



UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik



Entwicklung einer modularen Pfadplanungsbibliothek

Evaluation unterschiedlicher Pfadplanungsalgorithmen

Masterarbeit
zur Erlangung des Grades
MASTER OF SCIENCE
im Studiengang Informatik

vorgelegt von

Benedikt Jöbgen

Betreuer: Dipl.-Inform. Christian Winkens, Institut für
Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Erstgutachter: Dipl.-Inform. Christian Winkens, Institut für
Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Zweitgutachter: Prof. Dr.-Ing. Dietrich Paulus, Institut für
Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Koblenz, im Januar 2015

Kurzfassung

In der Forschung der autonomen mobilen Roboter, ist besonders die Pfadplanung immer noch ein sehr aktuelles Thema. Diese Masterarbeit befasst sich mit verschiedenen Pfadplanungsalgorithmen zur Navigation solcher mobilen Systeme. Hierbei ist nicht nur eine kollisionsfreie Trajektorie von einem Punkt zu einem anderen zu ermitteln, sondern sollte diese auch noch möglichst optimal sein und alle Fahrzeug-gegebenen Einschränkungen einhalten. Besonders die autonome Fahrt in unbekannter dynamischer Umgebung stellt eine große Herausforderung dar, da hier eine geschlossene Regelung notwendig ist und dem Planer somit eine gewisse Dynamik abverlangt wird. In dieser Arbeit werden zwei Arten von Algorithmen vorgestellt. Zum einen die Pfadplaner, welche auf dem A^* aufbauen, der im eigentlichen Sinne ein Graphsuchalgorithmus ist: A^* , *Anytime Repairing A^** , *Lifelong Planning A^** , *D^* Lite*, *Field D^** , *hybrid A^** . Zum anderen die Algorithmen, welche auf dem probabilistischen Planungsalgorithmus *Rapidly-exploring Random Tree* basieren (*RRT*, *RRT**, *Lifelong Planning RRT**), sowie einige Erweiterungen und Heuristiken. Außerdem werden Methoden zur Kollisionsvermeidung und Pfadglättung vorgestellt. Abschließend findet eine Evaluation der verschiedenen Algorithmen statt.

Abstract

In current research of the autonomous mobile robots, path planning is still a very important issue. This master's thesis deals with various path planning algorithms for the navigation of such mobile systems. This is not only to determine a collision-free trajectory from one point to another. The path should still be optimal and comply with all vehicle-given constraints. Especially the autonomous driving in an unknown and dynamic environment poses a major challenge, because a closed-loop control is necessary and thus a certain dynamic of the planner is demanded. In this paper, two types of algorithms are presented. First, the path planner, based on A^* , which is a common graph search algorithm: A^* , *Anytime Repairing A^** , *Lifelong Planning A^** , *D^* Lite*, *Field D^** , *hybrid A^** . Second, the algorithms which are based on the probabilistic planning algorithm *Rapidly-exploring Random Tree* (*Rapidly-exploring Random Tree*, *RRT**, *Lifelong Planning RRT**), as well as some extensions and heuristics. In addition, methods for collision avoidance and path smoothing are presented. Finally, these different algorithms are evaluated and compared with each other.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich ja nein
einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. ja nein

Koblenz, den 21. Januar 2015

Inhaltsverzeichnis

1	Einleitung	15
1.1	Motivation	15
1.2	Problemdefinition	17
2	Pfadplanungsalgorithmen	21
2.1	A*	21
2.1.1	Heuristik	22
2.1.2	Der Algorithmus	22
2.1.3	Gewichteter A*	25
2.1.4	Einschränkungen für die Pfadplanung	25
2.2	Anytime Repairing A*	27
2.2.1	Der Algorithmus	27
2.2.2	Ergebnisse	29
2.3	Lifelong Planning A*	31
2.3.1	Vorgehensweise	31
2.3.2	Der Algorithmus	32
2.3.3	Ergebnisse	35
2.4	D*-Lite	37
2.4.1	Vorgehensweise	37
2.4.2	Der Algorithmus	37
2.4.3	Ergebnisse	39
2.5	Field D*	41
2.5.1	Vorgehensweise	41
2.5.2	Algorithmus	43
2.5.3	Field Backtracing	45
2.5.4	Das Zwei-Eltern Dilemma	47
2.6	Hybrider A*	49
2.6.1	Vorgehensweise	49
2.6.2	Nachteile des hybriden A*	50
2.6.3	hybrider D*	52
2.7	Rapidly-exploring Random Tree	54

2.7.1	Algorithmus	54
2.7.2	Rapidly Exploring	56
2.7.3	Ergebnisse	57
2.8	RRT*	59
2.8.1	Algorithmus	59
2.8.2	Ergebnisse	63
2.8.3	Erweiterungen	63
2.9	Lifelong Planning RRT*	74
2.9.1	Änderung des Fahrzeugzustandes	74
2.9.2	Änderung des Umgebungsmodells	75
2.9.3	Backtracing	77
3	Pfadplanungsbibliothek	81
3.1	Kollisionsvermeidung	81
3.1.1	Fahrzeugmodellierung und Ausschwenkmaße	82
3.1.2	Linien Rasterisierung	84
3.2	Glatte Pfadstücke	87
3.2.1	Einfache Kreisfahrt	87
3.2.2	Glatte Kreisübergänge zwischen gegebenen Zuständen	87
3.3	Pfadglättung	92
3.3.1	Kreis-Interpolation am Pfad	92
3.3.2	Vergleich mit Catmull-Rom Splines	96
3.4	Pfadplanungsbibliothek ppLib	99
4	Evaluation	103
4.1	Vergleich ARA* mit A*	103
4.2	A*, LPA* und D* Lite Laufzeitmessung	107
4.3	Graphgrößenvergleich zwischen repeated A* und D* Lite	108
4.4	Field D* Evaluation	110
4.4.1	Pfadkosteneinsparung des <i>Field D*</i>	110
4.4.2	Mehrfachoptimierung durch das Zwei-Eltern Dilemma	112
4.5	Vergleich hybrider A* mit A*	114
4.6	Vergleich RRT mit A*	115
4.7	RRT* Evaluation	117
4.7.1	FLANN im RRT*	118
4.7.2	Rekursives ReWire im RRT*	119
4.7.3	Dynamic Domain RRT Evaluation	120
4.8	Statische LP-RRT* Evaluation	122
4.9	Laufzeitmessung der Kreis Interpolation	124
5	Zusammenfassung	125

<i>INHALTSVERZEICHNIS</i>	9
A Field-Backtrace Algorithmus	131
B Linien-Rasterisierungs Algorithmus	137
C Evaluationsumgebungen	141
D ppLib - readme	145

Abbildungsverzeichnis

1.1	Umgebung und Interaktion autonomer mobiler Systeme	16
1.2	Das optimale Pfadplanungsproblem	18
2.1	A^* -Algorithmus	23
2.2	Vergleich verschiedener Gewichte im A^*	24
2.3	Beispiele für Optimalitätseinbußen der graphgebundenen Pfadsuche	26
2.4	ARA^* -Algorithmus pt. 1	28
2.5	ARA^* -Algorithmus pt. 2	29
2.6	Beispielverlauf des <i>Anytime Repairing A^*</i> mit $\epsilon_{max} = 2.5$ und $\Delta\epsilon = 0.2$ (hellblaue Felder sind die jeweils aktualisierte Knoten)	30
2.7	<i>Lifelong Planning A^*</i> - Algorithmus pt. 1	32
2.8	<i>Lifelong Planning A^*</i> - Algorithmus pt. 2	33
2.9	Beispielsituationen des <i>Lifelong Planning A^*</i> (blau: <i>OpenList</i> , braun:Hinderniss, hellgrün:Start, dunkelgrün:Ziel)	36
2.10	D^* <i>Lite</i> - Algorithmus pt. 1	38
2.11	D^* <i>Lite</i> Algorithmus pt. 2	39
2.12	D^* <i>Lite</i> Beispiel (blau: <i>OpenList</i> , braun:Hindernisse, hellblau:zurückgelegter Weg, rot:bearbeitete Knoten)	40
2.13	Beispiel zu Verdeutlichung der graphgebundenen Einschränkung in der Pfadplanung (blau: gefundener Pfad, braun: Hindernisse, grau: schwer befahrbares Gebiet, hellgrün: Startpunkt, dunkelgrün: Ziel) .	41
2.14	verschieden Graphtypen	42
2.15	verschiedene Kostenfunktionen	42
2.16	Interpolierende Kostenberechnung von v mit: v_1 (direkter Nachbar), v_2 (diagonaler Nachbar), c_a (Zellenkosten der Zelle zwischen v, v_1, v_2), c_b (Zellenkosten der Zelle zwischen v, v_1 aber nicht v_2)	44
2.17	Beispiele für Field D^* backtracing (blau:Zielpfad, braun:Hindernisse, hellgrau:schwer befahrbares Gebiet, dunkelgrau:Knoten im Graph mit zugehörigen Vorgängerrichtungen, lila:Knoten der <i>OpenList</i>) . .	45
2.18	Field D^* backtracing	46

2.19	Field- <i>LPA*</i> Beispiel (hellgrün:Startpose, dunkelgrün:Zielpose, blau:Zielpfad, grau:Knoten im Graph mit zugehörigen Vorgängerrichtungen, rot:Verdeutlichung des Zwei-Eltern Dilemmas)	48
2.20	Vom <i>A*</i> zum <i>hybrid A*</i>	50
2.21	Nachteile des hybriden <i>A*</i>	51
2.22	Gerüst des Rapidly-exploring Random Tree Algorithmus	54
2.23	Die extend-Funktion	55
2.24	Hinzufügen eines neuen Knotens in den Graphen	56
2.25	Voronoi-Gebiete eines rrt-Graphen (rot,schwarz: <i>RRT</i> -Graph, blau: Voronoi-Kanten	57
2.26	Der <i>RRT*</i> -Algorithmus	59
2.27	Hinzufügen eines neuen Knotens in den Graphen mittels chooseParent-Funktion	60
2.28	Die chooseParent - Funktion	61
2.29	Die reWire - Funktion	62
2.30	Optimierung des Baumes mittels reWire-Funktion (weiterführend von Abb. 2.27)	62
2.31	goal-bias sampling	64
2.32	Das <i>local-bias</i> Sampling	65
2.33	Ein <i>local-bias</i> Beispiel	66
2.34	Beweis zum <i>Sample-Region</i> -Radius	66
2.35	Das Node-Rejection sampling	67
2.36	Sampling Bereiche in der „bugtrap“ [YJSL05] (blau: <i>RRT</i> -Graph, schwarz: Hindernis, rot: Voronoi-Grenzen)	68
2.37	Der <i>Dynamic Domain RRT</i> - Algorithmus	69
2.38	Sampling Bereich von v [YJSL05]	70
2.39	Motivation(a&b) und Ergebnis(c) des rekursiven reWire (grau: Hindernis, schwarz: <i>RRT*</i> -Graph, blau: Zielregion und Pfad)	71
2.40	Einfache rekursive reWire-Funktion	71
2.41	Rekursive reWire-Funktion mittels Breitensuche	72
2.42	Die fahrzeugunabhängige Baumstruktur des <i>LP-RRT*</i> (rot-schwarz: Graph, hellgrün: Startpose, dunkelgrün: Zielpose, blau: Zielpfad)	75
2.43	Neue Hindernisse erscheinen (rot-schwarz: Graph, hellgrün: Startpose, dunkelgrün: Zielpose, blau: Zielpfad, braun: Hindernisse)	76
2.44	Hindernisse verschwinden	77
2.45	Algorithmus zu Verarbeitung von Umgebungsänderungen	78
2.46	Der findPath - Algorithmus: intelligentes Backtracing	79
3.1	Erweiterung der Hindernisse um e'	81
3.2	Fahrzeugmodell	82
3.3	Fahrzeug schwenkt aus	83

3.4	Linien Rasterisierung (dunkelgrau: Bresenham ähnlich, hellgrau: eigener Algorithmus)	84
3.5	Linien Rasterisierung, drei verschiedene Fälle	85
3.6	Einfache Kurvenfahrt	88
3.7	Circular-Linear Interpolation	89
3.8	Circular-Circular Interpolation	91
3.9	B-Splines(grün) vs. Catmull-Rom Splines (rot) (braun: Hindernisse, schwarz: ursprünglicher Pfad mit Kontrollpunkten)	92
3.10	Kreis-Konstruktion für die Pfad-Interpolation pt. 1	93
3.11	Kreis-Konstruktion für die Pfad-Interpolation pt. 2	95
3.12	Interpolierter Pfad mit Interpolationskreisen pro Kontrollpunkt	96
3.13	Catmull-Rom Spline mit Interpolationsvektoren pro Kontrollpunkt	97
3.14	Kreisinterpolation (blau) vs. Catmull-Rom Spline (rot)	97
3.15	Planungsbibliothek Grundgerüst	100
3.16	Knoten der Planungsalgorithmengraphen	100
3.17	Planungsalgorithmen	101
3.18	Graphische Benutzeroberfläche	102
4.1	Der direkte Vergleich der Graphgröße zwischen A^* und ARA^*	104
4.2	Der direkte Vergleich der Pfadlänge zwischen A^* und ARA^*	104
4.3	Der direkte Laufzeitvergleich zwischen A^* und ARA^*	105
4.4	Der vollständige Vergleich zwischen A^* und ARA^*	106
4.5	Laufzeitvergleich zwischen A^* , LPA^* und $D^* Lite$	107
4.6	Größenvergleich des Graphen des $D^* Lite$ mit dem A^* -Graphen	108
4.7	Graph-Reste(rot: Knoten im Graph; dunkelblau: <i>OpenList</i>) bei einer Fahrt mittels $D^* Lite$. Die dunkelblaue Strecke zeigt den geplanten, noch zu fahrenden Weg an, die hellblaue Strecke den bereits zurückgelegten Weg	109
4.8	Vergleich der Pfadlängen zwischen $D^* Lite$ und <i>Field</i> D^* in der Szene aus C.6	110
4.9	Maximale Pfadverkürzung durch die Field-Kostenrechnung	111
4.10	Mehrfachberechnung der Knoten bei der Field-Heuristik	113
4.11	Messergebnisse eines Vergleichs von A^* und <i>hybrid</i> A^*	114
4.12	Vergleich des RRT und A^* in der Umgebung C.1	115
4.13	Vergleich des RRT und A^* in der Umgebung C.5	116
4.14	Evaluationsergebnisse des RRT^*	117
4.15	Geschwindigkeitsmessung der Graphausbreitung beim RRT^* mit verschiedenen Erweiterungen	118
4.16	Vergleich der Expandierungsgeschwindigkeit des RRT^* mit und ohne rekursiven reWire	119
4.17	Vergleichsmessungen des RRT^* mit und ohne rekursiven reWire	120

4.18	Dynamic Domain <i>RRT</i> Zeitmessungen in der „bugtrap“	121
4.19	Dynamic Domain <i>RRT*</i> Zeitmessungen	121
4.20	Evaluationsergebnisse des <i>Lifelong Planning RRT*</i>	123
A.1	Field-Backtrace - pt.1	132
A.2	Field-Backtrace - pt.2	133
A.3	Field-Backtrace - pt.3	134
A.4	Skizze zur Verdeutlichung der Variablen im <i>Field D*</i> -Algorithmus	135
B.1	Linien-Rasterisierungs-Algorithmus	138
B.2	Skizze zur Verdeutlichung der Variablen im Linien-Rasterisierungs- Algorithmus	139
C.1	Karte 1: 60×40 Zellen	142
C.2	Karte 2: 60×40 Zellen	142
C.3	Karte 3: 60×50 Zellen	142
C.4	Karte 4: 100×100 Zellen	143
C.5	Karte 3, 200×200 Zellen	143
C.6	Die simulierte Fahrt mit begrenzter Sichtweite durch eine dynamische Umgebung zur Evaluation des <i>D* Lite</i> mit 200×200 Zellen. (hellgrüner Punkt: aktuelle Position; dunkelgrüner Punkt: Zielposition)	144

Kapitel 1

Einleitung

1.1 Motivation

Die Entwicklung mobiler Roboter und deren autonome Fortbewegung stellt ein aktuelles Forschungsgebiet in der Robotik dar. Autonomie bedeutet hierbei, dass ein System selbstständig agiert, ohne weiteres Eingreifen von Außen. Generell beinhaltet die Pfadplanung zum Beispiel auch die autonome Manipulation der Umgebung mit Armen und Greifern. In dieser Arbeit liegt der Fokus jedoch primär auf der autonomen Navigation mobiler Systeme. Hierbei stellt das autonome Fahren ein geschlossenes System aus der Umgebung, der Umgebungswahrnehmung, der Lokalisierung und Kartierung, der Pfadplanung bis hin zur Ausführung des Plans und der daraus resultierenden Bewegung dar, welches in Abbildung 1.1 zu erkennen ist. Diese Masterarbeit ist dabei im Bereich der Planung einzuordnen. In der Pfadplanung ist zu unterscheiden, ob diese in vollständig bekanntem Gebiet stattfindet, oder in teils oder vollständig unbekanntem Gebiet. Ist das Terrain nicht vollständig bekannt oder verändert es sich dynamisch, so besitzt das System, aufgrund der begrenzten sensorbasierten Erfassung der Umgebung, nur lokale Teilinformationen der Umgebung. Das sich hierdurch nur inkrementell aufbauende Umgebungsmodell bedingt eine inkrementelle Berechnung des Pfades. Gerade in diesem sehr komplexen Gebiet befindet sich die aktuelle Forschung noch in den Anfängen, wodurch diese Masterarbeit ihre Bedeutung erlangt.

Die Pfadplanung bedeutet nicht nur die Ermittlung einer kollisionsfreien Trajektorie von einem Punkt zum Anderen, sondern sollte diese auch möglichst optimal sein und alle nicht-holonomen Einschränkungen des Vehikels einhalten. Mit der genauen Beschreibung der Problemdefinition startet diese Arbeit in Abschnitt 1.2. Die daraufhin in Kapitel 2 vorgestellten Pfadplanungsalgorithmen lassen sich grob in zwei Bereiche aufteilen.

