Universität Stuttgart
Institut für Fördertechnik
und Logistik

IFT

Universität Stuttgart

Fakultät Konstruktions-, Produktions- und Fahrzeugtechnik

Institut für Fördertechnik und Logistik

# Implementierung und simulativer Vergleich von dynamischen Pfadplanungsalgorithmen für autonome Transportfahrzeuge

# Implementation and Simulative Comparison of Dynamic Path Planning Algorithms for Autonomous Transport Vehicles

BACHELORARBEIT

24.09.2021

| | |
|---|---|
| Verfasser | Lorenz J. Kolb |
| | Mat.-Nr.: 3406177 |
| Prüfer | Univ.-Prof. Dr.-Ing. Robert Schulz |
| Betreuerin | Carolin Brenner |

# Eidesstattliche Erklärung

„Hiermit erkläre ich, Lorenz Kolb, ehrenwörtlich, dass ich entsprechend der PO Bachelor Maschinenbau § 25 Abs. 7 und der PO Master Maschinenbau § 23 Abs. 7 und § 24 Abs. 8 die vorliegende Bachelorarbeit selbstständig ohne fremde Hilfe verfasst habe, keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe. Des Weiteren war die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahren und wurde weder vollständig noch in Teilen bereits veröffentlicht. Das elektronische Exemplar stimmt mit allen anderen Exemplaren überein.“

Stuttgart, den 24.09.2021

Lorenz Kolb

# Kurzfassung

Da der Materialfluss in Fabriken für jedes Produkt immer individueller wird sind neue, flexible Logistikkonzepte nötig. Flächenbewegliche autonome Transportfahrzeuge sind ein Ansatz, um diese Aufgabe zu erfüllen. Für die Navigation dieser Fahrzeuge sind Pfadplanungsalgorithmen nötig.

Im Rahmen dieser Arbeit sollen verschiedene Pfadplanungsalgorithmen verglichen werden. Ziel ist es einen Algorithmus auszuwählen, der dynamische Pfadplanungsaufgaben bestmöglich erfüllt. Hierfür wurden Quadtrees und Framed Quadtrees in Kombination mit A* und D* Lite, RRT* und RRT$^X$ betrachtet. Somit wurden Vertreter der suchbasierten Algorithmen mit Vertretern der stichprobenbasierten Algorithmen verglichen.

Zuerst wurden die zuvor genannten Algorithmen in Matlab implementiert. In verschiedenen Simulationsexperimenten wurden diese anschließend verglichen. Als Vergleichsparameter wurden die Pfadlänge, die Rechenzeit und die Glattheit des Pfades gewählt. Die Simulationsexperimente waren dynamische und statische Pfadplanungen auf verschiedenen Karten mit unterschiedlichen Größen. Durch die Variation der Kartengröße und durch Erzwingen von größeren und kleineren Neuplanungen wurden Auswirkungen der Zeitkomplexität untersucht.

Die Simulationen ergaben, dass der RRT$^X$ Algorithmus bei vertretbarer Zeit die kürzesten und glattesten Pfade liefert. Außerdem lässt sich die Planungsdauer von RRT$^X$ durch teilweise gleichzeitige Planung und Pfadausführung enorm verringern. Damit ist RRT$^X$ der vielversprechendste Algorithmus und sollte in weiteren Arbeiten vor allem in Bezug auf nichtholonome Bindungen, simultane Pfadplanung und Pfadausführung und auf hybride Anwendung mit anderen Algorithmen untersucht werden.

**Stichwörter:** Dynamische Pfadplanung, RRT$^X$, RRT*, RRT, A*, D* Lite, Quadtrees, Framed Quadtrees, simulativer Vergleich

# Abstract

Due to the individualisation of the flow of material for every single product new, flexible concepts for logistics are required. Autonomous transport vehicles may be an approach to comply with these requirements. To navigate these vehicles path planning algorithms are needed.

Within the framework of this thesis, different path planning algorithms are to be compared. The main goal is to identify the algorithm, that is the most capable of solving dynamic path planning problems. For this purpose, the following algorithms were compared: Quadtrees and Framed Quadtrees in combination with A* and D* Lite, RRT* and RRT$^X$. Thus, representatives of sampling-based algorithms were compared with representatives of search-based algorithms.

As a first step, the previously listed algorithms were implemented in Matlab. After that, the algorithms were compared in various simulation experiments. The length of the path, the computation time and the average angle between path segments were used as comparative parameters. The simulation experiments were static as well as dynamic path planning problems on various maps with different sizes. Through the variation of the map size and by enforcing replanning of different lengths, effects of the runtime complexity were examined.

The results of the simulations have shown that the RRT$^X$ algorithm calculates the shortest and smoothest paths within reasonable time. Furthermore, the computation time can be drastically reduced when the planning and the execution of the path happen partially at the same time. Therefore, RRT$^X$ is the most promising algorithm. In further research especially nonholonomic constraints, simultaneous planning and execution of the path and hybrid applications with other algorithms should be investigated.

**Keywords:** dynamic path planning, RRT$^X$, RRT*, RRT, A*, D* Lite, Quadtrees, Framed Quadtrees, simulative comparison

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

| Kurzzeichen | Benennung |
|---|---|
| **FQT** | Framed Quadtree |
| **FSM** | Finite State Machine |
| **QT** | Quadtree |
| **RRT** | Rapidly-Exploring Random Tree |

# Verzeichnis der Formelzeichen

| Kurzzeichen | Einheit | Benennung |
|---|---|---|
| $\boldsymbol{c(n,u)}$ | m | Euklidischer Abstand von zwei Knoten $n$ und $u$ |
| $\boldsymbol{C}$ | - | Konfigurationsraum |
| $\boldsymbol{C_{free}}$ | - | Freier Teil des Konfigurationsraumes |
| $\boldsymbol{C_{obj}}$ | - | Teil des Konfigurationsraumes, der durch Objekte blockiert ist |
| $\boldsymbol{dim}$ | - | Dimension des Konfigurationsraumes |
| $\boldsymbol{f(n)}$ | m | Gesamtkosten eines Knotens $n$ |
| $\boldsymbol{g(n)}$ | m | Minimale Wegkosten zu einem Knoten $n$ |
| $\boldsymbol{h(n)}$ | m | Heuristik eines Knotens $n$ |
| $\boldsymbol{k_1(n), k_2(n)}$ | m | Kennzahlen zur Ordnung der open-list |
| $\boldsymbol{km}$ | m | Differenz der Kennzahlen bei Änderung des Starts |
| $\boldsymbol{lmc(n)}$ | m | Über Nachbarn berechnete minimale Wegkosten |
| $\boldsymbol{\mu(C_{free})}$ | m² | Lebesgue-Maß von $\boldsymbol{C_{free}}$ |
| $\boldsymbol{num(n)}$ | - | Anzahl der Knoten |
| $\boldsymbol{q}$ | m | Beliebige Konfiguration |
| $\boldsymbol{r}$ | m | Innerhalb dieses Radius werden Nachbarn betrachtet |
| $\boldsymbol{rhs(n)}$ | m | Über Nachbarn berechnete minimale Wegkosten |
| $\boldsymbol{\vartheta_d}$ | m³ | Volumen der Einheitskugel im d-dimensionalen Raum |

# 1. Einleitung und Problemstellung

„Any customer can have a car painted any colour that he wants so long as it is black."
Henry Ford, 1922 [For1922, 72]

Mit diesen Worten beschrieb Henry Ford, einer der größten Automobilproduzenten seiner Zeit, die Grundidee der Massenproduktion: Ein möglichst großes Produktionsvolumen wird durch minimale Produktvariation und damit einhergehende einfache Automatisierung erreicht.

In den vergangenen 100 Jahren hat sich die Automobilbranche jedoch drastisch verändert. Automobilhersteller beschränken sich längst nichtmehr auf ein Modell und eine Farbe, weshalb in vielen Werken mehrere Modelle gleichzeitig gefertigt werden müssen. Das Porsche Werk in Zuffenhausen ist mit der Fertigung von neun verschiedenen Modellvarianten Spitzenreiter in Deutschland [Hof2018, 2]. Werden auf einem Fließband verschiedene Modelle gefertigt gibt es entlang der Fertigungslinie zwangsläufig Bearbeitungsschritte, die nicht bei jedem Fahrzeug durchgeführt werden müssen. Trotzdem müssen die Fahrzeuge diese Bearbeitungsstationen auf dem Fließband durchlaufen und dort für einen Taktzyklus verweilen, bis der nächste Bearbeitungsschritt frei ist. Daraus folgen erhöhte Durchlaufzeiten für alle Modelle.

Forderungen von Politik und Gesellschaft nach neuen Antriebskonzepten im Individualverkehr verschärfen dieses Problem noch zusätzlich. Während zwei verschiedene Dieselautos sich in vielen Bearbeitungsschritten ähneln, unterscheiden sich die Baugruppen und Arbeitsschritte eines Dieselautos und eines Elektroautos enorm. Hierdurch wird eine zeitgleiche, ökonomisch sinnvolle Fließbandproduktion nahezu unmöglich.

Neben verschiedenen Fahrzeugmodellen können zunehmend auch Details, wie zum Beispiel das Interieur, von Kunde zu Kunde variieren. Dieser Trend ist nicht nur bei Automobilen, sondern auch bei vielen anderen Produkten wie Kleidung, Küchenartikeln oder Möbeln zu erkennen und wird als Mass Customization bezeichnet, was die Begriffe „Mass Production" und „Customization" vereint [Pil2006, 174]. Mass Customization ist nur dann rentabel, wenn die Kosten in der Produktion vergleichbar mit den Kosten der Massenproduktion sind. Hierfür muss der Fluss von Werkstücken, Werkzeugen, Materialien und Hilfsstoffen flexibel sein [And2017, 138]. Der Weg eines Produktes durch die Fertigung soll demnach von den nötigen Fertigungsschritten und nicht von der Struktur des Fließbandes vorgegeben werden [Hof2018, 3].

Im Extremfall von Mass-Customization würde jedes Produkt mit einer Losgröße von eins produziert werden. Folglich würde jedes Produkt auch einen individuellen Weg durch die Fertigungsstätte benötigen. Daraus resultiert eine große Menge verschiedener Transportwege, die zu komplex sind, um sie mit einem System aus Fließbändern zu

verbinden. Außerdem ist das Lohnniveau in Deutschland zu hoch, um die Logistik durch menschliche Arbeitskraft zu bewältigen.

Somit verbleiben nur flächenbewegliche fahrerlose Transportfahrzeuge (FTF) zur Bewältigung der Logistik in den Fabriken der Zukunft. Bewegen sich diese Fahrzeuge aber ausschließlich entlang von vordefinierten Spuren oder auf Schienen hat das einerseits geringe Flexibilität, aber auch eine hohe Anfälligkeit für Störungen zur Folge. Bei jeder Änderung des Fabrikaufbaus muss das Transportnetz dementsprechend angepasst werden. Wird die Spur oder Schiene durch ein Objekt oder einen Menschen blockiert steht der Transport auf dieser Route still, da die Spur nicht verlassen werden kann.

Die Navigation durch den Freiraum ist am sinnvollsten, da sie ein Maximum an Flexibilität und Skalierbarkeit bietet. Bei einer Erweiterung des Produktportfolios können zusätzliche Arbeitsschritte ohne bauliche Maßnahmen eingegliedert werden. Soll das Produktionsvolumen erhöht werden kann die Zahl der FTF äquivalent erhöht werden.

# 2. Aufgabenstellung und Zielsetzung

Die Navigation durch den freien Raum soll im Kontext von Inhouse-Logistik und Containerterminals betrachtet werden. Flächenbewegliche fahrerlose Transportfahrzeuge sollen in diesen Umgebungen jeden Punkt anfahren können. Diese Flexibilität wird ermöglicht durch Pfadplanungsalgorithmen, welche für Roboter bereits seit über 30 Jahren erforscht werden. Durch die steigende Nachfrage in der Logistik, aufgrund der in Kapitel 1 erklärten Treiber, werden diese Algorithmen zunehmend auch Gegenstand der Forschung zur Steuerung von flächenbeweglichen Transportfahrzeugen. Pfadplanungsalgorithmen sollen aufgrund der höheren Flexibilität die Navigation entlang von Leitlinien ersetzten, die aktuell noch in vielen Fabriken angewandt wird.

Das Finden eines Pfades durch den freien Raum ist deutlich komplexer als ein FTF einer Leitlinie folgen zu lassen. Dies liegt vor allem daran, dass ohne vorgegebene Spuren die Anzahl möglicher Pfade unendlich ist. In diesem unendlichen Suchraum muss dann der optimale Pfad gefunden werden. Am Institut für Fördertechnik und Logistik soll diese Herausforderung im Rahmen der Forschung an FTF bewältigt werden. Diese Arbeit stellt den Beginn der Forschung dar und soll bekannte Pfadplanungsalgorithmen näher betrachten.

Zuerst sollen verschiedene Vertreter der suchbasierten und stichprobenbasierten Algorithmen in Matlab implementiert werden. Jeder der Algorithmen muss dieselben Eingaben verarbeiten können und als Ausgabe einen möglichst optimalen Pfad liefern.

Bei der Planung dieser Pfade soll ein Roboter mit omnidirektionalem Antrieb angenommen werden. Ein solcher Roboter kann sich ohne vorherige Drehung in jede beliebige Richtung bewegen. Als Ergebnis liefern die Algorithmen also einen Pfad, der die Orientierung nicht beachtet.

Mit Hilfe von Simulationen sollen die Berechnungsmethoden genauer verglichen werden. Hierbei sollen die Rechenzeit, die Pfadlänge, die Glattheit des Pfades und die Flexibilität als Vergleichskriterien dienen. Unter Flexibilität ist das Verhalten des Algorithmus zu verstehen, falls in einer dynamischen Umgebung ein zuvor geplanter Pfad während der Ausführung unbrauchbar wird und neu geplant werden muss.

Die Simulationsszenarien sollen sowohl manuell als auch durch probabilistische Testfallgenerierung erstellt werden. Um Auswirkungen der Zeitkomplexität besser erkennen zu können sollen auch Umgebungen mit verschiedenen Größen verwendet werden.

Zum Schluss soll basierend auf den Erkenntnissen entschieden werden, ob die Algorithmen geeignet sind dynamische Pfadplanungsaufgaben im Umfeld einer Produktionshalle oder Containerterminals zu lösen und welche Algorithmen in weiteren Arbeiten näher untersucht werden sollen.

# 3. Stand der Technik

In diesem Kapitel sollen zuerst grundliegende Konzepte der Pfadplanung erklärt werden. Danach werden die Prinzipien von suchbasierten und stichprobenbasierten Pfadplanungsalgorithmen erläutert. Beide Arten von Algorithmen werden durch Beispiele veranschaulicht und deren Funktionsweise erklärt.

## 3.1 Das grundliegende Pfadplanungsproblem

Das grundliegende Pfadplanungsproblem lässt sich nach [Lat1991, 6] wie folgt definieren:

Zwischen einem gegebenen Anfangszustand und Endzustand, die aus der Position und Orientierung eines Fahrzeugs in einer realen Umgebung bestehen, soll eine kontinuierliche Abfolge von Zuständen gefunden werden, die den Kontakt mit einem Hindernis ausschließen und somit den Start und das Ziel verbinden. Wenn keine solche Abfolge von Zuständen existiert, muss die Suche terminieren.

Diese allgemeine Definition kann noch um eine weitere Forderung ergänzt werden: Es soll ein Pfad gefunden werden, der die Kosten zwischen Start und Ziel minimiert. Für die Bestimmung der Kosten wird immer der Parameter verwendet, der minimiert werden soll. Beispiele hierfür sind die Ausführdauer, die Pfadlänge oder der Energieverbrauch. Im Rahmen dieser Arbeit entsprechen die Kosten immer der euklidischen Länge.

## 3.2 Arbeitsraum und Konfigurationsraum

Die reale Umgebung eines Systems wird als Arbeitsraum $W$ bezeichnet. Hierbei gilt:

$$W = \mathbb{R}^N; N \in \{2; 3\}$$

In diesem Raum bewegt sich das System und die Position und Orientierung wird durch kartesische Koordinaten der Teilkörper beschrieben [Kav2016, 140].

Im Arbeitsraum unterscheidet sich die Bewegung eines Roboterarms mit 3 Gliedern offensichtlich von der ebenen Bewegung eines mobilen Roboters. In beiden Fällen besteht das Ziel jedoch darin das System von dem aktuellen Zustand in einen anderen, gewünschten Zustand zu überführen, ohne dabei mit Hindernissen zu kollidieren [Kav2016, 140].

Um nicht für jede neue Pfadplanungsaufgabe einen neuen Algorithmus entwickeln zu müssen ist es sinnvoll das zu betrachtende System zu abstrahieren, indem man es durch

die minimal nötige Information repräsentiert. Die minimale Anzahl an Parametern, die den Zustand eines Systems vollständig beschreiben können, entspricht den Freiheitsgraden von eben diesem.

Im Falle des Roboterarmes sind dies die drei Gelenkwinkel und im Falle des mobilen Roboters die kartesischen Koordinaten in der Ebene und die Orientierung zu den Achsen. Von den Parametern beider Systeme wird also ein dreidimensionaler Raum aufgespannt. Die Koordinaten jedes Punktes in diesem Raum stellen genau einen Zustand des Systems dar. Unter der Annahme, dass die Systeme steif sind, können aus einem Zustand und der Systemgeometrie die Positionen aller Punkte des Systems berechnet werden. Eine vollständige Beschreibung der Positionen aller Punkte des Systems zu einem Zeitpunkt nennt man Konfiguration $q$ [Cho2005, 40]. Der Raum, der alle möglichen Konfigurationen des Systems enthält, wird als Konfigurationsraum $Q$ bezeichnet [Lat1991, 8]. Eine Konfiguration ist nur dann möglich, wenn das System sie einnehmen kann, ohne mit einem Hindernis zu kollidieren. Deshalb werden Hindernisse in den Konfigurationsraum übertragen und bilden dort $C_{obj} \subseteq C$. Der freie Raum wird als $C_{free}$ bezeichnet und ist ebenfalls eine Teilmenge von $C$.

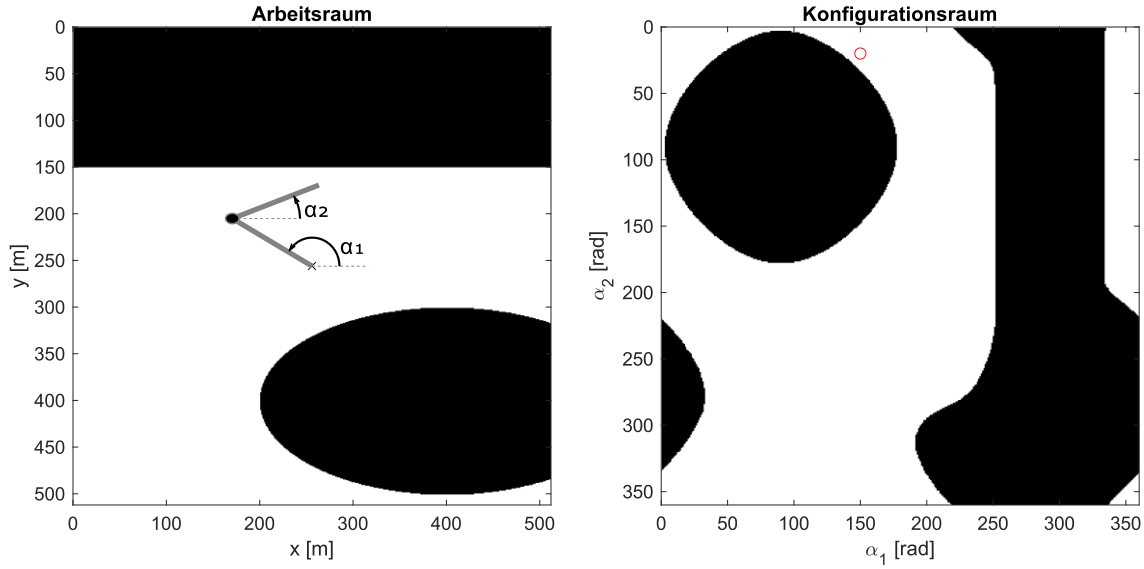


*Abbildung 3.1: Links ist ein Zweiarmroboter in seinem Arbeitsraum zu sehen. Schwarze Felder stellen Hindernisse dar. Rechts ist der zugehörige Konfigurationsraum mit den zwei verallgemeinerten Koordinaten $\alpha_1$ und $\alpha_2$ zu sehen. Die Hindernisse wurden in den Konfigurationsraum transformiert und sind ebenfalls in schwarz dargestellt. Der rote Kreis im Konfigurationsraum entspricht dem Zustand des Roboters im Arbeitsraum. Wegen der Periodizität der Robotergelenke muss man sich den Konfigurationsraum an den jeweils gegenüberliegenden Kanten als zusammengerollt vorstellen, wodurch ein Torus entsteht.*

Für einen mobilen omnidirektionalen Roboter, der sich in der Ebene bewegt ist der Arbeitsraum identisch zum Konfigurationsraum. Die kartesischen Koordinaten des Arbeitsraumes sind identisch zu den betrachteten verallgemeinerten Koordinaten des Roboters, weshalb sich Hindernisse und Freiräume ohne Verzerrung übertragen lassen.

Bewegt sich ein System im Arbeitsraum, so ändert sich auch seine Konfiguration. Zwischen einer Startkonfiguration $q_{init}$ und einer Endkonfiguration $q_{goal}$ liegen eine Vielzahl anderer Konfigurationen, die durchlaufen werden müssen. So ergibt sich ein

kontinuierlicher Pfad im Konfigurationsraum, der die gesamte Bewegung des Systems im Arbeitsraum repräsentiert.

Die Planung einer Bewegung von Volumenkörpern im Arbeitsraum wird damit zur Planung der Bewegung eines einzelnen Punktes im Konfigurationsraum abstrahiert [Lat1991, 7]. So können verschiedenste Pfadplanungsprobleme mit denselben Algorithmen gelöst werden. Im Folgenden sollen nun verschiedene Algorithmen vorgestellt werden.

## 3.3 Suchbasierte Algorithmen

Suchbasierte Algorithmen sind Graphsuchalgorithmen, die in einem Netz aus Knoten und Kanten kürzeste Pfade finden können. Um suchbasierte Algorithmen für Pfadplanung nutzen zu können muss also zuerst der freie Teil des Konfigurationsraumes mit einem Netz aus Kanten und Knoten ausgefüllt werden. Die Konfigurationen von Start und Ziel werden nach der Erstellung des Netztes mit diesem auf dem kürzesten Weg verbunden. Danach wird mit suchbasierten Algorithmen in diesem Netz ein kürzester Pfad vom Start zum Ziel gesucht.

Im Kontext der Pfadplanung werden diese Netze als Straßenkarten bezeichnet. Nachfolgend werden Straßenkarten und deren Generierung genauer betrachtet. Anschließend werden verschiedene suchbasierte Algorithmen vorgestellt.

