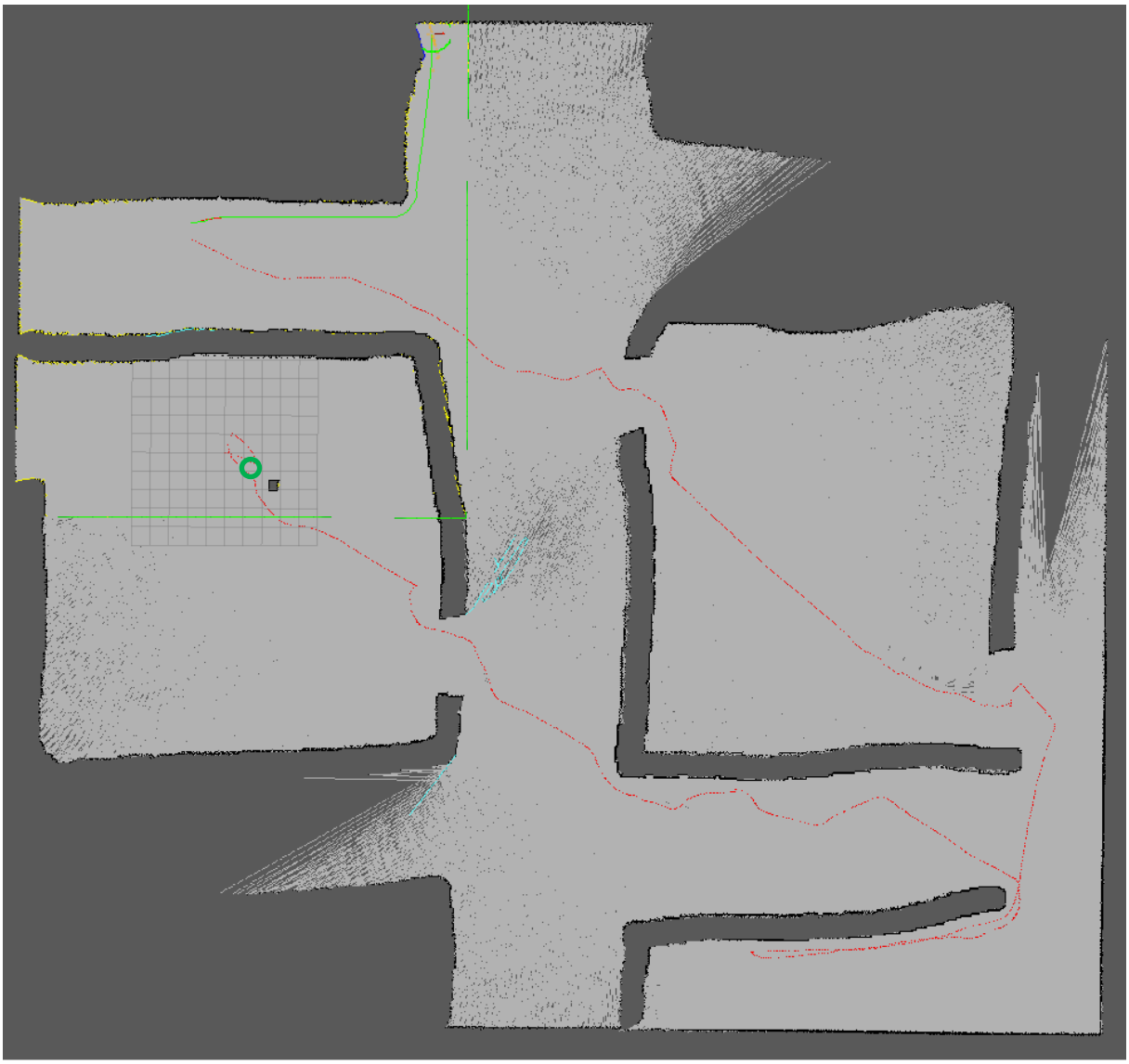


Abb 5.9: Explorationsergebnis 3_3_5_0 nach 30 min - dritter Lauf

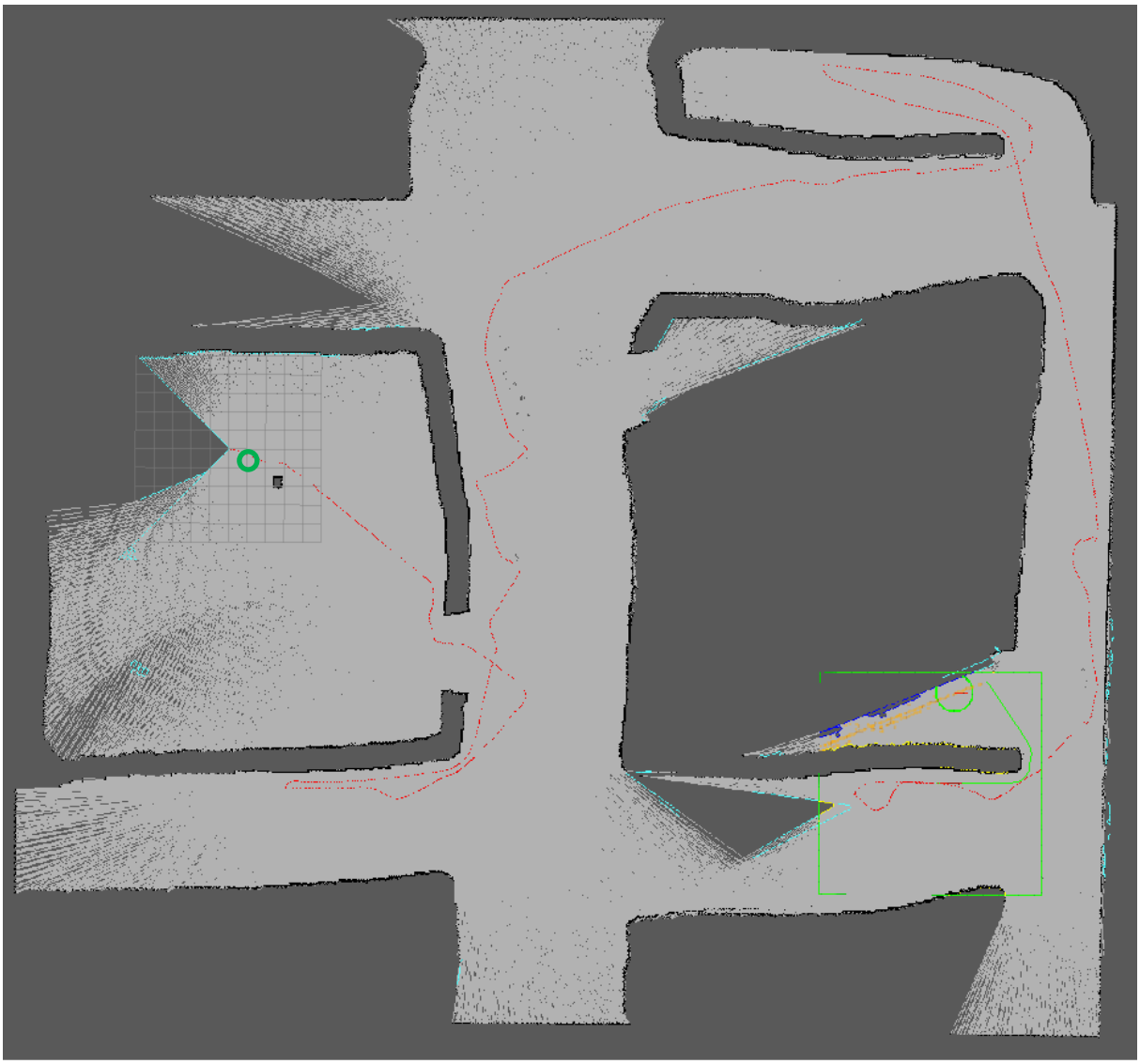


Abb 5.10: Explorationsergebnis 6_3_4_0 nach 30 min - zweiter Lauf

Abb 5.3 bis Abb 5.10 zeigen Ergebnisse in simulierter Umgebung basierend auf Abb 5.2. (rot) zurückgelgter Weg; (orange) *portentialGoalArea; (*blau) beste *frontierRegion;* (gelb) *frontierRegions;* (grün) Suchbereich; (cyan) *whitelist;* (grüner Kreis) Startpunkt; Verwendet wurde ein 270° Scanner mit 250 Samples und einer Reichweite von 20m. Das zu sehende Gitter hat eine Ausdehnung von 10m x 10m.