Der erste Bereich sind die Algorithmen, welche auf dem altbewährten A^* (Kapitel

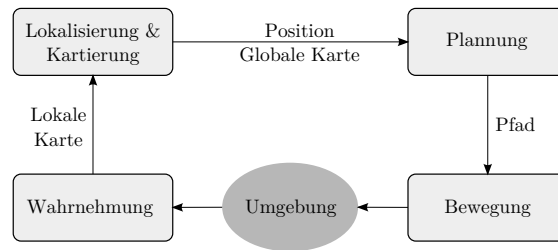


Abbildung 1.1: Umgebung und Interaktion autonomer mobiler Systeme

2.1) aufbauen. Hierzu zählt der *Anytime Repairing A** (Kapitel 2.2), welcher den Pfad schneller findet als der A^* und diesen inkrementell verbessert. Der *Lifelong Planning A** (Kapitel 2.3) zeigt eine Methode, Änderungen des Umgebungsmodells in den bereits bestehenden A^* -Graphen einzubauen, ohne ihn von Grund auf neu zu planen. Der *D* Lite* (Kapitel 2.4) erweitert diesen noch, sodass auch eine Veränderung der Startposition während des inkrementellen Planens möglich ist. Die bisherigen Algorithmen erzeugen suboptimale Pfade, da sie gewissen Einschränkungen der Graphsuche unterliegen. Der *Field D**, welcher in Kapitel 2.5 näher beschrieben wird, versucht dieses aufzuheben. In Kapitel 2.6 wird der *hybrid A** vorgestellt, welcher den A^* so erweitert, dass sofort die fahrzeuggegebenen Einschränkungen mitberücksichtigt werden. Im Rahmen dieser Arbeit wurde versucht eine Kreuzung des *hybrid A** und des *D* Lite* zu erzeugen. In Kapitel 2.6.3 wird erläutert, aus welchem Grund dies nicht gut umsetzbar ist.

Der zweite Bereich der hier vorgestellten Algorithmen basiert auf dem *Rapidly-exploring Random Tree* (Kapitel 2.7) von Steven M. LaValle. Man spricht von einem probabilistischen Pfadplaner, welcher im Gegensatz zum A^* im kontinuierlichen Raum arbeiten kann und nicht auf vordefinierte diskrete Knotenpunkte angewiesen ist. Der *RRT* alleine liefert jedoch sehr suboptimale Pfade, was der *RRT** (Kapitel 2.8) zu beheben versucht. Als letzter Algorithmus wird der eigens entwickelte *Lifelong Planning RRT** (Kapitel 2.9) vorgestellt, welcher ähnlich dem *D* Lite* ein inkrementeller Planer für dynamische Umgebungen und mobile Systeme geeignet ist.

Anschließend werden in Kapitel 3 die Planer-übergreifenden Algorithmen vorgestellt, welche in der Pfadplanungs-Bibliothek ppLib (Kapitel 3.4) genutzt werden. Hierzu zählen Methoden der Kollisionsvermeidung (Kapitel 3.1), Konstruktionen glatter Pfadstücke (Kapitel 3.2) und die abschließende Pfadglättung (Kapitel 3.3). Abschließend finden sich in Kapitel 4 verschiedene Laufzeit- und Speicherbedarfsmessungen der einzelnen Algorithmen und darauf basierende Vergleiche.

1.2 Problemdefinition

Das sogenannte **Pfadplanungsproblem** (*motion planning problem* oder *piano movers problem*) beschreibt die Suche einer Trajektorie von einem Startzustand zu einem Zielzustand, ohne dabei mit Hindernissen zu kollidieren. In der vorliegenden Arbeit wird hierbei hauptsächlich auf die Pfadfindung für ein autonomes Fahrzeug von einer Startpose hin zu einer möglichen Zielpose eingegangen. Diese Posen sind Punkte im Zustandsraum des Fahrzeuges $\mathcal{Z} \subseteq \mathbb{R}^n$. Neben diesem muss zusätzlich der Raum der möglichen Steuerbefehle $\mathcal{U} \subseteq \mathbb{R}^m$ beschrieben werden, sowie die Ausdehnung des Vehikels und dessen nicht-holonomen Bewegungseinschränkungen. Des Weiteren wird eine ausreichende Beschreibung des Arbeitsraumes \mathcal{W} verlangt, um eine Kollisionsvermeidung durchführen zu können. In der Pfadplanung wird hierzu meist eine zweidimensionale Projektion der Umwelt auf eine Befahrbarkeitskarte genutzt, in der definiert ist, welche Zustände durch Hindernisse $B_i \subseteq \mathcal{W}$ blockiert werden und welche nicht:

$$\begin{aligned}\mathcal{Z}_{obs} &:= \{z \in \mathcal{Z} \mid \forall B_i : vehicle(z) \cap B_i \neq \emptyset\} \\ \mathcal{Z}_{free} &:= \mathcal{Z} \setminus \mathcal{Z}_{obs}\end{aligned}$$

Gegebenenfalls kann die Belegtheitskarte auch zu einer Kostenkarte erweitert werden, falls dementsprechende Informationen vorliegen:

$$c : \mathcal{Z}_{free} \rightarrow \mathbb{R}$$

Ziel der Pfadplanung ist es einen Pfad $z(t)$ im Zustandsraum \mathcal{Z} von einem vordefinierten Startzustand bis zu einem Zielzustand zu finden:

$$\begin{aligned}z(t) \text{ mit } \forall t : z(t) \in \mathcal{Z} \\ t \in [0, T], T \in \mathbb{R}_{>0} \\ z(0) = z_{start} \\ z(T) \in \mathcal{Z}_{goal}\end{aligned}$$

Des Weiteren muss auch die Sequenz von Kontrollbefehlen $u(t)$ bekannt sein, welche das Fahrzeug entlang des Pfades führt und somit die gesuchte Trajektorie definiert. Während der Ausführung dieser Trajektorie darf das Subjekt mit keinen Hindernissen kollidieren, was bedeutet, dass es sich zu jeder Zeit in einem freien Zustand befinden muss:

$$\forall t : z(t) \in \mathcal{Z}_{free}$$

gegeben:

- Zustandsraum (Fahrzeug): $\mathcal{Z} \subseteq \mathbb{R}^n$
- Arbeitsraum (Umwelt): $\mathcal{W} \subseteq \mathbb{R}^l; \quad c : \mathcal{Z}_{free} \rightarrow \mathbb{R}$
- Subjektbeschreibung: Ausmaße,
Bewegungseinschränkungen,..
- initialer Zustand: z_{start}
- Menge von Zielzuständen: \mathcal{Z}_{goal}

gesucht:

- Sequenz von Kontrollbefehlen: $u : [0, T] \rightarrow \mathcal{U}$
- zum Ausführen einer Trajektorie
entlang eines Pfades: $z : [0, T] \rightarrow \mathcal{Z}$
 $T \in \mathbb{R}_{>0}$
- vom initialen Zustand zu einem Zielzustand: $z(0) = z_{start}; \quad z(T) \in \mathcal{Z}_{goal}$
- ohne ein Hindernis zu treffen: $\forall t : z(t) \in \mathcal{Z}_{free}$
- mit minimalen Kosten: $J(\mathbf{x}) = \int_0^T c(z(t))dt$

Abbildung 1.2: Das optimale Pfadplanungsproblem

Wird nun außerdem auch noch versucht einen optimalen Weg zu finden, so spricht man von dem **optimalen Pfadplanungsproblem** (*optimal motion planning problem*). Hierzu muss die Kostenfunktion minimiert werden:

$$J(x) = \int_0^T c(z(t))dt$$

Die Definition der Kosten ist hierbei nicht weiter eingeschränkt. Sie können beispielsweise die Streckenlänge oder die benötigte Zeit für ihre Ausführung sein. Zusätzlich können aber auch noch weitere Aspekte mit einfließen, wie beispielsweise die Sicherheit des Pfades.

Zusammengefasst findet sich das Pfadplanungsproblem in Abbildung 1.2. Im Rahmen dieser Arbeit wird jedoch abweichend hierzu meist von nur einem Zielzustand z_{goal} ausgegangen. Des Weiteren wird nicht auf die Planung der Steuerbefehle eingegangen, welche die geplante Trajektorie abfahren würden, sondern lediglich auf die Bestimmung eines zweidimensionalen Pfades. Damit das Fahrzeug diesen entlang fährt, wird in der Praxis oft ein einfacher Regelalgorithmus (z.B. ein PID-Regler) verwendet.

Der wohl bekannteste Algorithmus der dieses Problem grundlegend löst ist der bereits 1968 entwickelte A^* [HNB68](Kap. 2.1) von Peter Hart, Nils J. Nilsson und Bertram Raphael. Der A^* -Algorithmus ist unter den richtigen Bedingungen nicht nur optimal, findet also immer den *besten* Pfad, sondern auch optimal effizient, was bedeutet, dass es keinen Algorithmus geben kann, welcher einen kürzeren

Pfad mit weniger Rechenaufwand findet. Dies gilt in der Pfadplanung zwar nur in eingeschränkter Weise (mehr dazu in Kapitel 2.1.4), dient aber dennoch als Grundlage für viele Erweiterungen, von denen einige in dieser Arbeit untersucht werden. Der große Nachteil des A^* ist jedoch, dass er ein Graph-Suchalgorithmus ist, weshalb die Umgebung als Graph dargestellt und somit auf Knoten beschränkend diskretisiert werden muss.

Eine etwas unkonventionelle probabilistische Lösung des Pfadplanungs Problems bietet der *Rapidly-exploring Random Tree* [LaV98](Kap. 2.7) von Steven M. LaValle, welcher im Gegensatz zum A^* im kontinuierlichem Raum plant. Auch für diesen Algorithmus gibt es Erweiterungen, welche aus dem ursprünglich eher unbrauchbaren Algorithmus einen passablen Pfadplaner machen.

Soll der Planungsalgorithmus für die Pfadplanung autonomer Fahrzeuge genutzt werden, so genügt es in der Regel nicht, zu Beginn einmal einen Pfad zu planen, welcher dann genau so ausgeführt wird. Da sich die Umgebung während der Fahrt ändern kann oder unbekanntes Gebiet eventuell erst während der Fahrt erkundet wird, ist es notwendig den ermittelten Pfad während der Fahrt zu aktualisieren, um die Kollisionsfreiheit und Optimalität ständig zu gewährleisten. Allerdings ist es sehr aufwändig bei jeder neuen Umgebungswahrnehmung von Grund auf neu zu planen. Daher geht die Entwicklung der Pfadplaner in Richtung **Replanning-Lifelong-** oder **Anytime-** Algorithmen. Dies bedeutet, dass der Algorithmus zu Beginn einmal gestartet wird und bis zur endgültigen Zielfindung aktiv bleibt (*lifelong*). Ändert sich das Wissen über die Umgebung, so plant er nicht von Grund auf neu, sondern nutzt das bereits aufgebaute Wissen und ändert es entsprechend ab (*replanning*), wodurch er in der Lage ist zu jeder Zeit einen gültigen Pfad zu liefern. Ist zu Beginn eine sehr schnelle Pfadfindung gewünscht, so kann dies auf Kosten von Optimalitätseinbußen erreicht werden. Der so ermittelte anfangs noch nicht optimale Pfad, wird dann jedoch im weiteren Verlauf stetig verbessert (*anytime*).

Kapitel 2

Pfadplanungsalgorithmen

2.1 A*

Der A^* -Algorithmus ist ein bekannter Graphsuchalgorithmus, welcher ursprünglich 1968 von Peter Hart, Nils J. Nilsson und Bertram Raphael entwickelt wurde [HNB68]. Er zählt zu den informierten Suchalgorithmen, was bedeutet, dass er eine Heuristik (Kap. 2.1.1) nutzt, um der Suche eine gewisse Richtung zu geben und sie so zu beschleunigen. Da der A^* im eigentlichen Sinne ein Graphsuchalgorithmus ist, ist es notwendig den Suchraum mittels eines Graphen darzustellen. In diesem Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ stellen die Knotenpunkte $v \in \mathcal{V}$ Posen im Zustandsraum \mathcal{Z} des Fahrzeuges dar, welche um algorithmusspezifische Parameter erweitert werden. Die Kanten \mathcal{E} beschreiben die direkten lokalen Nachbarschaften zwischen diesen Knotenpunkten und besitzen Gewichte ≥ 1 definiert durch die Kostenfunktion c . Mehr dazu findet sich in Kapitel 2.1.4. Des Weiteren wird unbekanntes Gebiet als frei befahrbar angenommen.

Dass der A^* -Algorithmus so populär ist, verdankt er einerseits seiner *Vollständigkeit*, was bedeutet, dass er sicher einen Pfad vom Start- zum Zielpunkt findet, falls einer existiert. Des Weiteren ist er gegeben dem unterliegenden Graphen *optimal*: er findet immer den kürzesten Weg hinsichtlich der Pfadkosten, welche über die Kantengewichte im Graphen definiert werden, solange die Kantengewichte stets positiv sind. Gibt es mehrerer kürzeste Wege, so findet er einen von diesen. Ebenso ist der Algorithmus *optimal effizient*, was bedeutet, dass es keinen Algorithmus geben kann, welcher mit der selben Heuristik den optimalen Weg schneller findet. Die Schnelligkeit bezieht sich hierbei auf die Anzahl der expandierten Knoten.

2.1.1 Heuristik

Die Heuristik ist eine Funktion $h: \mathcal{Z} \rightarrow \mathbb{R}$ welche jedem Punkt im Zustandsraum eine Schätzung der Kosten zuordnet. Damit der A^* -Algorithmus immer den optimalen Weg findet, ist es wichtig, dass die Heuristik zum Ziel hin monoton abfallend ist und folgende Ungleichung für jeder Knotenpaar erfüllt ist:

$$h(v) \leq c(v, v') + h(v'), \quad \forall v, v' \in \mathcal{V} \quad (2.1)$$

Hierbei sind $c(v, v')$ die tatsächlichen Kosten zwischen v und v' . In dem Fall gilt die Heuristik als konsistent und ist damit zulässig für den A^* -Algorithmus. Für die Optimalität des Planers ist es sehr wichtig, dass die Heuristik unterschätzend ist. Je näher diese Schätzung jedoch an den tatsächlichen Kosten liegt, desto schneller findet der Algorithmus das Ziel. Ein Beispiel für eine gültige Heuristik in der Pfadplanung ist die euklidische Distanz zwischen der Fahrzeugposition und dem Zielzustand. Es besteht aber auch die Möglichkeit Umgebungswissen mit in die Heuristik einfließen zu lassen, um somit besser um Hindernisse herum zu führen. Hierfür könnte eine mittels *Field D** (Kap. 2.5) generierte Kostenkarte als Grundlage dienen. Außerdem ist es auch möglich verschiedene Heuristiken zu kombinieren, da das Maximum zweier konsistenter Heuristiken ebenfalls konsistent ist und somit zulässig. Liegen dem Algorithmus keinerlei Informationen vor, welche für eine Heuristik genutzt werden können, so ist sogar die Nullheuristik $h_0: \mathcal{Z} \rightarrow 0$ eine zulässige Wahl, da sie monoton und unterschätzend ist und somit die Dreiecksungleichung 2.1 erfüllt. In diesem Fall würde aus dem A^* der *Dijkstra*-Algorithmus resultieren.

2.1.2 Der Algorithmus

Der A^* weist jedem Knoten v im Graphen einen f -Wert zu, bestehend aus der Summe der errechneten Kosten g , vom Startzustand v_{start} bis zum Knoten v , und den geschätzten Kosten h bis zum Zielzustand v_{goal} , die Heuristik. Initial werden die g -Werte aller Knoten auf ∞ gesetzt (Abb. 2.1, Zeile 1.1), da diese noch nicht bekannt sind. Lediglich die Kosten vom Startknoten können auf Null gesetzt werden (Zeile 1.2). Beginnend mit v_{start} beinhaltet die *OpenList* Q nun immer genau alle lokal inkonsistenten Knoten, von welchen aus der Algorithmus sein Wissen ausbreitet, indem er den noch nicht bearbeiteten Nachbarknoten die g -Werte zuweist (Zeile 2.8). Hierbei wird immer genau der Knoten als nächstes behandelt, welcher den geringsten f -Wert besitzt (Zeile 2.2). Sobald der Zielknoten erreicht wurde, kann der Algorithmus abbrechen und den kürzesten Pfad mittels *backtracing* zurückverfolgt (Zeilen 5-6). Hierzu wird ausgehend vom v_{goal} immer derjenige Nachbarknoten als Vorgänger gewählt, welcher den geringsten g -Wert hat, bis der Startknoten v_{start} erreicht wird. Alternativ kann auch jedem Knoten

1. initialize()

```

1 foreach  $v \in \mathcal{V}$  do  $g(v) = \infty$ ;
2  $g(v_{start}) \leftarrow 0$ ;
3  $Q \leftarrow \{v_{start}\}$ ;

```

2. computePath()

```

1 while  $Q \neq \emptyset$  do
2    $v \leftarrow \operatorname{argmin}_{v' \in Q} (g(v') + h(v'))$ ;
3    $Q \leftarrow Q \setminus \{v\}$ ;
4    $C \leftarrow C \cup \{v\}$ ;
5   if  $v = v_{goal}$  then
6     return  $\operatorname{backtrace}(v)$ ;
7   foreach  $v_{succ} : \{v, v_{succ}\} \in \mathcal{E}, v_{succ} \notin C$  do
8     if  $g(v_{succ}) > g(v) + c(v, v_{succ})$  then
9        $g(v_{succ}) \leftarrow g(v) + c(v, v_{succ})$ ;
10       $Q \leftarrow Q \cup \{v_{succ}\}$ ;
11     end
12   end
13 end

```

3. $path_{v_{start} \rightarrow v_{goal}} \leftarrow \operatorname{main}_{A^*}(\mathcal{G} = (\mathcal{V}, \mathcal{E}), v_{start}, v_{goal} \in \mathcal{V})$

```

1 initialize();
2 computePath();

```

Abbildung 2.1: A^* -Algorithmus

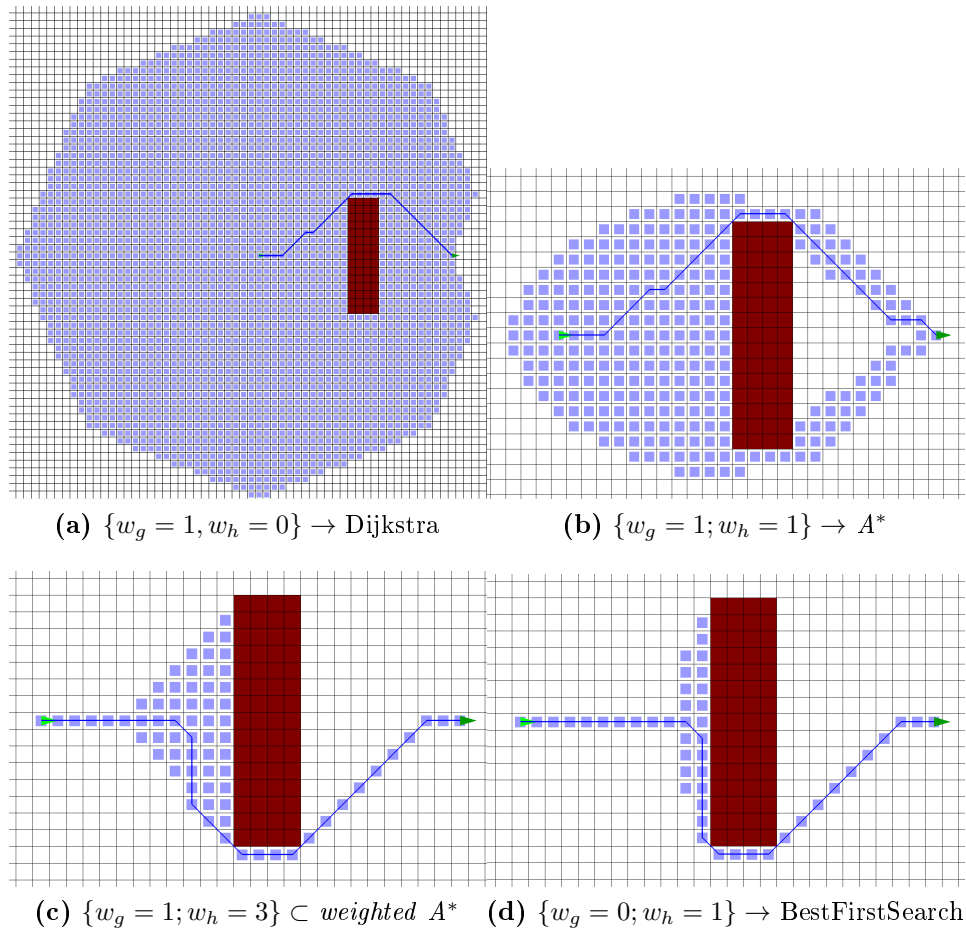


Abbildung 2.2: Vergleich verschiedener Gewichte im A^*

bereits während des Expandierens ein Vaterknoten zugewiesen werden, von welchem aus dieser erreicht wurde (in Zeile 7).