### 3.3.1 Straßenkarten

Durch Straßenkarten (Englisch: roadmaps) wird die Konnektivität des Freiraums $C_{free}$ von einem Netzwerk aus eindimensionalen Kurven repräsentiert [Lat1991, 12]. Diese Karten können entweder mit Hilfe geometrischer Eigenschaften des Konfigurationsraumes oder auf Zufall basierend erstellt werden.

#### 3.3.1.1 Gleichmäßige Rasterkarten

Den einfachsten Weg, um den freien Teil des Konfigurationsraumes mit einem Netz aus Knoten und Kanten auszufüllen, stellen gleichmäßige Rasterkarten dar. Hier muss anfangs eine Gitterauflösung gewählt werden mit der $C$ aufgeteilt wird. Die Mitte jeder Gitterzelle stellt einen potenziellen Knoten dar. Ergibt sich aus der Kollisionsprüfung der Konfiguration, die zu einer Zellenmitte gehört, dass diese Konfiguration in $C_{free}$ liegt wird

der Knoten mit allen umliegenden Knoten verbunden, die ebenfalls kollisionsfrei sind. Wird ein quadratischer Raum betrachtet steigt die Zahl der potenziellen Knoten quadratisch mit der Kantenlänge des Raumes, weshalb Rechenzeit und Speicherbedarf auch quadratisch steigen. Außerdem weisen enge Passagen eine gleiche Dichte an Knoten auf, wie große freie Flächen [Yah1998, 651]. Bei gleichmäßigen Rasterkarten existieren zudem nur maximal acht Richtungen, auf denen ein Knoten verlassen werden kann.

#### 3.3.1.2 Quadtrees

Werden Freiflächen stattdessen durch größere Zellen repräsentiert müssen weniger Zellen gespeichert werden und die Pfadplanung erfolgt in einem Netz mit weniger Knoten, wodurch diese schneller wird. Dieser Ansatz wird in einem zweidimensionalen Raum durch Quadtrees realisiert. Hierbei wird zu Beginn der gesamte Raum geviertelt. Anschließend wird überprüft, ob diese kleineren Zellen vollständig in $C_{free}$ liegen. Ist das für eine Zelle der Fall wird die Zelle in die Straßenkarte übernommen. Liegt eine Zelle vollständig in $C_{obj}$ wird sie nicht weiter betrachtet. Trifft auf eine Zelle keiner der zuvor gennannten Fälle zu wird sie geviertelt und diese noch kleineren Zellen werden erneut geprüft. Dieses Prozedere wird so lange wiederholt, bis alle Zellen eindeutig $C_{obj}$ oder $C_{free}$ zugeordnet sind oder eine zuvor festgelegte minimale Rastergröße erreicht ist. Wird diese untere Schranke erreicht werden alle, auf dieser Rastergröße nicht entscheidbaren Zellen, als belegt gewertet, um keine Kollisionen zu riskieren [Her2012, 165f.].

Zwar wurde mit Hilfe von Quadtrees die Rasterdichte an die Objektdichte des Raumes angepasst, aber es gibt bei gleich großen Nachbarzellen weiterhin maximal 8 Richtungen, in die eine Bewegung passieren kann. Das führt zu nichtoptimalen Pfaden, denn Zellen müssen in Richtungen durchquert werden, die nicht direkt in Richtung Ziel führen.

Bei Quadtrees fällt ein weiteres Problem, besonders bei großen Zellen, ins Gewicht. Sind diese am geplanten Pfad beteiligt muss stets die Mitte der Zelle durchlaufen werden, auch wenn dies nicht der Richtung entspricht, in der die Zelle im Optimalfall durchquert werden sollte. So entstehen besonders zackige Pfade, die nicht dem optimalen Pfad entsprechen. In Abbildung 3.2 ist dieser Effekt links unten auf der linken Karte zu sehen.

### 3.3.1.3 Framed Quadtrees

Framed Quadtrees erlauben mehr Flexibilität bei der Richtung, in der eine Zelle durchquert wird, indem ein zusätzlicher Schritt bei Erstellung der Straßenkarte durchgeführt wird. Wurde eine Zelle als vollständig frei erkannt wird sie nicht sofort in die Straßenkarte übernommen. Stattdessen wird sie durch eine Struktur von Zellen der minimalen Größe ersetzt, die rundherum am inneren Rand der Zelle platziert werden. Diese kleineren Zellen werden nun statt der großen Zelle in die Straßenkarte integriert. Der Raum, der zuvor von der großen Zelle eingenommen wurde, kann nun auf der Verbindungslinie zwischen beliebigen Randzellen durchquert werden, wodurch mehr Bewegungsrichtungen möglich werden [Yah1998, 651]. Ein Nachteil dieser Methode besteht darin, dass im Vergleich zu Quadtrees ein erhöhter Speicherbedarf besteht. Ist der Arbeitsraum sehr fragmentiert ist die Zellenzahl bei Framed Quadtrees nahe an der Zellenzahl einer gleichmäßigen Rasterkarte, da nur sehr kleine freie Quadrate gefunden werden können, die schon nahe an der minimalen Auflösung sind. Werden diese kleinen Quadrate dann noch, gemäß den Framed Quadtrees, entlang ihres Randes von Quadraten der minimalen Auflösung ersetzt wird nahezu der gesamte Raum mit der Minimalauflösung dargestellt.

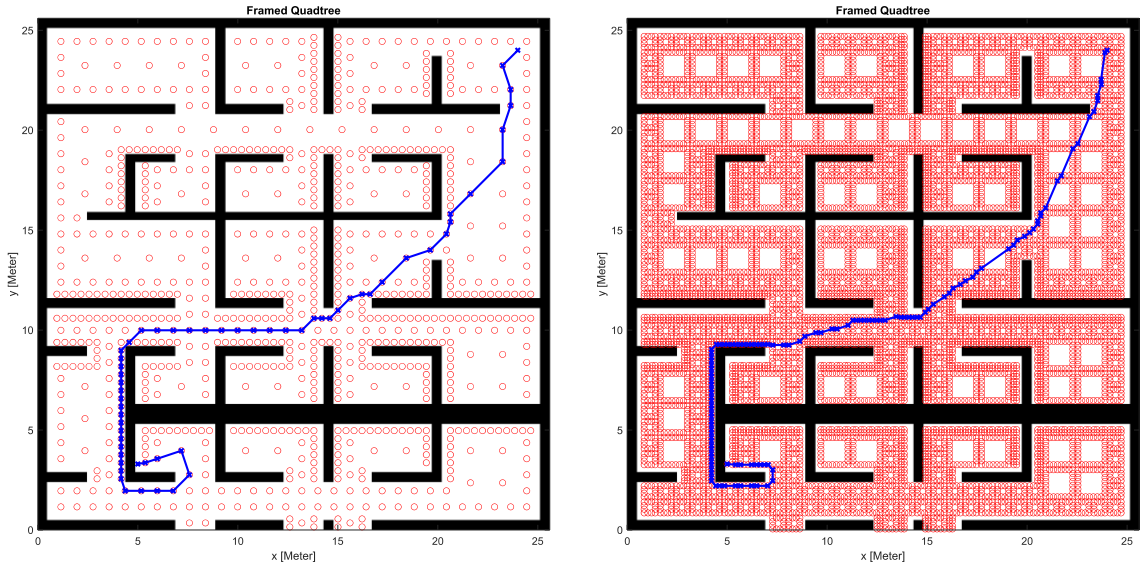


*Abbildung 3.2: Links ist ein Pfad zu sehen, dessen Straßenkarte mit normalen Quadtrees erstellt wurde. Die Straßenkarte für die Pfadplanung, die auf der rechten Seite zu sehen ist, wurde mit Framed Quadtrees erstellt. Die Vorteile der Framed Quadtrees sind deutlich zu erkennen: Der Pfad ist kürzer und enthält weniger Zacken.*

Die zuvor beschriebenen Verfahren können auch in den dreidimensionalen Raum übertragen werden, indem der Raum in Würfelelemente zerlegt wird. Im Fall von nicht gleichmäßigen Gittern werden die Datenstrukturen Octrees genannt. Äquivalent zu Quadtrees werden bei Octrees nicht eindeutige Würfelelemente in acht kleinere Elemente zerlegt [Her2012, 363f.].

### 3.3.2 A* Algorithmus

Liegt zu einem Pfadplanungsproblem eine Straßenkarte vor muss als Nächstes ein Pfad in diesem Netz gefunden werden. Der A* („A-Star“) Algorithmus ermöglicht diese Pfadsuche.

Eine gewöhnliche Breitensuche oder Tiefensuche in einer Straßenkarte würde nacheinander alle möglichen Wege ausprobieren und miteinander vergleichen. Der A*-Algorithmus hingegen untersucht zuerst solche Wege genauer, die am wahrscheinlichsten zum Ziel führen. Die Bewertung wie wahrscheinlich ein Weg zum Ziel führt wird ermöglicht, indem zusätzliche Informationen über das Problem genutzt werden. Eine zusätzliche Information, die die Lösung mit einem Algorithmus beschleunigt wird Heuristik genannt. Bedient sich ein Suchalgorithmus einer Heuristik wird er als informierter Suchalgorithmus bezeichnet [Her2012, 357].

Bezogen auf die Pfadplanung mit A* ist die Zusatzinformation eine Kostenabschätzung $h(n)$ vom aktuellen Knoten $n$ zum Ziel. Zusammen mit den bisher entstandenen Wegkosten $g(n)$, um den aktuellen Knoten $n$ zu erreichen, ergibt sich ein geschätzter Wert für die Gesamtkosten $f(n)$ des Pfades, der über $n$ zum Ziel führt.

$$f(n) = g(n) + h(n)$$

Hierbei ist es wichtig, dass die geschätzten Restkosten $h(n)$ niemals größer sind als die tatsächlichen Kosten zum Ziel, da sonst die Optimalität von A* verloren geht [Har1968, 102ff.]. Überschätzt man die Restkosten eines optimalen Pfades kann es passieren, dass stattdessen ein Pfad mit größeren Kosten verfolgt wird, was in einer nicht optimalen Lösung resultiert. Ein Beispiel hierfür ist in [Cho2005, 535f.] zu finden.

Im Folgenden soll die Funktionsweise von A* beschrieben werden.

Während der Ausführung des Algorithmus werden zwei Listen geführt. In der closed-list befinden sich alle Knoten, deren Untersuchung abgeschlossen ist. In der open-list befinden sich alle Knoten, die im Moment betrachtet werden.

Außerdem müssen zu allen Knoten alle Nachbarn und die Kosten, um die Nachbarn zu erreichen, bekannt sein.

1. Füge den Startknoten zur open-list hinzu. Die Wegkosten betragen 0.
2. Entferne den Knoten mit den niedrigsten Gesamtkosten aus der open-list und füge ihn zur closed-list hinzu. Dieser Knoten wird „expandiert“.
3. Beende den Algorithmus, wenn:
    a. Der expandierte Knoten dem Zielknoten entspricht, was bedeutet, dass das Ziel erreicht wurde.
    b. Kein Knoten expandiert wurde, weil die open-list leer ist. Daraus folgt, dass kein Pfad existiert.
4. Füge jeden Nachbarn des expandierten Knotens zur open-list hinzu, wenn:
    a. Der Nachbar ist nicht in der closed-list enthalten.

   b. Der Nachbar ist nicht in der open-list enthalten. Ist der Nachbar in der open-list enthalten prüfe die gespeicherten Wegkosten zu diesem. Sind die gespeicherten Wegkosten geringer als die Summe der Wegkosten des expandierten Knotens und der Kosten zum betrachteten Nachbarn geschieht nichts. Sind die gespeicherten Kosten jedoch größer wird der geringere Wert eingetragen und der expandierte Knoten als neuer Vorgänger gespeichert. Es wurde also ein schnellerer Weg zu dem Nachbar gefunden.
5. Berechne für die neuen Elemente der open-list die Wegkosten durch Addieren der Wegkosten zum expandierten Knoten mit den Kosten zu den Nachbarn. Berechne danach die Heuristik der neuen Elemente und bestimme damit die Gesamtkosten $f(n) = g(n) + h(n)$. Vermerke den expandierten Knoten als Vorgänger der neuen Elemente.
6. Gehe zu Schritt 2.

Wie bei der Darstellung des grundliegenden Pfadplanungsproblems in Kapitel 1 gefordert terminiert A*, falls kein möglicher Pfad existiert. Besitzt ein Algorithmus diese Eigenschaft wird er als komplett bezeichnet. Wird mit Hilfe von A* eine Lösung gefunden so ist diese optimal. Es gibt also keinen anderen Pfad durch das Netzwerk, der geringere Kosten aufweist [Har1968, 104ff.].

Da die Straßenkarte aber nur eine vereinfachte Darstellung von $C_{free}$ ist kann daraus nicht gefolgert werden, dass der Pfad auch optimal durch den Freiraum führt. Nur für eine Straßenkarte, die jeden möglichen Zustand enthält, wäre der Pfad wirklich optimal. Diese Abschwächung der Optimalität wird aber zu Gunsten der Rechenzeit in Kauf genommen [Kar2011a, 4].

### 3.3.3 D* Lite

Es existieren nahezu keine realen Umgebungen, die vollkommen statisch sind. Deshalb ist es naheliegend Algorithmen einzusetzen, die, im Gegensatz zu A*, Änderungen des Konfigurationsraums verarbeiten können, anstatt neu zu planen. Der D* Algorithmus erfüllt diese Anforderung. D* steht für „Dynamic A*“ und wurde erstmals von Anthony Stenz vorgestellt [Ste1995]. Im Rahmen dieser Arbeit soll jedoch ausschließlich der D* Lite Algorithmus betrachtet werden. Dieser ist verwandt mit D*, kann jedoch in weniger Zeilen Code implementiert werden und ist mindestens so leistungsfähig wie D* [Koe2002, 483].

Ein grundliegender Unterschied zwischen A* und D* Lite ist, dass bei D* Lite die Suche am Zielknoten beginnt und am Startknoten endet. Daraus resultiert, dass die Wegkosten $g(n)$ eines Knotens $n$ den Restkosten bis zum Ziel entsprechen. Die Richtungsänderung wird durchgeführt, weil der Startknoten immer der aktuellen Roboterposition entspricht und sich deshalb ändert, wenn der Roboter sich bewegt. Wird dann eine Neuplanung durchgeführt sind die zuvor berechneten Werte wie zum Beispiel die Restkosten bis zum

Ziel immer noch valide und müssen nicht wegen dem neuen Startpunkt erneut berechnet werden.

Aufgrund der Richtungsänderung ist eine Klarstellung weiterer Bezeichnungen nötig: Während der Planung stellen die Nachbarn $u$ eines Knotens $n$ potenzielle Vorgänger dar. Bei erfolgreicher Expansion wird $n$ zum Nachfolger seiner Nachbarn $u$. Während der Pfadsuche werden also vom Ziel aus Vorgängerknoten durchsucht. Bei der Pfadausführung selbst werden vom Startknoten ausgehend Nachfolger betrachtet.

Neben den Wegkosten $g(n)$ und der, von A* bekannten, Heuristik $h(n)$ benutzt D* Lite noch zusätzlich einen Wert $rhs(n)$. Dieser Wert berechnet sich aus den Wegkosten der Nachbarn $g(u)$ und den Kosten der Nachbarn zu $n$. Die Kosten einer Kante zwischen zwei Knoten werden durch die Funktion $c(n_1, n_1)$ beschrieben.

$$rhs(n) = \begin{cases} 0 & , wenn\, n = Start \\ min_{s \in Vorgänger(n)}(g(s) + c(s,n)) & , sonst \end{cases}$$

Da der rhs-Wert auf den Wegkosten der Nachbarn und den Kosten zu diesen basiert kann an ihm abgelesen werden, ob sich die Umgebung des Knotens verändert hat. Der g-Wert hingegen speichert den günstigsten Wert, mit dem der Knoten bisher erreicht wurde. Wenn $g(n) = rhs(n)$ gilt wird der Knoten als lokal konsistent bezeichnet [Koe2002, 477]. Im Fall $g(n) > rhs(n)$ wird der Knoten als lokal überkonsistent und im Fall $g(n) < rhs(n)$ als lokal unterkonsistent bezeichnet [Koe2002, 478].

Ist ein Knoten lokal konsistent bedeutet das, dass er aus Planungssicht nicht weiter betrachtet werden muss, da es keinen Nachbarn gibt, mit welchem die Wegkosten weiter verringert werden können. Lokale Überkonsistenz bedeutet, dass der Pfad mit geringeren Kosten erreicht werden kann als bisher angenommen. Lokale Unterkonsistenz kann nur dann eintreten, wenn sich Kosten von Kanten zwischen Nachbarn erhöhen und der Knoten deshalb nur mit höheren Kosten als bisher angenommen erreicht werden kann.

Genau wie A* benutzt auch D* Lite eine open-list, in der jedem Knoten $n$ eine Kennzahl $k(n)$ zugeordnet wird. Der Wert $km$ ist relevant, falls sich der Start bei Neuplanung ändert und wird später genauer betrachtet. Im Fall von D* Lite besteht diese Priorität aus zwei Werten, die wie folgt berechnet werden:

$$k_1(n) = \min\left(g(n) + h(n, Ziel) + km\,, rhs(n) + h(n, Ziel) + km\right)$$

$$k_2(n) = \min\big(g(n), rhs(n)\big)$$

$k_1(n)$ entspricht damit, äquivalent zu A*, den Gesamtkosten. Besitzen zwei Knoten die gleiche Kennzahl $k_1(n)$ wird $k_2(n)$ verwendet, um zu beurteilen welche Kennzahl geringer ist. Die Kennzahl ist eine Abschätzung der Kosten eines Pfades, der durch den betrachteten Knoten zum Ziel führt. Eine geringere Kennzahl bedeutet also, dass es vielversprechend ist mit diesem Knoten einen Pfad mit niedrigeren Kosten zu finden.

Im Folgenden soll die Funktionsweise des Algorithmus erläutert werden. Dies geschieht in Anlehnung an [Koe2002, 479].

Zu Beginn wird ein Initialisierungsschritt durchgeführt, bei dem alle rhs-Werte und alle g-Werte gleich unendlich gesetzt werden, was eine Obergrenze für alle tatsächlichen Kosten darstellt. Danach wird der rhs-Wert des Zielknotens gleich 0 gesetzt und der Zielknoten auf die open-List gesetzt. Der Zielknoten ist nun lokal überkonsistent.

Anschließend wird zum ersten Mal ein Pfad berechnet, indem das folgende Prozedere so lange wiederholt wird, bis eine Abbruchbedingung erfüllt ist.

Äquivalent zu A* wird von allen Knoten auf der open-list immer der Knoten $n$ mit der niedrigsten Kennzahl betrachtet, da es für diesen Knoten am wahrscheinlichsten ist, dass er in Richtung Ziel führt. Aufgrund von veränderten Werten in vorherigen Iterationen kann sich die tatsächliche Kennzahl jedoch geändert haben. Deshalb wird zu Beginn jeder Iteration die gespeicherte Kennzahl mit der tatsächlichen Kennzahl verglichen. Ist die tatsächliche Kennzahl höher wird die gespeicherte Kennzahl aktualisiert und die Iteration wird beendet. Ist die tatsächliche Kennzahl geringer kann die Iteration fortgeführt werden, da durch ein Verringern der gespeicherten Kennzahl der Knoten immer noch der Knoten mit der niedrigsten Kennzahl ist.

Auch wenn die Kennzahl für den betrachteten Knoten $n$ korrekt ist kann die Iteration fortgeführt werden. Beim ersten Durchgang der Pfadplanung können Knoten nur lokal überkonsistent sein, da sich die Kosten keiner Kante geändert haben.

Ein lokal überkonsistenter Knoten wir als erstes lokal konsistent gemacht, indem $g(n) = rhs(n)$ gesetzt wird. Der Knoten $n$ wird dann von der open-list entfernt, da er nun konsistent ist und die Planung für diesen Knoten somit abgeschlossen ist. Anschließend werden alle potenziellen Vorgänger untersucht und deren rhs-Wert auf Basis des betrachteten Knoten berechnet. Die Information über die bisherigen Kosten wird so an die Vorgänger weitergegeben.

Jeder betrachtete Vorgänger durchläuft am Ende dasselbe Prozedere, das im Folgenden als aktualisieren eines Knotens bezeichnet werden soll. Alle Vorgänger müssen aktualisiert werden, da sich deren Zustand aufgrund der vorherigen Schritte geändert haben kann und sie deshalb von der open-list anders behandelt werden müssen. Ist der Vorgänger lokal inkonsistent und nicht auf der open-list, wird er dieser hinzugefügt. Ist er lokal inkonsistent und auf der open-list wird seine Kennzahl aktualisiert. Ist er lokal konsistent und auf der open-list wird er von dieser entfernt. Die Iteration ist hiermit beendet und beginnt von vorn, vorausgesetzt die Abbruchbedingung ist nicht erfüllt.

Durch dieses Vorgehen werden in jeder Iteration neue Vorgänger berechnet und es entstehen lokal inkonsistente Knoten, die der open-list hinzugefügt werden und in späteren Iterationen wiederum ihre Vorgänger auf die open-list setzen. So breitet sich die Information über die bisher entstandenen Kosten immer weiter aus. Dieses Prozedere wird so lange wiederholt bis der Startknoten lokal konsistent ist oder der Startknoten die niedrigste Kennzahl besitzt. Letztere Abbruchbedingung ist möglich, da der Startknoten dann als nächstes expandiert werden würde und im nächsten Schritt konsistent werden würde.

Nachdem eine Abbruchbedingung erfüllt ist, wird der Pfad bestimmt. Dies erfolgt schrittweise, beginnend beim Startknoten bis der Zielknoten erreicht ist. Der nächste Schritt wird mit folgender Formel berechnet:

$$n_{next} = min_{s \in Nachfolger(n)}(c(n,s) + g(s))$$

Somit wurde ein initialer Pfad gefunden, der so lange ausgeführt werden kann, bis eine Änderung des Konfigurationsraumes erkannt wird.

Ändert sich der Konfigurationsraum so, dass sich Kosten von Kanten ändern, muss der Pfad repariert werden. Befindet sich zum Beispiel ein Hindernis zwischen zwei Knoten werden die Kosten der betroffenen Kante unendlich groß.

Als Erstes nach einer Änderung des Konfigurationsraumes werden die neuen Kantenkosten gespeichert und zudem wird der rhs-Wert von allen Knoten, die an einer veränderten Kante liegen aktualisiert. Außerdem werden alle diese Knoten auch aktualisiert, da sich die rhs-Werte teilweise geändert haben, wodurch die Knoten inkonsistent werden und wieder auf die open-list gesetzt werden müssen.