Konfiguration		Zeit [min]	Zyklen	Distanz [m]	Radienverteilung					**Exploriert [%]**
					3	6	9	12	15	
radius [m]	**3**	10	37	43,42	31	13	1	1	-	**27,1**
stepping [m]	**3**	20	68	98,12	40	20	10	9	1	**36,47**
attempts	**5**	30	97	149,4	59	29	14	14	4	**52,23**
driveToGoal	**0**	40	131	208,24	86	30	21	19	6	**68,83**

Konfiguration		Zeit [min]	Zyklen	Distanz [m]	Radienverteilung				**Exploriert [%]**
					6	9	12	15	
radius [m]	**6**	10	44	59,06	40	4	2	-	**30,12**
stepping [m]	**3**	20	78	121,9	67	8	9	2	**37,68**
attempts	**4**	30	115	185,4	108	13	10	2	**55,98**
driveToGoal	**0**								

Konfiguration		Zeit [min]	Zyklen	Distanz [m]	Radienverteilung	**Exploriert [%]**
					15	
radius [m]	**15**	10	46	75,4	46	**33,76**
stepping [m]	**5**	20	96	136,7	96	**42,57**
attempts	**2**	30	144	221	144	**64,98**
driveToGoal	**0**					

Konfiguration		Zeit [min]	Verarbeitete Zyklen	Gesamt Zyklen	Distanz [m]	Radienverteilung					**Exploriert [%]**
						3	6	9	12	15	
radius [m]	**3**	10	39	48	69,7	45	7	-	1	-	**35,44**
stepping [m]	**3**	20	72	94	165,5	91	8	1	1	1	**63,79**
attempts	**5**	25	73	101	184,07	91	9	1	2	1	
driveToGoal	**1**										

Konfiguration		Zeit [min]	Verarbeitete Zyklen	Gesamt Zyklen	Distanz [m]	Radienverteilung				**Exploriert [%]**
						6	9	12	15	
radius [m]	**6**	10	24	62	88,6	29	-	2	1	**29,19**
stepping [m]	**3**	20	42	137	279,9	49	3	2	1	**62,23**
attempts	**4**	30	62	210	375,9	68	4	4	1	**71,41**
driveToGoal	**1**									

Tabelle 5.1: Verschiedene Explorationsszenarien

Auflistung der untersuchten Explorationsszenarien und –strategien. Gemessen wurde in zehn Minuten Intervallen. Wichtig an dieser Stelle sind die zurückgelegte Strecke und der explorierte Anteil der Karte.

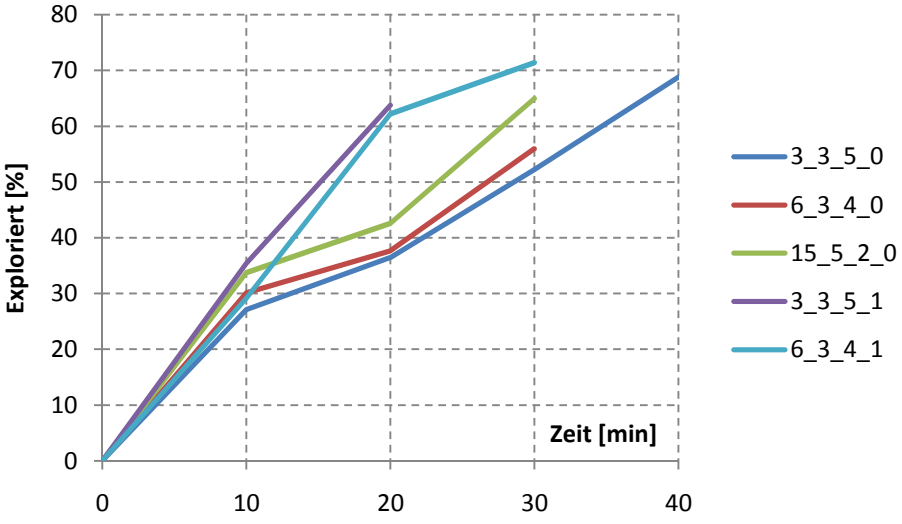


Diagramm 5.1: Explorierter Anteil der Karte

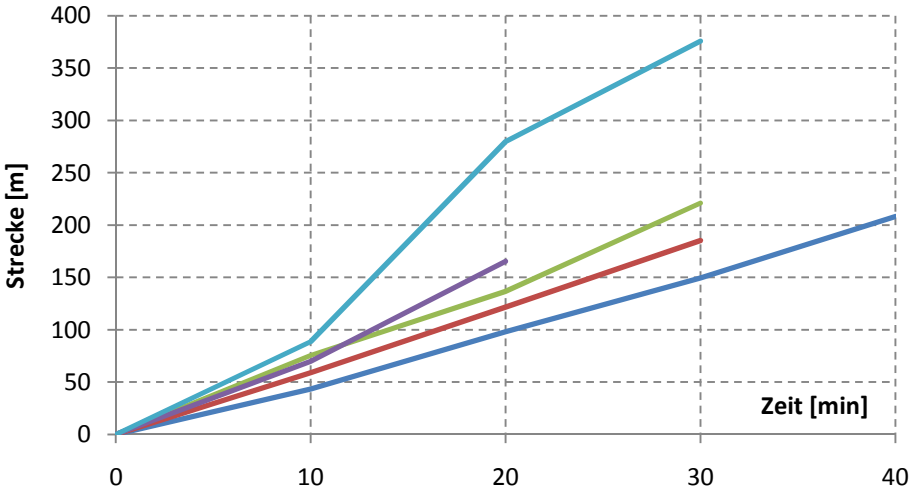


Diagramm 5.2: Zurückgelegte Strecke

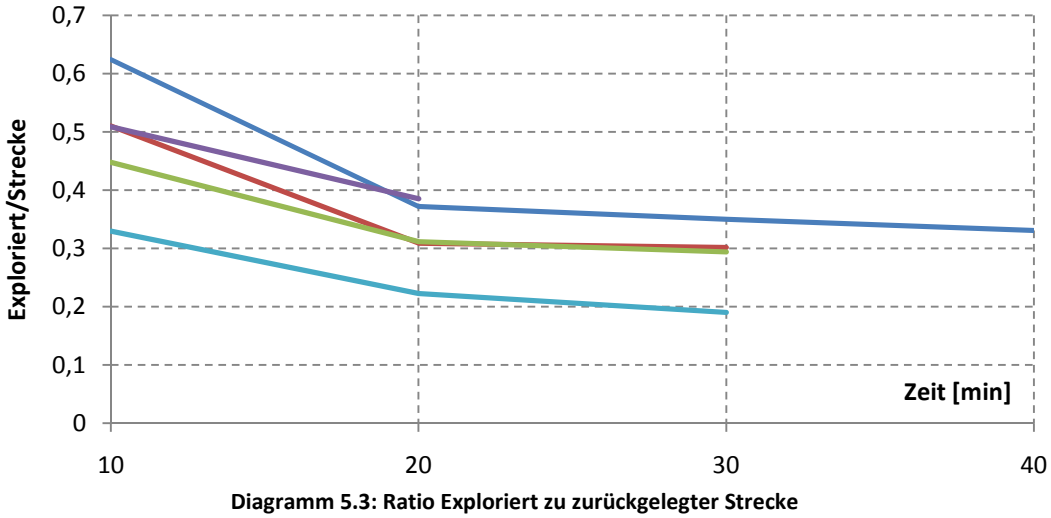


Diagramm 5.3: Ratio Explored zu zurückgelegter Strecke

Diagramm 5.1, Diagramm 5.2 und Diagramm 5.3 stellen die zuvor aufgeführten Ergebnisse grafisch aufgearbeitet dar.