Lokal inkonsistent ist ein Knoten immer dann, wenn er bereits ein g-Wert zugewiesen bekam (Zeile 2.9), aber noch nicht abschließend behandelt wurde (Zeilen 2.2 - 2.4). Somit sind dies alle Knoten, welche bereits eine erste Abschätzung der Kosten haben, jedoch nicht klar ist, ob dies die endgültigen Kosten sind, da sie sich noch verringern könnten. Erst nach abschließender Behandlung (Zeilen 2.2 - 2.4) steht der korrekte g-Wert des Knotens fest und dieser kann somit in die *ClosedList C* eingefügt werden. Die Bezeichnung „lokal inkonsistent“ ist für den einfachen A^* kaum von Bedeutung, wird jedoch für die Erklärung einiger Erweiterungen benötigt.

2.1.3 Gewichteter A^*

Das Verhalten des A^* lässt sich beeinflussen, indem die Kosten und die Heuristik mit den Faktoren w_g und w_h gewichtet werden. So lässt sich beispielsweise die Zielfindung beschleunigen, indem der Heuristik ein Gewicht > 1 gegeben wird. In diesem Fall spricht man von dem *gewichteten A^** (*weighted A^**) [Poh70]. Da nun bei der Expansion die Heuristik h stärker gewichtet wird als die bisherigen Kosten g , steuert der Graph viel stärker auf das Ziel zu und geht somit weniger in die Breite. Die Heuristik ist nun aber nicht mehr zwangsläufig unterschätzend, was dazu führen kann, dass nicht mehr der kürzeste Weg gefunden wird: *Der Algorithmus ist nicht mehr optimal*. Zu erkennen ist dies in Abbildung 2.2. In (b) sieht man den normalen A^* mit gleich gewichteter Heuristik und in (c) ist ein gewichteter A^* mit einem dreifachem Heuristik-Gewicht. Der gewichtete A^* expandiert weniger Knoten (hellblau), findet dafür jedoch einen suboptimalen Pfad (blaue Linie, von links nach rechts).

Mithilfe der Gewichte erkennt man gut die Verwandtschaft des A^* zu dem Dijkstra-Algorithmus und der greedy best first - Suche. Gibt man der Heuristik ein Nullgewicht, nutzt also gar keine Heuristik, so ergibt sich aus dem A^* der Dijkstra-Algorithmus, welcher mittels Breitensuche den Graphen durchsucht bis er das Ziel erreicht hat. In Abbildung 2.2 (a) erkennt man, dass er wesentlich mehr Knoten expandiert, jedoch wie der normale A^* immer einen optimalen Pfad findet. Die *best-first* Suche expandiert im Gegensatz dazu immer genau den Knoten, welcher die geringsten geschätzten Kosten bis zum Ziel hat. Dieser Algorithmus ist sehr schnell, aber nicht mehr optimal. In Abbildung 2.2 (d) sieht man ein Beispiel hierzu.

2.1.4 Einschränkungen für die Pfadplanung

Der A^* ist eigentlich ein Algorithmus der Graphsuche. Dies bedeutet, dass der Arbeitsraum, welcher in der Pfadplanung die Umgebung des Fahrzeuges ist, als Graph dargestellt werden muss. Für das autonome Fahren wird hierzu eine zweidimensionale Belegtheitskarte beziehungsweise Kostenkarte angelegt. Sie repräsentiert einen Graphen mit achter-Nachbarschaft mit den Distanzen zwischen den Knotenmittelpunkten als Gewichte der Kanten. Liegen außerdem Informationen über die Qualität der Befahrbarkeit des Terrain vor, so können diese über die Kantengewichte mit einbezogen werden. Diese Wahl der Umgebungsrepräsentation ist sehr verbreitet, da viele Terrainklassifikationsalgorithmen einen solchen Graphen als Ergebnis liefern. Außerdem lassen sich solche Karten effizient speichern. Der A^* verspricht optimal und vollständig zu sein, gegeben den vorliegenden Graphen. Jedoch verhindert genau dieser, durch die Diskretisierung des Arbeitsraumes, sehr oft die Ermittlung des tatsächlich optimalen Pfades. Zu erkennen ist dies in Ab-

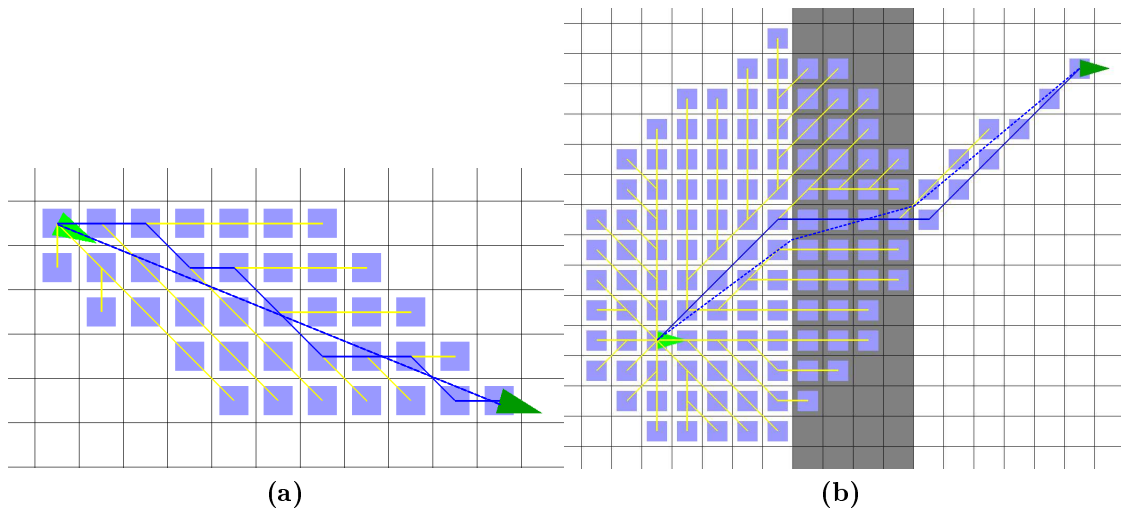


Abbildung 2.3: Beispiele für Optimalitätseinbußen der graphgebundenen Pfadsuche

bildung 2.3. Das grau markierte Gebiet hat doppelt so hohe Pfadkosten wie das weiße Gebiet. Die durchgezogene blaue Linie zeigt den von A^* gefunden Pfad an, die gestrichelte Linie wäre der tatsächlich optimale Pfad. Wie groß die Abweichung der Strecke genau werden kann, wir in Kapitel 4.4.1 ermittelt. Dies lässt leicht erkennen, dass der A^* zwar ein optimaler Algorithmus der Graphsuche ist, jedoch nicht zwangsweise optimal in der Pfadplanung von Fahrzeugen.

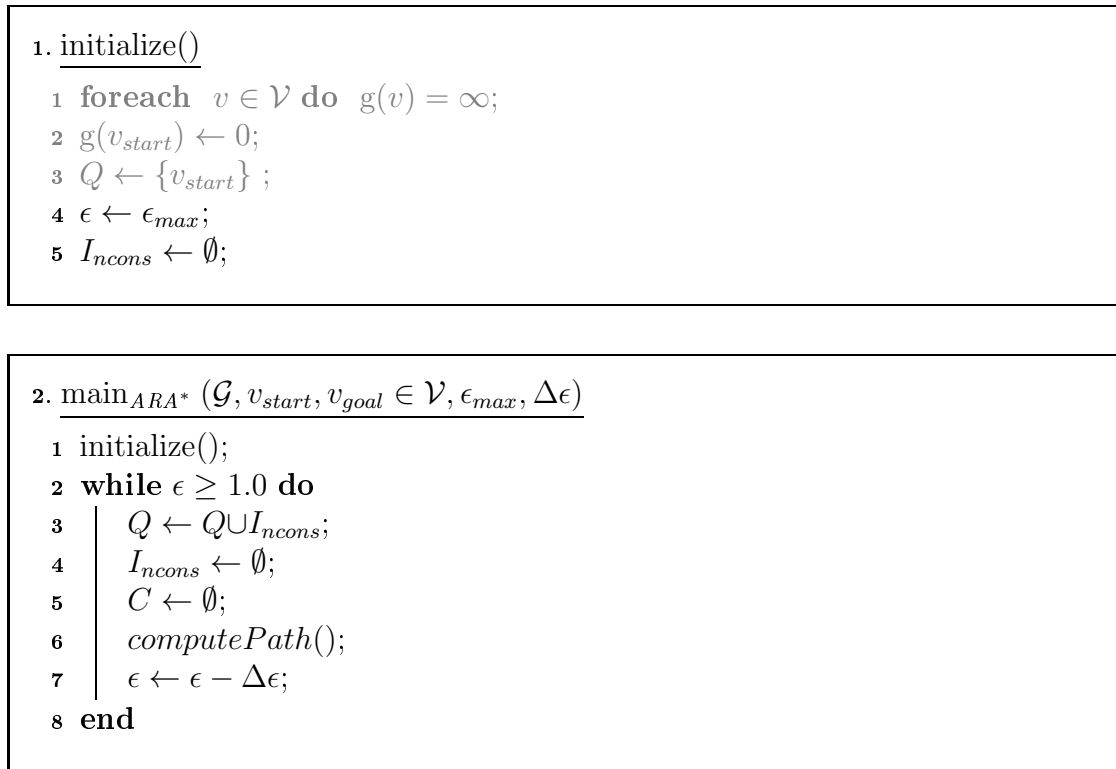
2.2 Anytime Repairing A^*

Der *Anytime Repairing A^** (ARA^*) [LGT04] ist eine *anytime*-Abwandlung des A^* . Dies bedeutet, dass der Algorithmus sehr schnell einen ausführbaren, nicht zwangsläufig optimalen Plan findet, welcher sich mit der Anzahl der Iterationen stetig verbessert. Diese Art von Algorithmus kommt besonders dann zum Einsatz, wenn eine sehr große Strecke geplant werden muss, das Fahrzeug aber sehr früh starten soll. Hierbei wird in Kauf genommen, dass der erste Plan gegebenenfalls nicht optimal ist. Dennoch lässt sich an diesem bereits festmachen, ob es überhaupt einen ausführbaren Pfad gibt. In der Regel genügt dieser auch schon, um dem Fahrzeug bereits eine grobe Richtung vorzugeben. Der Pfad welcher vom ARA^* ausgegeben wird ist ϵ -suboptimal, was bedeutet, dass der Pfad maximal ϵ mal so lang ist wie der optimale Pfad. Beim ARA^* ist es jederzeit leicht möglich dieses ϵ zu ermitteln. Der Pfad wird mit jedem Iterationsschritt des Algorithmus verbessert, bis letztendlich der selbe optimale Pfad gefunden wird, welcher auch ein A^* mit gleicher Heuristik finden würde. Hierbei wird in jeder Iteration das bereits erlangte Wissen aus den vorherigen Iterationen genutzt, wodurch Rechenaufwand und damit Laufzeit eingespart werden kann. Außerdem kann gezeigt werden, dass im Gegensatz zu anderen ähnlichen Algorithmen (z.B. der Anytime A^* von Zhou und Hansen [ZH02]), jeder Knoten pro Iteration maximal einmal bearbeitet wird, wodurch die Laufzeit nach oben hin beschränkt ist [LGT04].

2.2.1 Der Algorithmus

Die Grundidee des ARA^* besagt, dass ein gewichteter A^* (2.1.3) mit relativ großem Heuristikgewicht ϵ (Abb. 2.4, Zeile 1.4) gestartet wird, wodurch mit vergleichsweise wenigen Expansionsschritten ein gültiger Pfad gefunden wird. Anschließend wird mit jeder Iteration die Heuristik etwas (Zeile 2.7) weniger gewichtet, bis ϵ den Wert 1 hat und somit der Algorithmus das gleiche Ergebnis liefert wie ein „normaler“ A^* . Da der ARA^* jedoch das Vorwissen aus den vorherigen Iterationen nutzt, ist er effizienter, als würde man jedes Mal den gewichteten A^* mit entsprechendem Heuristikgewicht von Grund auf neu planen lassen. Genauere Daten dazu finden sich in der Evaluation (Kap: 4.1). Um das Wissens aus vorherigen Iterationen nutzen zu können, werden die berechneten Kosten g der Knoten nicht gelöscht. Lediglich die *ClosedList* wird geleert (Zeile 2.5), um die Aktualisierung bestehender Knoten nicht zu hindern.

Abbildung 2.5 zeigt, wie eine einzelne Iteration des Planers aussieht. Die grau markierten Zeilen sind hierbei Zeilen, welche genau so auch im A^* vorkommen, die daher auch keine genauere Betrachtung mehr bedürfen. Nutzt der A^* eine gültige konsistente Heuristik, so ist garantiert dass jeder Knoten nur einmal behandelt wird. Da der ARA^* jedoch zu Beginn ein Heuristikgewicht $\epsilon > 1$ nutzt, ist die

Abbildung 2.4: ARA^* -Algorithmus pt. 1

Heuristik nicht mehr zwangsläufig unterschätzend und somit nicht mehr konsistent. Dies kann dazu führen, dass Knoten mehrfach besucht werden, was jedoch unerwünscht ist. Um dies zu verhindern, pflegt der ARA^* nicht nur eine einzelne Liste Q für die inkonsistenten Knoten, sondern führt eine neue Liste I_{incons} ein. Stellt sich heraus, dass ein Knoten seine Kosten weiter reduzieren kann (Abb 2.5, Zeile 3.8), er somit lokal inkonsistent ist, sich aber bereits in der *ClosedList* C befindet, dann wird er in der I_{incons} -Liste vorgemerkt (Zeile 3.11), anstatt ihn ein zweites mal zu expandieren (Zeile 3.10). Für den nächsten Durchlauf werden alle Knoten von dieser Liste mit in die *OpenList* verschoben (Abb 2.4, Zeilen 2.3 - 2.4), um in diesem Durchlauf abschließend behandelt zu werden. Die Entwickler von ARA^* bewiesen in [LGT03], dass trotz der frühzeitigen Unterbrechung der Expandierwelle die ϵ -Suboptimalität gewährleistet ist.

Die Suche innerhalb einer Iteration kann beendet werden, sobald der f-Wert des „besten“ Knotens größer ist, als der f-Wert des Zielknotens, da dies bedeutet, dass dieser den Zielpfad hinsichtlich des aktuellen ϵ nicht weiter verbessern kann (Abb. 2.5, Zeile 3.1). Durch die Inkonsistenz der Heuristik, kann dies jedoch dazu führen, dass selbst ein von Grund auf neu planender A^* mit gleichem Heuristikgewicht


```

3. computePath()
1  while  $\min_{v' \in Q} (g(v') + \epsilon \cdot h(v')) \geq g(v_{goal}) + \epsilon \cdot h(v_{goal})$  do
2     $v \leftarrow \operatorname{argmin}_{v' \in Q} (g(v') + h(v'))$ ;
3     $Q \leftarrow Q \setminus \{v\}$ ;
4     $C \leftarrow C \cup \{v\}$ ;
5    if  $v = v_{goal}$  then
6      return  $\operatorname{backtrace}(v)$ ;
7    foreach  $v_{succ} : \{v, v_{succ}\} \in \mathcal{E}$  do
8      if  $g(v_{succ}) > g(v) + c(v, v_{succ})$  then
9         $g(v_{succ}) \leftarrow g(v) + c(v, v_{succ})$ ;
10       if  $v_{succ} \notin C$  then  $Q \leftarrow Q \cup \{v_{succ}\}$ ;
11       else  $I_{ncons} \leftarrow I_{ncons} \cup \{v_{succ}\}$ ;
12     end
13   end
14 end

```

Abbildung 2.5: ARA*-Algorithmus pt. 2

einen kürzeren Pfad zum Ziel findet, obwohl auch hier die ϵ -Suboptimalität weiterhin eingehalten wird, wie in Kapitel 4.1 genauer nachzulesen ist.

2.2.2 Ergebnisse

Einen beispielhaften Durchlauf in zwei Dimensionen ist in Abbildung 2.6 zu sehen. Gut zu erkennen ist, dass der Graph in den früheren Iterationen geringere Ausmaße hat, wodurch der Zielpfad auch schneller gefunden werden kann. Im weiteren Verlauf breitet sich der Graph jedoch aus, wodurch kürzere Wege gefunden werden können. Außerdem erkennt man in der Abbildung, dass am Ende der Iterationen nicht immer alle Knoten behandelt wurden und somit nicht in die *ClosedList* kamen (hellblaue Knoten). Auf diese Weise konnten Kosten eingespart werden.