Nachdem alle Knoten mit veränderten Kanten aktualisiert wurden, wird die zuvor beschriebene Pfadplanung erneut gestartet. Zusätzlich können Knoten jetzt auch lokal unterkonsistent sein. Ein lokal überkonsistenter Knoten wird gleichbehandelt, wie bei der initialen Planung.

Ist ein Knoten $n$ lokal unterkonsistent wird zuerst $g(n) = \infty$ gesetzt, da die bisher im g-Wert gespeicherten minimalen Wegkosten nicht mehr erreichbar sind. Außerdem wird der Knoten dadurch lokal überkonsistent. Für alle Nachbarn $s$ von $n$ wird dann ein neuer rhs-Wert bestimmt. War $n$ der Vorgänger, mit dem der rhs-Wert von $s$ zuvor berechnet wurde, wird auch $s$ inkonsistent werden, da der rhs-Wert nun mit dem höheren rhs-Wert von $n$ berechnet wird. Auch wenn der rhs-Wert von $s$ nun durch einen anderen Knoten $v$ berechnet wird steigt er. Ansonsten wäre der rhs-Wert schon zuvor mit dem anderen Knoten berechnet worden. Der Knoten $v$ ist nun der Nachfolger von $s$ wodurch sich der Pfad verändert hat. Anschließend werden alle Nachbarn $s$ aktualisiert. Hiermit ist die Iteration beendet und beginnt, falls keine Abbruchbedingung erfüllt ist, von vorne.

Da immer neue lokal unterkonsistente Knoten entstehen, werden immer neue Knoten auf die open-list gesetzt und neue Nachfolger bestimmt. Anschaulich breiten sich mehrere „Wellen“ von allen Stellen mit veränderten Kanten aus. Die erste Welle setzt neue Knoten auf die open-list und macht sie lokal unterkonsistent, indem sie die erhöhten Kosten schrittweise weitergibt. Die zweite Welle macht die Knoten lokal unterkonsistent und sucht neue Nachfolger, wodurch der Pfad repariert wird. Eine dritte Welle macht die Knoten schließlich lokal konsistent und repariert den Pfad damit endgültig. Die Wellen breiten sich aus, bis der Pfad repariert wurde.

Damit ist die Neuplanung des Pfades abgeschlossen und der Pfad kann weiter ausgeführt werden. Ändern sich erneut Kosten von Kanten wird das zuvor beschriebene Prozedere wiederholt.

Da sich der Roboter zwischen den Neuplanungsphasen bewegt besitzt er bei jeder Neuplanung einen neuen Startknoten. Da die Heuristik, und damit die Kennzahl in Bezug

auf den Startknoten berechnet werden ändern sich diese mit jeder Neuplanung. Ein Ziel von D* Lite ist es bei jeder Neuplanung die Daten aus dem vorherigen Durchgang verwenden zu können. Bei einem veränderten Startknoten würden die Kennzahlen, die mit dem alten Starknoten berechnet wurden wegen der veränderten Heuristik, nicht mit neu berechneten Kennzahlen zusammenpassen und müssten deshalb für jeden Knoten neu berechnet werden. Darum wird vor jeder Neuplanung ein Wert $km$ bestimmt, der den euklidischen Abstand zwischen dem alten Start und dem neuen Start darstellt. Die erste Komponente der Kennzahl kann sich zwischen diesen beiden Knoten maximal um den euklidischen Abstand verringert haben, da sich die Restkosten zum Ziel nicht um mehr als den euklidischen Abstand verringern können. Man könnte also für jeden Knoten auf der open-list die Kennzahl $k_1(n)$ um $km = h(start_{neu,} start_{alt})$ verringern. Dies würde die Reihenfolge der open-list jedoch nicht verändern, weshalb diese Subtraktion nicht durchgeführt wird. Wird nun ein neuer Knoten hinzugefügt wird seine Kennzahl in Bezug auf den neuen Start berechnet und wäre damit, verglichen mit den Kennzahlen, die bereits in der open-list sind, um $km = h(start_{neu,} start_{alt})$ zu klein. Anstatt von allen bereits berechneten Kennzahlen $km$ abzuziehen wird $km$ zu jeder neu berechneten Kennzahl addiert [Koe2002, 480].

## 3.4 Stichprobenbasierte Algorithmen

Mit Hilfe von stichprobenbasierten Algorithmen soll der Teilraum $C_{free}$ von Knoten, die durch Kanten verbunden sind, bestmöglich ausgefüllt werden. Die Grundidee besteht darin eine zufällige Konfiguration zu generieren und diese optimal mit den bereits existierenden Knoten und Kanten zu verbinden.

Im Gegensatz zur Erstellung von Straßenkarten für suchbasierte Algorithmen wird bei stichprobenbasierten Algorithmen nicht der gesamte Konfigurationsraum betrachtet. Stattdessen wird der zufällig ausgewählte Punkt auf Kollisionen überprüft und somit immer nur ein kleiner Teil von $C$ betrachtet [Kar2011a, 2].

Im Verlauf dieses Kapitels sollen Rapidly-Exploring Random Trees (RRT) und die Nachfolger RRT* und RRT$^X$ vorgestellt werden.

### 3.4.1 Rapidly-Exploring Random Tree (RRT)

RRTs wurden erstmals von Steven LaValle in [LaV1998] vorgestellt. Ihren Namen haben sie aufgrund ihrer baumähnlichen Struktur, die sich, beginnend beim Starknoten, immer weiter verzweigt und sich dadurch im Konfigurationsraum ausbreitet. Der Startknoten wird als Wurzel bezeichnet und die Ausläufer des Baumes als Äste.

Die Funktionsweise von RRT lässt sich in wenigen Schritten erklären. Wie bei jedem Pfadplanungsalgorithmus besteht das Ziel darin den Start und das Ziel auf dem besten Weg miteinander zu verbinden. Zu Beginn enthält der freie Teil des Konfigurationsraums $C_{free}$ nur den Startknoten. Es beginnt nun ein Prozedere, das so lange wiederholt wird, bis ein neu hinzugefügter Knoten dem Ziel entspricht oder sich in einer ausreichend kleinen Umgebung um das Ziel befindet.

Zuerst wird ein zufälliger Punkt $p_{rand}$ im Konfigurationsraum bestimmt. Anschließend wird aus allen Knoten $n$, die bereits im Baum vorhanden sind, der Knoten $n_{near}$ herausgesucht, der am nächsten an $p_{rand}$ liegt. Weil der Baum schrittweise wachsen soll und nicht zu sehr springen soll, existiert eine maximale Verbindungsdistanz $d$. Falls $|p_{rand} - n_{near}| > d$ wird ein neuer Knoten $n_{new}$ so auf der Verbindungsstecke zwischen $p_{rand}$ und $n_{near}$ platziert, dass $|n_{new} - n_{near}| \leq d$ gilt. Für den Fall $|p_{rand} - n_{near}| < d$ wird $n_{new} = p_{rand}$ gesetzt. Abschießend muss noch geprüft werden, ob $n_{new}$ und die gesamte Verbindungsstecke von $n_{near}$ zu $n_{new}$ in $C_{free}$ liegt. Ist dies der Fall wird $n_{new}$ in die Menge der Knoten $N$ aufgenommen und $n_{near}$ ist damit der Vorfahre von $n_{new}$. Ansonsten wird $n_{new}$ verworfen. Anschließend beginnt das Prozedere erneut [Cho2005].

RRT tendieren dazu sich in Gebiete auszubreiten, in denen bisher wenige Knoten existieren. Betrachtet man das Voronoi-Diagramm eines Baumes erkennt man, dass die Knoten am Rand die größten Voronoi-Regionen besitzen. Innerhalb dieser Voronoi-Regionen ist per Definition immer der Knoten, zu dem die Voronoi-Region gehört der nächstgelegene. Die Wahrscheinlichkeit, dass ein neuer, zufällig ausgewählter Knoten an einen existierenden Knoten $n$ anschließt, ist proportional zur Größe der Voronoi-Region von $n$. Damit ist es für einen Baum immer am wahrscheinlichsten in die Gebiete vorzudringen, die eine geringe Dichte an Knoten aufweisen, was für einen Baum im frühen Stadium besonders auf die Randgebiete zutrifft [Kav2016, 144].

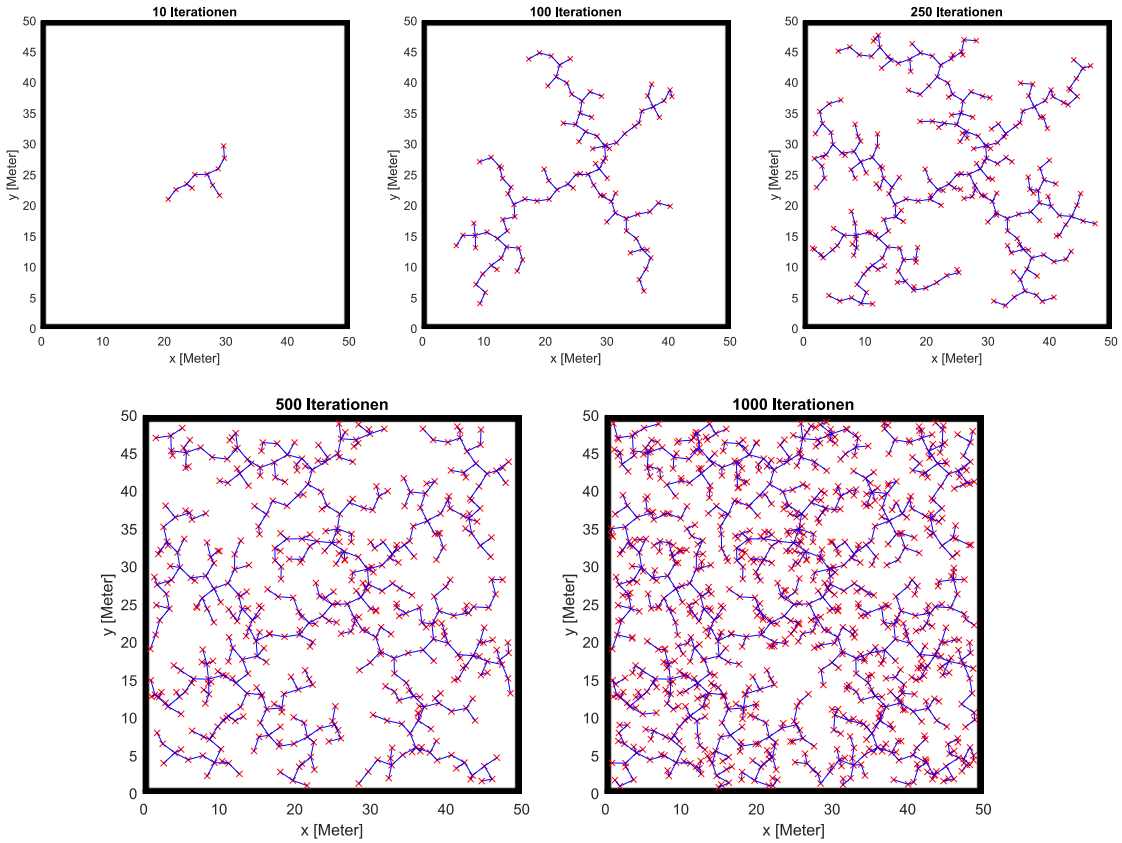


*Abbildung 3.3: Ein RRT breitet sich in einem freien Konfigurationsraum aus. Zu Beginn wächst der RRT stark in Richtung der Ränder. Danach breitet er sich in Regionen mit wenigen Knoten aus und füllt so den Konfigurationsraum immer gleichmäßiger aus.*

Klassische RRT Strukturen sind meist sehr zackig, da ein neu hinzugefügter Knoten die bereits existierenden Knoten nicht beeinflusst. Beispielsweise wird nicht berücksichtigt, wenn ein neu hinzugefügter Knoten einen schnelleren Weg zu einem bereits existierenden Knoten liefern kann oder ein anderer Knoten als $n_{near}$ besser als Vorfahre für $n_{new}$ geeignet wäre. Daraus resultiert, dass der RRT-Algorithmus nicht optimal ist und demnach auch bei unendlich langer Laufzeit nicht gegen die optimale Lösung konvergiert [Kar2011a, 51ff.]. Aus diesem Grund soll der reine RRT-Algorithmus im Rahmen dieser Arbeit nicht näher betrachtet werden. Stattdessen werden im Folgenden Weiterentwicklungen des Algorithmus vorgestellt.

### 3.4.2 Rapidly-Exploring Random Tree* (RRT*)

Der RRT* Algorithmus ist ein asymptotisch optimaler und statistisch kompletter Algorithmus [Kar2011a, 72ff., Kar2011a, 29]. Diese Eigenschaften erreicht er, indem er im Gegensatz zum RRT-Algorithmus ein erneutes Verbinden der Knoten zulässt.

RRT* führt alle Schritte durch, die auch RRT durchführt. Das Erstellen neuer Knoten und das Vermindern der Distanz zum Baum geschieht auf die gleiche Weise und soll nicht näher dargestellt werden. Nachdem ein Knoten $n_{new}$ erfolgreich erstellt wurde startet die

Suche nach dem besten Vorfahren und die Umstrukturierung. Um bei einer wachsenden Knotenanzahl nicht immer mehr Knoten umstrukturieren zu müssen werden nur Knoten in einem Radius $r$ um $n_{new}$ betrachtet. Dieser Radius berechnet sich wie folgt:

$$r = \min\left(\left(\gamma * \frac{\log(num(n))}{num(n)}\right)^{\frac{1}{dim}}, d\right)$$

$num(n)$ beschreibt hier die Anzahl der Knoten im Baum. $dim$ ist die Dimension des Konfigurationsraumes, und $d$ die maximale Verbindungsdistanz eines neuen Knotens. Für $\gamma$ gilt die Bedingung

$$\gamma > \left(2\left(1 + \frac{1}{dim}\right)\right)^{\frac{1}{dim}} * \left(\frac{\mu(C_{free})}{\vartheta_d}\right)^{\frac{1}{dim}}$$

wobei $\mu(C_{free})$ das Lebesgue-Maß von $C_{free}$ darstellt und $\vartheta_d$ das Volumen der Einheitskugel im d-dimensionalen Raum. Diese Forderung ist nötig, um die asymptotische Optimalität zu garantieren [Kar2011a, 29]. Der Beweis hierfür kann [Kar2011a, 72ff.] entnommen werden.

Alle Knoten, die innerhalb des Radius $r$ um $n_{new}$ liegen, werden nun erneut betrachtet und in der Menge $N_{near}$ zusammengefasst. Es muss nicht immer sein, dass $n_{near}$ auch die geringsten Kosten zu $n_{new}$ ermöglicht, da möglicherweise ein Umweg nötig ist, der größer ist als die Distanzdifferenz zu einem Knoten ohne Umweg. Deshalb muss unter allen $n \in N_{near}$ der Knoten gefunden werden, der $g_{min} = g(n) + c(n, n_{new})$ minimiert. Hierbei bezeichnet $g(n)$ die Kosten, um einen Knoten $n$ zu erreichen und $c(a, b)$ die benötigten Kosten, um von $a$ nach $b$ zu gelangen. Der Knoten, der diese Bedingung minimiert wird dann als Vorfahre von $n_{new}$ gespeichert (siehe Abbildung 3.4). Außerdem ist es auch möglich, dass $n_{near}$ zwar die geringsten Kosten bietet aber sich auf der Verbindungsstrecke zwischen $n_{new}$ und $n_{near}$ ein Hindernis befindet. Auch dann würde ein anderer Vorgänger aus der Menge $N_{near}$ bestimmt werden.

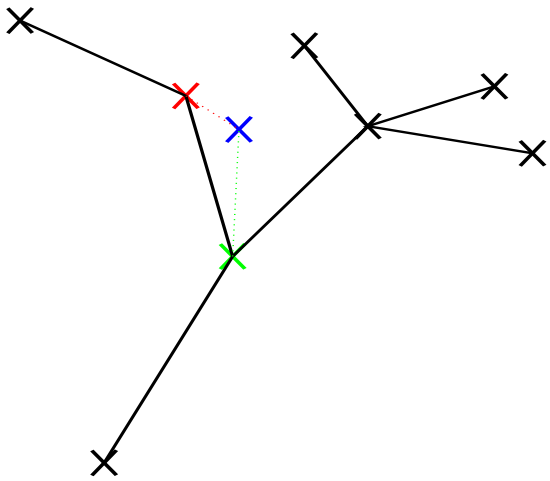

*Abbildung 3.4: Ein neuer Knoten (blau) wird dem Konfigurationsraum hinzugefügt. Der RRT Algorithmus würde $n_{near}$ (rot) als Vorgänger wählen. RRT* hingegen wählt den grün markierten Knoten als Vorgänger, da dieser geringere Kosten zum blauen Knoten ermöglicht.*

Danach werden alle Knoten aus $N_{near}$ erneut betrachtet und geprüft, ob deren Kosten über $n_{new}$ reduziert werden können. Falls die Strecke zwischen $n \in N_{near}$ und $n_{new}$ frei ist und die Bedingung $g(n) > g(n_{new}) + c(n, n_{near})$ erfüllt ist wird $n_{new}$ als neuer Vorfahre von $n$ gespeichert. Die Kante im Baum zum alten Vorfahren von $n$ wird dann verworfen und durch die neue Kante ersetzt, damit keine Schleifen im Baum entstehen. Durch dieses Vorgehen werden die Knoten immer neu verknüpft, wenn sich dadurch Kosten minimieren lassen, und der Baum konvergiert zu einer Struktur, die den optimalen Weg zwischen dem Startpunkt und jedem beliebigen Punkt innerhalb von $C_{free}$ enthält (siehe Abbildung 3.5).

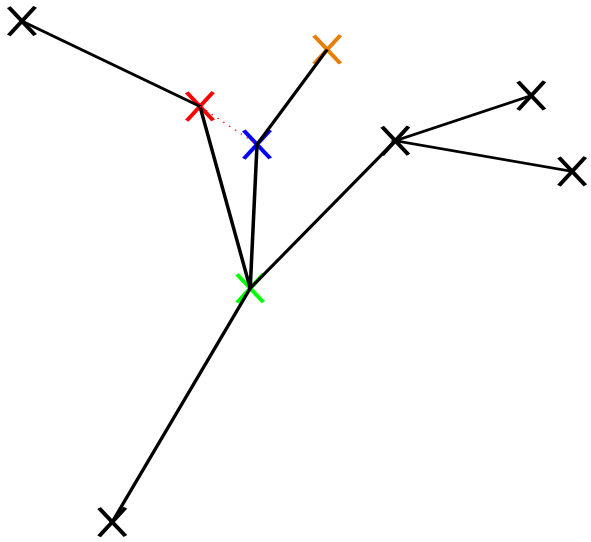

*Abbildung 3.5: Der orangene Knoten wird über den blauen Knoten neu verbunden, da dieser die Kosten minimieren kann. Die alte Verbindung wird verworfen.*

### 3.4.3 Rapidly-Exploring Random Tree$^X$ (RRT$^X$)

Mit RRT* können nur statische Umgebungen behandelt werden. Sobald es Veränderungen im Konfigurationsraum gibt, muss der Baum verworfen und anschließend neu erstellt werden. Der RRT$^X$-Algorithmus wurde erstmals in [Ott2015] vorgestellt und ermöglicht eine schnelle Neuplanung in dynamischen Umgebungen.

In statischen Umgebungen verhält sich RRT$^X$ äquivalent zu RRT*. Daraus resultiert, dass auch RRT$^X$ asymptotisch optimal und statistisch komplett ist und außerdem eine vergleichbare Laufzeit bei statischen Planungsaufgaben besitzt.

Wie bei jedem dynamischen Pfadplanungsalgorithmus benötigt auch RRT$^X$ einen initialen Pfad, der später repariert wird. Da die Erstellung von diesem aber gleich zum RRT*-Algorithmus geschieht wird an dieser Stelle auf eine erneute vollständige Erklärung verzichtet. Im Gegensatz zum RRT*-Algorithmus besteht jedoch der Unterschied, dass Informationen für die Neuplanung gespeichert werden. Zum Beispiel

werden für jeden Knoten die Nachbarn innerhalb des Radius $r$ gespeichert, damit diese später nicht erneut bestimmt werden müssen.

Die Nachbarschaftsbeziehungen werden immer einseitig gespeichert, weshalb es für jeden Knoten $n$ zwei verschiedene Mengen an Nachbarn gibt. Die Menge $N^-(n)$ beschreibt die eingehenden Nachbarschaftsbeziehungen und die Menge $N^+(n)$ die ausgehenden Nachbarschaftsbeziehungen. Diese beiden Mengen werden in zwei weitere Mengen, nämlich die anfänglichen Nachbarn $N_0^-(n)$ beziehungsweise $N_0^+(n)$ und die laufenden Nachbarn $N_r^-(n)$ beziehungsweise $N_r^+(n)$ aufgeteilt [Ott2016, 806]. Diese Aufteilung ist wichtig, da im späteren Verlauf Nachbarschaftsbeziehungen entfernt werden, um zu verhindern, dass die Zahl der Nachbarn proportional mit der Anzahl der Knoten wächst. In dem Moment, in dem ein Knoten hinzugefügt wird, werden ihm seine anfänglichen Nachbarn zugewiesen. Zu diesem Zeitpunkt ist der Baum intakt und die Menge der anfänglichen Nachbarn garantiert damit ein Mindestmaß an Verbindungen durch $C_{free}$. Deshalb dürfen bei der Entfernung von Nachbarn keine anfänglichen Nachbarschaftsbeziehungen gelöscht werden. Die Folge davon wäre der Verlust der statistischen Vollständigkeit und der asymptotischen Optimalität [Ott2016, 808].

Äquivalent zu D*Lite besitzt auch bei RRT$^X$ jeder Knoten $n$ einen Wert $g(n)$, der die Restkosten zum Ziel darstellt, und einen Wert $lmc(n)$, der die geringsten Restkosten über die Nachbarn berechnet und somit besser über Änderungen informiert ist als $g(n)$.

Nun soll die Neuplanung mit RRT$^X$ erläutert werden. Dies geschieht in Anlehnung an [Ott2016].

Wird eine Änderung im Konfigurationsraum, wie zum Beispiel ein neues Hindernis, erkannt müssen alle Kanten, die von diesem Hindernis blockiert werden, betrachtet werden. Zuerst werden die Kantenkosten auf unendlich gesetzt. Werden die Kantenkosten zwischen einem Knoten $n$ und seinem Vorfahren $v$ unendlich, bedeutet das, dass $n$ vom restlichen Baum getrennt wurde. $n$ wird deshalb in die Menge der Waisen aufgenommen.

Die Veränderungen der Kosten einzelner Kanten haben immer auch Auswirkungen auf die Werte von Knoten, die weiter in Richtung der Äste liegen. Wird ein Knoten $n$ zum Waisen werden also auch alle Knoten, die sich in Richtung der Äste von $n$ befinden zu Waisen. So breitet sich die Information über neue Hindernisse wellenartig in Richtung der Äste aus. Allgemein gilt für alle Waisen $n$: $g(n) = \infty$ und $lmc(n) = \infty$.