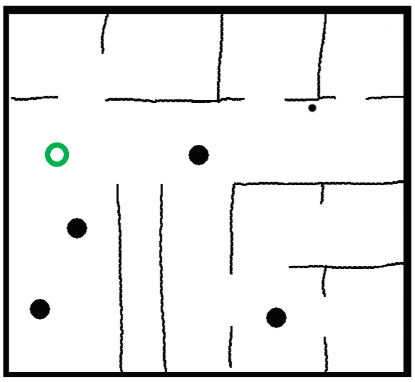


Abb 5.11: Simulierte Umgebung 2

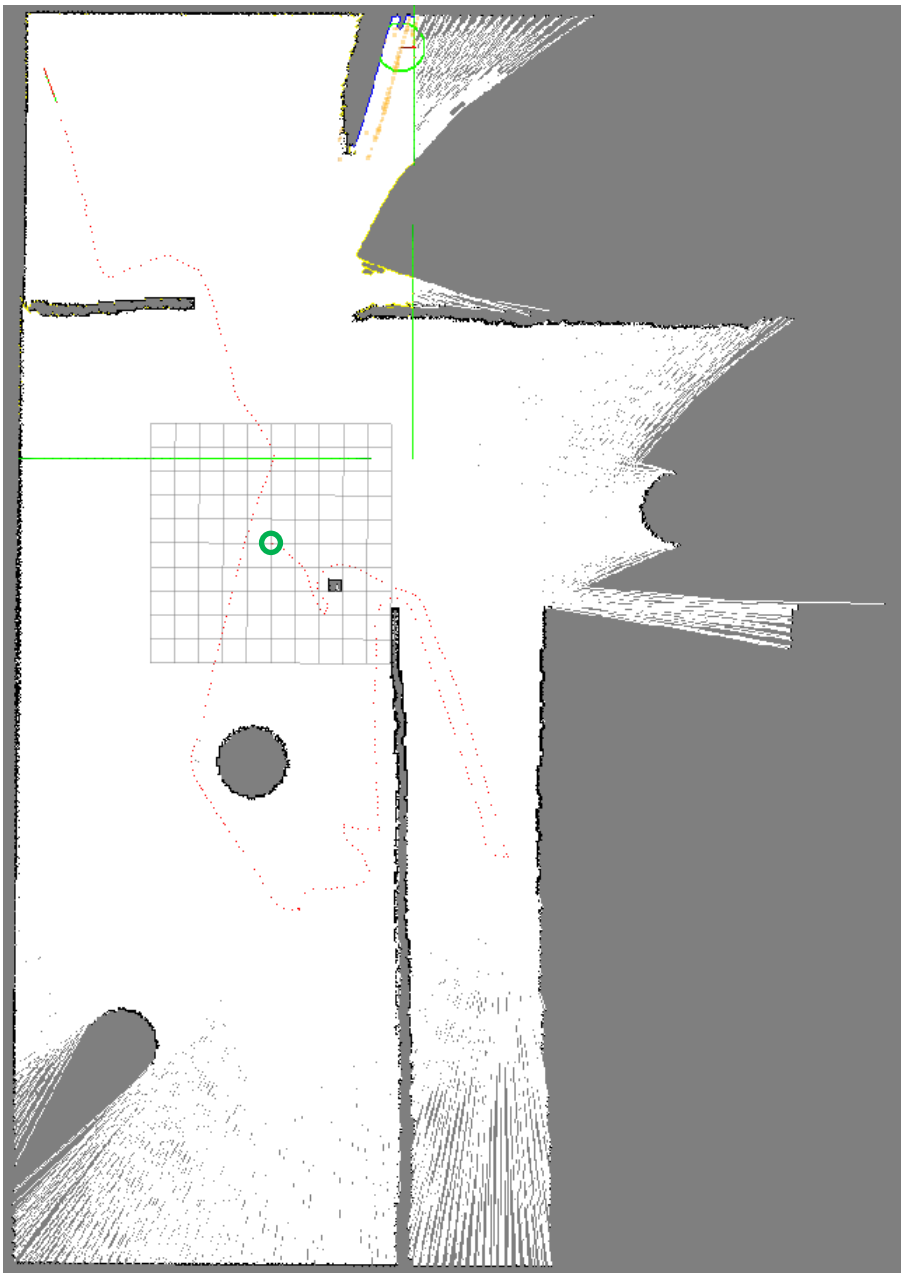


Abb 5.12: Explorationsergebnis 3_3_5_1 nach 10 min

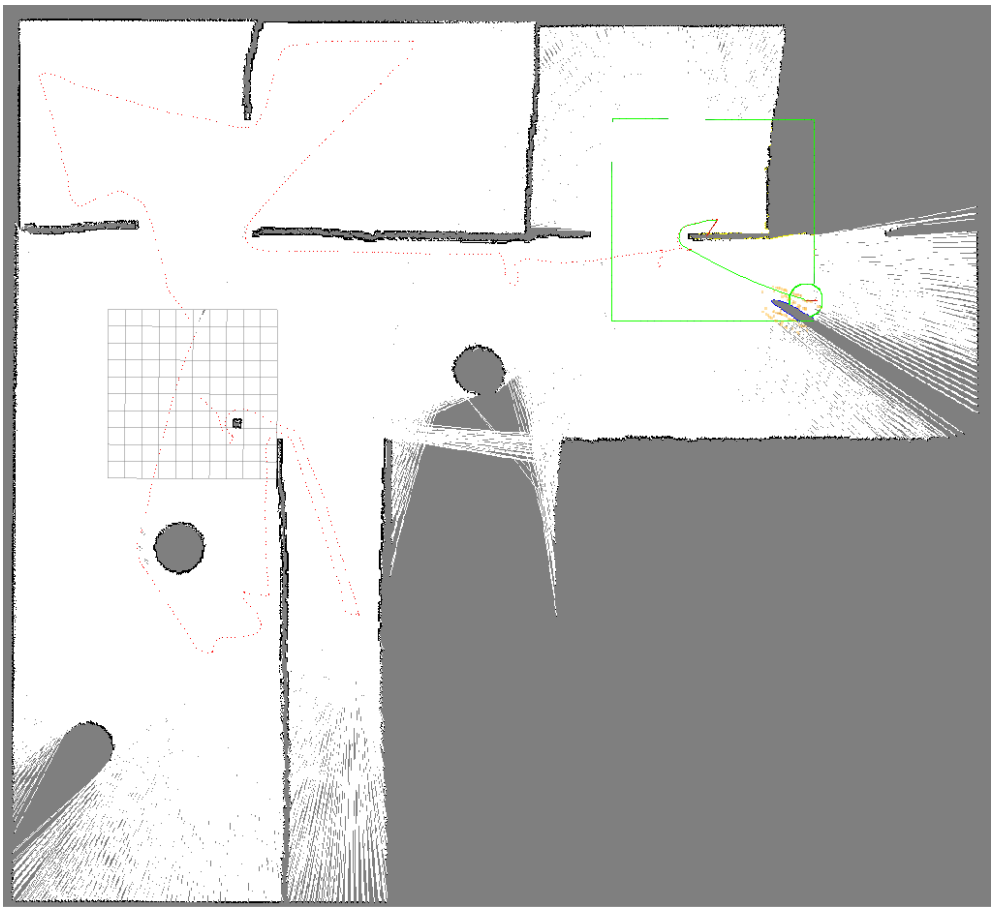


Abb 5.13: Explorationsergebnis 3_3_5_1 nach 20 min

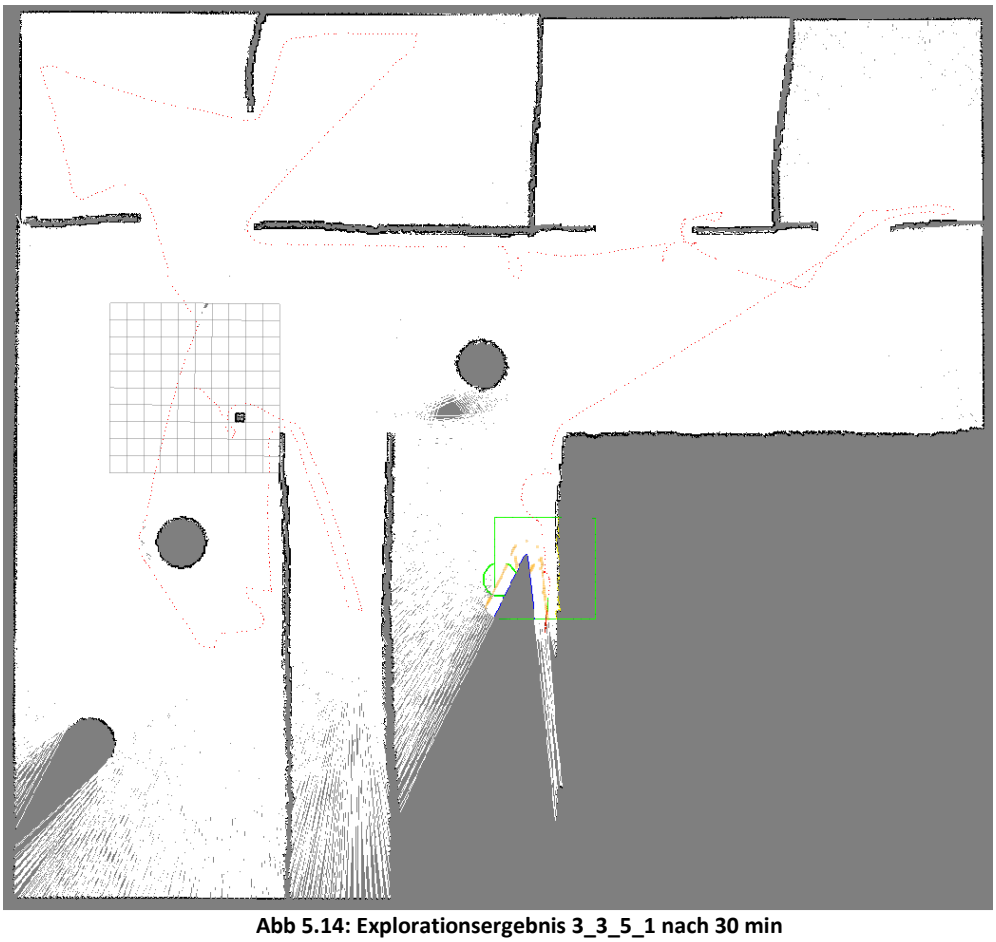


Abb 5.14: Explorationsergebnis 3_3_5_1 nach 30 min

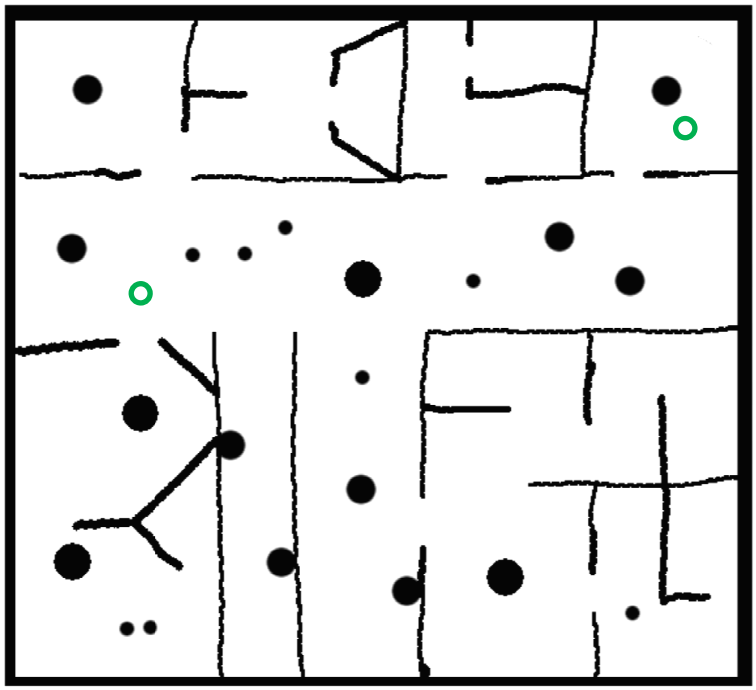


Abb 5.15: Simulierte Umgebung 3

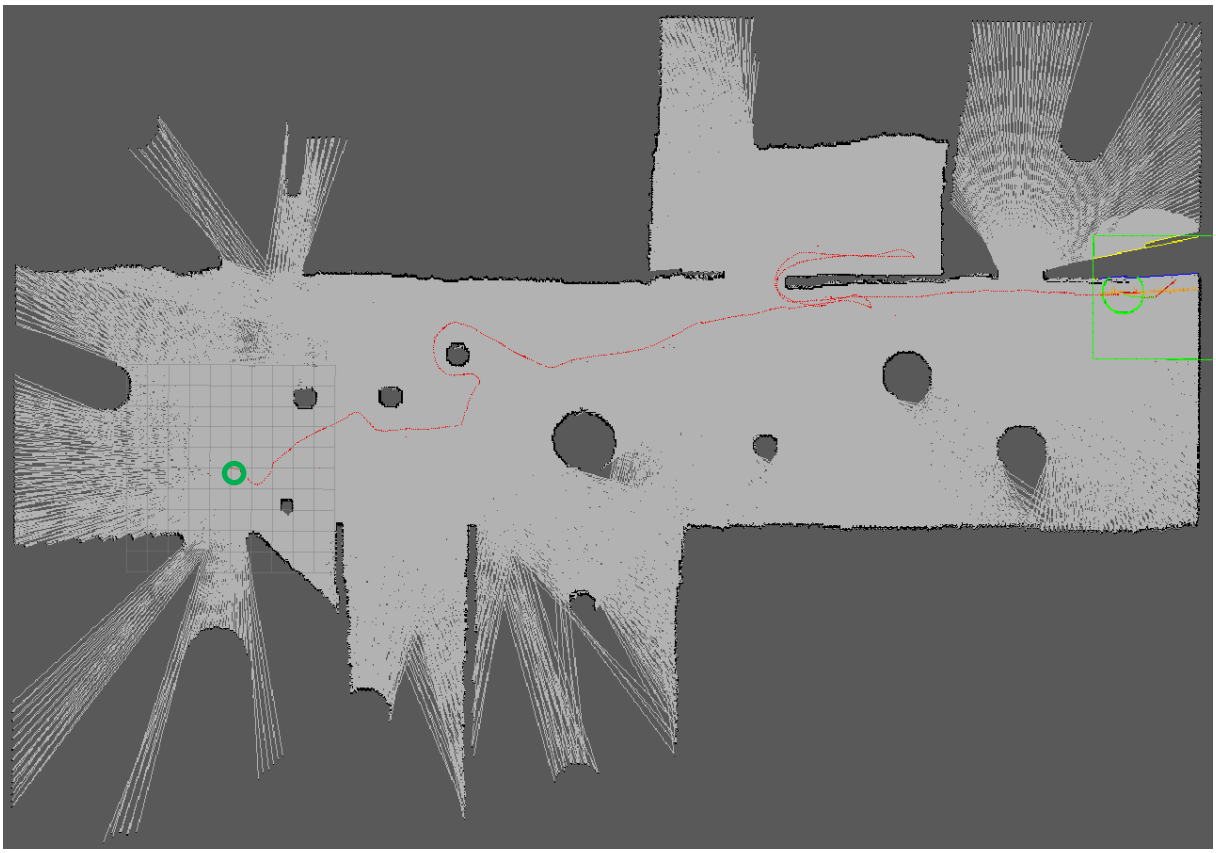