Sebastian Thrun et al. zeigt in [LGT04], dass ihr autonomes Fahrzeug durch das planen mittels ARA* statt eines optimalen Planers einen Startvorteil von über 10.5 Sekunden hatte, wodurch das Fahrzeug früher losfahren und somit früher ans Ziel kommen konnte. Hierbei planten sie in vier Dimensionen (x,y-Position, Orientierung, Geschwindigkeit) auf einem 2D-Grid mit über 50000-Zellen.

Der praktische Nutzen des ARA* ist jedoch etwas zweifelhaft, da der Algorithmus zwar in mehreren Iterationen, also auch während der Fahrt, plant, jedoch nicht mit einer Änderung der Startposition umgehen kann. Sollte sich also das Fahrzeug, zum

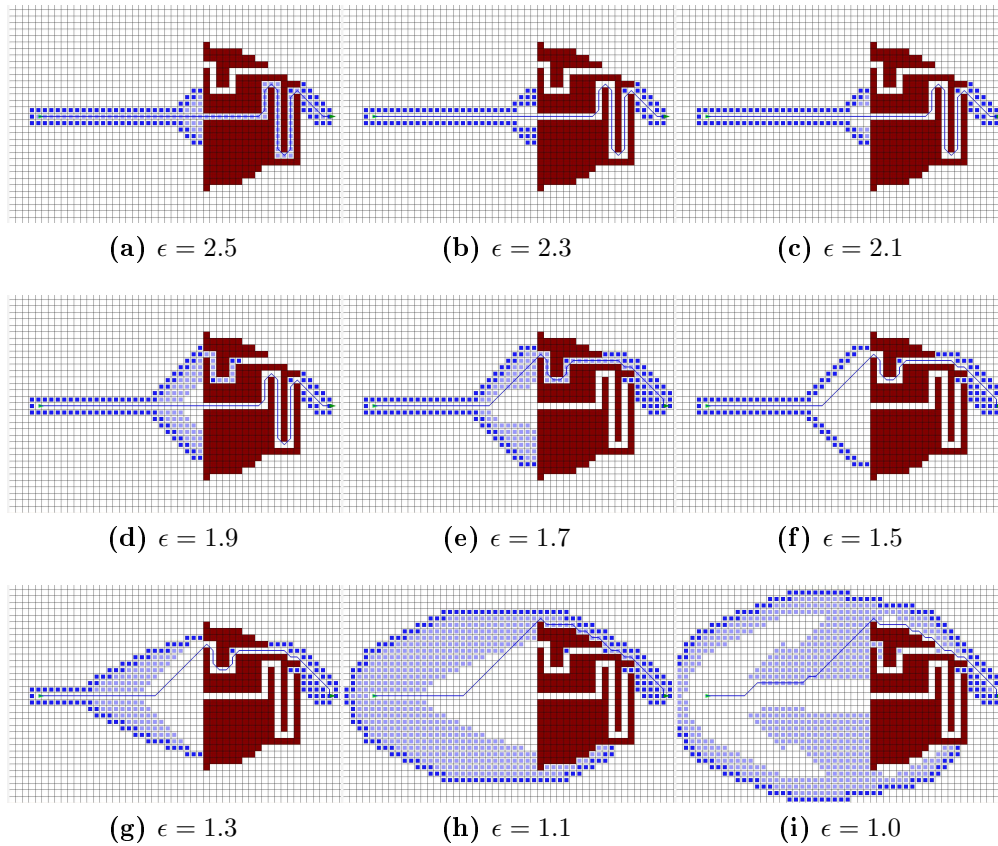


Abbildung 2.6: Beispielverlauf des *Anytime Repairing A** mit $\epsilon_{max} = 2.5$ und $\Delta\epsilon = 0.2$ (hellblaue Felder sind die jeweils aktualisierte Knoten)

Beispiel durch eine Änderung des Pfades, nicht mehr auf diesem befinden, so kann der *ARA** dies nicht berücksichtigen. Des Weiteren kann er nur in einer statischen Umgebung planen, da nicht definiert ist, wie sich Änderungen des Umgebungswissens in den Plan einbauen lassen.

2.3 Lifelong Planning A^*

Der nachfolgend beschriebene Algorithmus, ist ein inkrementelle Pfadsuche auf Basis des A^* . Dies bedeutet, dass eine Änderung des Umgebungsmodells direkt in einen bereits bestehenden Plan eingebaut werden kann, anstatt von Grund auf neu zu planen. Im Gegensatz zum A^* eignet sich der *Lifelong Planning A^** (LPA^*) [KLF04] dafür, über einen längeren Zeitraum aktiv zu bleiben, da er sein erlangtes Wissen in späteren Planungen wiederverwenden kann. Der LPA^* wird in dieser Arbeit behandelt, da er als Grundlage des $D^* Lite$ (Kapitel 2.4) dient und hilft, diesen besser zu verstehen.

2.3.1 Vorgehensweise

Wird der *Lifelong Planning A^** das erste Mal initial ausgeführt, so plant er genau so wie auch der normale A^* und resultiert in dem selben Graphen mit dem selben gefundenen Pfad (siehe Abbildung 2.9 (a)). Die Besonderheit des LPA^* besteht nun darin, dass er bei Änderungen im Umgebungsmodell, diesen Graphen so abändert, dass letztendlich wieder ein gültiger A^* -Graph entsteht. Hierbei werden die Knoten, bei denen sich nichts ändert, nicht nochmal neu berechnet, es werden lediglich die Knoten aktualisiert, welche tatsächliche geänderte Kostenwerte erhalten. Würde der A^* zu einem Umgebungsmodell U_1 einen Graph G_1 mit dem Plan P_1 finden, und zu einem etwas abgewandelten Umgebungsmodell U_2 den Graph G_2 mit Plan P_2 , so ist die Grundidee des LPA^* , dass er nach erfolgreichen ermitteln von P_1 mit identischem Graph G_1 , diesen soweit zurück baut bis nur noch die gemeinsame Schnittmenge $G_1 \cap G_2$ bestehen bleibt (Abb. 2.9 (b)). Von dieser ausgehend plant der Algorithmus weiter, bis ebenfalls ein gültiger Graph G'_2 , mit gültigem Pfad P'_2 , ermittelt wurde. Da das Zurückbauen jedoch nur in dem Rahmen durchgeführt wird, wie es für den Plan P'_2 von Relevanz ist, kann dies dazu führen das irrelevante „Reste“ des alten Plans übrig bleiben (zu sehen in Abb. 2.9 (d)). Jedoch ist sicher gegeben, dass $G_2 \subseteq G'_2$, wodurch der gefundene Pfad P'_2 identisch mit P_2 ist.

Da auch das Zurücksetzen des Graphens genau so aufwändig ist wie das Erweitern, ist leicht zu erkennen, dass dieses Vorgehen je nach Situation aufwändiger sein kann, als das einfach Neuplanen mittels dem normalen A^* . Beim LPA^* wird jedoch garantiert, dass jeder Knoten maximal zweimal pro Iteration besucht wird ([KLF04] Kapitel 5.1). Dabei kann er maximal einmal unter- und einmal überkonsistent sein, womit die Laufzeit begrenzt ist solange der Graph begrenzt ist. Der *Lifelong Planning A^** lohnt sich immer dann, wenn nur kleine Änderungen in der Umgebung wahrgenommen werden. Daher ist der LPA^* (und somit auch der $D^* Lite$), besonders geeignet für das Planen einer autonomen Fahrt, wenn das Umgebungswissen mit einer relativ hohen Frequenz inkrementell aktualisiert wird.

```

1. initialize()
  1 foreach  $v \in \mathcal{V}$  do
  2   |  $g(v) \leftarrow \text{rhs}(v) \leftarrow \infty$ ;
  3 end
  4  $\text{rhs}(v_{\text{start}}) \leftarrow 0$ ;
  5  $Q \leftarrow \{v_{\text{start}}\}$ ;

```

```

2. key(v)
  1 return  $\left[ \min(g(v), \text{rhs}(v)) + h(v) ; \min(g(v), \text{rhs}(v)) \right]$ ;

```

```

3. mainLPA*( $\mathcal{G}, v_{\text{start}}, v_{\text{goal}} \in \mathcal{V}$ )
  1 initialize();
  2 while 1 do
  3   | computePath();
  4   | waitForChanges();
  5   | foreach  $v : \{v', v\} \in (\mathcal{E} \cap \text{Changes})$  do updateNode(v);
  6 end

```

Abbildung 2.7: Lifelong Planning A* - Algorithmus pt. 1

2.3.2 Der Algorithmus

Auf Grund der Tatsache, dass der *Lifelong Planning A** einen bereits bestehenden Graphen gegebenenfalls abändert muss, um ihn an gegebene Umweltänderungen anzupassen, kann ein Knoten niemals als *endgültig bearbeitet* bezeichnet werden. Dieser Fakt verdeutlicht wieso der *LPA** auf die *ClosedList C* verzichten kann. Stattdessen benötigt der Algorithmus jedoch zu jedem Knoten, neben dem im *A** bereits verwendeten g-Wert (die bereits festgestellten Kosten vom Start bis zu besagtem Knoten) und dem h-Wert (die geschätzten Kosten von diesem Knoten bis zum Zielzustand) noch einen weiteren Wert: der rhs-Wert (*right-hand-side*). Dieser wird simultan zum g-Wert mit ∞ initialisiert (vgl. Abb 2.7 Zeile 1.2). Die rhs-Variable enthält invariant die minimalen g-Kosten die der Knoten gegeben den

4. updateNode(v)

```

1 if  $v \neq v_{start}$  then
2   |  $rhs(v) = \min_{v':\{v',v\} \in E} (g(v') + c(v', v));$ 
3 if  $g(v) \neq rhs(v)$  then
4   |  $Q \leftarrow Q \cup \{v\};$ 
5 else  $Q \leftarrow Q \setminus \{v\};$ 

```

5. computePath()

```

1 while  $\min_{v \in Q} key(v) \leq key(v_{goal})$  or  $rhs(v_{goal}) \neq g(v_{goal})$  do
2   |  $v \leftarrow \operatorname{argmin}_{v \in Q} key(v);$ 
3   | if  $g(v) > rhs(v)$  then
4     |  $g(v) \leftarrow rhs(v);$  ▷ überkonsistent
5   | else
6     |  $g(v) \leftarrow \infty;$  ▷ unterkonsistent
7     |  $updateNode(v);$ 
8   | foreach  $v_{succ} : \{v, v_{succ}\} \in E$  do  $updateNode(v_{succ})$ 
9 end

```

Abbildung 2.8: Lifelong Planning A* - Algorithmus pt. 2

g-Werten seiner Vorgängerknoten, haben kann. Es gilt also zu jeder Zeit für jeden Knoten v :

$$\text{rhs}(v) = \begin{cases} 0 & \text{wenn } v = v_{start}, \\ \min_{v':\{v',v\} \in \mathcal{E}} (g(v') + c(v',v)) & \text{sonst.} \end{cases}$$

Es gelten genau die Knoten als *lokal inkonsistent* (vgl. Sektion 2.1.2), deren g-Wert ungleich dem rhs-Wert ist. Dies sind immer genau die Knoten, welche in der *OpenList* zu finden sind, da die Ungleichheit auf eine Änderung der g-Kosten hinweist. Der rhs-Wert eines Knotens hat eine ähnliche Bedeutung wie der g-Wert, gilt jedoch als aktueller, da er bereits beim ersten Erreichen eines Knotens gesetzt wird (vgl. Abb. 2.8 Zeile 5.9), wohingegen der g-Wert, im Gegensatz zum Vorgehen beim A^* , erst beim abschließenden Bearbeiten einer Planungsiteration gesetzt wird (Zeilen 5.3 - 5.8). Hierdurch lässt sich der rhs-Wert als Indiz dafür wählen, ob zu diesem Knoten ein schnellerer Weg gefunden wurde, als der vorher Bekannte, oder ob der bisher angenommene Pfad nun ungültig geworden ist und somit ein längerer Pfad geplant werden muss. Im ersten Fall wird der Knoten *überkonsistent* genannt und ist daran zu erkennen, dass der rhs-Wert geringer ist als der g-Wert. In diesem Fall ist der g-Wert mit dem rhs-Wert gleich zu setzen, da der neu ermittelte Weg zu diesem Knoten der schnellere ist. Dies ist in Abbildung 2.8 in den Zeilen 5.3 - 5.4 zu erkennen. Ist der g-Wert jedoch geringer als der rhs-Wert, so gilt dieser Knoten als *unterkonsistent*. Dies tritt ein, wenn der bisher geplante Pfad zu diesem Knoten durch neu entdeckte Hindernisse unterbrochen wurde und der g-Wert dadurch ungültig geworden ist. Weil nicht davon ausgegangen werden kann, dass der rhs-Wert in dieser Situation die korrekten Kosten wiedergibt, da auch dieser auf ungültigen Knoten basieren kann, besteht zu diesem Zeitpunkt noch kein Wissen über die tatsächlichen Kosten, weshalb der g-Wert auf ∞ gesetzt wird. Nachdem der g-Wert eines Knotens verändert wurde, muss, unabhängig davon ob dieser über- oder unterkonsistent war, diese Information an die Nachfolgeknoten weitergegeben werden, damit diese ihren rhs-Wert aktualisieren können (vgl. Zeile 5.3).

Im Gegensatz zum A^* plant der LPA^* nicht pro Iteration so lange bis der Zielknoten erreicht wird, da dieser im Normalfall bereits in der Vorrunde erreicht wurde, aber gegebenenfalls falsche veralteten Werte besitzt. Daher wird genau so lange geplant bis der aktuell geringste Schlüsselwert aus der *OpenList* höher ist als die Kosten des Zielknotens (Zeile 5.1), da dies bedeutet, dass der „beste“ inkonsistente Knoten den Zielknoten nicht mehr verbessern kann. Hierbei ist der Schlüsselwert eines Knotens v (zu vergleichen mit dem f -Wert des A^* : $f(v) = g(v) + h(v)$) ein Tupel aus $\min(g(v), \text{rhs}(v)) + h(v)$ und $\min(g(v), \text{rhs}(v))$, welcher einer lexikographischen Ordnung unterliegt (Zeile 2.1).

2.3.3 Ergebnisse

Durch das oben beschriebene Vorgehen, plant der LPA^* dort wo Kosten gesenkt werden können, genau so wie der normale A^* , unabhängig ob die Werte der Knoten vorher unbekannt (also ∞) waren, oder bereits einen älteren, höheren Wert hatten. Tauchen in dem Umgebungsmodell neue Hindernisse auf oder erhöhen sich die Kosten einiger Terrainzellen, so breitet sich diese Information in der gleichen Form aus, jedoch werden hier die alten Kosten der betroffenen Knoten gelöscht.

Beobachtet man den Algorithmus in Abbildung 2.9, so kann man deutlich Graph-ausbreitende und Graph-verkleinernde Wellen erkennen. Hierbei ist unten links der Startknoten und oben rechts das Ziel. Selbiges Verhalten ist auch zu beobachten, wenn sich lediglich die Kosten einzelner Kanten ändern. Je nach Situation kann es außerdem auftreten, dass sich mehrere verschiedene Wellen gleichzeitig ausbreiten. Außerdem ist es möglich diese Umgebungsänderung auch während dem Planen einzubauen. Es ist nicht unbedingt erforderlich auf das beenden einer Iteration zu warten, wie es in der Abbildung 2.7 Zeile z 3.4 - 3.5 umgesetzt ist (vgl. [KLF04], Kapitel 7).

Der *Lifelong Planning A^** wandelt den A^* nun so ab, dass Änderungen im Umgebungsmodell in den aktuellen Plan eingebaut werden können. Jedoch kann er, so wie er bis jetzt beschrieben wurde, nicht mit einer Veränderung der Fahrzeugposition umgehen. Daher eignet er sich noch nicht vollständig als ein *lifelong*-Planungsalgorithmus einer autonomen Fahrt in unbekannter dynamischer Umgebung.

2.4 D^* -Lite

Wie in Kapitel 2.3 beschrieben kann der *Lifelong Planning A** Änderungen der Umwelt in den bereits bestehenden Plan einbauen. Er kann jedoch nicht damit umgehen, wenn sich die Start-Position v_{start} verändert. Dies ist jedoch unumgänglich, wenn während einer autonomen Fahrt der Plan erweitert und aktualisiert werden soll. Hierzu entwickelten Koenig und Likhachev den *D* Lite* ([KL02]), welcher auf den *Lifelong Planning A** aufbaut und ihn so erweitert, dass Fahrzeug-Positionsänderungen verarbeitet werden können. Der *D* Lite*-Algorithmus folgt hierbei der selben Strategie wie der *Focussed Dynamic A** (D^*) von Anthony Stentz [Ste95], arbeitet jedoch algorithmisch etwas anders. Der *D* Lite* ist im Vergleich nicht so komplex wie der D^* und damit einfacher zu verstehen, zu analysieren und zu erweitern.

2.4.1 Vorgehensweise

Die markanteste Änderung des *D* Lite* gegenüber dem LPA^* ist, dass er rückwärts, also vom Zielknoten v_{goal} zum Startknoten v_{start} plant. Dies ist notwendig, da eine Verschiebung von v_{start} ansonsten zur Folge hätte, dass die g -Werte aller bereits expandierten Knoten falsch wären und neu berechnet werden müssten. Wird jedoch rückwärts geplant, so führt dies dazu, dass die geschätzten Heuristiken inkorrekt werden. Dies hat wiederum ausschließlich auf die Reihenfolge der *OpenList* Auswirkungen, welche hierdurch neu sortiert werden muss. Wie in Kapitel 2.3 beschrieben, wird für Kantenänderungen an den äußeren Gebieten weniger Aufwand benötigt, als für Änderungen in der Nähe des Startknotens. Geht man nun davon aus, dass beim Pfadplanen im großen, teils unbekanntem Gelände eher Umweltänderungen in Fahrzeughöhe wahrgenommen werden, so erkennt man einen weiteren Grund für das Rückwärtsplanen.