Wie schon bei D*Lite gibt es auch bei RRT$^X$ eine open-list, die alle inkonsistenten Knoten beinhaltet. Es werden nun alle Nachbarn der Waisen durchsucht, um Knoten zu finden, die keine Waisen sind. Dies können nur Knoten sein, die sich auf anderen Ästen befinden und stellen damit eine Möglichkeit dar den abgeschnittenen Ast wieder mit dem Baum zu verbinden. Wird ein solcher Knoten $n$ gefunden wird $g(n) = \infty$ gesetzt und der Knoten damit inkonsistent. Hierdurch wird garantiert, dass alle Knoten, die einen abgetrennten Ast wieder verbinden können, später erneut betrachtet werden, da sich alle inkonsistenten Knoten auf der open-list befinden. Da für jeden Knoten auf der open-list später untersucht wird, ob er die Kosten seiner Nachbarn verringern kann wird sichergestellt, dass die Kosten der Waisen in der Nachbarschaft verringert werden.

Wie bei D*Lite wird nun dafür gesorgt, dass im Baum keine inkonsistenten Knoten existieren. Bei RRT$^X$ wird jedoch eine Toleranz $\varepsilon$ zugelassen. Ein Knoten $n$ ist also nur dann inkonsistent, wenn $|g(n) - lmc(n)| > \varepsilon$ gilt. Diese Lockerung der Konsistenz ist nötig, um eine schnelle Konvergenz zu garantieren. Da $\varepsilon$ kleiner als der Radius des Roboters gewählt wird, betrifft diese Lockerung auch nur Änderungen im Konfigurationsraum, welche die Kosten leicht verändern und nicht gesamte Passagen unbrauchbar machen [Ott2016]. Die open-list wird beginnend bei dem Knoten mit der kleinsten Kennzahl geleert. Die Kennzahl berechnet sich wie folgt:

$$k_1(n) = \min(g(n), lmc(n))$$

$$k_2(n) = g(n)$$

Bei der Bestimmung der kleinsten Kennzahl wird zuerst $k_1(n)$ verwendet. Ergibt sich dann ein Gleichstand zwischen mehreren Knoten wird $k_2(n)$ verglichen. Wenn dann immer noch ein Gleichstand besteht, wird zufällig zwischen den Knoten entschieden. Für den Knoten $n$ mit der geringsten Kennzahl wird zuerst $lmc(n)$ erneut berechnet. Anschließend werden die Nachbarn von $n$ neu verbunden. Hierfür wird jeder Nachbarknoten $v$ von $n$ betrachtet und überprüft, ob $lmc(v)$ über n verringert werden kann. Anschaulich bedeutet das, dass überprüft wird, ob über den Knoten $n$ der Knoten $v$ mit geringeren Kosten als bisher erreicht werden kann. Ist das der Fall wird $n$ zum Vorfahren von $v$, $lmc(v)$ wird aktualisiert und $v$ wird der open-list hinzugefügt. Alle Knoten, die zuvor zu Waisen wurden, haben ihre Nachbarn, die keine Waisen sind, zuvor auf die open-list gesetzt. Es wurde also durch das zuvor beschriebene Szenario sichergestellt, dass ein neuer Weg identifiziert wird, um die Waisen zu erreichen und der abgetrennte Ast ist somit wieder mit dem Baum verbunden. Durch die Abarbeitung der open-list mit Hilfe der Kennzahlen werden die Knoten, die sich näher an der Wurzel des Baumes befinden zuerst abgearbeitet. Es wird also versucht die abgetrennten Äste möglichst nah an der Wurzel wieder mit dem Baum zu verbinden. Wurde ein neuer Ast so neu verbunden wird der Knoten, an dem er verbunden wurde auf die open-list gesetzt und dieser Knoten gibt seine verringerten Kosten an die Nachbarn im abgetrennten Ast weiter. Diese geben die verringerten Kosten wiederrum an ihre Nachbarn weiter und setzten diese auf die open-list. So breitet sich die neue Kosteninformation wellenartig von der neuen Anschlussstelle aus. Dieser Vorgang wird so lange wiederholt, bis die open-list leer ist und damit der gesamte Baum konsistent ist. Die Neuplanung ist damit abgeschlossen.

# 4. Implementierung der Algorithmen

In diesem Kapitel soll die Implementierung der in Kapitel 3 beschriebenen Algorithmen betrachtet werden. Außerdem sollen alle weiteren Algorithmen und Daten vorgestellt werden, die zur Nutzung der Algorithmen nötig sind.

Alle im Folgenden beschriebenen oder erwähnten Programmcodes befinden sich im Anhang.

## 4.1 Karten für die Pfadplanung

Für jede Pfadplanungsaufgabe werden Karten benötigt. Diese müssen den Konfigurationsraum mit einer vorgegebenen Auflösung repräsentieren. Da jeder Punkt des Konfigurationsraumes entweder Teil von $C_{free}$ oder $C_{obj}$ ist eignet sich eine binäre Matrix, in der jeder Eintrag einem Pixel entspricht und einen Wert von null oder eins enthält.

Um verschiedene Arbeitsräume für die Pfadplanung zu erhalten können mit Hilfe der Funktion $mapCreator()$ zufällige quadratische Karten erstellt werden. Hierfür muss die gewünschte Kartengröße in Metern, die gewünschte Auflösung und die Koordinaten von Start und Ziel übergeben werden. Durch den Zufallsgenerator von Matlab wird eine Karte mit zufälligen Zahlenwerten im Intervall $[-1,1]$ erstellt. Zu diesem Zeitpunkt ist die Karte absolut zufällig und zeigt keine regelmäßigen Strukturen, die einem echten Szenario ähneln. Deshalb werden die Werte mit Hilfe von laufenden Mittelwerten geglättet. Ein laufender Mittelwert mit einem großen Intervall sorgt zuerst für grobe große Strukturen. Danach sorgen zwei laufende Mittelwerte mit kleineren Intervallen für klare Kanten. Zum Schluss wird noch das Gebiet um den Start und das Ziel von Hindernissen befreit und ein Rahmen eingefügt.

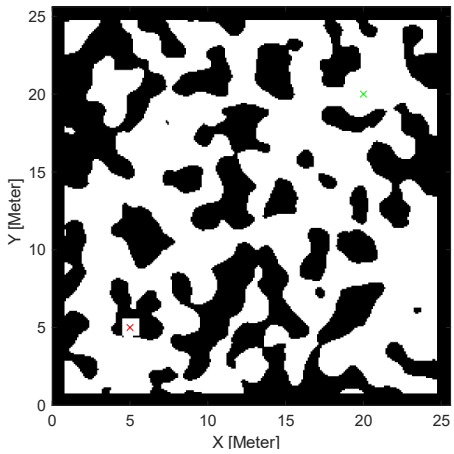


*Abbildung 4.1: Eine zufällige von mapCreator() erstellte Karte. Der Start ist mit einem roten Kreuz markiert. Das Ziel ist mit einem grünen Kreuz markiert. Im Bereich um den Start wurde das Hindernis von $mapCreator()$ entfernt, um eine Pfadplanung möglich zu machen.*

Die zufällig erstellten Karten bieten zwar viele verschiedene Szenarien aber gleichen eher einer Pfadplanung im Außenbereich. Da in dieser Arbeit aber besonders die Pfadplanung in Warenhäusern und Containerterminals betrachtet werden soll wird eine zweite Karte benötigt. Die Matlab Navigation Toolbox bietet hierfür Beispielkarten an. Da diese jedoch eine sehr geringe Auflösung besitzen und nicht quadratisch sind wird eine leicht veränderte Karte verwendet. Sie besitzt eine höhere Auflösung und ein Teil der Originalkarte ist gespiegelt an die Unterseite angehängt. Im Folgenden soll diese Karte als Lagerhauskarte bezeichnet werden.

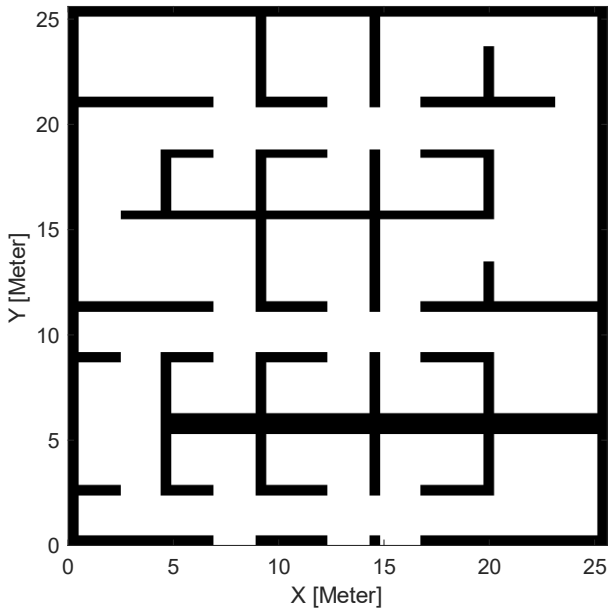


*Abbildung 4.2: Lagerhauskarte, die in den Simulationsexperimenten verwendet wird.*

## 4.2 Allgemeine Programmcodes

In vielen Situationen muss überprüft werden, ob eine Verbindungsstrecke zwischen zwei Knoten zulässig ist. Immer dann wird die Funktion $motionValidatorBinary()$ verwendet. Dieser Funktion muss die binäre Repräsentation des Konfigurationsraumes, die Auflösung der Karte, die zwei Konfigurationen, deren Verbindungsstrecke kontrolliert werden soll und die Prüfdistanz, mit der die Strecke überprüft werden soll, übergeben werden. Hierfür wird die Verbindungsstrecke in Schritten durchlaufen, die der übergebenen Prüfdistanz entsprechen. Jeder dieser Schritte wird dann auf Kollisionen überprüft. Mit Hilfe der Auflösung kann bestimmt werden welchem Pixel in x-Richtung die Konfiguration entspricht:

$$px_{Nummer,x} = floor(x_{Konfiguration} * Auflösung)$$

Für die y-Richtung muss die Formel umgedreht werden, da Matrixindizierung hier entgegen der Koordinatenachse läuft:

$$px_{Nummer,y} = Pixelzahl_y - ceil(y_{Konfiguration} * Auflösung)$$

Sobald die Position und damit der Index des Pixels bekannt ist kann dieses Pixel auf der binären Karte überprüft werden. Ist nur eines der Pixel auf der Verbindungsstrecke belegt, so gilt die Strecke als nicht zulässig und die Funktion kann sofort beendet werden.

Für alle im weiteren Verlauf vorgestellten Pfadplanungsalgorithmen existiert ein Skript, das alle für die Pfadplanung nötigen Variablen initialisiert. Dazu gehört ein Konfigurationsraum, der Start, das Ziel und einige frei wählbare Parameter, die die Genauigkeit der Algorithmen bestimmen. Außerdem verwaltet dieses Skript Änderungen im Konfigurationsraum vor Neuplanungen und ruft die Hauptfunktionen der Pfadplanungsalgorithmen auf. Dieses Skript ist in den Ordnern der jeweiligen Algorithmen zu finden und trägt den Namen des Algorithmus.

## 4.3 Quadtrees

In Kapitel 3 wurde die Funktionsweise von Quadtrees beschrieben. Im Folgenden soll die Umsetzung in Matlab und der entwickelte Algorithmus genauer erklärt werden. Die Erstellung des Quadtrees geschieht in zwei Schritten. Zuerst werden die Knoten erstellt und mit einem Index versehen. Danach werden mit Hilfe einer Finite State Machine (FSM) die Nachbarn der Knoten ermittelt.

Als Eingangsdaten erhält der Algorithmus eine binäre Repräsentation des Konfigurationsraumes, wobei ein Pixelwert von 1 für ein belegtes Pixel steht. Außerdem wird eine Matrix übergeben, die für jedes Pixel die entsprechende Konfiguration enthält. Zudem wird die minimal zulässige Quadratgröße übergeben.

Die Ausgabedaten bestehen aus den Konfigurationen der Knoten, deren Nachbarn und den Kosten zu den Nachbarn.

### 4.3.1 Knotensuche

Jeder Knoten des Quadtrees besitzt zwei Identifikationsmerkmale. Die Adresse soll im Folgenden die Zeile in der Liste der Knoten beschreiben, in der ein Knoten gespeichert ist. Der Index eines Knotens berechnet sich aus seiner Position im Quadtree und wird zur Berechnung der Nachbarn benötigt. Es wurde ein Index in Anlehnung an [Rob2006] verwendet. Bei jeder neuen Aufteilung eines großen Quadrates in vier kleinere ergibt sich der Index der kleineren Quadrate, indem eine Ziffer an den Index des großen Quadrates angehängt wird. Abhängig von der Position des Quadrates wird eine andere Ziffer angehängt. Das Schema ist der nachfolgenden Abbildung 4.3 zu entnehmen.

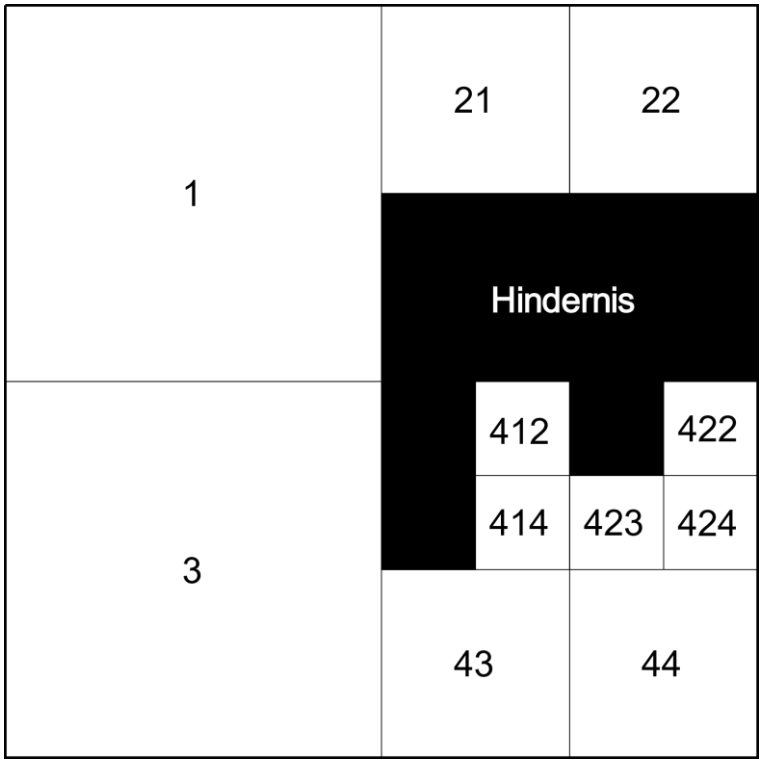


*Abbildung 4.3: Die Abbildung zeigt die Quadtreezerlegung eines Konfigurationsraums mit Hindernis. Die Zahlen in den Quadraten entsprechen den Indizes der Quadrate.*

Anders als in [Rob2006] wurden die Indizes aus den Zahlen eins, zwei, drei und vier zusammengesetzt. Der Grund hierfür ist, dass beispielsweise der Index „000021“ in Matlab nicht in einer Variablen des Typs double gespeichert werden kann, da der Index dann in „21“ umgewandelt werden würde. Alternativ könnten die Indizes als String gespeichert werden. Da die Bearbeitung von Strings aber langsamer ist als einfache Rechenoperationen mit Variablen des Typs double, wurde eine Änderung der Indexbausteine bevorzugt.

Eine weitere Modifikation der FSM ist die Verwendung von Zahlen als Repräsentation für die Richtungen. Der Grund hierfür ist, dass häufig die Richtung in logischen Operationen auf Gleichheit mit einem bestimmten Wert überprüft werden muss und solche Operationen mit ganzen Zahlen schneller sind als mit char-Variablen. Den Richtungen werden in folgender Reihenfolge die Zahlen eins bis neun zugeordnet: Rechts, Links, Unten, Oben, Rechtsunten, Rechtsoben, Linksunten, Linksoben, Halt. Diese Konvention gilt für alle Funktionen, die Teil der Implementierung der Quadtrees oder Framed-Quadtrees sind.

Der Index der Knoten wird durch die Funktion $divideAndReturnSquares()$ erstellt, die bei jedem Aufruf eine weitere Ziffer an den übergebenen Index anhängt.

Wie in Kapitel 3 beschrieben, wird der betrachtete Raum so lange in kleinere Quadrate aufgeteilt, bis ein eindeutiger Zustand oder eine minimale Auflösung erreicht ist. Ob ein eindeutiger Zustand erreicht ist, wird mit der Funktion $checkstatus()$ überprüft. Diese erhält als Eingabeparameter das aktuell betrachtete Quadrat und die minimal zulässige Quadratgröße. Zu Beginn wird die Größe des aktuellen Quadrates geprüft. Ist die

minimal zulässige Größe erreicht wird die Funktion beendet und das Quadrat wird als belegt gewertet.

Ansonsten wird das erste Pixel des Quadrates betrachtet. Dieses gibt den Überprüfungsmodus vor. Besitzt das erste Pixel einen Wert von eins wird das Quadrat so lange Pixel für Pixel überprüft, bis ein Pixel den Wert null enthält. Das Quadrat kann dann als nicht eindeutig bewertet werden und die Funktion wird beendet. Durch dieses Vorgehen wird die Rechenzeit verringert, da bei nicht eindeutigen Quadraten nicht alle Pixel überprüft werden müssen. Wird das Ende des Quadrates erreicht, ohne dass ein Pixel den Wert null enthält, wird das Quadrat als belegt gewertet und die Funktion endet ebenfalls. Besitzt das erste Pixel einen Wert von null verhält sich die Funktion konträr und bricht beim ersten Pixel mit dem Wert eins ab.

Die Ausgabe von $checkstatus()$ gibt also vor, ob das Quadrat weiter aufgeteilt werden muss oder nicht. Die Funktion $divideAndReturnSquares()$ ist rekursiv und ruft sich so lange selbst auf, bis der Status des Quadrates eindeutig ist. Bei jeder Aufteilung der binären Matrix wird die Matrix der Konfigurationen genau gleich aufgeteilt und rekursiv übergeben. So kann jedem Quadrat weiterhin die Position im gesamten Konfigurationsraum zugeordnet werden, was nötig ist, um bei einem eindeutigen Quadrat die Konfiguration in der Quadratmitte anzugeben. Das wäre mit nur einem Teil der binären Darstellung nicht möglich. Wird ein Quadrat als frei identifiziert berechnet die Funktion $getNodeFromValidSquare()$ die Konfiguration, die der Mitte des Quadrates entspricht, indem sie den Mittelwert der Konfigurationen am Rand berechnet. Diese Konfiguration ist nun ein Knoten des Quadtrees und der Index wird in die Liste $indexcodes\_all$ der Knoten übernommen. Wird ein Quadrat als komplett belegt identifiziert wird der Index in der Matrix $indexcodes\_occupied$ gespeichert. Nur wenn ein Quadrat eindeutig frei oder belegt ist endet der rekursive Funktionsaufruf und der gefundene Knoten wird als Ausgabewert übergeben. Somit garantiert die rekursive Struktur das Finden aller möglichen Knoten.

### 4.3.2 Suche nach Nachbarn

Vorweg sollen einige Begrifflichkeiten definiert werden. Als Nachbarn eines Quadrates sind jene Quadrate zu verstehe, die direkt an das betrachtete Quadrat angrenzen, oder eine gemeinsame Ecke mit dem betrachteten Quadrat besitzen. Ist das angrenzende Quadrat größer als das betrachtete Quadrat spricht man von einem größeren Nachbarn. Außerdem gibt es noch kleinere und gleich große Nachbarn. Als Elternquadrat wird jenes Quadrat bezeichnet, aus dem das betrachtete Quadrat direkt entstanden ist. Als Vorfahren werden das Elternquadrat und die Quadrate bezeichnet, die mehr als einen Aufteilungsschritt vom betrachteten Quadrat zurückliegen. Für das Quadrat mit dem Index 1234 wäre das Quadrat mit dem Index 123 das Elternquadrat und die Quadrate mit den Indizes 123, 12 und 1 die Vorfahren.

Zur Berechnung der Nachbarn werden die Funktionen $findNeighboursSameSize()$ und $newIndexAndDirection()$ verwendet. Diese Funktionen setzten die Finite State Machine aus [Rob2006, 251] um, welche aber nur geeignet ist, um Nachbarn der gleichen Größe zu finden.

| Suchrichtung | Quadrant 1 | Quadrant 2 | Quadrant 3 | Quadrant 4 |
|---|---|---|---|---|
| **Rechts** | 2, halt | 1, rechts | 4, halt | 3, rechts |
| **Links** | 2, links | 1, halt | 4, links | 3, halt |
| **Unten** | 3, halt | 4, halt | 1, unten | 2, unten |
| **Oben** | 3, oben | 4, oben | 1, halt | 2, halt |
| **Rechtsunten** | 4, halt | 3, rechts | 2, unten | 1, rechtsunten |
| **Rechtsoben** | 4, oben | 3, rechtsoben | 2, halt | 1, rechts |
| **Linksunten** | 4, links | 3, halt | 2, linksunten | 1, unten |
| **Linksoben** | 4, linksoben | 3, oben | 2, links | 1, halt |

*Tabelle 4.1: FSM nach [Rob2006, 251] zur Bestimmung von gleich großen Nachbarn in Quadtrees.*

Die FSM nutzt aus, dass die Quadrate immer in der gleichen Reihenfolge angeordnet sind. Ein Quadrat, dessen Index auf 1 endet, hat als rechten Nachbarn also immer ein Quadrat, dessen Index auf 2 endet. Beide Quadrate sind in diesem Beispiel aus dem gleichen Elternquadrat entstanden. Auch ein linker Nachbar eines Quadrates, dessen Index auf 1 endet, besitzt als letzte Indexziffer eine zwei ist aber nicht aus demselben Elternquadrat entstanden. Deshalb müssen weitere Ziffern des Index betrachtet werden. Dies entspricht der Suche nach einem ersten gemeinsamen Vorfahren.

Für die Suche nach dem gleich großen Nachbar auf der rechten Seite des Quadrates 414 wird die FSM aus Tabelle 1 wie folgt genutzt: Die letzte Ziffer und die initiale Suchrichtung geben mit der Tabelle die letzte Ziffer des Nachbarnindex und die neue Suchrichtung vor. In diesem Fall wird der Index zu 413 und die Suchrichtung bleibt rechts. Nun wird die zweitletzte Ziffer, also die Ziffer 1 betrachtet. Der Index wird zu 423 und als neue Suchrichtung gibt die FSM „halt“ aus. Damit ist die Suche abgeschlossen und der Index des Nachbarn auf der rechten Seite lautet 423

Die Funktion $findNeighboursSameSize()$ berechnet mit Hilfe der FSM die Indizes aller acht gleich großen Nachbarn eines Knoten. Es wird zu diesem Zeitpunkt nicht bestimmt, ob diese Nachbarn tatsächlich existieren oder nicht, denn die Indizes werden später auch benötigt, um nach größeren Nachbarn suchen zu können.