Abb 5.16: Explorationsergebnis 3_3_5_1 nach 10 min

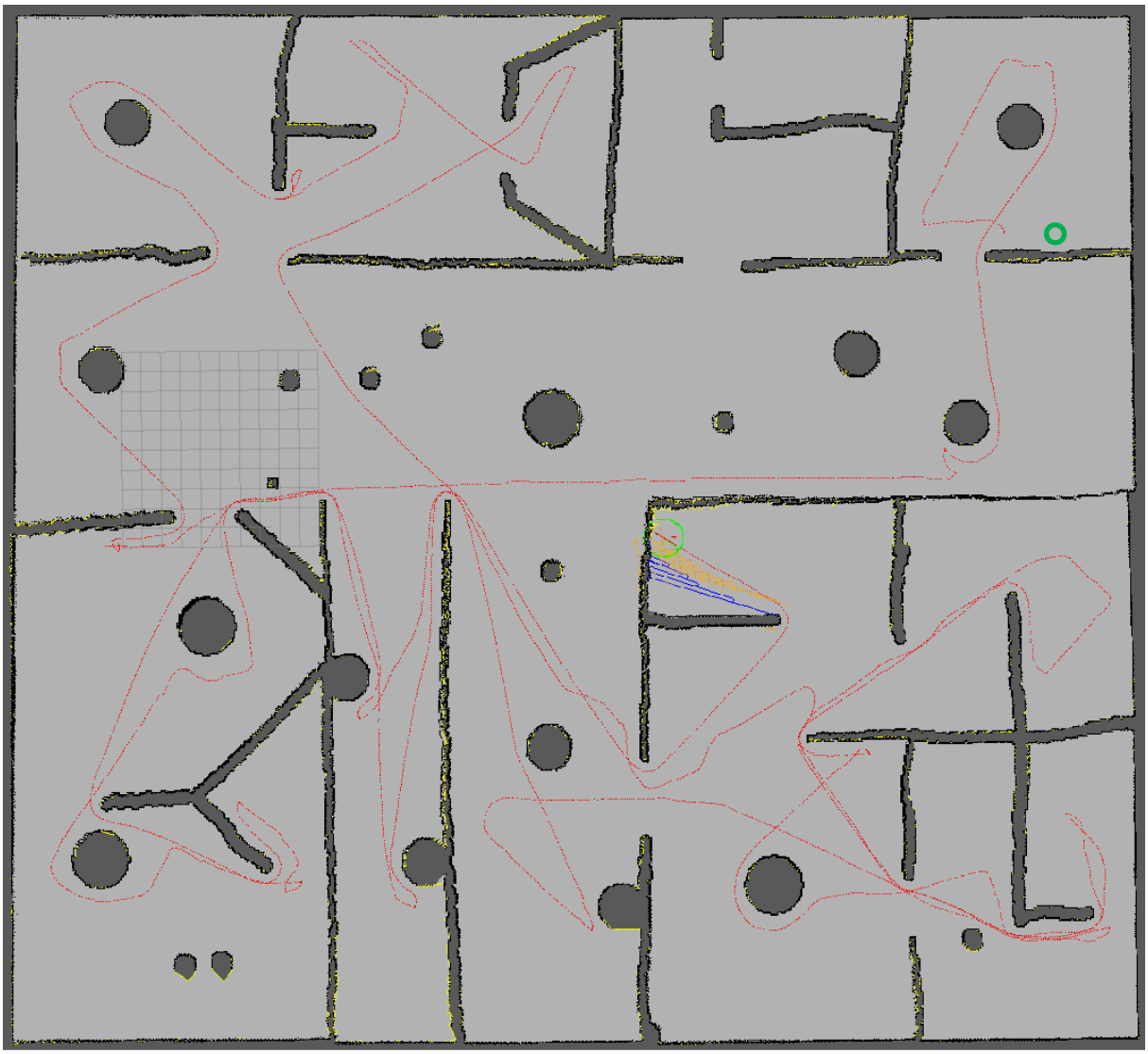


Abb 5.17: Explorationsergebnis 3_3_5_1 100%

(rot) zurückgelgter Weg; (orange) ***portentialGoalArea;*** **(blau) beste** ***frontierRegion;*** **(gelb)** ***frontierRegions;*** **(grün) Suchbereich; (cyan)** ***whitelist;*** **(grüner Kreis) Startpunkt; Verwendet wurde ein 270° Scanner mit 250 Samples und einer Reichweite von 20m. Das zu sehende Gitter hat eine Ausdehnung von 10m x 10m. In Abb 