2.4.2 Der Algorithmus

Abgesehen von der Tatsache, dass der *D* Lite* vom Ziel- zum Startknoten plant (Abb. 2.10 Zeilen 1.3 - 1.4) und daher auch die Heuristik angepasst werden muss, gibt es nur noch eine weitere kleine Veränderung gegenüber dem LPA^* . Um nicht nach jeder Startzustandsänderung die komplette *OpenList* neu zu sortieren, wird eine weitere Variable k_m benötigt. Dieser Wert summiert über die Fahrt hinweg die Heuristik entlang der zurückgelegte Strecke auf, wie in Abbildung 2.10 Zeile 3.6 zu erkennen ist. Dieser Wert wird nun bei Neuberechnung des Schlüsselwerts auf den linken Wert aufaddiert (Zeile 2.1). Wird ein Knoten als potentiell bester Knoten aus der *OpenList* ausgewählt (Abb 2.11 Zeile 4.2), so kann an einer Änderung des Schlüssels nach Neuberechnung erkannt werden, ob dieser aktuell oder veraltet

1. updateNode(v)

```

1 foreach  $v \in \mathcal{V}$  do  $g(v) \leftarrow rhs(v) \leftarrow \infty$ ;
2  $k_m = 0$ ;
3  $rhs(v_{goal}) \leftarrow 0$ ;
4  $Q \leftarrow \{v_{goal}\}$ ;

```

2. key(v)

```

1 return  $\left[ \min(g(v), rhs(v)) + h(v) + k_m ; \min(g(v), rhs(v)) \right]$ ;

```

3. $main_{D^* Lite}(\mathcal{G}, v_{start}, v_{goal} \in \mathcal{V})$

```

1 initialize();
2 while !goalReached() do
3   | computePath();
4   | waitForChanges();
5   | foreach  $v : \{v', v\} \in (\mathcal{E} \cap Changes)$  do updateNode( $v$ );
6   |  $k_m \leftarrow k_m + h(v_{last}, v_{start})$ ;
7 end

```

Abbildung 2.10: $D^* Lite$ - Algorithmus pt. 1

```

4. computePath()
1 while  $\min_{v \in Q} \text{key}_{old}(v) \leq \text{key}(v_{goal})$  or  $\text{rhs}(v_{goal}) \neq g(v_{goal})$  do
2   |  $v \leftarrow \text{argmin}_{v \in Q} \text{key}_{old}(v)$ ;
3   | if  $\text{key}_{old}(v) < \text{key}(v)$  then
4   |   |  $Q \leftarrow Q \cup \{v\}$ ;
5   |   | else if  $g(v) > \text{rhs}(v)$  then
6   |   |   |  $g(v) \leftarrow \text{rhs}(v)$ ;
7   |   | else
8   |   |   |  $g(v) \leftarrow \infty$ ;
9   |   |   |  $\text{updateNode}(v)$ ;
10  |   | foreach  $v_{succ} : \{v, v_{succ}\} \in \mathcal{E}$  do  $\text{updateNode}(v_{succ})$ 
11 end

```

Abbildung 2.11: D^* Lite Algorithmus pt. 2

ist (Zeile 4.3). Ist er nicht aktuell, so wird er mit neuem aktualisierten Schlüssel abermals in die *OpenList* eingereiht (Zeile 4.4). Dieses Vorgehen hat gegenüber der kompletten Neusortierung der *OpenList* nach jeder Zustandsänderung den Vorteil, dass nur relevante Knoten umsortiert werden. Außerdem kann mit einer weiteren Zustandsänderung weitergearbeitet werden, obwohl die alte Umsortierung noch nicht vollständig beendet wurde. Ein Vorteil besteht darin, dass nun Zustandsänderungen mit sehr hoher Frequenz verarbeitet werden können.

2.4.3 Ergebnisse

Wie Koenig und Likhachev zeigten, ist der D^* Lite optimal, optimal effizient und liefert das gleiche Ergebnis wie der D^* von Stentz [KL05]. Im Gegensatz zum normalen A^* muss bei dem D^* Lite bedacht werden, dass die Heuristik nicht nur vorwärts-konsistent, sondern vorwärts-rückwärts-konsistent ist. Hierzu muss für jedes Knoten-Tripel $v, v', v'' \in \mathcal{V}$ folgende Dreiecksungleichung gelten:

$$h(v, v'') \leq h(v, v') + h(v', v'')$$

Betrachtet man das Beispiel des *Lifelong Planning A^** in Abbildung 2.9 (d) so erkennt man bereits, dass es vorkommen kann, dass ältere Teile des Graphen bestehen bleiben, obwohl sie für die aktuelle Planung nicht relevant sind. Dies tritt besonders dann auf, wenn sich das Fahrzeug in Richtung des Ziels bewegt. Hierbei hinterlässt der D^* Lite eine Spur aus „Graphresten“, unter anderem mit inkonsistenten Knoten, welche die OpenList unnötig groß halten. Dies führt zu



Abbildung 2.12: *D* Lite* Beispiel (blau: *OpenList*, braun: Hindernisse, hellblau: zurückgelegter Weg, rot: bearbeitete Knoten)

einem immer größer werdenden Graphen, und somit zu einem wachsendem Speicherbedarf, obwohl ein Großteil des Graphens ohne relevante Informationen und damit unerwünscht ist. Des Weiteren führt die Größe der *OpenList* zu einer unnötigen Verlangsamung. Dies zeigt sich in dem Beispiel in Abbildung 2.12. Hier hat das Fahrzeug nur eine begrenzte Sichtweite (roter Kreis) und fuhr in kleinen Schritten die hellblaue Linie entlang in Richtung Ziel (oben rechts). Die blau markierten Knoten sind hierbei die inkonsistenten Knoten.

2.5 Field D^*

Der A^* ist zwar ein optimaler Graphsuchalgorithmus, das macht ihn jedoch nicht zum optimalen Pfadplanungsalgorithmus, da er lediglich direkte Verbindungen zwischen Nachbarknoten erlaubt und daher nicht im kontinuierlichen Raum arbeitet, wie in Kapitel 2.1.4 bereits beschrieben. Dies äußert sich in einem zweidimensionalen Graphen mit achter-Nachbarschaften darin, dass der gefundene Pfad nur Winkel in 45° Schritten annehmen kann (siehe Abbildung 2.13 (a)). Um wirklich den tatsächlich optimalen Pfad zu finden, muss diese Einschränkung jedoch gebrochen werden. Aus diesem Grund entwickelten Ferguson und Stentz den *Field D^** [FS05]. Dieser ist zwar auch an diskrete Knotenpunkte gebunden, jedoch interpoliert er so zwischen diesen, dass ein optimaler Pfad gefunden werden kann, wie in Abbildung 2.13 (b) zu erkennen ist. Ferguson und Stentz entwickelten den *Field D^** zwar nur für den D^* Lite, jedoch kann er genauso auch für den normalen A^* und jede seiner Erweiterungen genutzt werden. Im Rahmen dieser Arbeit wurde er in der beiliegenden Bibliothek auf dem D^* Lite (Kap. 2.4) und dem LPA^* (Kap. 2.3) angewendet.

2.5.1 Vorgehensweise

Bisher wurde die Position eines Knotens immer als Mittelpunkt der zugehörigen Terrainzelle betrachtet. Für den *Field D^** muss jedoch, statt auf einem solchen Zentrum-basierten Graphen, auf einem Eckpunkt-basierten Graphen geplant werden. In Abbildung 2.14 veranschaulicht dies ein kleiner Beispielgraph. Da die Belegtheitskarte pessimistisch aufgebaut ist, sind auch Kanten zwischen einem Hindernis und einer freien Zelle erlaubt. Außerdem zählt bei Kanten, die genau zwischen zwei verschiedenen Zellen entlang führen, der geringere der beiden Kostenwerte.

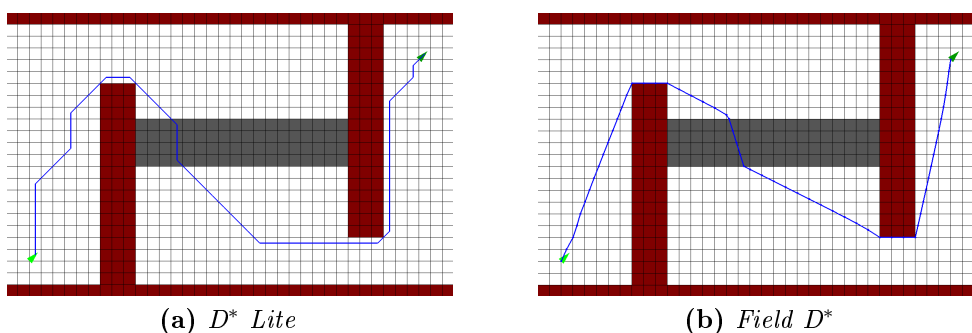


Abbildung 2.13: Beispiel zu Verdeutlichung der graphgebundenen Einschränkung in der Pfadplanung (blau: gefundener Pfad, braun: Hindernisse, grau: schwer befahrbares Gebiet, hellgrün: Startpunkt, dunkelgrün: Ziel)

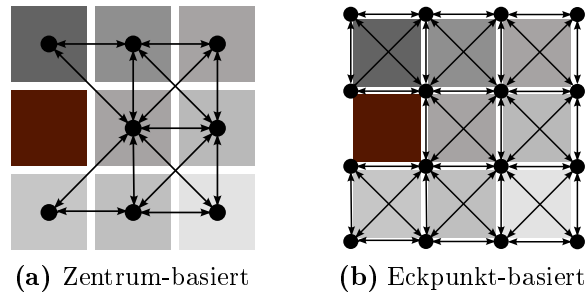


Abbildung 2.14: verschieden Graphypen

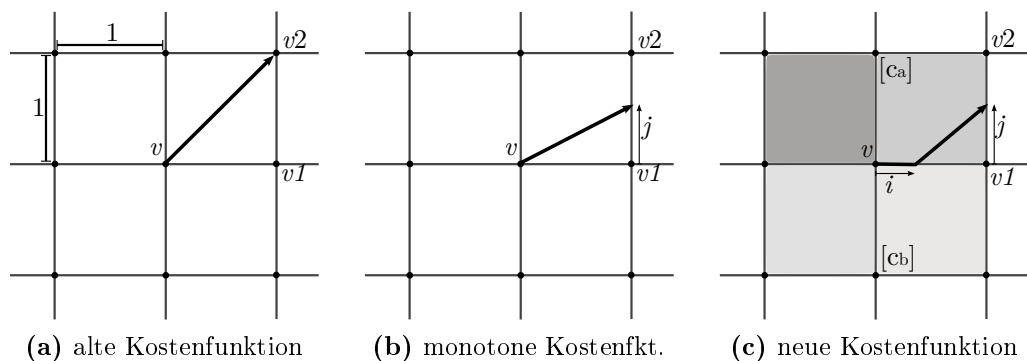


Abbildung 2.15: verschiedene Kostenfunktionen

Bisher wurde in dieser Arbeit immer davon ausgegangen, dass der Pfad nur von Knotenpunkt zu Knotenpunkt springen kann. Aufgrund dessen wurden auch bei der Kostenberechnung eines Knoten lediglich die diskreten Nachbarknoten als potentielle Vorgänger in Betracht gezogen. Der *Field D**-Algorithmus rechnet hingegen auch damit, dass der Pfad zwischen den Knotenpunkten verläuft, ohne diese zu „treffen“. Deswegen ist auch jede Koordinate zwischen den Knotenpunkten ein potentieller Vorgängerpunkt und wird bei der Kostenberechnung berücksichtigt. Hierfür interpoliert der *Field D** die *g*-Werte der benachbarten Knoten auf den Verbindungsstrecken. Wie dies im Detail geschieht wird im folgenden Kapitel 2.5.2 dargelegt. Für das bessere Verständnis, sind in den Abbildungen 2.17 und 2.19 zu den Knoten im Graph die jeweils ermittelten Vorgängerrichtungen (später *targets* genannt) eingezeichnet. Dieses Vorgehen ist zwar mathematisch nicht vollkommen korrekt, führt jedoch zu einer ausreichend guten Annäherung. Diese wird umso genauer, je weiter der Knoten vom Startzustand entfernt ist, wodurch sich die neue Kostenfunktion besonders für den rückwärtsplanenden *D* Lite* eignet.

2.5.2 Algorithmus

Wie bereits beschrieben werden zum Berechnen der g -Kosten eines Knoten nun nicht nur die Kosten der benachbarten Knoten beachtet, sondern diese Kosten werden zwischen den zugehörigen Positionen interpoliert. Vorher hingegen wurden die Kosten eines Knotens als Minimum der g -Werte der Nachbarknoten zusammen mit den Pfadkosten zwischen diesen angenommen:

$$g(v) \leftarrow \min_{\{v_{pred}, v\} \in \mathcal{E}} \left[g(v_{pred}) + c(v_{pred}, v) \right] \quad (2.2)$$

Folglich ergeben sich Vorgänger-Verbindungen wie sie in Abbildung 2.15 (a) zu sehen sind. Interpoliert man jedoch die g -Werte der Nachbarknoten in einer monotonen Umgebung, so ergibt sich eine neue Kostenfunktion:

$$g(v) \leftarrow \min_{v_1, v_2, j} \left[c_a \sqrt{1^2 + j^2} + jg(v_2) + (1-j)g(v_1) \right] \quad (2.3)$$

Hierbei sind v_1 und v_2 zwei konsekutive Nachbarknoten, $j \in [0, 1]$ der Interpolationswert und c_a der Kostenwert der entsprechenden Zelle. Eine solche Situation ist in Abbildung 2.15 (b) aufgeführt. Bedenkt man nun noch, dass die an v und v_1 aber nicht an v_2 anliegende Zelle einen geringeren Kostenwert $c_b < c_a$ hat so ergibt sich die endgültige Kostenfunktion:

$$g(v) \leftarrow \min_{v_1, v_2, i, j} \left[c_b \cdot i + c_a \sqrt{(1-i)^2 + j^2} + jg(v_2) + (1-j)g(v_1) \right] \quad (2.4)$$

Dabei ist v_1 immer der direkte und v_2 der diagonale Nachbar. Verbildlicht wurde diese Situation in Abbildung 2.15 (c). Um diesen Wert zu bestimmen, gilt es nun die Interpolationswerte $i, j \in [0, 1]$ zu ermitteln:

$$(i, j) \leftarrow \operatorname{argmin}_{i, j} \left[c_b \cdot i + c_a \sqrt{(1-i)^2 + j^2} + j \cdot g(v_2) + (1-j)g(v_1) \right] \quad (2.5)$$

Diese Interpolationswerte werden nicht nur zur Kostenberechnung genutzt, sondern auch zum späteren *backtracen* (Kapitel 2.5.3), weshalb sie gespeichert werden sollten. Ferguson und Stentz zeigen, dass entweder $i = 0$ ist, oder $j = 1$, und beweisen dies in [FS]. Dadurch genügt es die Gleichung 2.3 nach j abzuleiten und 0 zu setzen, um damit je nach Situation die Kosten zu berechnen. Für j ergibt sich dabei:

$$j \leftarrow \sqrt{\frac{(g(v_1) - g(v_2))^2}{c_a^2 - (g(v_1) - g(v_2))^2}} \quad (2.6)$$

Die vollständige Herleitung hierfür findet sich in [FS05]. Für die komplette interpolierende Kostenberechnung ergibt sich der Algorithmus 2.16, welcher zwischen sechs verschiedenen Fällen unterscheidet und damit die Rechnung minimal klein hält.

$c(v, v_1, v_2, c_a, c_b)$	
1	$i \leftarrow j \leftarrow 0;$
2	if $\min(c_a, c_b) = \infty$ then
3	$costs \leftarrow \infty;$ ▷ Knoten unbekannt
4	else if $g(v_1) \leq g(v_2)$ then
5	$costs \leftarrow \min(c_a, c_b) + g(v_1);$ ▷ direkter Nachbar v_1
6	else
7	$f \leftarrow g(v_1) - g(v_2);$
8	if $f \leq c_b$ then
9	if $c_a \leq f$ then
10	$costs \leftarrow c_a\sqrt{2} + g(v_2);$ ▷ direkter Nachbar v_2
11	else
12	$j \leftarrow \min(\frac{f}{\sqrt{c_a^2 - f^2}}, 1.0);$ ▷ j-Interpolation
13	$costs \leftarrow c_a\sqrt{1 + j^2} + f(1 - j) + g(v_2);$
14	end
15	else
16	if $c_a \leq c_b$ then
17	$costs \leftarrow c_a\sqrt{2} + g(v_2);$ ▷ direkter Nachbar v_2
18	else
19	$i \leftarrow 1 - \min(\frac{c_b}{\sqrt{c_a^2 - c_b^2}}, 1.0);$ ▷ i-Interpolation
20	$costs \leftarrow c_a\sqrt{1 + (1 - i)^2} + c_b \cdot i + g(v_2);$
21	end
22	end
23	end
24	return $(costs, i, j);$

Abbildung 2.16: Interpolierende Kostenberechnung von v mit: v_1 (direkter Nachbar), v_2 (diagonaler Nachbar), c_a (Zellenkosten der Zelle zwischen v, v_1, v_2), c_b (Zellenkosten der Zelle zwischen v, v_1 aber nicht v_2)

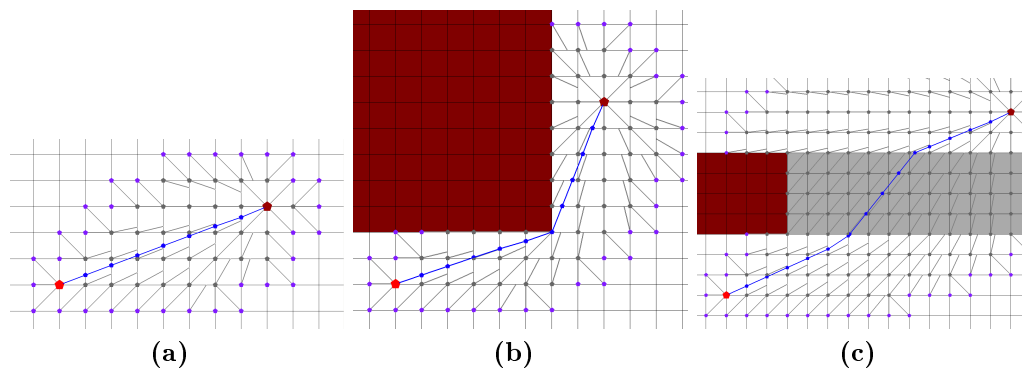


Abbildung 2.17: Beispiele für Field D* backtracing (blau:Zielpfad, braun:Hindernisse, hellgrau:schwer befahrbares Gebiet, dunkelgrau:Knoten im Graph mit zugehörigen Vorgängerrichtungen, lila:Knoten der *OpenList*)

2.5.3 Field Backtracing

Im *Field D** hat nun nicht mehr jeder Knoten einen eindeutigen Vorgängerknoten, welcher das Backtracing zuvor sehr vereinfacht hat. Es kann nun nicht mehr einfach von Knotenpunkt zu Knotenpunkt gesprungen werden, um den Zielpfad zu ermitteln, da der gesuchte Pfad meistens zwischen den Knoten hindurch führt (siehe Abbildungen 2.17). In den Veröffentlichungen von Ferguson und Stentz, wie zum Beispiel [FS05], wird genau erklärt wie das Expandieren des *Field D** funktioniert, jedoch wird nicht darauf eingegangen, wie das Zurückverfolgen des Pfades vom gefundenen Zielknoten bis zum Start durchzuführen ist. Aus diesem Grund war es erforderlich ein eigenes *Backtracing* zu entwickeln, welches im Folgenden vorgestellt wird.