$getAllNeighbours()$ berechnet alle Nachbarn eines Knoten, die wirklich existieren. Hierfür werden die Indizes der aller theoretisch existenten, gleich großen Nachbarn und der Index des aktuellen Knoten an die Funktion übergeben. Zusätzlich benötigt die Funktion Zugriff auf $indexcodes\_all$ und $indexcodes\_occupied$. Die Funktion betrachtet nacheinander alle acht Richtungen. Zuerst wird in den Listen der Indizes nach dem gleich großen Nachbar in dieser Richtung gesucht, um zu überprüfen, ob dieser tatsächlich existiert. Wird er in $indexcoes\_all$, also der Liste der freien Knoten, gefunden

wird die Adresse des Nachbarn gespeichert. Wird der Index des Nachbarn in $indexcodes\_occupied$ gefunden wird die Adresse nicht gespeichert und die Suche kann in der nächsten Richtung fortgesetzt werden.

Existiert kein gleich großer Nachbar wird anschließend nach einem größeren Nachbar in der Richtung gesucht. Ein größerer Nachbar kann nur dann existieren, wenn außerhalb des Elternquadrates gesucht wird, da alle anderen Nachbarn mit dem gleichen Elternquadrat maximal gleich groß sein können. Die FSM gibt nicht „halt“ aus, wenn die Suchrichtung das Elternquadrat verlässt und kann so genutzt werden, um zu bestimmen ob in einer Richtung nach einem größeren Nachbarn gesucht werden soll. Existiert ein größerer Nachbar in der Suchrichtung muss er zwangsläufig ein Vorfahre des gleich großen Nachbarn sein. Der Index eines Vorfahren wird ermittelt, indem der Index des gleich großen Nachbarn in dieser Richtung schrittweise um eine Ziffer verkürzt wird. Nach diesem verkürzten Index wird in den Listen der Indizes gesucht. Es wird so lange nach einem größeren Nachbarn gesucht, bis er gefunden wird oder der Suchindex keine Ziffern mehr enthält. Wurde ein größerer Nachbar in $indexcodes\_all$ gefunden ist er ein valider Nachbar. Befindet sich der Index in $indexcodes\_occupied$ wird die nächste Richtung betrachtet. Wurde kein größerer Nachbar gefunden folgt daraus, dass in dieser Richtung ein kleinerer Nachbar existieren muss. Diese Nachbarschaftsbeziehung wird jedoch zunächst ignoriert, da das kleinere Quadrat das größere Quadrat später als größeren Nachbar identifizieren wird.

Ist die Nachbarsuche in alle Richtungen beendet übergibt die Funktion $getAllNeighbours()$ die Adressen aller gefundener Nachbarn als Matrizen. Es wird eine Matrix mit den Adressen der gleich großen Nachbarn und eine Matrix mit den Adressen der größeren Nachbarn übergeben. In der Hauptfunktion $createQuadtree()$ wird für jede der 3 Nachbargrößen ein Zellarray erstellt. Es müssen Zellarrays verwendet werden, da die Anzahl an Nachbarn von Knoten zu Knoten variieren kann. Die Matrizen mit den Nachbaradressen werden in den Zellarrays in der Zeile gespeichert, die der Adresse des betrachteten Knoten entspricht. Das Zellarray der kleineren Nachbarn bleibt zunächst leer.

Die gesamte zuvor beschriebene Suche nach Nachbarn wird für jeden Knoten wiederholt.

Anschließend werden mit der Funktion $calculateCostToNeighbours()$ die Kosten von einem Knoten zu all seinen Nachbarn berechnet. Dies passiert über eine Berechnung des euklidischen Abstands der Knoten.

Nachbarschaftsbeziehungen sind immer beidseitig. Besitzt beispielsweise ein Knoten A einen größeren Nachbar B so ist der Knoten A automatisch ein kleinerer Nachbar von Knoten B. Bisher ist nur Knoten B als Nachbar des Knotens A eingetragen. Die Funktion $transferNeighbourhoodFromBigToSmall()$ überträgt diese Nachbarschaftsbeziehung, sodass auch A als Nachbar von B gespeichert wird. Gleichzeitig werden auch die entsprechenden Kosten übertragen und gemeinsam mit den übertragenen Adressen in dem Zellarray der kleineren Nachbarn gespeichert.

Die Struktur $nodes$ in der die Arrays der Nachbarn enthalten sind enthält drei Felder für jede Art von Nachbarn. Da Framed-Quadtrees aber beispielsweise nur zwei Klassen von Nachbarn haben ist es sinnvoll eine Standardstruktur zu nutzen, um eine Straßenkarte unabhängig vom Algorithmus, mit dem sie erstellt wurde, weiterverwenden zu können. Eine Straßenkarte soll demnach nur ein Feld $nodes.states$ mit den Konfigurationen der Knoten und ein Feld $nodes.neighbours$ mit allen Nachbarn und deren Kosten enthalten. Die Funktion $restructureNodesStructQT()$ überführt $nodes$ in die gewünschte Struktur.

Damit ist die Berechnung des Quadtrees abgeschlossen und die Straßenkarte kann mit einem Graphsuchalgorithmus weiterverwendet werden.

## 4.4 Framed Quadtrees

Die Erstellung der Framed Quadtrees unterscheidet sich in 2 Punkten von der Erstellung eines normalen Quadtrees. Erstens muss ein Rahmen aus kleinen Quadraten erstellt werden, sobald ein freies Quadrat gefunden wurde. Zweitens besteht ein Framed-Quadtree wegen des Rahmens am Ende nur aus gleich großen Quadraten. Letzteres erleichtert die Suche nach Nachbarn.

Ein Parameter, der bei Framed-Quadtrees zusätzlich benötigt wird, ist die maximal zulässige Indexlänge. Diese berechnet sich aus der Größe der Karte und der minimal zulässigen Quadratgröße mit der Funktion $getMaxIndexLength()$. Das Prinzip der Funktion besteht darin die Indexlänge eines Quadrates zu bestimmen, das die minimal zulässige Größe besitzt. Um Rechenkapazität zu sparen, wird hierfür nicht der tatsächliche Konfigurationsraum genutzt und nach einem minimalen Quadrat gesucht, sondern die Höhe und Breite der Karte werden so lange halbiert, bis die minimale Größe erreicht ist. Bei jeder Vierteilung des Konfigurationsraumes würde sich der Index eines Quadrates um eine Ziffer verlängern, weshalb bei jeder Halbierung der Höhe und Breite der maximal zulässige Index inkrementiert wird.

Die Suche nach freien Quadraten verläuft gleich wie bei den normalen Quadtrees mit der Funktion $divideAndReturnNodes()$. Wird ein freies Quadrat identifiziert wird die Funktion $getFrame()$ zur Erstellung des Rahmens aufgerufen. Diese Funktion erhält als Eingabewerte die zuvor berechnete maximal zulässige Indexlänge, den Index des gefundenen Elternquadrates, die Konfigurationen des Elternquadrates und die minimal zulässige Quadratgröße. Zuerst werden die Indizes der Quadrate des Rahmens berechnet, die im Folgenden als Rahmenquadrate bezeichnet werden. Hierfür wird wieder die Funktionalität der Indizes verwendet. Die Quadrate des oberen Randes enden beispielsweise immer auf eins oder zwei. Um den oberen Rand zu erhalten werden, beginnend mit dem Index des Elternquadrates, immer jeweils die Ziffer 1 und die Ziffer 2 an alle Indizes angehangen, wodurch sich die Anzahl der Indizes in jedem Schritt verdoppelt. Dies wird so lange wiederholt, bis die Indizes die maximal zulässige Länge erreicht haben. Für den rechten, linken und unteren Rand verläuft das Prozedere äquivalent, jedoch werden der Richtung entsprechend andere Ziffern angehängt.

Als nächstes werden die Konfigurationen berechnet, die zu den jeweiligen Indizes gehören. Da die Quadrate des Rahmens gleichmäßig über diesen verteilt sind können die Konfigurationen unabhängig von der Erstellung der Indizes bestimmt werden. Aus der Matrix der Konfigurationen werden zunächst die Zustände der Rahmenquadrate in den Ecken bestimmt. Hierfür werden in jeder Ecke der Matrix der Konfigurationen Quadrate entfernt, die der minimalen Größe entsprechen. Mit der Funktion $getNodeFromValidSquare()$ werden die Konfigurationen in der Mitte der Eckquadrate bestimmt. Aus der Konfiguration des Quadrates rechts oben und links oben können die Konfigurationen aller Quadrate am oberen Rand bestimmt werden. Hierfür wird ein linearer Verlauf zwischen den zwei Konfigurationen an den Ecken erstellt, der so viele Werte besitzt, wie Indizes für den oberen Rand existieren. Dasselbe wird für die anderen Ränder durchgeführt.

Die Indizes und Konfigurationen der vier Richtungen werden anschließend zu jeweils einer Matrix zusammengefügt. Um die Ecken nicht mehrfach zu speichern, werden bei den Matrizen des rechten und des linken Randes jeweils der erste und der letzte Eintrag weggelassen.

Im weiteren Verlauf wird nach Nachbarn gleicher Größe gesucht. Hierfür ist es wichtig auf welcher Seite des Rahmens ein Knoten liegt, da davon die Richtung abhängt, in der gleich große Nachbarn außerhalb des eigenen Elternquadrates existieren. Diese Richtung wird gemäß den Richtungen der FSM kodiert und für jeden Knoten in einer Matrix gespeichert. Es ist wichtig, dass die Information zu diesem Zeitpunkt gespeichert wird, weil sie nicht aus den Indizes gelesen werden kann, wenn der Index des Elternquadrates nicht bekannt ist. Da zu diesem Zeitpunkt alle Informationen vorliegen wird die Berechnung ausgeführt und der Index des Elternquadrates kann verworfen werden.

Jedes Rahmenquadrat soll als Nachbarn jedes andere Rahmenquadrat besitzen, das zum gleichen Elternquadrat gehört. Um diese Beziehung festzuhalten, werden zunächst lokale Adressen verwendet, da der Funktion $getFrame()$ nur die Rahmenquadrate des aktuellen Elternquadrates bekannt sind und deshalb nicht sagen kann welche Adressen bisher existieren. Die lokale Adresse der Nachbarn in dieser Funktion entspricht also der Zeile der Matrix, in der die Nachbarn gespeichert sind. Diese Adresse hat jedoch nichts mit der Adresse zu tun, die ein Nachbarknoten später in der Matrix $indexcodes\_all$ haben wird. Deshalb werden nach Beenden von $getFrame()$ die Nachbaradressen transformiert. Hierfür wird die Anzahl der bisher gefundenen Knoten, also die Länge von $indexcodes\_all$, zu den lokalen Adressen addiert. Dieser Schritt passiert auf jeder Ebene des rekursiven Funktionsaufrufes, bis alle Knoten mit ihren Nachbarn in dieselbe Liste übertragen wurden.

Nachdem alle Knoten bestimmt wurden und die Nachbarn innerhalb eines Rahmens vermerkt sind, müssen noch gleich große Nachbarn in anderen Rahmen gesucht werden. Wie zuvor erwähnt wurde für jeden Knoten seine Position innerhalb des Rahmens gespeichert, da so klar ist in welchen Richtungen nach Nachbarn gesucht werden muss. Ein Rahmenquadrat am oberen Rand kann beispielsweise keinen Nachbarn auf der unteren Seite haben, der zu einem anderen Rahmen gehört. Im Falle

des oberen Randes ohne die Ecken muss also nur oben, rechtsoben oder linksoben nach Nachbarn gesucht werden. Die zulässigen Suchrichtungen werden von der Funktion $getValidSearchDirections()$ bestimmt, die als Eingabe die Position im Rahmen erhält.

Bei der Erstellung der normalen Quadtrees wurde eine ähnliche Bedingung bei der Suche nach größeren Nachbarn gestellt. Es wurde nur in solchen Richtungen nach größeren Nachbarn gesucht, in denen des Elternquadrat verlassen wird. Dieses Vorgehen ist hier jedoch nicht anwendbar, da die Rahmenquadrate häufig viel kleiner sind als das Quadrat, aus dem sie entstanden sind. Es liegen also viele Generationen zwischen diesen Quadraten. Schaut man nur eine Generation zurück, wie es bei normalen Quadtrees der Fall ist, erkennt man, dass Quadrate zwar zum selben Rahmen gehören, aber nicht dasselbe direkte Elternquadrat haben. Bei normalen Quadtrees wäre eine Suche in dieser Richtung zulässig. Bei Framed-Quadtrees führt es aber nur dazu, dass Nachbarschaften sowohl bei den gleich großen Nachbarn als auch bei den Nachbarn im Rahmen eigentragen werden, wodurch die Rechenzeit verlängert wird. Ein weiterer Effekt der zuvor bestimmten Richtungen ist, dass die FSM seltener aufgerufen werden muss, um zu prüfen, ob eine Suchrichtung valide ist, woraus ebenfalls eine kürzere Rechenzeit resultiert.

Die folgenden Schritte sind äquivalent zur Erstellung eines normalen Quadtrees und sollen deshalb nur kurz genannt werden. Eine genauere Darstellung ist Kapitel 4.1.2 zu entnehmen.

Zuerst werden für alle validen Richtungen die Indizes der potenziellen Nachbarn in dieser Richtung bestimmt und ihre Existenz in indexcodes_all überprüft. Falls die Nachbarn existieren, wird deren Adresse gespeichert. Die Suche nach gleich großen Nachbarn wird für alle Knoten wiederholt. Danach werden für alle Knoten und all ihre Nachbarn noch die Kosten berechnet und die Straßenkarte wird mit $restructureNodesStructFQT()$ in die Standardform umstrukturiert.

Damit ist die Erstellung des Framed-Quadtrees abgeschlossen und die daraus entstandene Straßenkarte kann mit einem Graphsuchalgorithmus weiterverwendet werden.

## 4.5 A* Algorithmus

Im Folgenden sollen Besonderheiten der Implementierung des A* Algorithmus beschrieben werden. Die Funktionsweise und Idee des Algorithmus wurden bereits in 3.2.2 erklärt.

Die Eingabedaten des Algorithmus bestehen aus einer Straßenkarte, der Startkonfiguration und der Endkonfiguration. Die Straßenkarte muss alle Knoten, deren Nachbarn und die Kosten zu den Nachbarn enthalten.

Die open-list wird als eine Matrix mit 5 Spalten erstellt. In der ersten Spalte befindet sich die Adresse des jeweiligen Knoten. Die Adresse beschreibt die Zeile in der Matrix aller Knoten, in welcher der Knoten gespeichert wurde. Dies ist nötig, um auf die Nachbarn oder die Koordinaten eines Knotens zugreifen zu können, da diese Informationen nicht in der open-list gespeichert werden.

In der zweiten Spalte wird die Adresse des Vorgängers gespeichert. Die dritte Spalte enthält die Wegkosten. In der vierten Spalte wird die Heuristik vermerkt und in der fünften Spalte werden die Gesamtkosten gespeichert.

In vielen Implementierungen wird die open-list nach den Gesamtkosten sortiert, um den Knoten mit den geringsten Gesamtkosten zu erhalten. Da die Reihenfolge der Knoten aber irrelevant ist und nur der Knoten mit den geringsten Kosten von Interesse ist wird stattdessen nur nach dem minimalen Element der Liste gesucht und die restliche open-list verbleibt unsortiert.

Die closed-list besteht nur aus einer Spalte und besitzt so viele Zeilen wie es Knoten gibt. Zu Beginn ist die closed-list mit NaN gefüllt. Wird ein Knoten expandiert und dadurch auf die closed-list gesetzt wird in der Zeile, die der Adresse des Knotens entspricht, der Vorgänger des Knotens gespeichert. Alle anderen Informationen, die in der open-list gespeichert waren, sind nichtmehr relevant und können verworfen werden. Durch Belegung der closed-list anhand der Adressen wird ermöglicht, dass schnell überprüft werden kann, ob ein Knoten bereits expandiert wurde. Hierfür muss nur die entsprechende Zeile aufgerufen werden und überprüft werden, ob sie NaN oder einen Zahlenwert enthält.

Zu Beginn des Algorithmus müssen noch der Startknoten und der Zielknoten mit der Funktion $includeStartAndGoal()$ in das Netz integriert werden. Hierfür wird für jeden Knoten der Abstand zu Start und Ziel berechnet. Es werden jeweils die vier Knoten als Nachbarn gespeichert, die den geringsten Abstand zu Start und Ziel besitzen und kollisionsfrei sind. Für den Endknoten muss die Nachbarschaftsbeziehung auf die Nachbarn übertragen werden, sodass sie beidseitig besteht. Bei dem Startknoten ist dies nicht nötig, da die Pfadplanung bei ihm beginnt, er also nie von anderen Knoten gefunden werden muss.

Wurde ein Pfad gefunden wird die Hauptschleife des Algorithmus beendet und der Pfad rekonstruiert. Hierfür wird die closed-list verwendet, da sie die Vorgänger aller expandierten Knoten enthält. Beginnend bei dem Zielknoten werden die Konfigurationen des Pfades in einer Matrix abgespeichert. In einer while-Schleife werden so lange die Vorgänger bestimmt und die Konfigurationen gespeichert, bis es keinen Vorgänger mehr gibt, was nur für den Startknoten der Fall sein kann.

Die Berechnung des Pfades ist damit beendet und als Ausgabewerte werden, falls existent, die Konfigurationen der Punkte des Pfades übergeben.

## 4.6 D* Lite

Die Implementierung des D* Lite Algorithmus wurde basierend auf dem Pseudocode aus [Koe2002, 480] erstellt.

Zuerst sollen die gewählten Datenstrukturen und verwendeten Variablen erklärt werden.

Jedem Knoten werden im Verlauf der Berechnung Werte zugewiesen, die gespeichert werden müssen. Zu diesen Werten zählen $g(n)$, $rhs(n)$, $h(n)$, $k_1(n)$ und $k_2(n)$, die in der Matrix $allNodes$ gespeichert werden. Wie bei A* besitzt jeder Knoten eine Adresse, die ihm eine Zeile in Matrizen zuordnet, die alle Informationen zu allen Knoten enthalten. $allNodes$ besitzt demnach so viele Zeilen wie Knoten existieren und speichert in 5 Spalten für jeden Knoten die zuvor genannten Werte in der zuvor genannten Reihenfolge.

Die Matrix $openList$ enthält in einer Spalte die Adressen aller Knoten, die sich aktuell in der open-list befinden.

Die Nachfolger aller Knoten werden in dem Zellarray $successors$ gespeichert, da nicht alle Knoten die gleiche Anzahl an Nachfolgern besitzt. In der $n$-ten Zeile und der ersten Spalte befindet sich der Zeilenvektor mit den Adressen aller Nachfolger. In der zweiten Spalte wird der Zeilenvektor mit den Kosten zu den Nachfolgern gespeichert.

Die Straßenkarte wird in der Matlab-structure $nodes$ gespeichert und enthält das Feld $nodes.states$, welches die Konfiguration jedes Knotens enthält, und das Feld $nodes.neighbours$, in welchem alle Nachbarn aller Knoten und die Kosten zu diesen Nachbarn gespeichert werden.

Im Folgenden sollen die einzelnen Funktionen und deren Annahmen in Bezug auf den Pseudocode erklärt werden.

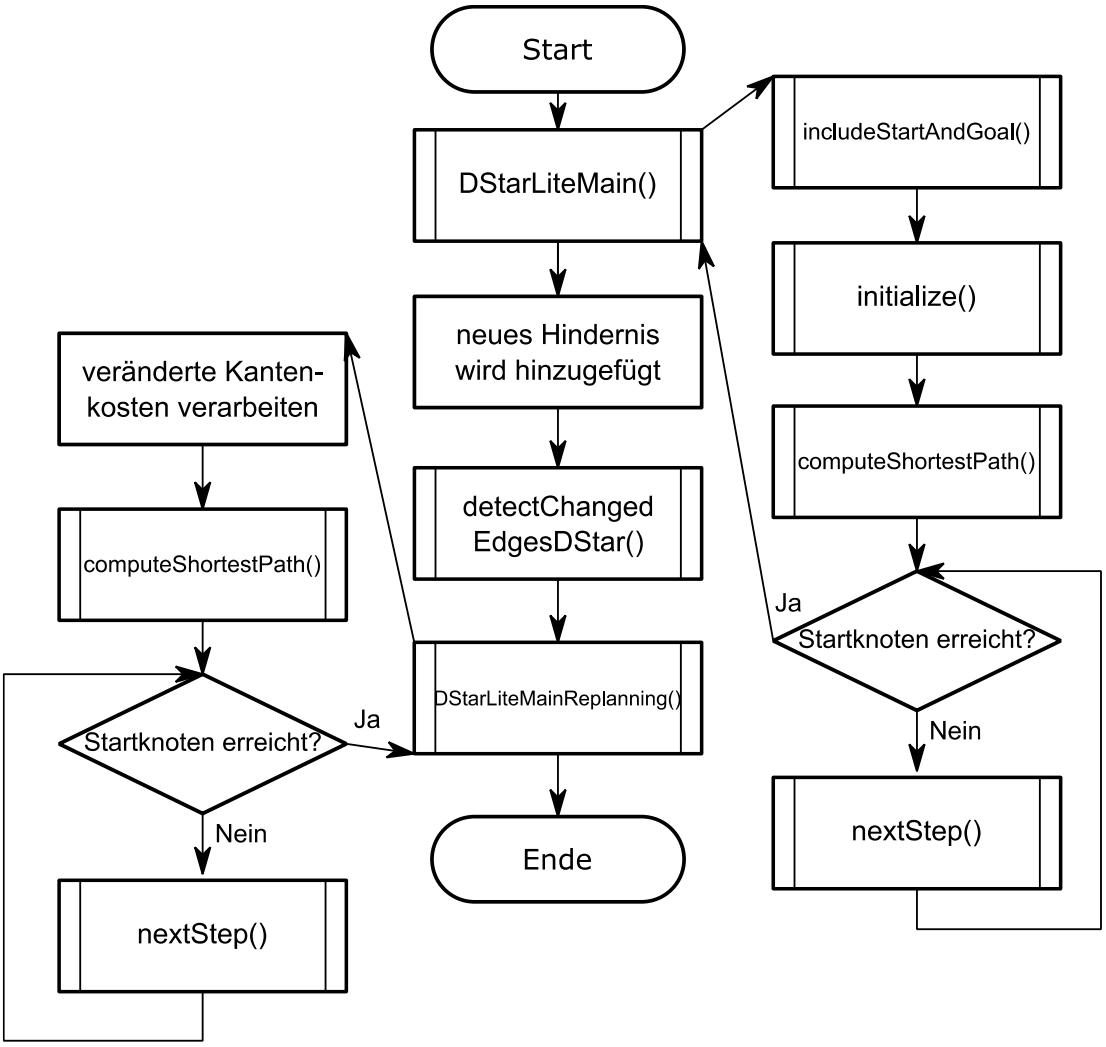


*Abbildung 4.4: Schematischer Ablauf der wichtigsten Funktionen der Implementierung des D* Lite Algorithmus.*

Im Pseudocode wird nur eine Hauptfunktion verwendet. In jeder Iteration sucht diese Hauptfunktion nach veränderten Kanten und bewegt den Roboter einen Schritt in Richtung Ziel. Im Rahmen dieser Arbeit soll aber gezielt das Verhalten des Roboters bei einer Neuplanung untersucht werden, wofür diese von der initialen Pfadplanung getrennt werden muss. Zudem soll sich der Roboter die gesamte Zeit an der Startposition befinden. Da sich das Verhalten der Hauptfunktion sehr zwischen der initialen Pfadplanung und der Neuplanung unterscheidet wurde sie in der Implementierung in die zwei Funktionen $DStarLiteMain()$ und $DStarLiteMainReplanning()$ unterteilt. Dies bringt außerdem den Vorteil, dass der Konfigurationsraum zwischen den Pfadplanungen verändert werden kann, um aussagekräftige Simulationsexperimente zu erhalten.