5.17 ist zu erkennen, dass die berechneten Wege bei Explorationsbeginn noch effizient sind, bei zunehmender Aufdeckung aber an Effizienz verlieren, da Wege doppelt gefahren werden.**

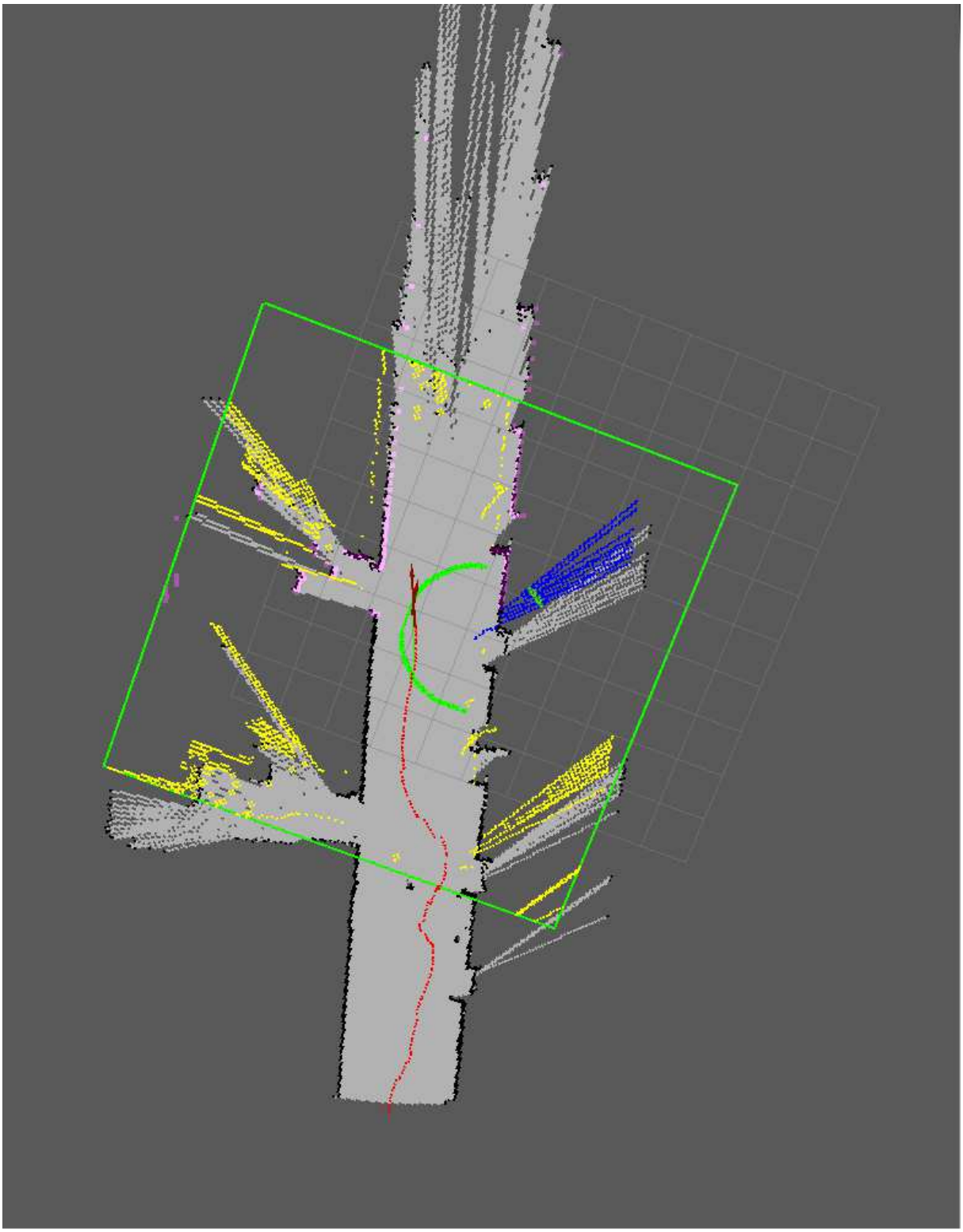


Abb 5.18: Explorationsergebnis in echter Umgebung

Versuch einen Flur im 2.Stock von Gebäude 38 zu explorieren. Auf der rechten Seite sind Türen zu angrenzenden Büroräumen zu erkennen. Auf der linken Seite sind Fenster zu sehen, die bis auf den Boden reichen.