Ähnlich wie bei dem „normalen“ Backtracing wird auch hier, am Zielknoten beginnend, von Punkt zu Punkt gesprungen, wobei die Punkte nicht zwangsweise genau auf den Knoten im Graphen liegen müssen, wie in Abbildung 2.17 zu sehen ist. Die Punkte werden jedoch so gewählt, dass sie immer genau auf einer Verbindungslinie zwischen zwei Knoten v_a und v_b liegen. In Abbildung 2.18 (a) ist dies der Punkt \mathbf{p} auf dem Zielpfad, mit den beiden Positionen \mathbf{p}_a und \mathbf{p}_b der Nachbarknoten v_a und v_b . Diese Knoten haben jeweils ein *Elterntarget* \mathbf{t}_a und \mathbf{t}_b , welche jeweils durch das Nachbarpaar $[\mathbf{p}_1, \mathbf{p}_2]$ und dem Interpolationswert j definiert werden (siehe Kapitel 2.5.1):

$$\mathbf{t}_b = \mathbf{p}_1 + j \cdot (\mathbf{p}_2 - \mathbf{p}_1) \quad (2.7)$$

$$\mathbf{s}_b = \mathbf{t}_b - \mathbf{p}_b \quad (2.8)$$

$$\text{analog für } \mathbf{t}_a \text{ und } \mathbf{s}_a \quad (2.9)$$

Gesucht ist das *target* \mathbf{t}_p für den Punkt \mathbf{p} , um im nächsten Iterationsschritt den

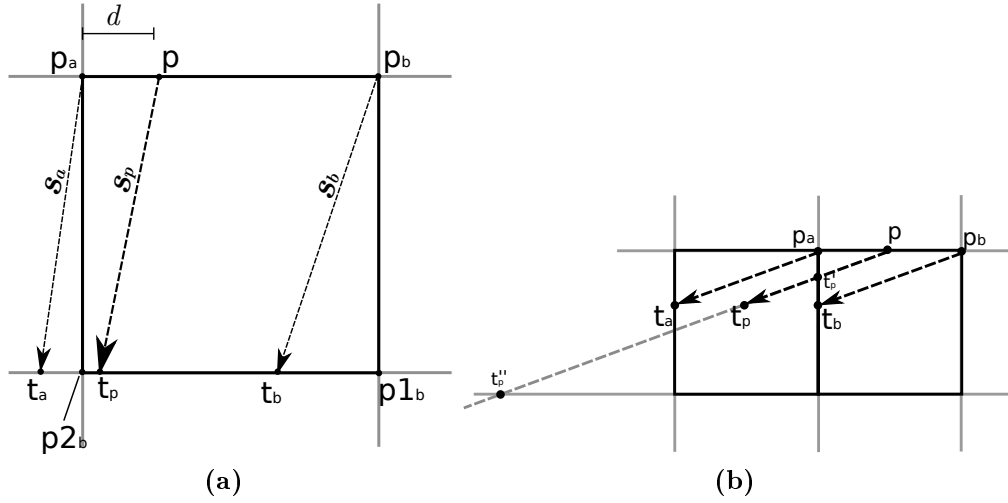


Abbildung 2.18: Field D* backtracing

Pfad von diesem aus weiter zu verfolgen, bis der Startknoten erreicht wird. Durch die Normierung des Grids liegen p_a und p_b genau um 1.0 auseinander, wodurch sich für die Interpolation folgendes ergibt:

$$d = \|p - p_a\| \quad (2.10)$$

$$s_p = (1 - d) \cdot s_a + d \cdot s_b \quad (2.11)$$

Die oben beschriebene Berechnung kann dazu führen, dass der Punkt

$$t_p = p + s_p \quad (2.12)$$

nicht genau auf einer Kante zwischen zwei Knoten liegt, wie es beispielsweise in Abbildung 2.18 (b) dargestellt wird. Um dem entgegen zu wirken, wird der Vektor s_p einfach durch $s_{p.y}$ bzw. $s_{p.x}$ geteilt, wodurch der Vektor auf dieser Dimension normiert wird und t_p'' damit wieder auf einer Kante liegt. Es besteht dabei jedoch die Möglichkeit, dass mit einem Schritt sehr große Distanzen überschritten werden. Das Vorgehen kann dann zu falschen Pfaden führen, wenn dabei Zellen mit verschiedenen Zellenkosten überschritten werden, wie beispielsweise Abbildung 2.17 (c) zeigt. Dieser Fehler rührt daher, dass die beiden Nachbar v_a und v_b lediglich über die zwei direkt anliegenden, in Abbildung 2.18 (b) dick umrahmten Felder informiert sind. Daher wird in diesem Fall der Vektor s_p an der nächsten Kante abgeschnitten. In Abbildung 2.18 (b) geschieht dies beispielsweise durch

$$s_p = s_p \cdot \frac{(p.x - p_a.x)}{s_{p.x}} \quad (2.13)$$

Abschließend kann

$$t'_p = p + s_p \quad (2.14)$$

bestimmt werden, sowie die neuen Nachbarn p'_a und p'_b , um den nächsten Iterationsschritt mit $p' = t_p$ starten zu können. Der vollständige Algorithmus findet sich im Anhang A.1 bis A.3.

2.5.4 Das Zwei-Eltern Dilemma

Glaubt man den Veröffentlichungen von Ferguson und Stentz [FS05], so ist der *Field D^** -Algorithmus funktionsfähig, wenn die neue Kostenfunktion in den *D* Lite* eingebunden wird. Lässt man diesen Algorithmus jedoch so laufen, so fällt sehr schnell ein großes Problem auf, welches es zu beheben gilt.

Aus der neuen Kostenfunktion ergibt sich, dass nun jeder Knoten nicht mehr einfach nur einen sondern zwei Vaterknoten referenziert, zwischen denen die Kosten interpoliert werden. Durch den Heuristik-gegebenen Drang in eine Richtung, kann es bei der Graphausbreitung jedoch passieren, dass ein Knoten expandiert wird, von dem nur einer der beiden am besten geeigneten Elternknoten bereits bearbeitet wurde. Dies veranschaulicht beispielsweise der rot markierte Knoten v in Abbildung 2.19, von dessen idealen Elternknoten v_1 und v_2 (rote Rahmen) lediglich einer bereits erreicht wurde. Dies bedeutet, dass der neue Knoten nun lediglich diesen einen Knoten als direkten Vaterknoten annimmt und über diesen seinen g -Wert berechnet, welcher dadurch größer ist als der optimale Wert. Wird im weiteren Verlaufe des Planens nun irgendwann der zweite Elternknoten erreicht und erhält einen g -Wert $< \infty$, so gerät der Knoten v in einen inkonsistenten Zustand und kommt in die *OpenList*. Infolgedessen wird dieser kurz darauf aktualisiert, woraufhin die Knoten, welche direkt von v abhängen, inkonsistent werden. Es werden also nach und nach alle davon abhängigen Knoten optimiert. Dieses Problem setzt sich rekursiv fort und führt nicht nur an den Randregionen zu Veränderungen und somit zu Rechenaufwand, sondern zieht sich durch große Teile des Graphen. Dies geht soweit, dass der *Field D^** irgendwann fast ausschließlich mit Optimierungen beschäftigt ist und sich kaum noch ausbreitet, bis er irgendwann fast zum Stillstand kommt.

Ebenso liegt der in der Abbildung zu erkennende Knick in der Nähe des Ziels daran, dass die zielnahen Knoten, aufgrund der Heuristik, expandiert wurden, bevor ihre Elternknoten vollständig berechnet waren.

Vollständig lösen lässt sich das Problem nur dann, wenn keine Heuristik genutzt wird und sich der Algorithmus wie der Dijkstra ausbreitet. Dies führt jedoch zu unerwünscht großen Graphen. Besonders dann, wenn in einer teils unbekanntenen Umgebung geplant wird, führt der rückwärts planende *D* Lite* zu einem sehr

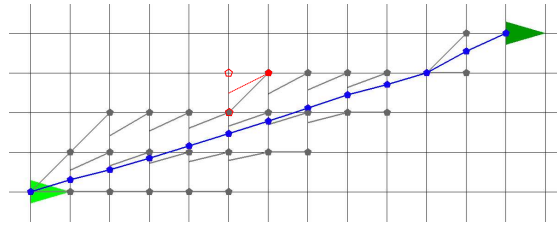


Abbildung 2.19: Field - LPA^* Beispiel (hellgrün:Startpose, dunkelgrün:Zielpose, blau:Zielpfad, grau:Knoten im Graph mit zugehörigen Vorgängerrichtungen, rot:Verdeutlichung des Zwei-Eltern Dilemmas)

großen kugelförmigen Graphen. In der Praxis genügt es jedoch auch schon, die Heuristik geringer zu gewichten. In Umgebungen mit 200×200 Zellen genügt circa ein Gewicht von $w_h = 0.9$, um einen flüssigen Verlauf zu erlangen. Je größer das Terrain desto geringer muss jedoch das Gewicht sein.

Eine weitere Möglichkeit ist es, eine gewisse Toleranz einzuführen, welche man zwischen dem g - und dem rhs -Wert zulässt, bevor ein Knoten wegen Ungleichheit dieser beiden Werte in die *OpenList* eingereiht wird. Wie groß die Auswirkungen der Heuristikgewichtung und der Toleranz genau sind, ist in Kapitel 4.4.2 nachzulesen.

Eine dritte Variante besteht darin, dem *Field* D^* eine Heuristik zu geben, welche parallel einen normalen D^* Lite laufen lässt und genau dann eine 0 liefert, wenn der D^* diese Zelle bereits erreicht hat und sonst ∞ :

$$h_D(v) = \begin{cases} 0 & \text{wenn } g(\varphi(v)) \leq \infty, \\ \infty & \text{sonst.} \end{cases} \quad \text{mit } \varphi : \mathcal{G}_{FieldD^*} \rightarrow \mathcal{G}_{D^*Lite} \quad (2.15)$$

Dies ist im eigentlichen Sinne keine gültige Heuristik, da sie nicht immer unterschätzt. Jedoch kann sie in diesem Fall trotzdem genutzt werden, da die Ausmaße des Graphen \mathcal{G}_{FieldD^*} des *Field* D^* immer vollständig in denen des Graphen \mathcal{G}_{D^*Lite} des D^* Lite enthalten sind. Somit breitet sich der Graph des *Field* D^* Dijkstra-mäßig ohne teure vielfach-optimierungs-Rechnungen in den Grenzen des D^* Lite - Graphen aus. Diese Methode lässt sich auch so verstehen, dass der D^* Lite normal plant, bis er das Ziel erreicht hat, und dann anschließende seine g -Werte mittels Interpolation optimiert. Auch diese Methode wird, von der in dieser Arbeit beigelegten Planungsbibliothek, unterstützt.

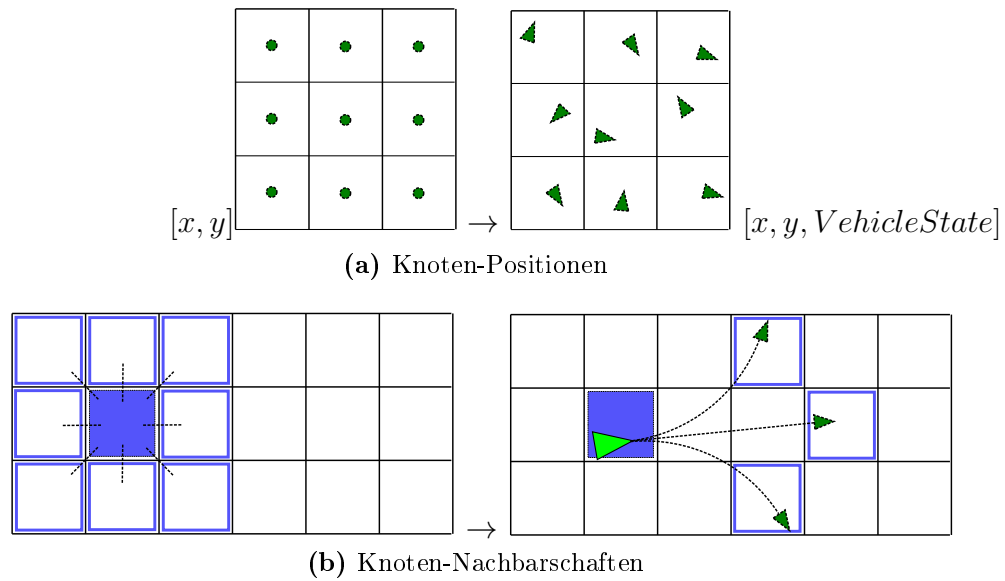
2.6 Hybrider A^*

Der *hybrid A^** von Dolgov und Thrun [DTMD10] stellt trotz der graphgegebenen Gebundenheit an diskrete Knoten des A^* , einen weiteren Versuch dar, im Kontinuierlichen zu planen. Im Gegensatz zum *Field D^** plant er jedoch mit Berücksichtigung der Einschränkungen nicht-holonomer Fahrzeuge. Dolgov und Thrun entwickelten hierzu einen Algorithmus, welcher aus zwei Teilen besteht. Im ersten Teil wird ein ausführbarer gültiger Pfad gefunden, den es im zweiten Teil zu optimieren gilt. Der zweite Teil setzt dabei eine Pfadglättung mittels konjugierter Gradienten (*Conjugate Gradient Descent*) um, welche sowohl den Abstand zu Hindernissen als auch die Pfadkrümmung berücksichtigt. Im Rahmen dieser Arbeit wird jedoch nur der erste Teil näher betrachtet.

2.6.1 Vorgehensweise

Beim normalen A^* wird jedem Knoten ein fest definierter Zustand zugewiesen, wie zum Beispiel der Mittelpunkt oder die Eckpunkte der zugehörigen Terrainzelle. Der *hybrid A^** weist jedem Knoten jedoch einen Fahrzeugzustand im kontinuierlichen Raum zu, welcher frei innerhalb der Zellengrenzen liegen kann. Dies ist in Abbildung 2.20 (a) zu erkennen. Es sollte hierbei in mindestens drei Dimensionen (x,y - Position, Winkel) geplant werden, damit mehr Zustände pro Terrainzelle zugelassen werden und somit ein besserer Pfad gefunden werden kann. Hierbei ist zu beachten, dass die dritte Winkel-Dimension ebenfalls in diskrete Abschnitte unterteilt wird, ähnlich den x- und y-Werten der Position, innerhalb derer sich der kontinuierliche Zustand befinden kann.

Die einzige weitere Veränderung gegenüber dem normalen A^* ist nun, dass sich Nachfolger-Beziehungen nicht über eine Nachbarschaft im Grid definieren, wie in Abbildung 2.20 (b) zu erkennen ist. Stattdessen werden auf den Fahrzeugzustand eines Knotens vordefinierte Pfad-Stücke addiert. Die Zellen in denen die so erreichten neuen Zustände fallen, gelten als Nachfolgeknoten. Des Weiteren werden auf diese Weise die Zustände der neuen Zellen definiert. Die vordefinierten Pfadstücke werden dabei so gewählt, dass sie von dem Fahrzeug unter Berücksichtigung der nicht-holonomen Einschränkungen ausgeführt werden können. Da diese Teilstücke länger sein sollten als die Breite der Zellen, erlaubt der *hybrid A^** mit weniger Schritten ans Ziel zu gelangen als der normale A^* . Je größer die Anzahl der vordefinierten Pfadstücke, desto höher sollte die Auflösung der dritten Dimension sein. Dies ist notwendig, denn sobald mehrere Zustände in einer Zelle enden, gewinnt nur der Beste und die anderen werden gelöscht. Je mehr Pfadstücke definiert sind und je höher die Auflösung ist, desto größer wird der Graph und die Suche dauert dementsprechend länger. Jedoch wird der gefundene Pfad näher

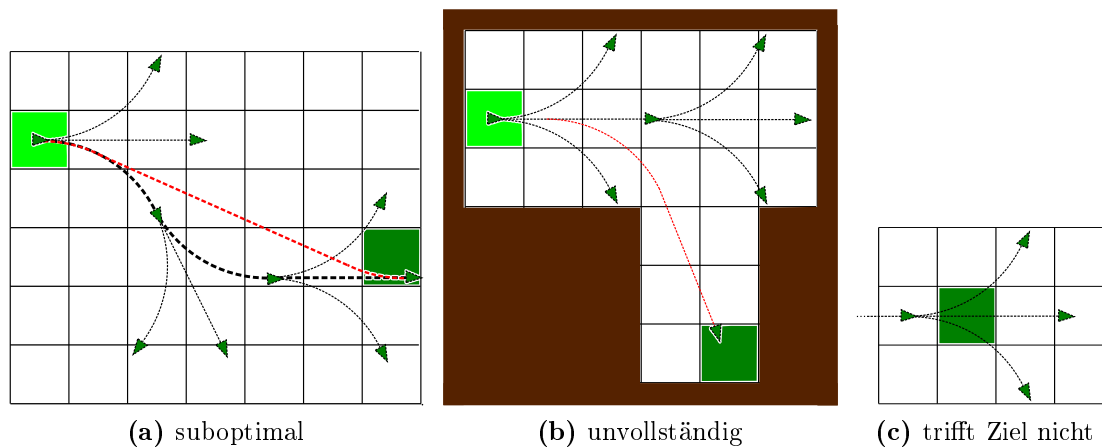
Abbildung 2.20: Vom A^* zum *hybrid* A^*

am Optimum liegen und die Wahrscheinlichkeit, dass ein gültiger existierender Pfad nicht gefunden wird sinkt.