$DStarLiteMain()$ ist die Funktion, die bei der initialen Pfadplanung verwendet wird. Als Eingabewerte erhält sie $nodes$, und die Start- und Zielkoordinaten. Wie bei A* werden zu Beginn Start und Ziel in das Netzt der Knoten eingefügt (siehe Kapitel 4.5). Danach folgt der Initialisierungsschritt, welcher mit Hilfe der Funktion $initialize()$ durchgeführt wird. Diese Funktion erstellt $openList$ und setzt die Adresse des Zielknotens darauf. Außerdem wird die Matrix $allNodes$ mit fünf Spalten und so viele Zeilen wie es Knoten

gibt erstellt. Die erste und zweite Spalte von $allNodes$ wird vollständig mit ∞ besetzt, da diese Spalten den Werten $g(n)$ und $rhs(n)$ entsprechen und die Initialisierung es so vorsieht. Anschließend wird für jeden Knoten die Heuristik $h(n)$ berechnet und in der dritten Spalte gespeichert. Zuletzt wird noch die Kennzahl des Zielknotens berechnet, da dieser sich bereits auf der open-list befindet.

Anschließend wird $computeShortestPath()$ ausgeführt. In dieser Funktion geschieht die eigentliche Pfadplanung nach dem Prinzip, wie es in Kapitel 3.3.3 beschrieben wurde. Die Eingabedaten sind $km$, die Adresse des Starknotens, $nodes.neighbours$, $nodes.states$, die Koordinaten des Ziels, $successors$, $allNodes$ und $openList$.

Innerhalb des Pseudocodes werden verschiedene Funktionen verwendet, die mit der open-list interagieren. $U.Remove(u)$ wird genutzt, um Knoten von der open-list zu entfernen. Da diese Operation aber nur eine Zeile benötigt wird statt einer Funktion direkt der Befehl zum Entfernen des Elements genutzt, wodurch der Code beschleunigt wird. Auch die im Pseudocode verwendete Funktion $U.Update()$ kann in der Implementierung durch eine Zeile Code ersetzt werden, da sie ausschließlich eine neue Kennzahl für einen Knoten abspeichert.

Außerdem werden im Pseudocode zu Beginn jeder Iteration die Funktionen $U.Top()$ und $U.TopKey()$ verwendet. $U.Top()$ berechnet den Knoten mit der geringsten Kennzahl und $U.TopKey()$ berechnet die geringste Kennzahl. Da eine Bestimmung des Knotens mit der geringsten Kennzahl aber nicht möglich ist, ohne auch die geringste Kennzahl zu bestimmen ist es nicht notwendig hierfür zwei Funktionen zu verwenden. In der Implementierung wird deshalb nur die Funktion $top()$ verwendet. Sie berechnet die Stelle an der open-list, die den Knoten mit der geringsten Kennzahl enthält und die Kennzahl an dieser Stelle. Als Eingabewerte erhält $top()$ die Kennzahlen aller Elemente auf der open-list.

In den Bedingungen der while-Schleife muss die geringste Kennzahl mit der Kennzahl des Startknotens verglichen werden. Weil Kennzahlen aber immer zwei Elemente haben und Funktionen, die in Matlab in Bedingungen eingebunden werden, nur einen Ausgabewert verwenden können müssen jeweils zwei Funktionen genutzt werden. $top1()$ und $calculateKey1()$ vergleichen $k_1$ während $top2()$ und $calculateKey2()$ $k_2$ vergleichen.

Während der Ausführung müssen häufig Knoten aktualisiert werden. Das geschieht mit der Funktion $updateVertex()$. Für den Fall, dass der aktuell betrachtete Knoten $n$ lokal überkonsistent ist, hat $updateVertex()$ in der Implementierung noch eine zusätzliche Funktion. Wird ein lokal überkonsistenter Knoten $n$ betrachtet wird dieser zuerst lokal konsistent gemacht und anschließend werden alle Nachbarn $s$ dieses Knotens überprüft. Bei der Überprüfung wird $rhs(s)$ neu berechnet. Wenn $rhs(s)$ basierend auf dem jetzt lokal konsistenten Knoten $n$ berechnet wird, wird $s$ zwangsläufig lokal inkonsistent. $s$ kann seine Kosten also verringern, wenn sie basierend auf $n$ berechnet werden. Somit ist $s$ ein Nachfolger von $n$, wenn $s$ nach der Expansion von $n$ inkonsistent wird. Da für alle Nachbarn von $n$ $updateVertex()$ aufgerufen wird und $updateVertex()$ die Konsistenz prüft kann mit dem Funktionsaufruf auch festgestellt werden, ob es sich bei

dem Knoten um einen Nachfolger handelt. Hierfür wird $validSuccessor$ entweder $true$ oder $false$. Abhängig davon wird danach der Knoten $s$ in $successors$ aufgenommen oder nicht.

Im Pseudocode wird der rhs-Wert eines Knotens n wie folgt berechnet:

$$rhs(n) = min_{s \in Nachfolger(n)} (c(n,s) + g(s))$$

In der Implementierung wird hierfür die Funktion $nextStep()$ verwendet, die als Eingabedaten die Nachfolger von $n$ und $allNodes$ erhält. Die Funktion iteriert über alle Nachfolger und bestimmt den Nachfolger, bei dem die geringsten Kosten entstehen. Anschließend werden die Adresse und die Kosten des besten Nachfolgers ausgegeben.

Wie zuvor erwähnt wurde die Hauptfunktion in zwei Funktionen aufgeteilt. Es wurden nun bereits die Besonderheiten von $DStarLiteMain()$ und den Unterfunktionen aufgezeigt. Die zweite Hauptfunktion $DStarLiteMainReplanning()$ unterscheidet sich nicht in den verwendeten Unterfunktionen, sondern es wird anstatt der Initialisierung zu Beginn die Veränderung des Konfigurationsraums verarbeitet. Wie diese Veränderung bestimmt werden soll, wird im Pseudocode nicht näher angegeben. In der Implementierung wird hierfür die Funktion $detectChangedEdgesDStar()$ verwendet. Als Eingabedaten erhält sie $nodes$, die binäre Darstellung der Karte, die Auflösung der Karte und $validationDistance$. Die Funktion iteriert über alle Knoten und überprüft mit $motionValidatorBinary()$, ob jeder Nachbar jedes Knotens erreichbar ist. Ist das nicht der Fall wird die Kante als belegt in der Matrix $changedEdges$ gespeichert. $changedEdges$ enthält in der ersten Zeile den Anfangsknoten, in der zweiten Zeile den Endknoten und in der dritten Zeile die neuen Kosten der veränderten Kante. Die Spalten der Matrix entsprechen jeweils einer Kante. Sobald über alle Kanten iteriert wurde ist die Funktion beendet und $changedEdges$ wird übergeben.

$DStarLiteMainReplanning()$ kann nun mit den Eingabevariablen $nodes$, der Startadresse, der Zieladresse, $changedEdges$, $km$, $successors$, $allNodes$ und $openList$ aufgerufen werden. Für den Rest der Funktion kann der Pseudocode in [Koe2002, 480] ab Zeile 38 ohne Besonderheiten in Matlab übersetzt werden.

Da der Pseudocode von einer schrittweisen Ausführung der Pfadplanung ausgeht, bei der sich der Roboter in jeder Iteration ein Stück bewegt wird zu keinem Zeitpunkt der globale Pfad berechnet. Die schrittweise Vorschrift, um den nächsten Schritt zu bestimmen kann jedoch beginnend beim Startknoten so lange angewandt werden bis der Zielknoten erreicht ist. Diese Vorschrift lautet:

$$n_{next} = min_{s \in Nachfolger(n)} (c(n,s) + g(s))$$

Wie zuvor beschrieben wird genau diese Berechnung von der Funktion $nextStep()$ ausgeführt. Am Ende der beiden Hauptfunktionen wird deshalb mit Hilfe von $nextStep()$ der gesamte Pfad bestimmt.

## 4.7 RRT*

Der RRT* Algorithmus wurde im Rahmen dieser Arbeit nicht implementiert, da dieser in der Matlab Navigation Toolbox enthalten ist. Die Funktionsweise und Anwendung dieses Algorithmus kann der Dokumentation der Toolbox entnommen werden.

Eine Einstellungsmöglichkeit bei der Nutzung dieses Algorithmus ist der Parameter $ContinueAfterGoalReached$. Wird dieser Parameter gleich $false$ gesetzt stoppt die Berechnung des Pfades, sobald das Ziel in den Baum integriert wurde und der Baum wird nicht durch weitere Knoten optimiert.

Mit dieser Einstellung liefert der RRT* Algorithmus einen Anhaltswert ab welchem Zeitpunkt ein Pfad vorliegt, der theoretisch ausgeführt werden könnte. Es wäre also denkbar die Ausführung des Pfades zu beginnen und den Baum währenddessen zu optimieren.

Da RRT* und RRT$^X$ die gleiche Zeitkomplexität besitzen und den initialen Pfad auf die Gleiche Art berechnen ist die Planungsdauer, die der RRT* Algorithmus mit $ContinueAfterGoalReached = false$ benötigt auch als untere Schranke für die Planungsdauer des RRT$^X$ Algorithmus repräsentativ [Ott2016, 826].

## 4.8 RRT$^X$

Der RRT$^X$-Algorithmus wurde auf Basis des Pseudocodes in [Ott2016] implementiert. Ein Unterschied besteht jedoch darin, dass im Rahmen dieser Arbeit nur der Fall eines hinzugefügten Hindernisses und nicht der Fall eines entfernten Hindernisses betrachtet werden soll. Diese Vereinfachung wurde getroffen, da D*Lite in Kombination mit klassischen Quadtrees nicht in der Lage ist entfernte Hindernisse zu verarbeiten und so eine bessere Vergleichbarkeit besteht. Der Grund hierfür ist, dass bei einer Quadtreezerlegung keine Knoten im Bereich von Hindernissen erstellt werden und deshalb auch keine Kantenkosten verändert werden können, wenn das Hindernis entfernt wird. Da das Entfernen von Hindernissen in der Berechnung sehr ähnlich zum Hinzufügen von Hindernissen ist sind in Sachen Laufzeit auch keine zusätzlichen Ergebnisse zu erwarten. Im Falle von RRT$^X$ kann die Funktion mit sehr geringem und im Falle der Quadtrees mit etwas höherem Aufwand eingefügt werden.

Außerdem wird bei dem Pseudocode davon ausgegangen, dass sich der Roboter zwischen jeder Iteration bewegt und die aktuelle Position immer in der Variablen $vBot$ gespeichert wird. Im Rahmen dieser Arbeit wird angenommen, dass der Roboter an der Startposition verbleibt und sich auch zum Zeitpunkt der Neuplanung immer am Startpunkt befindet. Somit entspricht die Position des Roboters $vBot$ immer dem Start. Wenn die Position des Roboters später in Form von Sensordaten vorliegt, können die

Werte einfach in $vBot$ gespeichert werden, weshalb eine nachträgliche Implementierung dieser Funktion wenig Aufwand darstellt.

Für die Implementierung von RRT$^X$ wird als Datenstruktur erneut eine Matlab-Struktur mit mehreren Feldern verwendet. Die einzelnen Felder sind Listen, in denen die verschiedenen Daten zu den Knoten gespeichert werden. Die Zeile, in der die Daten gespeichert sind, stellt die Adresse des Knotens dar. Die Struktur $nodes$ enthält folgende Felder: $nodes.values$ enthält $g(n)$, $lmc(n)$, die Adresse des Vorfahren und die Kosten, um den Vorfahren zu erreichen. Das Feld $nodes.states$ enthält die Konfigurationen der Knoten und in $nodes.childs$ sind die Nachfolger der Knoten gespeichert. Die Nachbarn werden entsprechend der in Kapitel 3.4.3 erwähnten Gruppen in die vier Felder $nodes.neighboursOutInitial$ und $nodes.neighboursOutRunning$, sowie $nodes.neighboursInInitial$ und $nodes.neighboursInRunning$ eingeteilt. Zusätzlich gibt es noch die Matrix $openList$, die in der ersten Spalte die Adressen der Knoten auf der open-list enthält und in der zweiten und dritten Zeile die dem Knoten zugehörigen Kennzahlen $k_1(n)$ und $k_2(n)$.

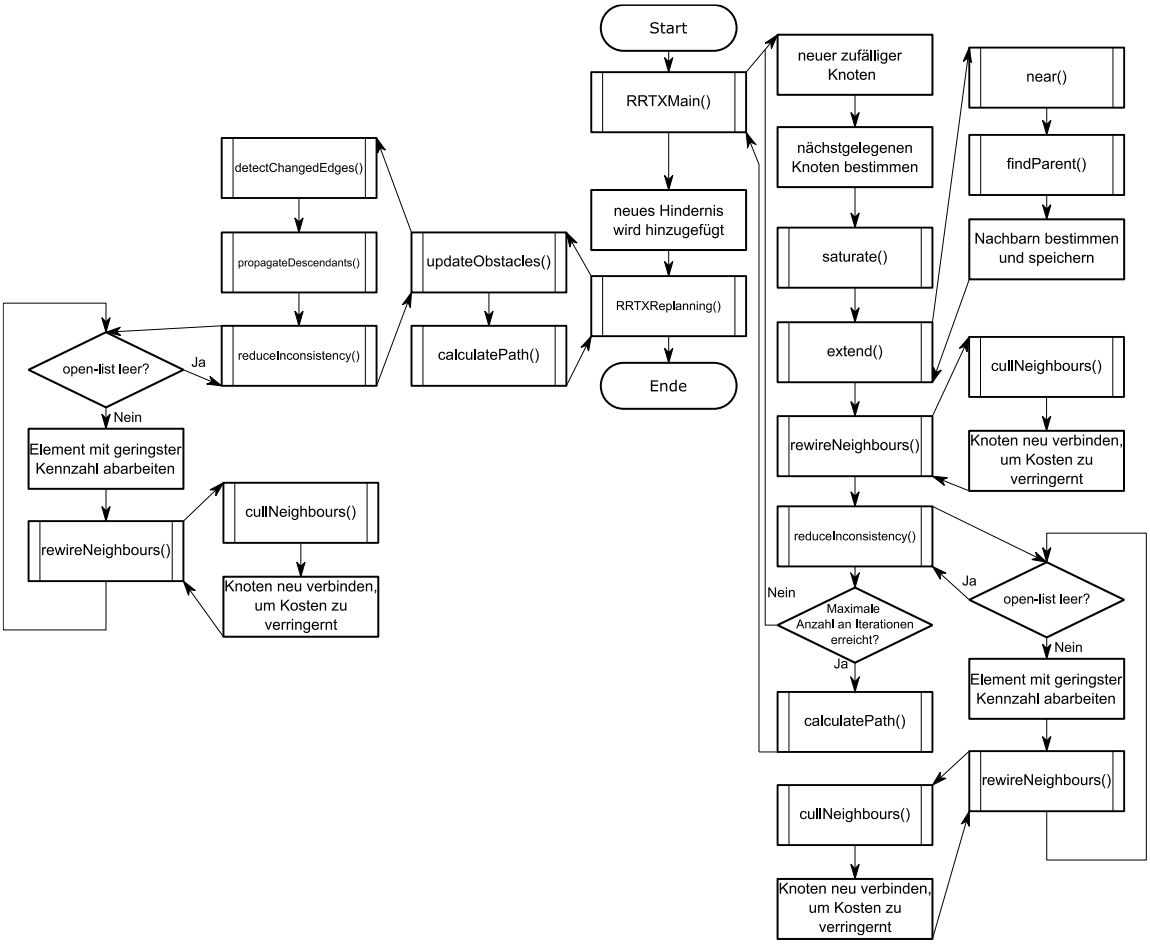


*Abbildung 4.5: Schematischer Aufbau der wichtigsten Funktionen der Implementierung des RRT$^X$ Algorithmus.*

Um die initiale Pfadplanung und die Neuplanung besser getrennt betrachten zu können und dazwischen gezielte Änderungen am Konfigurationsraum vornehmen zu können enthält die Implementierung zwei verschiedene Hauptfunktionen. $RRTXMain()$ ist die

Hauptfunktion der initialen Pfadplanung und $RRTXReplanning()$ ist die Hauptfunktion für die Neuplanung.

Zuerst soll die initiale Pfadplanung und damit $RRTXMain()$ betrachtet werden. Als Input erhält $RRTXMain()$ die binäre Darstellung des Konfigurationsraums $mapBinary$, die Konfiguration des Starts $start$ und die Konfiguration des Ziels $goal$. Außerdem wird die Auflösung der Karte $res$, die Anzahl der Iterationen $iterations$ und $validationDistance$ übergeben. Zudem wird die Variable $maxConectionDistance$ übergeben. Diese stellt die maximale Distanz dar, die zwischen einem neu eingefügten Knoten und seinem Vorfahren bestehen darf.

Zu Beginn der Funktion $RRTXMain()$ werden alle nötigen Variablen initialisiert. Der Start und das Ziel werden zu den Knoten hinzugefügt und deren Werte bestimmt. Da $vBot$, wie zuvor erwähnt, immer der Startposition entspricht kann es nun auch bestimmt werden. Die Adresse des Startknoten ist zwei und damit gilt $vBot = 2$. Hier wird die Adresse, statt der Konfiguration verwendet, da später $g(vBot)$ und $lmc(vBot)$ benötigt werden und so leichter auf diese zugegriffen werden kann.

Anschließend beginnt eine for-Schleife, in der neue Knoten hinzugefügt werden und der Baum aufgebaut wird. Neue Knoten werden durch die Matlab interne Funktion $rand()$ bestimmt. $saturate()$ verändert den neuen Knoten, falls nötig, sodass $maxConnectionDistance$ nicht überschritten wird. Hierfür wird der neue Knoten, der nächste Knoten im Baum und $maxConnectionDistance$ übergeben.

Die Funktion $extend()$ untersucht die Umgebung des neuen Knotens und wählt einen passenden Vorgänger. $extend()$ enthält die Funktionen $near()$ und $findParent()$ und erhält als Eingabeparameter die Variablen, die $near()$ und $findParent()$ benötigen. $near()$ bestimmt alle Knoten, die innerhalb des Radius $r$ um den neuen Knoten liegen. Hierfür wird der neue Knoten, der Radius und $nodes.states$ übergeben. Die Ausgabe besteht aus den Adressen der neuen Knoten und der Distanz zu ihnen. Anschließend wird mit $findParent()$ der beste Vorgänger identifiziert. Als Eingangsgrößen erhält die Funktion die Adressen der nahen Knoten, $nodes$, $mapBinary$, die Auflösung der Karte und die Auflösung der Kollisionskontrolle $validationDistance$. Es wird nun jeder Knoten $p$, der in der Liste der nahen Knoten enthalten ist als potenzieller Vorgänger von $v$ überprüft. Ist die Iteration über alle Knoten abgeschlossen übergibt $findParent()$ die Adresse des Vorfahren, die Distanz des Vorfahren und $lmc(v)$ zurück an $extend()$.

Anschließend werden in $extend()$ die Nachbarn des neuen Knoten bestimmt. Da bereits in der Funktion $near()$ alle potenziellen Nachbarn bestimmt wurden muss nur noch überprüft werden, ob eine Bewegung zwischen dem Knoten und den Knoten in der Nähe möglich ist. Ist das der Fall wird der nahe Knoten zum Nachbarn. Im Pseudocode wird zwischen den Richtungen der Nachbarschaftsbeziehungen unterschieden. Im Rahmen dieser Arbeit werden alle Nachbarschaftsbeziehungen als beidseitig angenommen, wobei in beiden Richtungen die gleichen Kosten entstehen. Diese Annahme ist valide,

da die Kosten nur über euklidische Abstände bestimmt werden und besondere äußere Einflüsse[1] oder nichtholonome Bindungen nicht berücksichtigt werden.

Konnte ein neuer Knoten erfolgreich eingefügt werden übergibt $extend()$ die Adresse $vAdress$ des neuen Knotens und die aktualisierte Struktur $nodes$ an $RRTXMain()$. In $nodes$ wurden die neuen Nachbarschaftsbeziehungen, und alle Werte des neuen Knotens gespeichert. Wenn das Hinzufügen des neuen Knotens nicht erfolgreich war, wird $vAdress$ als leere Matrix übergeben.

Falls $vAdress$ keinen Wert enthält wird die Iteration beendet, ansonsten werden weitere Schritte durchgeführt. Die Funktion $rewireNeighbours()$ untersucht, ob Nachbarn des neuen Knotens mit geringeren Kosten als bisher erreicht werden können, wenn sie den neuen Knoten als Vorgänger wählen. Hierfür werden $vAdress$, $nodes$, $\varepsilon$, der Radius $r$ und die open-list übergeben. Im Verlauf der Pfadplanung erhöht sich die Zahl der Knoten, wodurch sich der Radius $r$ verringert. Damit die Zahl der Nachbarn nicht proportional mit der Knotenanzahl wächst wird zu Beginn jeder Ausführung von $rewireNeighbours()$ die Funktion $cullNeighbours()$ mit dem aktuellen Knoten, dem Radius $r$ und $nodes$ als Eingabewerten aufgerufen. $cullNeighbours()$ überprüft die Distanz zu jedem Nachbarn und entfernt ihn, wenn die Distanz größer als $r$ ist und der Nachbar nicht der Vorfahre des aktuellen Knotens ist.