6. Zusammenfassung

Das Problem der autonomen Exploration ist aktuell viel diskutiert und besteht aus mehreren Teilgebieten (siehe Abb 1.1). Für jedes dieser Teilgebiete bedarf es einer effizienten sowie effektiven algorithmischen Umsetzung und einer Zusammenführung der einzelnen Lösungen.
Die von Yamauchi eingeführten *frontiers* [YAM97, YAM98] liefern hierbei einen Lösungsansatz für das Teilproblem der Exploration. Bis heute entstehen aus davon abgeleiteten Variationen situationsbezogene Verfahren und Beiträge zur autonomen Exploration.
Im Gegenzug zu zufälliger Exploration oder dem Folgen von Wänden bietet diese Technik selbst in Umgebungen mit zufällig verteilten Hindernissen und Wänden eine stabile Möglichkeit um autonom zu explorieren.
Auch in dieser Ausarbeitung wird ein Ansatz vorgestellt, welcher *frontiers* zugrundelegend autonome Exploration ermöglicht. Hierfür wird eine genauere Unterteilung zwischen *frontierCells* und *frontierRegions* vorgenommen. *frontierCells* sind f-Space Zellen, die an u-Space angrenzen. *frontierRegions* werden durch eine miteinander verbundene Menge von *frontierCells* gebildet. Es wird desweiteren ein Filtermechanismus entwickelt, um unabhängig von sensorischen Ungenauigkeiten *frontiers* berechnen zu können. Kernpunkte sind hierbei das eingeführte Nachbarschaftssummenintervall [NSI], welches als Filterkriterium dient und ein iterativ ausgeführter Filtermechanismus, welcher sich einer Liste mit potentiellen Filterkandidaten bedient und diese in jedem Iterationsschritt aktualisiert. Desweiteren wird im Gegensatz zu vielen anderen Vorschlägen nicht auf der vollständigen Karte nach *frontiers* gesucht, sondern in iterativ erhöhten Radien um den Roboter herum. Das Finden möglichst gewinnbringender Ziele im Sinne einer effizienten Exploration stellt einen weiteren Aspekt dieser Arbeit dar. Um dies zu gewährleisten werden Kriterien herausgearbeitet, die eine qualitative Einschätzung von *frontiers* zulassen. Diese Kriterien werden gewichtet addiert, um dadurch qualitativ hochwertige Ziele finden zu können. Es wird außerdem eine Technik vorgestellt, welche die Verwaltung von Zielen ermöglicht. So ist es zum Beispiel möglich Ziele für spätere Explorationszwecke zu speichern oder Ziele zu sperren. Eine ausführliche Abhandlung der Leistungsfähigkeit wird sowohl theoretisch anhand von Laufzeitanalysen vollzogen als auch in Experimenten dargelegt.

7. Ausblick

Bei der Erörterung und Umsetzung der vorliegenden Aufgabenstellung wurden viele Lösungswege bedacht, Verfahren gegeneinder abgewogen und nützliche Komponenten entwickelt. Darauf aufbauend ist es stets möglich die Leistung zu erhöhen oder weitere Features umzusetzen. Im Folgenden werden in Kürze Optimierugnsszenarien durchgespielt, mögliche zusätzliche Features vorgestellt und ein Ausblick in die Verwendbarkeit der vorgestellten Lösung anhand anderer Projekte versucht.

7.1. Algorithmische Erweiterungen

Ausgehend von großen Karten und dem Umstand, dass sich die Fragestellung in einer Echtzeitumgebung beweget ist die Performance der einzelnen Teilsysteme ein elementarer Bestandteil der algorithmischen Entwicklung.

7.1.1. Adaptive Auflösung in Suchbereich

Verringert man die Auflösung in einem Kartenabschnitt derart, dass die Größe einer Zelle ungefähr den Ausmaßen des Roboters entspricht [KK12], so verringert sich die Anzahl der Zellen quadratisch. Es gilt festzustellen, dass der Roboter einen gewissen Raum um sich herum zum Navigieren benötigt, wodurch die lokale Anpassung der Auflösung keinen großen Qualitätsverlust im Sinne der Navigation oder des Findens von *frontierRegions* mit sich bringt. Für die Skalierung sind eine Metrik von Nöten, sowie die Betrachtung der benötigten Laufzeit. Letztendlich lässt sich dann der Aufwand der Skalierung mit dem Gewinn einer reduzierten Zellanzahl bei weiteren Schritten in Bezug setzen.

7.1.2. Verbessertes iteratives Suchen nach *frontierRegions*

Das in dieser Arbeit vorgeschlagene Verfahren erhöht solange iterativ den Suchbereich bis *frontierRegions* gefunden werden, oder bis andere Bedingungen greifen und fallback Strategien durchgeführt werden. Bisher wird in jeder Iteration der komplette Suchbereich mit d^2 Zellen gefiltert und anschließend nach *frontierRegions* gesucht. Eine deutliche Optimierung wäre durch das Verwenden der bereits in der vorhergehenden Iteration berechneten Informationen möglich. Die Anzahl der Zellen pro Iteration lässt sich somit durch

$$d^2 - (d - 2s)^2 = 4ds - 4s^2 \mid d = 2r \; und \; s = Schrittgröße \qquad \text{eq xxvii}$$

berechnen und ist somit nicht mehr quadratisch, sondern linear in Abhängigkeit von d. Speziell das Glätten des Suchbereichs als auch das Suchen nach *frontierRegions* ließe sich dadurch beschleunigen.

7.1.3. Situationsbezogene Anpassung der Explorationsstrategie

Tests haben gezeigt, dass veränderte Gegebenheiten auf der Karte unter Umständen zu ungünstigen Explorationsschritten führen können. So ist in großen, offenen Gebieten ein hoch gewichtetes Richtungskriterium für eine schnelle Exploration verantwortlich, wohingegen die gleichen Parameter in einer engen, hindernisreichen Umgebung zu keinem Erfolg führen. Anlehnend an 7.3. wäre die Fähigkeit Umgebungsstrukturen zu erkennen und diese in Klassen einzuordnen ein weiterer Schritt in Richtung der autonomen Exploration. Beispielhaft wäre es denkbar eine Menge an Klassen zu definieren, die je einem bestimmten Umgebungstyp entsprechen. Anhand von definierten Umgebungsmerkmalen kann nun die Klasse mit der größten Schnittmenge ausgewählt und der darin gespeicherte Parametersatz geladen werden. Der Roboter könnte somit sein Explorationsverhalten dynamisch an die erkannte Situation anpassen und dadurch seine Erfolge maximieren.

7.1.4. Zufallskomponente

[FO05] stellt bereits einen auf *sensor-based random trees* basierenden Ansatz vor. Hierbei wird um den Roboter eine *saveRegion* berechnet und innerhalb derer eine zufällige Explorationsrichtung gewählt. Eine rein zufallsbasierte Richtungswahl schien den Autoren wenig gewinnbringend, weshalb sie die Zufälle in Richtung *frontiers* lenken. Solch ein integrierter Ansatz wäre eine denkbare Erweiterung für diese Arbeit. Vorstellbar wäre es zum Beispiel unter den besten x *frontierRegions* zufällig eine auszuwählen oder innerhalb der berechneten *potentialGoalArea* ein Ziel zufällig zu bestimmen. Bedingt durch eine Zufallskomponente könnten auftretende Navigationsprobleme durch wiederholen der aktuellen Iteration eventuell gelöst werden ohne auf komplexere fallback Strategien zurückgreifen zu müssen.

7.2. Erweiterte Sensorik

Lediglich auf horizontale Laserscans gestützt entsteht in den beschriebenen Verfahren eine zwei dimensionale Karte, welche für die Exploration verarbeitet wird. Verschiedenste Erweiterungen hierfür sind denkbar um einen Mehrwert für die autonome Exploration zu generieren. So ist eine vertikale Ausdehnung der Karte ein wichtiger Schritt, um den Ausmaßen des Roboters gerecht zu werden. Weitergehend ist die optische Erfassung der Umgebung eine wichtige Voraussetzung für die semantische Auswertung von entdeckten Hindernissen.
Es ist jedoch klar, dass mit jeder hinzugefügten sensorischen Einheit ein größeres Datenvolumen entsteht, welches verarbeitet werden muss. So resultiert eine zusätzliche Kartendimension ad hoc in einer kubischen Laufzeit.