2.6.2 Nachteile des hybriden A^*

Optimalität Der *hybrid* A^* plant zwar nun im Gegensatz zum A^* im Kontinuierlichen, jedoch hat er trotzdem ähnliche Einschränkungen in der Optimalität. Dadurch dass nur bestimmte vordefinierte Pfadstücke zugelassen werden, kann es passieren, dass der gefundene Pfad nicht der kürzeste ist, beispielhaft in Abbildung 2.21 (a) dargestellt. Hier befindet sich der Startzustand in der hellgrün markierten Zelle und das Ziel ist die dunkelgrüne Zelle. Der *hybrid* A^* findet durch seine wenigen festen Manöver lediglich den schwarzen Pfad zu Ziel. Der rote Pfad ist jedoch deutlich kürzer und hält ebenfalls die selben Lenkeinschränkungen ein wie der schwarze Pfad. Allgemein lässt sich daraus Schlussfolgern, dass der Pfad umso optimaler wird, je mehr verschiedene Manöver angewandt werden. Zusätzlich führt dies jedoch zu einem immer größer werdenden Graphen und bleibt trotzdem nur eine Annäherung. Aus diesem Grund sollte der gefundene Pfad nach Ermittlung möglichst noch geglättet werden, um hierdurch näher an das Optimum zu kommen.

Vollständigkeit Die Länge der einzelnen Manöver ist beim *hybrid* A^* frei wählbar, jedoch sollte sie länger sein als $\sqrt{2}$ -fache der Zellbreite. Ist dies nicht gegeben, so kann der Nachfolgezustand nach dem geradeaus Fahren in die selbe Zelle fallen wie der Vorgängerknoten und könnte nicht in den Graphen eingesetzt werden. Dies

Abbildung 2.21: Nachteile des hybriden A^*

birgt die Gefahr, dass der *hybrid* A^* nicht durch Engstellen plant, obwohl die Durchfahrt eigentlich möglich wäre. Ein Beispiel hierzu ist in Abbildung 2.21 (b) zu sehen. Die gültige rote Lösung wird vom Algorithmus nicht gefunden, da dieser lediglich auf die vordefinierten Manöver zurückgreift, mit welchen der Lösungspfad nicht nachgebildet werden kann. Schlussfolgernd bedeutet es, dass der *hybrid* A^* **nicht vollständig** ist, was für einen Planungsalgorithmus nicht akzeptabel ist. Auch hier gilt, je mehr verschiedene vordefinierte Manöver genutzt werden, desto höher ist die Wahrscheinlichkeit, dass der Pfad gefunden wird. Jedoch darf nicht vergessen werden, dass kleinere Lenkradius-Abstufungen auch eine höhere Auflösung in der dritten Dimension verlangen, wodurch die Suche deutlich verlangsamt wird.

Ziel erreichen Aus dem gleichen Grund, der den *hybrid* A^* nicht immer durch Engstellen finden lässt, kann es geschehen, dass er nicht direkt das Ziel findet. Kommt ein Zustand sehr nahe an die Zielzustände ran, so kann es passieren, dass die Manöver über das Ziel hinaus steuern und dieses nicht treffen, obwohl dies eigentlich möglich wäre. Ein Beispiel ist in Abbildung 2.21 (c) zu sehen ist. In diesem Fall würde der Algorithmus so lange weiter planen bis er das Ziel über andere Knoten erreicht, was aber dazu führt, dass ein Pfad mit höheren Kosten als Lösung bestimmt wird. Im schlimmsten Fall kann es sogar sein, dass gar kein gültiger Plan gefunden wird. Dieses Problem ist jedoch leicht in den Griff zu bekommen, indem man einen Knoten, welcher in eine gewissen Nähe zum Zielzustand fällt, direkt mit dem Ziel verbindet und diesen Pfad auf Ausführbarkeit testet. Ist das Pfadstück gültig, so kann es, genauso wie die vordefinierten Pfadstücke, in den Graphen integriert werden. Im Rahmen dieser Arbeit wurde dies mittels

Splines gelöst, welche auf Kollisionen und Lenkwinkelüberschreitungen getestet werden.

2.6.3 hybrider D^*

Im Rahmen dieser Masterarbeit wurde versucht, den *hybrid A^** mit dem *D^* Lite* zu verbinden. Der hieraus resultierende *hybrid D^** sollte sowohl die *lifelong-* und *replanning-*Eigenschaften des *D^* Lite* besitzen, als auch das Expandieren mittels vordefinierten Pfadstücken, um Pfade zu erhalten, welche alle nicht-holonomen Einschränkungen einhalten. Dieser Algorithmus erwies sich jedoch als nicht annähernd zufriedenstellend, sodass er nicht in voller Reife umgesetzt wurde. Die bisherigen Ergebnisse sind zwar in der beigefügten Bibliothek zu finden, jedoch wird davon abgeraten, diese unverändert zu nutzen.

Das hauptsächliche Problem des *hybrid D^** liegt dabei in der Algorithmik. Im Gegensatz zum normalen A^* , auf welchem der *hybrid A^** basiert, benötigt der *hybrid D^** nicht nur eine *Nachfolger-*Funktion sondern auch eine *Vorgänger-*Funktion, welche alle potentiellen Vorgängerknoten im Graphen ermittelt. In der normalen Achter-Nachbarschaft ist beides mit der selben, schnellen Funktion zu lösen. Ist die Nachbarschaft jedoch über die diskreten Pfadstücke definiert, so erweist sich die Vorgänger-Funktion als komplexer. Hier müssen alle Knoten gefunden werden, welche bereits einen zugewiesenen Fahrzeugzustand haben, von welchem aus einer der Pfadstücke in die gesuchte Zelle 'trifft'. Selbst mit der Optimierung, dass nur in dem Umfeld gesucht wird, welches durch das rückwärts-Anwenden der Pfadstücke auf den Zellenmittelpunkt erreicht wird, ist die Funktion bereits im Dreidimensionalen sehr langsam.

Hinzu kommt, dass sich zwei Knoten nicht einfach neu verknüpfen lassen, da sich hierdurch der zugeordnete Fahrzeugzustand der hinteren Zelle ändert, und somit neu überprüft werden muss, ob seine Nachfolgeknoten noch erreichbar sind. Da sich dies rekursiv auf alle Nachfolgeknoten weiterführen würde, wurde hier darauf verzichtet, denn Fahrzeugzustand des Knotens zu ändern, stattdessen muss jedoch explizit geprüft werden, ob eine Trajektorie zwischen den beiden Zuständen ausführbar ist.

Aus diesen Gründen, war der *hybrid D^** so langsam, dass versucht wurde Zeit an anderen Stellen einzusparen. Beispielsweise bei dem Hinzufügen neuer Hindernisse. Hier wird der Abbau des Graphen nicht über die *OpenList* geregelt, wie es bei dem *D^* Lite* üblich ist, sondern es wurden direkt alle Nachfolgeknoten gelöscht, ähnlich wie es im *LP-RRT** (Kapitel 2.9) umgesetzt wurde. Hier ist darauf zu achten, dass Knoten an den Grenzgebieten des gelöschten Bereichs wieder in die *OpenList* eingereiht werden, um das Weiterplanen zu ermöglichen.

Trotz diesen Optimierungen ist der *hybrid D^** zu langsam um mit anderen Algorithmen mithalten zu können. Des Weiteren zeigen sich auch hier die gleichen Nachteile,

welche auch schon im *hybrid A^** festgestellt wurden (Kapitel 2.6.2). Außerdem erinnerten die Optimierungsversuche immer stärker an die Arbeitsweise des *LP-RRT** (Kapitel 2.9), welche jedoch im Gegensatz zum *hybrid D^** keine so großen Nachteile mit sich führt. Folglich verlor der *hybrid D^** für die Arbeit an Relevanz.

2.7 Rapidly-exploring Random Tree

Der *Rapidly-exploring Random Tree* (*RRT*) [LaV98] von Steven M. LaValle, Author des Buches *Planning Algorithms* [LaV06], ist ein probabilistischer Ansatz zum Lösen des Pfadplanungsproblems. Er baut ausgehend von dem Startzustand v_{start} einen Baum \mathcal{G} auf, welcher sich solange durch eine zufallsbestimmte Heuristik im Zustandsraum ausbreitet, bis ein Zielzustand erreicht wurde. Schließlich bricht der Algorithmus ab und der gesuchte Pfad kann, beginnend vom gefundenen Zielzustand aus, zurückverfolgt werden. Ähnlich wie der A^* baut auch der *RRT* einen Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ auf, welcher jedoch eine Baumstruktur besitzt. Im Gegensatz zum A^* sind die Knoten $v \in \mathcal{V}$ jedoch nicht an das Terrain-Grid gebunden und können beliebig im kontinuierlichen Raum \mathbb{R}^n liegen. Von daher kann auch die Schrittweite der Kanten beliebig gewählt werden, was gegebenenfalls einen Geschwindigkeitsvorteil bringen kann. Ein weiterer Unterschied zum A^* ist der, dass der *RRT* eine *uninformierte Suche* darstellt, er somit also ohne zielgebundene Heuristik arbeitet.

Der *Rapidly-exploring Random Tree* ist ein *single query* Pfadplaner (oder *single-shot* Planer), was bedeutet, dass der Algorithmus lediglich für ein spezielles Problem ausgeführt wird und er nicht, wie zum Beispiel die probabilistische road map (PRM) [KSLO96], einmal ausgeführt wird und dann mehrere Abfragen beantworten kann.

2.7.1 Algorithmus

In Abbildung 2.22 ist der RRT-Algorithmus beschrieben. Er beginnt mit einem leeren Baum \mathcal{G} , welchem die Startknoten v_{start} als Wurzel übergeben wird (Abb. 2.22 Zeile 1.1). Anschließend werden so lange neue Knoten, welche Konfigurationen

```

1.  $\mathcal{G} = (\mathcal{V}, \mathcal{E}) \leftarrow \text{main}_{RRT}(v_{start})$ 
   1  $G \leftarrow \text{insertNode}(\emptyset, v_{start});$ 
   2 for  $n = 1$  to  $N_{max}$ 
   3 do
   4    $v_{rand} \leftarrow \text{sample}(n);$ 
   5    $\text{extend}(\mathcal{G}, v_{rand});$ 
   6 end
   7 return  $\mathcal{G}$ 

```

Abbildung 2.22: Gerüst des Rapidly-exploring Random Tree Algorithmus

```

2. extend( $\mathcal{G}, v_{rand}$ )
  1  $v_{nearest} \leftarrow \text{nearest}(\mathcal{G}, v_{rand});$ 
  2  $(v_{new}, u_{new}, \mathbf{x}_{new}) \leftarrow \text{steer}(v_{nearest}, v_{rand}, \Delta t);$ 
  3 if !isBlocked( $\mathbf{x}_{new}$ ) then
  4   |  $\mathcal{G} \leftarrow \text{insertNode}(v_{nearest}, v_{new}, \mathcal{G});$ 
  5 end

```

Abbildung 2.23: Die extend-Funktion

des Fahrzeuges entsprechen, an diesen Baum hinzugefügt (Zeilen 1.3-1.4), bis einer dieser Knoten Teil der Zielmenge ist. Sollte allerdings nach N_{max} Knoten immer noch kein Zielzustand gefunden worden sein, so wird davon ausgegangen, dass kein ausführbarer Pfad von der Startkonfiguration bis zu einer Zielkonfiguration existiert, und der Algorithmus bricht ab (Zeile 1.2).

Zum Erstellen eines neuen Knotens, wird zunächst ein zufälliger Knoten v_{rand} gewählt (Abb. 2.22 Zeile 1.3, sowie Abb. 2.24(b) roter Punkt). Hierzu wird ein Punkt gleichverteilt aus dem kompletten Konfigurationsraum gezogen. Nun wächst der bereits bestehende Baum in Richtung dieser Zufallskonfiguration (Abb. 2.22 Zeile 4, sowie Abb. 2.23). Hierfür wird aus dem bereits bestehenden Graphen der nächstliegende Nachbarknoten $v_{nearest}$ gesucht (Abb. 2.23 Zeile 2.1, sowie Abb. 2.24(b) gestrichelte Linie). Als Distanzfunktion zum Ermitteln des nächsten Knotens, sollte hier die Kostenfunktion der minimalen Trajektorie gewählt werden, ohne auf die Kollisionsvermeidung zu achten. Annähernd kann aber auch beispielsweise die euklidische Distanz oder die Dubinspfadlänge herangezogen werden.

Ist $v_{nearest}$ nun bekannt, so wird von diesem aus eine gewisse Strecke (Δt) in Richtung v_{rand} gesteuert (Abb. 2.23 Zeile 2.2), wobei Δt lediglich als obere Schranke zu verstehen ist. Da diese steering-Funktion durch den *RRT* selber nicht weiter eingeschränkt wird, ist es möglich den *RRT* für alle möglichen kinematischen Modelle anzuwenden. Die steer-Funktion kann, neben dem nun neu erreichten Konfigurationspunkt v_{new} (Abb. 2.24(c) grüner Punkt), sowohl die abzufahrende Trajektorie \mathbf{x}_{new} von $v_{nearest}$ bis v_{new} als auch die Kontrollbefehle u_{new} , welche für die Ausführung dieser Strecke benötigt werden, zurückgeben. Falls diese neue Strecke \mathbf{x}_{new} hindernisfrei ist (Abb. 2.23 Zeile 2.3), kann v_{new} als neuer Punkt in den Baum eingetragen werden, mit einer Kante von $v_{nearest}$ zu v_{new} (Abb. 2.23 Zeile 2.4, sowie Abb. 2.24(d)). Genauer zur Kollisionsvermeidung findet sich in [LaV11].

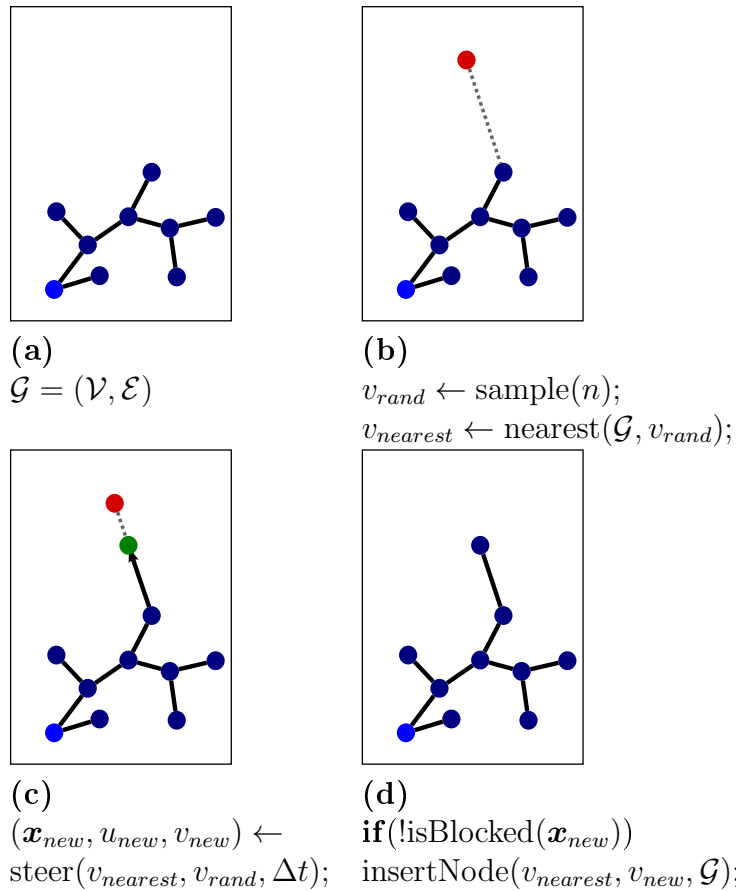


Abbildung 2.24: Hinzufügen eines neuen Knotens in den Graphen

2.7.2 Rapidly Exploring

Der folgende Abschnitt möchte darlegen, warum sich dieser Algorithmus „*rapidly-exploring*“, „sich schnell ausbreitend“ nennt. Hierzu ist es sinnvoll das Voronoi-Diagramm eines *RRT*-Graphen zu betrachten. Abbildung 2.25 zeigt ein solches Voronoi-Diagramm. Die roten Punkte und die schwarzen Linien stellen den *RRT*-Graphen mit seinen Knoten und Kanten dar. Die blauen Linien zeigen die Voronoi-Regionen der einzelnen Knoten. Durch die Definition der Voronoi-Regionen ist vorgegeben, dass jeder Punkt im Konfigurationsraum genau denjenigen Knoten im Baum als nächsten Knoten hat, in dessen Voronoi-Region er sich befindet. Wird bei dem *Sampling* nun gleichverteilt aus dem kompletten Konfigurationsraum gezogen und der nächstliegende Knoten erweitert, so ist die Wahrscheinlichkeit, dass ein spezieller Knoten erweitert wird, proportional zu der Größe seiner Voronoi-Region.

Betrachtet man nun Abbildung 2.25 so erkennt man, dass die Voronoi-Regionen der außen liegenden Knoten, auch *frontier*-Knoten genannt, wesentlich größer sind

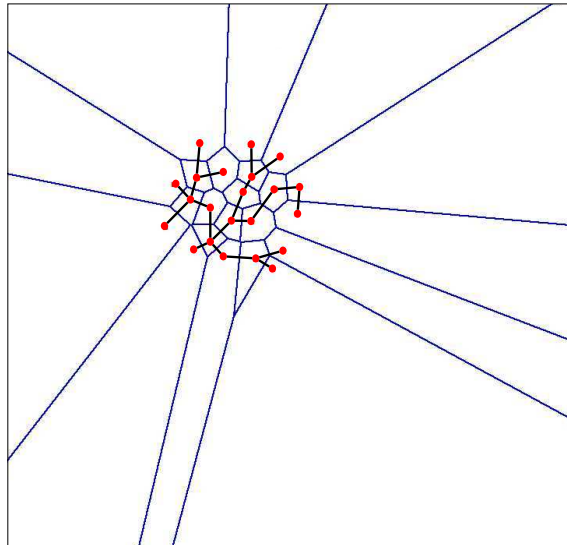


Abbildung 2.25: Voronoi-Gebiete eines rrt-Graphen (rot,schwarz: *RRT*-Graph, blau: Voronoi-Kanten)

als der Durchschnitt. Als Folge ist es viel wahrscheinlicher, dass sich der Baum in den nächsten Schritten nach außen weiter ausbreitet, als dass er sich nach innen weiter verdichtet. Analog könnte man auch sagen, dass der *RRT* ein Algorithmus ist, der versucht die Voronoi-Regionen möglichst gleichgroß zu halten.

Dies bewirkt den gewünschten Effekt, dass der *RRT* einen sehr starken Hang hat, sich in unbekannte Gebiete des Konfigurationsraumes auszubreiten. Dies ist eine der wichtigsten Eigenschaften, welche auch in allen Erweiterungen und Heuristiken dringend beibehalten werden muss.