$rewireNeighbours()$ kann dann damit beginnen die verbleibenden Nachbarn neu zu verbinden. Werden neue Verbindungen erstellt und alte Verbindungen gelöscht ist es wichtig, dass diese Veränderungen auch in $nodes.childs$ festgehalten werden. Im Pseudocode wird nur der neue Vorfahre gespeichert. Um später Änderungen von Kosten an die richtigen Nachfolger weitergeben zu können ist es wichtig, dass zu jedem Knoten seine tatsächlichen Nachfolger gespeichert sind. Wird beispielsweise der Knoten $v$ neu hinzugefügt und wegen besseren Wegkosten zum Vorfahren von Knoten $n$, muss $n$ aus den Nachfolgern seines alten Vorfahren entfernt werden und zu den Nachfolgern von $v$ hinzugefügt werden. Außerdem wurden in der Implementierung zwei for-Schleifen verwendet, um über alle eingehenden Nachbarn zu iterieren. Eine Schleife iteriert über die laufenden eingehenden Nachbarn und die andere Schleife über die ausgehenden laufenden Nachbarn. Das Verwenden von zwei Schleifen erleichtert die Indizierung und erspart das Definieren von Spezialfällen. Zum Schluss wird die Struktur $nodes$ mit neuen Werten und die open-list zurück an $RRTXMain()$ übergeben.

Anschließend wird $reduceInconsistency()$ mit den Eingabewerten $nodes$, $\varepsilon$,$r$ und $vBot$ aufgerufen. Beim Funktionsaufruf werden außerdem die open-list und die Liste der Waisen übergeben. Ziel der Funktion ist es Knoten, deren Werte sich aufgrund des neu hinzugefügten Knotens geändert haben wieder konsistent zu machen und diese Kostenänderung an die nachfolgenden Knoten weiterzugeben. Die open-list enthält alle

[1] Solch ein Einfluss ist beispielsweise eine konstante Strömung bei der Pfadplanung für ein Schiff auf einem Fluss. Strecken, die flussaufwärts führen verursachen mehr Kosten als Strecken, die flussabwärts führen. Die Nachbarschaftsbeziehung wäre hier also wegen den richtungsabhängigen Kosten zwar beidseitig aber mit verschiedenen Kosten. Der Planungsalgorithmus wird als Konsequenz einen Pfad bevorzugen, der mehr Streckenteile, die flussabwärts führen enthält.

inkonsistenten Knoten und wird deshalb nach aufsteigender Kennzahl abgearbeitet. Da sich aufgrund von $reduceInconsistency()$ lmc-Werte ändern, kann es wiederum sein, dass zu einem anderen Knoten ein günstigerer Weg möglich wird. Deswegen wird innerhalb von $reduceInconsistency()$ in jeder Iteration $rewireNeighbours()$ aufgerufen. Anschaulich wird also immer ein Knoten der open-list betrachtet und sein lmc-Wert angepasst. Daraufhin werden seine Nachbarn gegebenenfalls neu verbunden und auf die open-list gesetzt. In der nächsten Iteration werden die lmc-Werte der sich jetzt auf er open-list befindenden Nachbarn angepasst und wiederum deren Nachbarn neu verbunden und auch auf die open-list gesetzt. Dieser Vorgang setzt sich so lange fort, bis der gesamte Baum konsistent ist und nicht weiter optimiert werden kann.

Die Neuplanung wird durch die zweite Hauptfunktion $RRTXReplanning()$ gesteuert. Sie enthält die Unterfunktion $updateObstacles()$. Beide Funktionen erhalten die gleichen Eingabewerte: $nodes$, die open-list, $vBot$, $\varepsilon$, den Radius $r$, $mapBinary$, die Auflösung der Karte und $validationDistance$. Zu Beginn muss $updateObstacles()$ alle Kanten identifizieren, deren Kosten sich geändert haben. Wie zuvor erwähnt wird nur der Fall von hinzugefügten Hindernissen betrachtet. Im Pseudocode gibt es keine Hinweise darauf wie geänderte Kanten erkannt werden können. Deshalb wird statt der im Pseudocode erwähnten Funktion $addNewObstacle()$ die Funktion $detectChangedEdges()$ verwendet. Diese Funktion iteriert über alle Kanten und überprüft, ob diese unzulässig geworden sind. Außerdem identifiziert diese Funktion Waisen, indem überprüft wird, ob die Verbindung eines Knotens zu seinem Vorfahren ungültig geworden ist. Nur über alle Knoten zu iterieren und zu überprüfen, ob deren Konfiguration zulässig ist, wäre zwar einfacher, führt aber auch nicht zum Ziel, da sich Hindernisse auch ausschließlich zwischen Knoten befinden können.

Mit der Funktion $propagateDescendants()$ wird sichergestellt, dass die erhöhten Kosten der Knoten auch an Nachfolger weitergegeben werden. Außerdem sorgt sie dafür, dass alle Nachfolger von Waisen ebenfalls in die Menge der Waisen aufgenommen werden. Hierfür erhält die Funktion die Menge der Waisen, die open-list und $nodes$ als Eingabewerte. Ein Unterschied zum Pseudocode besteht wieder darin, dass über jede Menge von Nachbarn einzeln iteriert wird, da die eingehenden und ausgehenden, sowie die laufenden und anfänglichen Nachbarn in verschiedener Anzahl vorkommen. Eine wichtige Aufgabe dieser Funktion ist es auch den gespeicherten Vorfahren aller Waisen zu entfernen. In der Implementierung wird der Wert durch NaN ersetzt. Dies ist nötig, damit später entschieden werden kann, ob der Ast wieder an den Baum angeschlossen wurde, woraus folgt, dass die Neuplanung erfolgreich war. Die Ausgabewerte sind dieselben Variablen wie die Eingabewerte mit verändertem Inhalt. Auf der open-list sind nun beispielsweise alle Knoten, die den abgetrennten Ast wieder anknüpfen können.

Es wird nun noch ein letztes Mal $reduceInconsistency()$ aufgerufen, was die bekannte Reparaturwelle auslöst. Danach wird die Funktion $updateObstacles()$ beendet und $nodes$ und die open-list werden an $RRTXReplanning()$ übergeben.

Zuletzt muss noch der Pfad aus dem Baum gelesen werden. Im Pseudocode wird darauf nicht näher eingegangen. Falls der Zielknoten einen Vorfahren hat der ungleich NaN ist bedeutet das, dass nach einem Pfad gesucht werden kann. Hierfür wird beginnend vom

Zielknoten immer wieder der Vorfahre des betrachteten Knoten bestimmt, bis der Startknoten erreicht wurde. Diese Aufgabe wird durch die Funktion $calculatePath()$ erfüllt, die dafür $nodes.states$, $nodes.values$ und die Adresse der Roboterposition erhält. Die Ausgabe besteht aus den Konfigurationen der einzelnen Schritte des Pfades und beendet damit die Neuplanung.

# 5. Simulativer Vergleich der Algorithmen

In diesem Kapitel sollen die durchgeführten Simulationsexperimente vorgestellt werden. Zuerst sollen die verwendeten Vergleichsparameter und deren Berechnung erläutert werden. Danach sollen die einzelnen Simulationsexperimente und deren Zweck genauer vorgestellt werden. Zum Schluss soll die Auswertung der Experimente betrachtet werden.

Ziel des simulativen Vergleichs ist es, Aussagen über Vor- und Nachteile der Algorithmen in verschiedenen Szenarien zu treffen. Hieraus soll entschieden werden welche Algorithmen in Zukunft näher betrachtet werden.

In allen Experimenten werden die gleichen Werte für die einstellbaren Parameter der Algorithmen verwendet. Für beide Arten der Quadtrees wird eine minimale Kantenlänge der Quadrate von zwei Pixeln definiert. Die maximale Verbindungsstrecke bei RRT* und RRT$^X$ beträgt $15\,m$. Der RRT$^X$ Algorithmus wird außerdem mit $1500$ Iterationen verwendet.

In allen Experimenten werden folgende Algorithmen verglichen: Quadtrees + A*, Quadtrees + D* Lite, Framed Quadtrees + D* Lite, RRT* und RRT$^X$.

In einer Iteration der Simulationen wird mit allen Algorithmen die gleiche Pfadplanungsaufgabe gelöst. Jedes Experiment wird mit 1000 Iterationen durchgeführt. Bei dieser Anzahl an Iterationen variieren die Ergebnisse wenig, wenn das Experiment mehrfach ausgeführt wird. Dies bedeutet, dass die Ergebnisse reproduzierbar und damit valide sind.

Da es sich bei A* und RRT* nicht um dynamische Pfadplanungsalgorithmen handelt wird im Fall einer Neuplanung der Pfad nicht repariert, sondern komplett neu geplant.

## 5.1 Vergleichsparameter

Der erste Vergleichsparameter ist die Rechenzeit, die ein Algorithmus benötigt, um einen Pfad zu berechnen. Dieser Parameter ist relevant, da in der Anwendung von Pfadplanungsalgorithmen Wartezeiten möglichst vermieden werden sollten. Wartezeiten führen zu erhöhten Taktzeiten in der Produktion und verursachen Mehrkosten. Außerdem können zu lange Rechenzeiten dazu führen, dass nicht schnell genug auf eine Veränderung in dynamischen Umgebungen reagiert werden kann. Daraus können entweder Kollisionen oder eine dauerhafte Neuplanung resultieren, da sich der Konfigurationsraum schneller verändert als der Algorithmus diese Änderungen verarbeiten kann.

Die Rechenzeit wird mit den Matlab-Befehlen $tic$ und $toc$ bestimmt.

Der zweite Vergleichsparameter ist die Länge des Pfades. Ein kurzer Pfad ist wünschenswert, weil er die Ausführdauer und den Energieverbrauch minimiert. Da im Rahmen dieser Arbeit die Länge mit den Kosten gleich gesetzt wird ist hier keine Differenzierung nötig. Bei einer realen Pfadplanung mit mehr Einflüssen auf die Kosten wären die Kosten der sinnvollere Vergleichsparameter.

Die Länge eines Pfades wird aus der Summe der euklidischen Längen der einzelnen Pfadsegmente bestimmt. Die Funktion $calculatePathLength()$ führt diese Berechnung durch.

Der durchschnittliche Winkel des Pfades ist der dritte Vergleichsparameter. Mit diesem Parameter soll bewertet werden, wie abrupt die Richtungsänderungen in einem Pfad sind. Glatteren Pfaden kann in der Realität leichter gefolgt werden. Außerdem ist auf Pfaden mit geringeren Richtungsänderungen eine höhere Pfadgeschwindigkeit möglich.

Der durchschnittliche Winkel des Pfades wird mit der Funktion $pathSmoothness()$ bestimmt. Zuerst werden die Winkel zwischen den Pfadsegmenten bestimmt. Der durchschnittliche Winkel berechnet sich dann als Mittelwert aus allen Winkeln, die ungleich null sind.

## 5.2 Statisches Simulationsexperiment

Mit diesem Experiment soll das Verhalten der Algorithmen in einem statischen Umfeld untersucht werden. Das Experiment existiert in zwei Varianten: Bei der ersten Variante wird für jeden Durchgang eine neue Karte mit der Funktion $mapCreator()$ erstellt. Bei der zweiten Variante wird in jedem Durchgang die Lagerhauskarte verwendet.

Mit dem Skript $StaticComparison.m$ kann die Simulation ausgeführt werden.

Zu Beginn der Simulation werden notwendige Parameter, wie $validationDistance$ oder die Kartengröße festgelegt. Außerdem wir die Anzahl der Durchgänge und die gewünschte Simulationsvariante abgefragt. Über die Konsole können diese Werte eingegeben werden. Danach werden alle Matrizen initialisiert, die nötig sind, um die Vergleichsparameter für jede Iteration zu speichern. Falls in jedem Durchgang die gleiche Karte verwendet werden soll, wird diese vor der ersten Iteration definiert.

Im ersten Durchgang werden bei Verwendung der Variante zwei die Straßenkarten einmalig berechnet. Diese können in allen weiteren Iterationen erneut verwendet werden und werden deshalb nicht neu berechnet. Im Fall von Variante eins wird zu Beginn jedes Durchgangs die neue Karte erstellt. In jeder Iteration müssen demnach auch neue Straßenkarten erstellt werden.

Um nicht immer den gleichen Pfad zu berechnen werden Start und Ziel in jedem Durchgang neu definiert. Für Variante zwei werden so lange zufällige Start und Zielkoordinaten definiert, bis ihre Konfigurationen kollisionsfrei sind. Bei der Durchführung von Variante eins werden die Konfigurationen von Start und Ziel nur

einmalig bestimmt, da bei der anschließenden Erstellung der Karte mit $mapCreator()$ der Bereich um den Start und das Ziel freigehalten wird. In diesem Experiment werden Start und Ziel ohne Einschränkung zufällig auf der Karte platziert.

Anschließend wird in jeder Iteration mit allen zuvor gennannten Algorithmen nach einem Pfad gesucht und die Rechenzeit gemessen. Am Ende der Iteration werden die anderen Vergleichsparameter für alle geplanten Pfade berechnet.

Zum Schluss werden die Vergleichsparameter ausgewertet. Die Auswertung ist für alle Experimente gleich und wird später näher erläutert.

Um Auswirkungen der Zeitkomplexität identifizieren zu können soll das Experiment in der ersten Variante mit zwei verschiedenen Kartengrößen durchgeführt werden. Einmal mit einer Karte, die $25{,}6\ m$ hoch und breit ist und einmal mit einer Karte, die $51{,}2\ m$ hoch und breit ist. Beide Karten sollen eine Auflösung von $10\ \frac{Pixel}{m}$ besitzen.

Zusätzlich soll das Experiment noch in der zweiten Variante auf der Lagerhauskarte mit der Standardhöhe und Standardbreite von $25.6\ m$ und der Standardauflösung von $10\ \frac{Pixel}{m}$ durchgeführt werden.

## 5.3 Dynamisches Simulationsexperiment

Mit diesem Experiment soll das Verhalten der Algorithmen bei der initialen Pfadplanung und der Neuplanung untersucht werden. Es werden Karten der Standardgröße $25{,}6\ m$ und Standardauflösung $10\ \frac{Pixel}{m}$ verwendet. Das dynamische Simulationsexperiment kann mit dem Skript $DynamicComaparison.m$ durchgeführt werden.

Der Ablauf des Experiments unterscheidet sich nur geringfügig von dem des statischen Simulationsexperiment. Es gibt wieder zwei Varianten, die ausgeführt werden können. Da in jedem Durchgang von einer komplett neuen dynamischen Pfadplanung mit vollständiger initialen Pfadplanung ausgegangen werden soll werden die Straßenkarten in jedem Durchgang neu erstellt.

In diesem Experiment wird der Start nahe am linken Rand und das Ziel nahe am rechten Rand platziert. Für den Start gilt $x \in [0{,}8; 6{,}8]$. Für das Ziel gilt $x \in [18{,}8; 24{,}8]$. Für Start und Ziel gilt außerdem $y \in [0{,}8; 24{,}8]$. Durch diese Auswahl wird sichergestellt, dass das später hinzugefügte Hindernis circa in der Mitte der Karte platziert wird und dadurch nicht den Start oder das Ziel blockieren kann.

War die initiale Pfadplanung mit allen Algorithmen erfolgreich wird dem Konfigurationsraum ein Hindernis hinzugefügt. Das Hindernis ist vier Meter lang, $0{,}6$ Meter breit und wird vertikal auf der Karte platziert. Die Blockade wird auf der Hälfte des initialen Pfades eingefügt. Da sich die initialen Pfade unterscheiden können wird das Hindernis für jeden Algorithmus einzeln bestimmt.

Danach wird die Neuplanung durchgeführt. Alle nicht dynamischen Algorithmen führen in diesem Schritt eine komplett neue Pfadplanung aus.

Nachdem alle Iterationen durchgeführt wurden, geschieht die Auswertung für die initiale Pfadplanung und die Neuplanung. Die initiale Pfadplanung ist nahezu identisch mit einer statischen Pfadplanung

## 5.4 Dynamisches Simulationsexperiment mit geplantem Hindernis

In diesem Experiment soll die Zeitkomplexität der Algorithmen bei der Neuplanung untersucht werden. Es werden die gleichen Algorithmen wie beim dynamischen Simulationsexperiment betrachtet.

Die Zeitkomplexität der Neuplanung wird untersucht, indem verschieden große Umwege bei der Neuplanung erzwungen werden. Abhängig davon ändert sich die Anzahl der Knoten mit veränderten Werten und damit die Zahl der Knoten, die von den Algorithmen erneut betrachtet werden müssen. In diesem Experiment wird nur die Lagerhauskarte in der Standardgröße verwendet. Der Start befindet sich im Bereich $x \in [0{,}8; 6{,}8], y \in [19; 20{,}5]$ und das Ziel im Bereich $x \in [18{,}8; 24{,}8], y \in [19; 20{,}5]$. Der initiale Pfad führt wegen dieser Wahl immer durch den obersten horizontalen Gang. Beim Start des Experiments wird abgefragt, ob ein kleines oder ein großes Hindernis eingefügt werden soll. Abhängig davon wird vor der Neuplanung ein Hindernis eingefügt, das den oberen Gang entweder nur teilweise oder komplett blockiert.

Der Rest des Simulationsexperimentes ist identisch zum Dynamischen Simulationsexperiment.

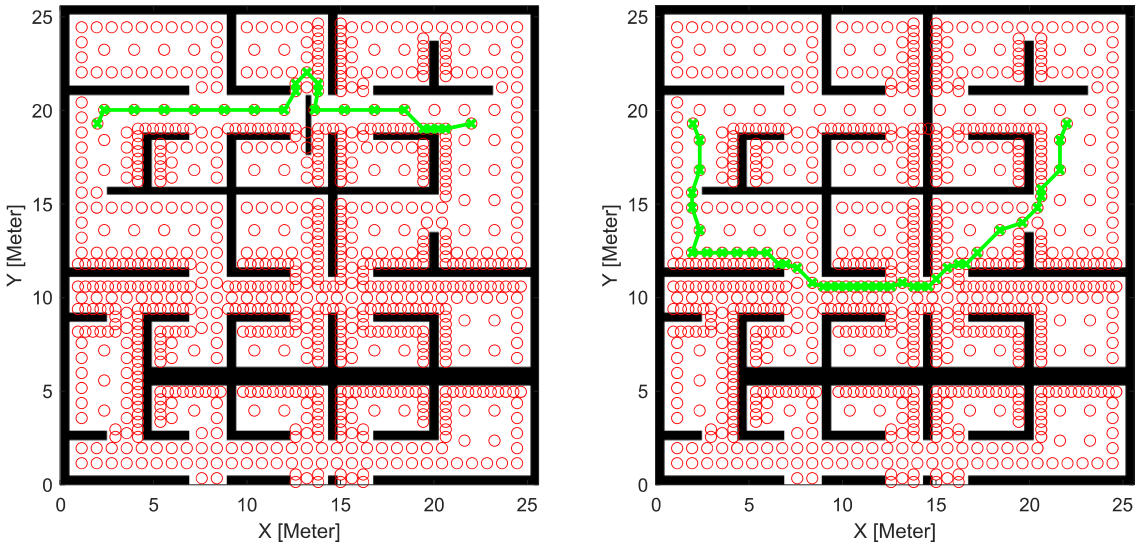


*Abbildung 5.1: Links ist die Neuplanung zu sehen, bei welcher der Gang nur teilweise blockiert wird. Rechts ist der Gang durch ein großes Objekt komplett blockiert.*

## 5.5 Auswertung der Simulationsexperimente

Wie zuvor erwähnt werden die Ergebnisse der Simulationsexperimente ausgewertet, nachdem alle Iterationen durchgeführt wurden. Dieser Schritt ist notwendig, um aussagekräftige und vergleichbare Daten zu erhalten. Nachdem alle Durchgänge eines Simulationsexperimentes durchgeführt wurden, liegen die Vergleichsparameter für jeden Durchgang einzeln vor, was einen Vergleich schwer macht.

Zuerst müssen diejenigen Durchgänge bestimmt werden, in denen alle Pfadplanungsalgorithmen erfolgreich einen Pfad finden konnten, da nur in diesen Durchgängen alle Algorithmen verglichen werden können. Anschließend werden für jeden Durchgang alle Vergleichsparameter aller Algorithmen miteinander verglichen. Die Vergleichsparameter der Planung mit Quadtrees und A* stellen die Basis für den Vergleich dar. Für alle anderen Algorithmen wird der relative Wert dieses Basiswertes bestimmt. Zum Schluss wird für alle relativen Werte der einzelnen Vergleichsparameter für jeden Algorithmus der Mittelwert berechnet. Zusätzlich wird noch der Mittelwert über alle Absolutwerte der Vergleichsparameter berechnet. Damit ist die Auswertung beendet und es können aussagen wie „Algorithmus A berechnet Pfade, die im Durchschnitt 10% kürzer sind als die von Algorithmus B berechneten Pfade.“ getätigt werden.

# 6. Ergebnisse der Simulationen

In diesem Kapitel sollen die Ergebnisse der verschiedenen Simulationsexperimente vorgestellt werden. Die ausführlichen Ergebnisse der Simulationen sind im Anhang zu finden und enthalten beispielsweise die Vergleichsparameter jeder einzelnen Iteration.

## 6.1 Statisches Simulationsexperiment

| Experiment 1 | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (21,73 m) | 95,91% | 95,14% | 139,13% | 100% |
| **Rechenzeit** | 100% (0.0091 s) | 11431,05% | 22759,04% | 4201,87% | 752,95% |
| **Durchschnittlicher Winkel** | 100% (33,73°) | 73,71% | 67,63% | 138,39% | 100,13% |

*Tabelle 6.1: Ergebnisse des statischen Simulationsexperimentes. Die Höhe und Breite der Karte betrugen 25,6 m. Es wurden 1000 Iterationen ausgeführt. Für jede Iteration wurde dieselbe Karte verwendet.*

Der RRT$^X$ Algorithmus liefert in diesem Experiment die kürzesten und glattesten Pfade. Dafür besitzt er mit großem Abstand die längste Ausführdauer. Die suchbasierten Algorithmen haben in diesem Experiment einen deutlichen Zeitvorteil, da die Straßenkarte in jeder Iteration wiederverwendet werden kann.

Auch zwischen den suchbasierten Algorithmen gibt es bei der Rechenzeit große Unterschiede.

| Experiment 2 | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (14,53 m) | 95,18% | 91,41% | 153,84% | 100% |
| **Rechenzeit** | 100% (0,60 s) | 447,64% | 323,66% | 11,91% | 125,42% |
| **Durchschnittlicher Winkel** | 100% (30,45°) | 76,89% | 45,41% | 167,89% | 100,16% |

*Tabelle 6.2: Ergebnisse des statischen Simulationsexperimentes. Die Höhe und Breite der Karte betrugen 25,6 m. Es wurden 1000 Iterationen ausgeführt. Für jede Iteration wurde eine neue Karte erstellt.*

Auch in diesem Experiment liefert der RRT$^X$ Algorithmus die Pfade mit der geringsten Länge und dem niedrigsten durchschnittlichen Winkel. Die Rechenzeit der stichprobenbasierten und suchbasierten Algorithmen unterscheidet sich weniger stark als beim vorherigen Experiment, da nun in jeder Iteration neue Straßenkarten erstellt werden müssen.