7.3. Semantische Erweiterungen

Die im Rahmen dieser Arbeit entstanden Algorithmen befassen sich zumeist direkt mit der Aufgabenstellung. An einigen Stellen scheint eine tiefer gehende Untersuchung der Problematik aber sinnvoll. So wird unter anderem den berechneten Daten nur oberflächlich Bedeutung zuteil (o-Space/u-Space/f-Space).
Zum Beispiel würde aber ein tiefer gehendes Verständnis des Objektes "Türe" im Sinne der Exploration einen deutlichen Mehrwert ergeben, da sich hinter einer Türe in der Regel neues Gebiet befindet. Ein möglicher Ansatz wird in [KLS+05] beschrieben.
Auch eine semantische Erfassung des Objektes Flur würde das Verfahren bereichern, da potentielle Ziele in der Mitte eines Flurs zu finden sind, und somit das Vorankommen beschleunigt würde.

7.4. Einsatzmöglichkeiten

Angefangen bei der Industrie in Fertigungsstraßen über die Haushaltshilfe bis hin zum Einsatz in Krisenregionen oder zur Exploration von Planeten lassen sich autonome Erkundungsroboter in vielerlei Hinsicht einsetzen. Es erleichtert dem Menschen die Arbeit, kann vor Gefahren schützen oder gelangt schlicht an Orte, an die ein Mensch aktuell nicht gelangen kann.
Die Exploration außen vor lassend ist ein autonomes System eine relevante Thematik in vielen Zweigen der Wissenschaft. Diese reichen von Fußballspielenden Robotern [RC], über autonome Fahrzeuge bis hin zum Einsatz für das Militär [SCJ+05].

7.4.1. Autonomes Fahren

Aktuelle Artikel und Diskussionen zeigen, dass Automobilhersteller aktiv an der Thematik autonomes Fahren arbeiten. Hierbei geht es nicht um die alleinstehende autonome Exploration, sondern vielmehr um ein selbstständig entscheidendes Fahrzeug im Ganzen.

In [ENG14] wird diese Thematik diskutiert und man kommt zu dem Schluß, dass es in den nächsten zehn Jahren Entwicklung noch viele Meilensteine zu bewältigen gilt. Datensicherheit und gesetzliche Hintergründe als Beispiel dafür. Es wird aber auch eine Prognose gewagt, die eine Verringerung der Unfallrate um circa 90% veranschlägt.

> *"The car that will take you home after you have had too much to drink is a long way off. But is that what we really want?"*
>
> Dieter Zetsche, Detroit motor show [ENG14]

[KUR14] kommt zu einem Ähnlichen Ergebnis und fügt die Frage "Wie weit und vor allem wie kann sich ein autonom fahrendes Fahrzeug vor uns Menschen etablieren?" hinzu.

Desweiteren wird auf technische Schwierigkeiten hingewiesen und eine Fusion verschiedener technischer Ansätze als unumgänglich angesehen.

> *Yet if snow is in the forecast, fancy LiDAR systems are useless – your car will be too busy scanning snowflakes, versus road signs and traffic signals.*
>
> Alberto Broggi, Italien [KUR14]

Basierend auf diesen zwei Artikeln und einigen anderen Quellen [AD14, BNZ14] ist die deutliche Bemühung unterschiedlicher Automobilhersteller zu erkennen in Richtung des autonomen Automobils zu entwickeln. Es lässt sich aber auch erkennen, dass der Weg dahin noch viele Hürden bereithält und dass die Akzeptanz in der Gesellschaft noch nicht vorhanden ist.
Im Gegensatz dazu sind Fahrassistenzsysteme weit verbreitet und bereits als Standard gefordert und umgesetzt. Diese ermöglichen jetzt schon automatisiertes und unterstütztes Fahren. Einparkhilfe, Abstandshalter und Spurassistent seien eine Auswahl davon. Diese Techniken sind noch weit entfernt vom autonomen Fahren, doch ist der erste Schritt dahin bereits getan.

7.4.2. Industrie und Militär

In [TTW+04] findet sich ein weiteres praxisbezogenens Beispiel. Hier wird gezeigt, wie ein autonom arbeitender Roboter - namentlich Groundhog - verlassene Minen an Stellen kartografiert, die für den Menschen ein zu hohes Risiko bergen. Vergleichbar mit vorliegender Arbeit wird basierend auf SLAM und wiederholten Laserscans iterativ eine Karte erstellt. Jedoch werden *goalRegions* in einer geraden Linie vor Groundhog in etwas fünf meter Entfernung definiert. Ist ein A* nicht in der Lage, einen Pfad dorthin zu finden, so wird die *goalRegion* vergrößert oder Groundhog fährt zulässig durch seine Architektur in die andere Richtung. Groundhogs chasis und Ausrüstung sind vorne und hinten identisch. Erfolgreiche Tests unter anderem in der Florence Mine in der Nähe von Burgettstown in Pennsylvania und in der Mathies Mine in der Nähe von Courtney bestätigen die Verwendbarkeit dieser Technik.

Abb 7.1: Groundhog [TTW+04]

[SCJ+05] präsentiert eine Technik, welche autonom die Trajektorie eines UAV in einem Stadtgebiet plant, um von einem Startpunkt A (Position des UAV) bis zu einem Zielpunkt B zu navigieren. Hauptaugenmerk liegt hierbei zwar mehr auf Pfadfindung als auf Exploration, jedoch wird auch hier der Aspekt des autonomen Verhaltens eines Roboters herausgearbeitet. Anhand eines Lasersensors wird eine *local obstacle map* im Umkreis des UAV erstellt. Ausgehend von einer initialen Trajektorie berechnet ein MPC (model predicitve control) Algorithmus eine kollisionsfreie angepasste Trajektorie. Eine Kostenfunktion bezogen auf die entdeckten Hindernisse und ein *gradient-search* Algorithmus liefern hierfür die benötigten Daten. Desweiteren wird die Frage nach einer geeigneten caching Strategie gestellt. Hierfür werden die Szenarien dynamische und statische Umgebung gegenübergestellt. Schlußendlich wird zusätzlich noch die Frage gestellt, unter welchen Bedingungen Hindernisse verworfen werden können. So stellen fallende Blätter dynamische Hindernisse dar, sind für die traversierung aber nicht relevant.

Abb 7.2: Berkeley UAV [SCJ+05]

Abb 7.3: Flugweg der UAV [SCJ+05]

Stichwortverzeichnis

Um dem Leser eine möglichst verständliche Lektüre zu ermöglichen stellt der Autor im Folgenden eine Liste der verwendeten Begrifflichkeiten zusammen.

free-Space
Gebiet, welches bereits erkundet wurde und frei von Hindernissen ist. In vorliegender Arbeit auch mit f-Space beschrieben. F_SPACE beschreibt die Konstante 0, welche im Zusammenhang mit occupancy grids verwendet wird.

frontierCell
Zelle, die sich in f-Space befindet und direkt an eine u-Space Zelle grenzt.

frontierRegion
Grenzbereich zwischen f-Space und u-Space. Für diese Arbeit von zentraler Bedeutung.

cartesian-Space
Gegenstück zum l-Space. Im Allgemeinen durch *pt* repräsentiert.

goalRegion
Definierter Bereich um das Ziel herum. Gewährleistet die Anwendung verschiedener Techniken.

height und width
Repräsentieren die Höhe und Breite der Karte. Mit h und w abgekürzt.

linear-Space
Vom Autor eingeführte Bezeichnung, um zwischen der linearen Indizierung von Punkten und den dazugehörigen kartesischen Punkten (vice versa) eindeutig unterscheiden zu können. Im Allgemeinen durch *i* repräsentiert.

map(i)
Abgekürzt für $map.data[i]$ um einen angenehmeren Lesefluss zu gewährleisten.

```
nav_msgs::occupancy_grid map = getMap();
return map.data[i];
```

Code 0.1: map.data

neighbours(i)
Repräsentiert alle Nachbarindizes um i in einem 3x3 Gebiet.

$$neighbours(i) = \{i \pm w \pm 1; i \pm w; i \pm 1\} \quad \text{eq xxviii}$$

neighbourValues(i)

$$neighbourValues(i) = \sum map\big(neighbours(i)\big) \quad \text{eq xxix}$$

NSI
Vom Autor eingeführte Abkürzung für ***N**achbarschafts **S**ummen **I**ntervall.* Beschreibt das Intervall, in welchem neighbourValues(i) liegen muss, um i zu glätten.

occupancy grid
Von ROS zur Verfügung gestelltes Datenmodell, um Karten speichern und bearbeiten zu können. Die Belegungswerte (F_SPACE, …) werden mittels eines Vektors repräsentiert.

occupied-Space
Erkundetes, aber belegtes Gebiet. Abgekürzt mit o-Space. $O\_SPACE = 100$.

ROS
Das **R**obot **O**perating **S**ystem ist ein open source Framework, welches eine Vielzahl an libraries, Geräte Treibern, Simulationswerkzeugen, Hardware Abstraktionen und vielem mehr zur Verfügung stellt um Entwicklern bei der Erstellung von Roboter Software zu unterstützen.

SLAM
Simultaneous **L**ocalization **a**nd **M**apping ist ein Verfahren, mit der ein Roboter iterativ eine Karte erstellen und gleichzeitig seine Position innerhalt dieser abschätzen kann [GTS+07].

unknown-Space
Zellen, welche keinen definierten Belegungszustand haben. Mit u-Space abgekürzt. $U\_SPACE = -1$

Abbildungsverzeichnis

Formelverzeichnis

Sonstige Verzeichnisse

Literaturverzeichnis

[AD14] Autonomous Driving. 2014. Link: <http://autonomous-driving.we-conect.com/en/>.

[BMF+00] Burgard, W.; Moors, M.; Fox, D.; Simmons R. und Thrun, S.: *Collaborative Multi-Robot exploration*. In: Proceedings of the IEEE Int. Conf. on Robotics and Automation. San Francisco, CA, USA, 2000. 476-481.

[BNZ14] Mercedes-Benz. *Autonomous Long-Distance Drive.* 2013. Link: <http://www5.mercedes-benz.com/en/innovation/autonomous-long-distance-drive-research-vehicle-s-500-intelligent-drive/>.

[DUQ04] Duque-Antón, M.: *Tipps zum Anfertigen einer Diplomarbeit.* In: FH Kaiserslautern, 2004. Link: <http://www.fh-kl.de/~duque/files/04ws/Anleitung.pdf>.

[ENG14] English, A.: *Autonomous cars – is this the end of driving?* In: The Telegraph. Link: <http://www.telegraph.co.uk/motoring/road-safety/10570935/Autonomous-cars-is-this-the-end-of-driving.html> 12.01.2014.

[ESP12] Esponda, M.: *Analyse von Algorithmen: Die O-Notation.* In: Freie Universität Berlin, 2012. Link: <http://w3.inf.fu-berlin.de/lehre/SS12/ALP2/slides/V6_Rekursion_vs_Iteration_ALP2.pdf>.

[FO05] Freda, L. und Oriolo, G.: *Frontier-based probabilistic strategies for sensor-based exploration.* In: Proceedings of the IEEE Int. Conf. on Robotics and Automation. Barcelona, Spain, 2005. 3881-3887.

[GKC03] Grabowski, R.; Khosla, P. und Choset, H.: *Autonomous Exploration via Regions of Interest.* In: Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems. Las Vegas, Nevada, USA, 2003. 1691-1696.

[GTS+07] Grisetti, G.; Tipaldi G.D.; Stachniss, C.; Burgard, W. und Nardi, D.: *Fast and accurate SLAM with Rao-Blackwellized particle filters.* In: Robotics and Autonomous Systems 55. 2007. 30-38.

[GL02] Gonzalez-Banos, H. H. und Latombe, J.: *Navigation Strategies for Exploring Indoor Environments.* In: The International Journal of Robotics Research. 2002. 829-848.

[IPA14] Fraunhofer IPA. Link: <http://www.ipa.fraunhofer.de/Navigation.25.0.html>.

[JOH07] Joho, Dominik: *Exploration für mobile Roboter unter Verwendung dreidimensionaler Umgebungsmodelle*. In: Diplomarbeit, Albert-Ludwigs-Universität Freiburg, 2007. Link: <http://ais.informatik.uni-freiburg.de/publications/papers/joho07diplom.pdf>.

[KK12] Keidar, M. und Kaminka G. A.: *Robot Exploration with Fast Frontier Detection: Theory and Experiments.* In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS). Valencia, Spain, 2012. 113-120.

[KLS+05] Kleinlützum, K.; Luksch, T.; Schmidt, D. und Berns, K.: *Selbstständige Exploration einer abstrakten topologiebasierten Karte für die autonome Exploration.* In: Technische Universität Kaiserslautern, 2005. Link: < https://agrosy.informatik.uni-kl.de/fileadmin/Literatur/Kleinluetzum05a.pdf>.

[KUR14] Kurczewski, N.: *Autonomous drive expert believes cars will be better drivers than humans.* In: Daily News. Link: <http://www.nydailynews.com/autos/buyers-guide/cars-safer-drivers-self-driving-vehicles-eliminate-traffic-accidents-article-1.1595616> 29.01.2014.

[LMB+02] Lacroix, S.; Mallet, A.; Bonnafous, D.; Bauzil, G.; Fleury, S.; Herrb, M. und Chatila, Raja: *Autonomous Rover Navigation on Unknown Terrains: Functions and Integration.* In: The International Journal of Robotics Research. 2002. 917-942.

[LR02] Levi, P. und Rembold U.: *Einführung in die Informatik: für Naturwissenschaftler und Ingenieure.* Carl Hanser Verlag GmbH & Co. KG. 2002.

[MH80] Marr, D. und Hildreth, E.: *Theory of Edge Detection.* In: Proceedings of the royal society. 1980. 187-217.

[MSW01] Moorehead, S. J.; Simmons, R. und Whittaker, W. L.: *Autonomous Exploration Using Multiple Sources of Information.* In: IEEE Int. Conf. on Robotics and Automation. Seoul, Korea, 2001.

[NÜC02] Nüchter, Andreas: *Autonome Exploration und 3D-Modellierung der Umgebung eines Roboters.* In: Diplomarbeit, Rheinische Friedriche-Wilhelms-Universität Bonn, 2002. Link: <http://plum.eecs.jacobs-university.de/download/it2002.pdf>.

[RC14] Robo Cup. Link: <http://www.robocup.org>.

[ROS14] Robot Operation System. Link: <http://wiki.ros.org/>.

[RUO13] Ruohonen, K.: *Graph Theory.* In: Tampere University of Technology, Finland, 2013. Link: <http://math.tut.fi/~ruohonen/GT_English.pdf>

[SCH05] Schöning, U.: *Logik für Informatiker.* Heidelberg; Berlin: Spektrum Akademischer Verlag GmbH. 2005.

[SCH13] Schiefer, B.: *Algorithmische Graphentheorie: Zusammenhangskomponenten.* In: Fachhoschule Kaiserslautern, 2013 Link: <http://www.informatik.fh-kl.de/~schiefer/lectures/download/GT_3.pdf>.

[SCJ+05] Shim, D. H.; Chung, H.; Jin Kim, H. und Sastry, S.: *Autonomous Exploration in Unknown Urban Environments for Unmanned Aerial Vehicles.* In: AIAA GN&C Conference. 2005.

[TTW+04] Thrun, S.; Thayer, S.; Whittaker, W.; Baker, C.; Burgard, W.; Ferguson, D.; Hähnel, D.; Montemerlo, M.; Morris, A.; Omohundro, Z.; Reverte, C. und Whittaker, W.: *Autonomous Exploration and Mapping of Abandoned Mines.* In: IEEE Robotics & Automation Magazine. 2004. 79-91.

[WAN06] Wang, Y.: *Image Filtering: Noise Removal, Sharpening, Deblurring.* In: Polytechnic University Brooklyn, 2006. Link: <http://eeweb.poly.edu/~yao/EE3414/image_filtering.pdf>.

[WOL14] Wolf, J.: *C++ Das umfassende Handbuch.* In: Galileo Computing. 2014.

[YAM97] Yamauchi, B.: *A frontier-based approach for autonomous exploration.* In: Computational Intelligence In Robotics and Automation. Monterey, CA, USA, 1997.

[YAM98] Yamauchi, B.: *Frontier-based exploration using multiple robots.* In: Proceedings of the second international conference on Autonomous agents. Minneaolis, MS, USA, 1998.

Alle Links wurden zuletzt am 20. März 2014 überprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, 25.03.2014