Der RRT* Algorithmus berechnet einen Pfad in der geringsten Zeit. Dafür ist der Pfad lang und besitzt einen großen durchschnittlichen Winkel.

Eine weitere Auffälligkeit ist die deutlich höhere Rechenzeit bei Verwendung von Framed Quadtrees anstatt normalen Quadtrees. Wird D* Lite anstatt A* verwendet fällt dies bei der Berechnungsdauer weniger ins Gewicht.

| **Experiment 3** | **QT + A*** | **FQT + D* Lite** | **RRT$^X$** | **RRT*** | **QT + D* Lite** |
|---|---|---|---|---|---|
| **Länge** | 100% (28,78 m) | 95,84% | 95,56% | 168,80% | 100% |
| **Rechenzeit** | 100% (4,84 s) | 296,79% | 33,04% | 3,71% | 106,76% |
| **Durchschnittlicher Winkel** | 100% (29,22°) | 70,35% | 66,58% | 182,88% | 100,02% |

*Tabelle 6.3: Ergebnisse des statischen Simulationsexperimentes. Die Höhe und Breite der Karte betrugen 51,2 m. Es wurden 1000 Iterationen ausgeführt. Für jede Iteration wurde eine neue Karte erstellt.*

In dem Simulationsexperiment mit doppelt so breiter und hoher Karte liefern die Framed Quadtrees in Kombination mit D* Lite zusammen mit RRT$^X$ die kürzesten Pfade. Der relative Wert der Framed Quadtrees mit D* Lite hat sich im Vergleich zu den vorherigen Experimenten nicht nennenswert verändert. Die relative Länge der Pfade, die mit RRT$^X$ berechnet wurden, ist im Vergleich zu Experiment 2 leicht gestiegen.

Der relative Wert des durchschnittlichen Winkels bei Verwendung von RRT$^X$ ist im Vergleich zu Experiment 2 zwar deutlich gestiegen, bleibt aber weiterhin der geringste Wert.

Die relative Rechenzeit der suchbasierten Algorithmen bewegt sich in ähnlichen Dimensionen wie bei Experiment 2. Die relative Rechenzeit der stichprobenbasierten Algorithmen ist, verglichen mit dem vorherigen Experiment, jedoch deutlich geringer. RRT$^X$ ist unter diesen Bedingungen durchschnittlich ungefähr dreimal schneller als die suchbasierten Algorithmen.

## 6.2 Dynamisches Simulationsexperiment

| Experiment 4a | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (23,11 m) | 95,96% | 92,81% | 143,15% | 100% |
| **Rechenzeit** | 100% (0,56 s) | 562,69% | 347,21% | 16,62% | 133,63% |
| **Durchschnittlicher Winkel** | 100% (29,09°) | 69,57% | 41,02% | 180,94% | 100,01% |

*Tabelle 6.4: Ergebnisse der initialen Pfadplanung des dynamischen Simulationsexperimentes. In jedem Durchgang wurde eine neue Karte mit mapCrator() erstellt. Insgesamt wurden 1000 Iterationen simuliert.*

Der einzige Unterschied von diesem Experiment zum statischen Simulationsexperiment ist die erhöhte Planungsdauer der Framed Quadtrees in Kombination mit D* Lite. Alle anderen Ergebnisse sind ähnlich zu den Ergebnissen aus Experiment zwei.

| Experiment 4b | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (24,45 m) | 96,05% | 91,87% | 135,60% | 107,53% |
| **Rechenzeit** | 100% (0,57 s) | 171,96% | 102,65% | 18,25% | 14,52% |
| **Durchschnittlicher Winkel** | 100% (30,48°) | 70,22% | 51,68% | 174,34% | 106,88% |

*Tabelle 6.5: Ergebnisse der Neuplanung des dynamischen Simulationsexperimentes. Es wurden 1000 Iterationen durchgeführt, wobei in jeder Iteration eine neue, mit mapCrator() erstellte Karte verwendet wurde.*

Bei der Neuplanung liefert der RRT$^X$ Algorithmus erneut den kürzesten und glattesten Weg. Im Vergleich zu Experiment 4a hat sich die relative Rechenzeit bei Nutzung des RRT$^X$ Algorithmus ebenfalls gedrittelt und ist damit nur knapp langsamer als eine komplette Neuplanung mit Quadtrees und A*.

Besonders auffällig ist auch die deutlich verkürzte Planungsdauer der Quadtrees und D* Lite. Die relative Länge und der durchschnittliche Winkel haben sich hier jedoch etwas verschlechtert.

Auch die relative Rechenzeit der Framed Quadtrees in Kombination mit D* Lite hat sich gegenüber der initialen Pfadplanung deutlich verringert. Die anderen Vergleichsparameter sind nahezu unverändert.

| Experiment 5a | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (26,97 m) | 96,75% | 95,79% | 130,88% | 100% |
| **Rechenzeit** | 100% (0,52 s) | 587,35% | 328,49% | 75,86% | 136,02% |
| **Durchschnittlicher Winkel** | 100% (33,41°) | 66,25% | 51,40% | 144,63% | 100,10% |

*Tabelle 6.6: Ergebnisse der initialen Pfadplanung nach 1000 Iterationen mit Verwendung der Lagerhauskarte.*

| Experiment 5b | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (35,26 m) | 101,09% | 95,36% | 99,52% | 104,78% |
| **Rechenzeit** | 100% (0,54 s) | 199,02% | 95,02% | 58,24% | 13,47% |
| **Durchschnittlicher Winkel** | 100% (33,77°) | 62,22% | 76,82% | 142,34% | 105,53% |

*Tabelle 6.7: Ergebnisse der Neuplanung des dynamischen Simulationsexperimentes. Es wurden 1000 Iterationen durchgeführt, wobei in jeder Iteration die Lagerhauskarte verwendet wurde.*

Erneut erhält man bei Nutzung des RRT$^X$ Algorithmus die durchschnittlich kürzesten Pfade. RRT* erreicht erstmals eine relative Länge kleiner als $100\%$.

Eine auffällige Änderung zwischen initialer Planung und Neuplanung ist außerdem, dass sich die relativen Längen von FQT mit D* Lite und QT mit D* Lite um etwa $5\%$ erhöht haben. Beide Kombinationen von Algorithmen konnten jedoch ihre relative Rechenzeit verringern.

## 6.3 Dynamisches Simulationsexperiment mit geplantem Hindernis

| Experiment 6 | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (19,36 m) | 104,14% | 95,30% | 180,97% | 111,41% |
| **Rechenzeit** | 100% (0,58 s) | 193,30% | 77,55% | 34,83% | 13,28% |
| **Durchschnittlicher Winkel** | 100% (36,80°) | 70,23% | 44,32% | 144,13% | 124,03% |

*Tabelle 6.8: Ergebnisse der Neuplanung bei der nur ein kleines Hindernis den Pfad blockiert. Es wurden 1000 Iterationen auf der Lagerhauskarte ausgeführt.*

| Experiment 7 | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|
| **Länge** | 100% (32,90 m) | 97,27% | 94,86% | 127,77% | 101,34% |
| **Rechenzeit** | 100% (0,63 s) | 195,64% | 70,69% | 51,71% | 12,81% |
| **Durchschnittlicher Winkel** | 100% (28,04°) | 63,77% | 64,69% | 186,65% | 106,63% |

*Tabelle 6.9: Ergebnisse der Neuplanung, wenn der gesamte obere Korridor auf der Lagerhauskarte blockiert wird. Es wurden 1000 Iterationen ausgeführt.*

Im Vergleich zur Neuplanung mit einem kleinen Hindernis weisen QT und FQT in Kombination mit D*Lite eine leicht verbesserte relative Pfadlänge und einen geringeren durchschnittlichen Winkel auf, wenn der gesamte obere Korridor blockiert wird. Die relative Berechnungsdauer steigt bei der Nutzung von Framed Quadtrees etwas an.

Der RRT$^X$ Algorithmus hat eine leicht verringerte relative Planungsdauer, wenn ein großes Hindernis eingefügt wird. Die relative Länge bleibt unverändert und der durchschnittliche Winkel erhöht sich eindeutig.

# 7. Diskussion

In diesem Kapitel sollen aus den in Kapitel 6 vorgestellten Ergebnisse Rückschlüsse auf die Eignung der Algorithmen in verschiedenen Szenarien gezogen werden.

In Tabelle 7.1 ist für alle relevanten Experimente und für jeden Vergleichsparameter eine Rangliste der Algorithmen zu sehen. Eine geringere Zahl steht hier für eine bessere Platzierung.

Das statische Experiment zeigt, dass der größte Anteil der Planungsdauer, bei Verwendung von suchbasierten Algorithmen, während der Erstellung der Straßenkarte entsteht. Wird diese mehrfach verwendet, können Pfade sehr schnell gefunden werden. Liegt als Objekt einer Pfadplanungsaufgabe beispielsweise ein Roboterarm vor, der in einer gesicherten Roboterzelle bewegt wird, können einmalig Straßenkarten mit hoher Auflösung erstellt werden. Diese können dann für jede Planung verwendet werden und kurze Pfad in sehr wenig Zeit berechnen. Da dies aber nur für statische Umgebungen relevant ist, kann dieser Vorteil für Pfadplanungen in dynamischen Umgebungen nicht genutzt werden und ist somit für die in dieser Arbeit angestrebte Anwendung nicht relevant. Aus diesem Grund wird das erste Experiment in der Rangliste in Tabelle 7.1 vernachlässigt.

Außerdem ist in diesem Experiment erkennbar, dass Quadtrees und besonders Framed Quadtrees bei größeren Karten deutlich langsamer sind. Dies liegt daran, dass bei größeren Karten mehr Knoten erstellt werden und somit bei der Nachbarsuche in längeren Listen nach potenziellen Nachbarn gesucht werden muss. Die Nachbarsuche muss zudem für deutlich mehr Nachbarn ausgeführt werden. Dafür ist die Knotendichte bei Straßenkarten, die mit Quadtrees oder Framed Quadtrees erstellt werden, konstant, wodurch auch die Optimalität des Pfades unabhängig von der Kartengröße ist.

Wird $RRT^X$ hingegen mit gleich vielen Iterationen auf einer größeren Karte verwendet verschlechtert sich die Pfadqualität im Vergleich zur kleineren Karte. Der Grund hierfür ist, dass mehr Knoten benötigt werden, um das Ziel zu erreichen und deshalb weniger Knoten genutzt werden können, um den Pfad zu optimieren. Die Dichte an Knoten sinkt, was die Anzahl an Verbesserungen für den Pfad reduziert.

Die Ergebnisse des dynamischen Simulationsexperimentes zeigen, dass die Pfadgüte von suchbasierten Algorithmen bei der Neuplanung schlechter ist als bei der initialen Pfadplanung. Der Grund hierfür ist, dass durch neu hinzugefügte Objekte der Konfigurationsraum nichtmehr optimal mit Knoten ausgefüllt ist. Dies fällt besonders dann ins Gewicht, wenn die Neuplanung einen ähnlichen Pfad bestimmt wie die initiale Pfadplanung, da dann der beschädigte Teil der Straßenkarte erneut durchquert wird. Dieser Effekt ist auch im dynamischen Experiment mit geplantem Hindernis zu erkennen.

Beim dynamischen Simulationsexperiment ist außerdem zu sehen, dass bei der initialen Pfadplanung der Zeitvorteil von Framed Quadtrees gegenüber $RRT^X$ auf das statische Experiment beschränkt bleibt. Dies liegt an der großen Menge an Daten, die für einen

Framed Quadtree benötigt werden und für die Neuplanung zurückgegeben und gespeichert werden müssen.

Eine Erkenntnis des dynamischen Simulationsexperimentes mit geplantem Hindernis ist, dass es nur bei Verwendung von Framed Quadtrees eine erkennbare Abhängigkeit der Planungsdauer von der Größe der Neuplanung gibt. Wegen der großen Anzahl der Knoten in einem Framed Quadtree muss D* Lite veränderte Kosten an sehr viele Knoten weitergeben. Bei allen anderen Algorithmen ist dieser Unterschied zu gering, um ihn zu messen.

Aus den zuvor genannten Erkenntnissen und der Rangfolge in Tabelle 7.1 soll nun ein Endresultat gezogen werden.

Wegen der schlechten Skalierbarkeit und der hohen Planungsdauer, wobei nicht die geringste Pfadlänge erreicht werden konnte, sind Framed Quadtrees nicht für die dynamische Pfadplanung geeignet. Hierbei spielt es keine Rolle, ob D* Lite oder A* verwendet wird, da die negativen Effekte von der Erstellung und der Beschaffenheit der Framed Quadtrees stammen. Auch in der Rangliste erreichen die Framed Quadtrees mit D* Lite nicht das beste Ergebnis.

Quadtrees und D* Lite liefern kurze Rechenzeiten, besonders bei der Neuplanung. Jedoch sind die erreichten Pfadlängen deutlich schlechter im Gegensatz zur Nutzung von RRT$^X$ oder Framed Quadtrees. Geht man von einer Ausführgeschwindigkeit von $7\ \frac{km}{h}$ aus und legt die Werte von Experiment 5a zugrunde wäre die Ausführung des Pfades, der mit RRT$^X$ erstellt wurde, $0{,}58\ s$ schneller als die Ausführung des mit Quadtrees und D* Lite bestimmten Pfades. Der Vorteil der geringeren Planungsdauer wird also durch den längeren Pfad verringert. Außerdem beschränkt sich der Zeitvorteil bei der initialen Pfadplanung auf kleine Karten, da die Erstellung von Quadtrees bei Skalierung auf große Karten eine deutlich höhere Rechenzeit benötigt. Dies zeigt sich auch in der Rangliste. Quadtrees und D* konnten hier nicht das beste Ergebnis erzielen.

Die Verwendung von RRT$^X$ erscheint auf Basis der Experimente damit am sinnvollsten, weshalb RRT$^X$ in Tabelle 7.1 das beste Ergebnis erreicht. RRT$^X$ liefert die kürzesten Pfade, die in den meisten Experimenten auch den geringsten durchschnittlichen Winkel aufweisen. Die hohe Planungsdauer kann verringert werden, indem der RRT$^X$ Algorithmus inkrementell verwendet wird. Dies bedeutet, dass die Ausführung beginnt, sobald ein erster Pfad verfügbar ist. Die durchgehend geringe Planungsdauer des RRT* Algorithmus, der terminiert, sobald ein erster Pfad gefunden wurde, zeigt wie sehr sich die Planungsdauer des initialen Pfades hierdurch reduzieren lässt. Während der Ausführung wird der Pfad dann weiter optimiert. Da die Ausführdauer weit größer ist als die Planungsdauer steht ein nahezu optimaler Pfad zur Verfügung, wenn erst ein kleiner Teil des Pfades zurückgelegt wurde. Die Verbesserung des Pfades während der Ausführung ist mit suchbasierten Algorithmen nur mit größerem Aufwand möglich [Ott2016, 802].

Die Neuplanung mit Quadtrees und D* Lite ist zwar schneller, jedoch überwiegen die Vorteile von RRT$^X$. Außerdem kann dieser Vorteil durch die verringerte initiale

Planungsdauer, die mit Hilfe inkrementeller Planung erreicht wird, und durch die geringere Ausführzeit etwas kompensiert werden.

Pfadplanung mit RRT ähnlichen Strukturen hat einen weiteren Vorteil, der ins Gewicht fällt, sobald eine Annahme, die im Rahmen dieser Arbeit gilt, verworfen wird. Wird nicht mehr von einem omnidirektionalen Roboter ausgegangen kann nicht jeder Pfad ausgeführt werden. Bei Verwendung einer Art von RRT ist die Erweiterung der Pfadplanung, sodass Roboter mit nichtholonomen Bindungen die Pfade ausführen können mit wenig Aufwand möglich [Kar2011b].

Tabelle 7.1 befindet sich auf der nächsten Seite.

| Experiment | Vergleichs-parameter | QT + A* | FQT + D* Lite | RRT$^X$ | RRT* | QT + D* Lite |
|---|---|---|---|---|---|---|
| **Experiment 2** | Länge | 3 | 2 | 1 | 4 | 3 |
| | Rechenzeit | 2 | 5 | 4 | 1 | 3 |
| | Winkel | 3 | 2 | 1 | 4 | 3 |
| **Experiment 3** | Länge | 2 | 1 | 1 | 3 | 2 |
| | Rechenzeit | 3 | 5 | 2 | 1 | 4 |
| | Winkel | 3 | 1 | 2 | 4 | 3 |
| **Experiment 4a** | Länge | 3 | 2 | 1 | 4 | 3 |
| | Rechenzeit | 2 | 5 | 4 | 1 | 3 |
| | Winkel | 3 | 2 | 1 | 4 | 3 |
| **Experiment 4b** | Länge | 3 | 2 | 1 | 5 | 4 |
| | Rechenzeit | 3 | 5 | 4 | 2 | 1 |
| | Winkel | 3 | 2 | 1 | 5 | 4 |
| **Experiment 5a** | Länge | 3 | 2 | 1 | 4 | 3 |
| | Rechenzeit | 2 | 5 | 4 | 1 | 3 |
| | Winkel | 3 | 2 | 1 | 4 | 3 |
| **Experiment 5b** | Länge | 3 | 4 | 1 | 2 | 5 |
| | Rechenzeit | 4 | 5 | 3 | 2 | 1 |
| | Winkel | 3 | 1 | 2 | 5 | 4 |
| **Experiment 6** | Länge | 2 | 3 | 1 | 5 | 4 |
| | Rechenzeit | 4 | 5 | 3 | 2 | 1 |
| | Winkel | 3 | 2 | 1 | 5 | 4 |
| **Experiment 7** | Länge | 3 | 2 | 1 | 5 | 4 |
| | Rechenzeit | 4 | 5 | 3 | 2 | 1 |
| | Winkel | 3 | 1 | 2 | 5 | 4 |
| **Summe** | - | 70 | 71 | 46 | 80 | 73 |

*Tabelle 7.1: Aufstellen einer Rangliste der verschiedenen Algorithmen. RRT$^X$ liefert das geringste und damit beste Ergebnis.*

# 8. Ausblick

Die Ergebnisse dieser Arbeit haben gezeigt, dass die Verwendung von $RRT^X$ für die dynamische Pfadplanung in Lagerhäusern, Containerterminals und zufälligen Umgebungen erfolgreich ist. Als nächstes ist die Betrachtung nichtholonomer Bindungen des Transportfahrzeuges von Interesse. Zusätzlich ist eine Modifikation der Implementierung sinnvoll, sodass der Algorithmus inkrementell verwendet werden kann. Eine Implementierung des Algorithmus in einer Compilersprache ist eine Maßnahme, um die Rechenzeit zu verringern und besonders vor dem letztendlichen Einsatz interessant.

Außerdem könnten in Zukunft die Vorteile der verschiedenen Algorithmen kombiniert werden. Die Kombination einer Straßenkarte, die für die Grobplanung die wichtigsten Punkte einer realen Umgebung verbindet und dem $RRT^X$ Algorithmus für die Feinplanung wäre denkbar.

Die Bestimmung des Konfigurationsraumes mit Hilfe von Sensordaten ist ein weiterer Schritt, der nötig ist bevor der Algorithmus in einer echten Umgebung eingesetzt werden kann.

Sind alle diese Schritte erfüllt, kann ein erster Prototyp in einer realen Umgebung getestet werden.

Auch wenn nach aktuellem Stand der Forschung eine Anwendung in Fabriken bereits machbar ist, können die bestehenden Algorithmen durch weitere Untersuchungen optimiert werden und die automatisierte Logistik in der Fabrik der Zukunft noch flüssiger gestaltet werden. Durch Anwendung in anderen Bereichen, wie der Servicerobotik oder dem autonomen Fahren werden uns Pfadplanungsalgorithmen in Zukunft auch immer mehr im Alltag begegnen.

# 9. Literaturverzeichnis

[And2017] Andelfinger, V. P.; Hänisch, T.: Industrie 4.0. Wie cyber-physische Systeme die Arbeitswelt verändern. Springer Gabler, Wiesbaden, 2017.

[Cho2005] Choset, H.M. Hrsg.: Principles of robot motion: theory, algorithms, and implementation. MIT Press, Cambridge, Massachusetts, 2005.

[For1922] Ford, H.: My life and work. Garden City, N.Y., Doubleday, Page & company, United States, 1922.

[Har1968] Hart, P.; Nilsson, N.; Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In IEEE Transactions on Systems Science and Cybernetics, 1968, 4; S. 100–107.

[Her2012] Hertzberg, J.; Lingemann, K.; Nüchter, A.: Mobile Roboter. Eine Einführung aus Sicht der Informatik. Springer Vieweg, Berlin, 2012.

[Hof2018] Hofmann, M.: Intralogistikkomponenten für die Automobilproduktion ohne Band und Takt – erste Prototypen. Wissenschaftliche Gesellschaft für Technische Logistik, 2018.

[Kar2011a] Karaman, S.; Frazzoli, E.: Sampling-based algorithms for optimal motion planning. In undefined, 2011.

[Kar2011b] Karaman, S.; Frazzoli, E.: Optimal Kinodynamic Motion Planning using Incremental Sampling-based Methods. In 49th IEEE Conference on Decision and Control (CDC), 2011.

[Kav2016] Kavraki, L. E.; LaValle, S. M.: Motion Planning: Springer Handbook of Robotics. Springer International Publishing, Cham, 2016; S. 139–162.

[Koe2002] Koenig, S.; Likhachev, M.: D* Lite: Eighteenth National Conference on Artificial Intelligence. American Association for Artificial Intelligence, USA, 2002; S. 476–483.

[Lat1991] Latombe, J.-C.: Robot Motion Planning. Springer US, Boston, MA, 1991.

[LaV1998] LaValle, S. M.: Rapidly-Exploring Random Trees: A New Tool for Path Planning, Ames, 1998.

[Ott2015] Otte, M.; Frazzoli, E.: RRTX  Real-Time Motion Planning/Replanning for Environments with Unpredictable Obstacles: Algorithmic Foundations of Robotics XI; S. 461–478.

[Ott2016] Otte, M.; Frazzoli, E.: RRTX  Asymptotically optimal single-query sampling-based motion planning with quick replanning. In The International Journal of Robotics Research, 2016, 35; S. 797–833.

[Pil2006] Piller, F. T.: Mass Customization. Ein wettbewerbsstrategisches Konzept im Informationszeitalter. Dt. Univ.-Verl., Wiesbaden, 2006.

[Rob2006] Robert Yoder; Peter A. Bloniarz: A Practical Algorithm for Computing Neighbors in Quadtrees, Octrees, and Hyperoctrees, 2006; S. 249–255.

[Ste1995] Stentz, A.: The Focussed D* Algorithm for Real-Time Replanning, 1995.

[Yah1998] Yahja, A. et al.: Framed-quadtree path planning for mobile robots operating in sparse environments: International Conference on Robotics and Automation,1998; S. 650–655.