2.7.3 Ergebnisse

Zusammenfassend kann nun gesagt werden, dass der *Rapidly-exploring Random Tree* ein *single query* Pfadplaner für statische Umgebungen ist. Ein sehr großer Vorteil des *RRT* liegt darin, dass er keine Einschränkungen in der Steuer-Funktion aufweist und somit auch auf **nichtholonomen** und **nicht-linearen Bewegungsmodellen** angewendet werden kann.

Soll ein Verfahren bewertet werden, so ist es interessant zu überprüfen, inwiefern sie korrekt (sound), vollständig (complete) und optimal sind. **Korrektheit** bedeutet in diesem Fall, dass es in der Realität auch einen solchen Pfad geben sollte, falls der *RRT* eine Lösung findet. Diese Eigenschaft wird bei dem *RRT* an die *ObstacleFree*- und *Steer*-Funktion weitergegeben. Solange diese korrekt sind, führt auch der *RRT* zu einem korrekten Ergebnis. Die Vollständigkeit im Sinne der Pfadplanung bedeutet, dass der Algorithmus einen Pfad finden muss, falls es

einen in der Realität gibt. Hier kann es jedoch tatsächlich vorkommen, dass der *RRT* sehr lange braucht, um Engstellen zu passieren, was auch bedeuten kann, dass vor Ablauf der Maximalzeit kein Pfad gefunden wurde, obwohl es einen gäbe. Allerdings steigt die Wahrscheinlichkeit einen Weg zu finden, falls einer existiert, mit der Anzahl der Knoten und somit mit der Laufzeit. Dies nennt man **probabilistische Vollständigkeit**. Ein sehr großer Makel hat der normale *RRT* in Hinsicht der Optimalität, welche in diesem Fall die Minimierung einer Kostenfunktion angewandt auf den Zielpfad bedeutet. Da die Knoten und Kanten zufällig erzeugt werden, kann es sein, dass der gefundene Zielpfad sehr große Umwege beinhaltet. Daher ist der normale *RRT* **nicht optimal**. Das Erreichen dieser Eigenschaft wird allerdings schon im folgenden Kapitel 2.8 angestrebt.

2.8 *RRT**

Der größte Mangel des einfachen *Rapidly-exploring Random Tree* ist, dass er keine Optimalität anstrebt. Wird ein neuer Knoten hinzugefügt, so wird bloß darauf geachtet, dass das neu entstandene Stück Pfad möglichst kurz ist. Die Gesamtkosten vom Startknoten bis zu dem neuen Knoten werden jedoch nicht berücksichtigt. Des Weiteren bleiben einmal gesetzte Kanten für immer fest bestehen, selbst wenn ein Knoten nach dem weiteren Wachstum des Baumes über einen anderen Weg schneller erreichbar wäre. Genau an diesen Punkten setzt die *RRT*-Variante *RRT** [KF10][KWP⁺11] an. Im Grunde erweitert sie den normalen *RRT* lediglich um zwei Funktionalitäten. Die `chooseParent`-Funktion sorgt dafür, dass eine neu entstehende Kante nicht lediglich lokal die kürzeste, sondern global gesehen die optimalste ist. Die `reWire`-Funktion wird dazu genutzt, um nach dem Setzen eines neuen Knotens darauf zu achten, ob durch diesen andere Knoten schneller erreicht werden können und strukturiert Kanten gegebenenfalls um. Im Folgenden werden diese Funktionen jedoch noch ausführlicher erklärt.

2.8.1 Algorithmus

```

 $\mathcal{G} = (\mathcal{V}, \mathcal{E}) \leftarrow \text{main}_{RRT^*}(v_{start})$ 

1  $\mathcal{G} \leftarrow \text{insertNode}(\emptyset, v_{start});$ 
2 for  $n = 1$  to  $N_{max}$  do
3    $v_{rand} \leftarrow \text{sample}(n);$ 
4    $v_{nearest} \leftarrow \text{nearest}(\mathcal{G}, v_{rand});$ 
5    $(v_{new}, u_{new}, \mathbf{x}_{new}) \leftarrow \text{steer}(v_{nearest}, v_{rand}, \Delta t);$ 
6   if !isBlocked( $\mathbf{x}_{new}$ ) then
7      $V_{near} \leftarrow \text{near}(\mathcal{G}, v_{new}, r);$ 
8      $v_{min} \leftarrow \text{chooseParent}(V_{near}, v_{nearest}, v_{new}, \mathbf{x}_{new});$ 
9      $\mathcal{G} \leftarrow \text{insertNode}(v_{min}, v_{new}, \mathcal{G});$ 
10     $\mathcal{G} \leftarrow \text{reWire}(\mathcal{G}, V_{near}, v_{min}, v_{new});$ 
11  end
12 end
13 return  $\mathcal{G}$ 

```

Abbildung 2.26: Der *RRT**-Algorithmus

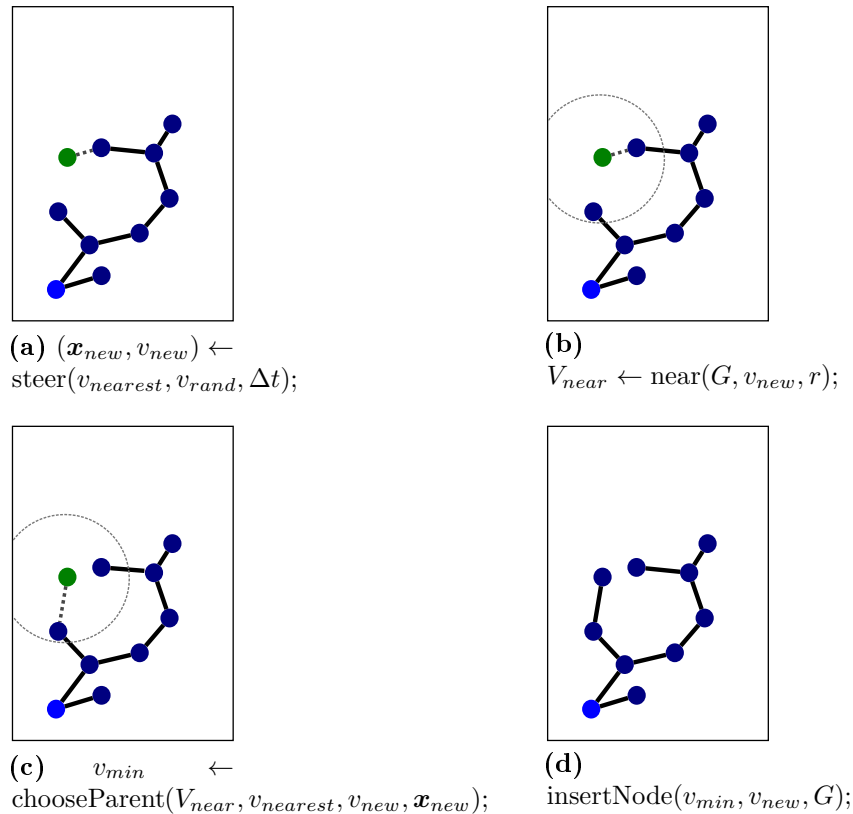


Abbildung 2.27: Hinzufügen eines neuen Knotens in den Graphen mittels chooseParent-Funktion

Der veränderte Algorithmus ist in Abbildung 2.26 zu sehen. Die grau markierten Zeilen unterscheiden sich dabei nicht von dem normalen *RRT* (Abb. 2.22). Erst nachdem ein neuer Knoten (v_{new}) mit gültigem Pfad (\mathbf{x}_{new}) gefunden wurde (Abb. 2.26, Zeilen 5-6), gibt es Unterschiede zum ursprünglichen Algorithmus. Bevor das chooseParent (Zeile 8) und reWire (Zeile 10) eingesetzt werden können, wird die von diesen Funktionen benötigte Menge von Knoten (V_{near}) gesucht, deren Elemente sich innerhalb einer vordefiniert-großen (r) Kugel um den neuen Knoten v_{new} befinden (Zeile 7). Hierbei sollte r so gewählt werden, dass alle Knoten in dieser Kugel liegen, von welchen aus v_{new} in einem Schritt(Δt) zu erreichen ist (siehe Abbildung 2.27 (b)).

Die ChooseParent-Funktion

Die Abbildung 2.27 veranschaulicht die Einordnung der chooseParent-Funktion und ihre Aufgabe. Nachdem mittels der steer-Funktion ein neuer Knoten v_{new} ermittelt wurde, schaut diese in dessen unmittelbarer Umgebung, ob es einen


```

chooseParent( $V_{near}, v_{nearest}, v_{new}, \mathbf{x}_{new}$ )
1  $v_{min} \leftarrow v_{nearest};$ 
2  $c_{min} \leftarrow c(v_{nearest}) + c(\mathbf{x}_{new});$ 
3 foreach  $v_{near} \in V_{near}$  do
4    $(v', u', \mathbf{x}') \leftarrow \text{steer}(v_{near}, v_{new}, \Delta t);$ 
5   if  $(\text{!isBlocked}(\mathbf{x}') \wedge (v' = v_{new}))$  then
6      $c' = c(v_{near}) + c(\mathbf{x}');$ 
7     if  $(c' < c_{min})$  then
8        $v_{min} \leftarrow v_{near};$ 
9        $c_{min} \leftarrow c';$ 
10    end
11  end
12 end
13 return  $v_{min}$ 

```

Abbildung 2.28: Die chooseParent - Funktion

optimaleren Vaterknoten als $v_{nearest}$ gibt, durch den die Kosten vom Startknoten bis zu v_{new} minimiert würden.

Wie der chooseParent-Algorithmus im Detail funktioniert, wird in Abbildung 2.28 aufgeführt. Nachdem die V_{near} -Kugel ermittelt wurde, geht die Funktion über jeden Knoten aus dieser Menge (Zeile 3) und überprüft, ob dieser ein potentieller Vaterknoten für v_{new} sein könnte (Zeilen 4-5). Dabei ist nicht nur darauf zu achten, dass der neu entstandene Pfad hindernisfrei ist, sondern auch, ob mit diesem trotz der Δt -Beschränkung der Zielknoten v_{new} wirklich erreicht wird (Zeile 5). Anschließend muss noch unter allen potentiellen Vaterkonfigurationen der bestimmt werden, über welche die Kosten von der Startkonfiguration bis hin zu dem neuen Zustand v_{new} minimal sind (Zeilen 6-7). Es wird als Vaterknoten also der Knoten gesucht, welcher folgende Gleichung erfüllt:

$$v_{min} = \arg \min_{v_{near} \in V_{near}} \left(c(v_{near}) + c(\mathbf{x}_{(v_{near}, v_{new})}) \right), \quad (2.16)$$

wobei $c(v)$ die Kosten vom Startzustand bis zu dem Knoten v beschreibt, $c(\mathbf{x})$ die Kosten des Pfades \mathbf{x} beschreibt (∞ wenn nicht hindernisfrei) und $\mathbf{x}_{(v_{near}, v_{new})}$ der neue Pfad von v_{near} bis v_{new} ist.

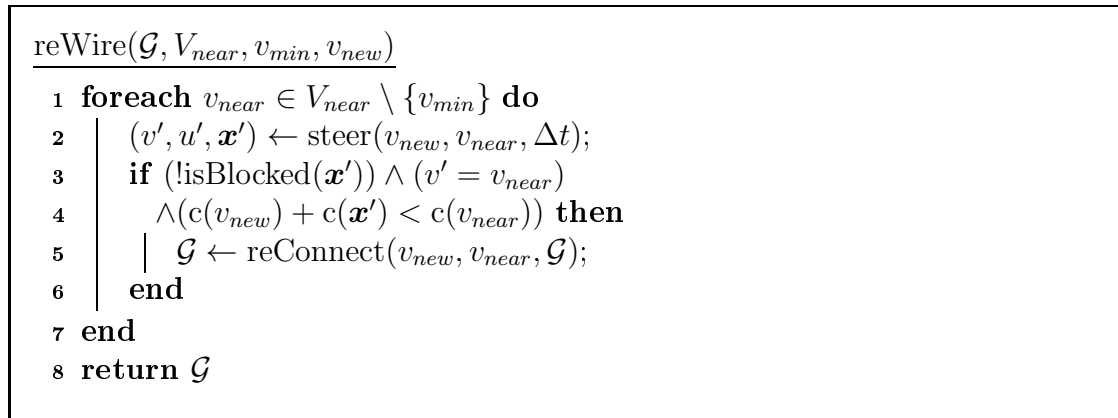


Abbildung 2.29: Die reWire - Funktion

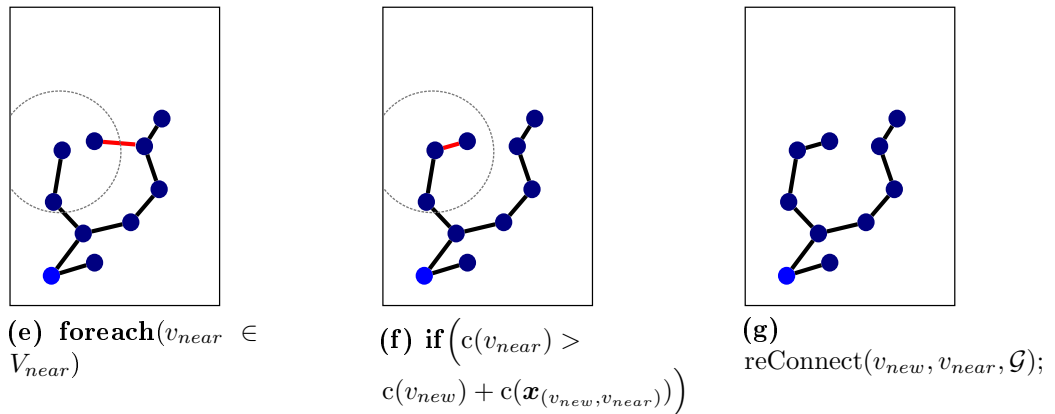


Abbildung 2.30: Optimierung des Baumes mittels reWire-Funktion (weiterführend von Abb. 2.27)

Die ReWire-Funktion

Nachdem mit der chooseParent-Funktion bereits der bestmögliche Vaterknoten für v_{new} ermittelt und in den Graphen eingetragen wurde, überprüft die reWire-Funktion, ob sich die Kosten von einem der Knotenpunkte aus der V_{near} -Kugel dadurch senken lassen, dass dieser nun die v_{new} -Konfiguration als Vaterknoten nimmt.

Eine solche Beispielsituation ist in Abbildung 2.30 als Fortführung von Abbildung 2.27 abgebildet. Den genauen Algorithmus hierzu sieht man in Abbildung 2.29. Hierin wird wieder über jeden Knoten aus V_{near} außer v_{min} iteriert und überprüft, ob diese Knoten von v_{new} aus erreichbar sind (Zeilen 1-4). Ist dies der Fall, wird

kontrolliert, ob die Kosten sinken würden und gegebenenfalls wird v_{new} als neuer Vaterknoten deklariert (Zeilen 5-6).

2.8.2 Ergebnisse

Vergleicht man nun den RRT^* mit dem normalen RRT , so erkennt man, dass sich an Korrektheit und der probabilistischen Vollständigkeit nichts geändert hat. Ebenso blieb die steer-Funktion unverändert und somit ist auch weiterhin gegeben, dass der Algorithmus auch für nichtholonome und nicht-lineare Fahrzeugmodelle geeignet ist. Die signifikante Veränderung des RRT^* ist ein großer Schritt hin zur Optimalität. Da der RRT^* während er läuft, nicht nur den Baum vergrößert, sondern auch existierende Bereiche optimiert, wird dieser auch nach der Zielfindung nicht gestoppt. Sobald ein gültiger Pfad gefunden wurde, wird dieser stetig weiter verbessert, solange der Algorithmus weiter läuft. Dadurch, dass Pfadstücke nur verändert werden, wenn dadurch auch die Kosten sinken, ist es nicht möglich, dass sich die Gesamtkosten wieder verschlechtern. Sie nähern sich also immer weiter dem absoluten Optimum an. In diesem Fall spricht man davon, dass der RRT^* **asymptotisch optimal** ist [KF11] [KF10]. Was im Rahmen dieser Arbeit leider nicht mehr umgesetzt werden konnte, ist eine Regelung, die festlegt, wie lange und wie stark der Baum nach der Zielfindung noch wachsen soll. Der Baum würde somit nicht unendlich weiter wachsen, obwohl kein Informationsgewinn mehr zu erwarten ist. Dies zu regeln wäre sinnvoll, um benötigte Rechenleistung und Speicherbedarf abhängig von der gewünschten Qualität des Pfades abstimmen zu können.

2.8.3 Erweiterungen

Goal Bias

Der *goal-bias* [AS11] ist eine Heuristik, welche sowohl bei dem normalen RRT als auch bei dem RRT^* eingesetzt werden kann. Bei Letzterem macht sie allerdings nur Sinn, solange ein Zielpfad noch nicht gefunden wurde. Absicht dieser Heuristik ist es, dem Baumwachstum eine Tendenz in Richtung des Zielzustandes zu geben, um einen Zielpfad schneller zu finden. Dies funktioniert besonders gut, wenn es bereits eine Stelle gibt, an der sich zwischen den Baumblättern und dem Zielzustand keine Hindernisse mehr befinden. Erreicht wird diese Orientierung dadurch, dass die sample-Funktion zu einem gewissen Anteil ein Zielknoten v_{goal} anstatt eines zufälligen Knotens zurück gibt, wie im Algorithmus in Abbildung 2.31 zu sehen ist. In dem Bild in Abbildung 2.31 erkennt man die Auswirkung der Heuristik. Sobald die Fläche zwischen einem Ast des RRT -Baums und dem Zielzustand (dunkelgrün) frei ist, hilft der *goal-bias* schnell einen direkten Weg zum Ziel zu finden.

```

 $v \leftarrow \text{sample}_{gb}(i)$ 
1  $p \leftarrow \text{random}(0, 1);$ 
2 if  $((p < \alpha) \wedge (\neg \text{Pathfound}))$  then
3   | return  $v_{goal};$ 
4 else
5   | return  $\text{sample}(i);$ 
6 end

```

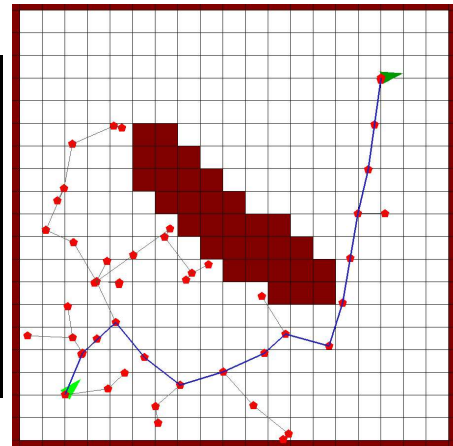


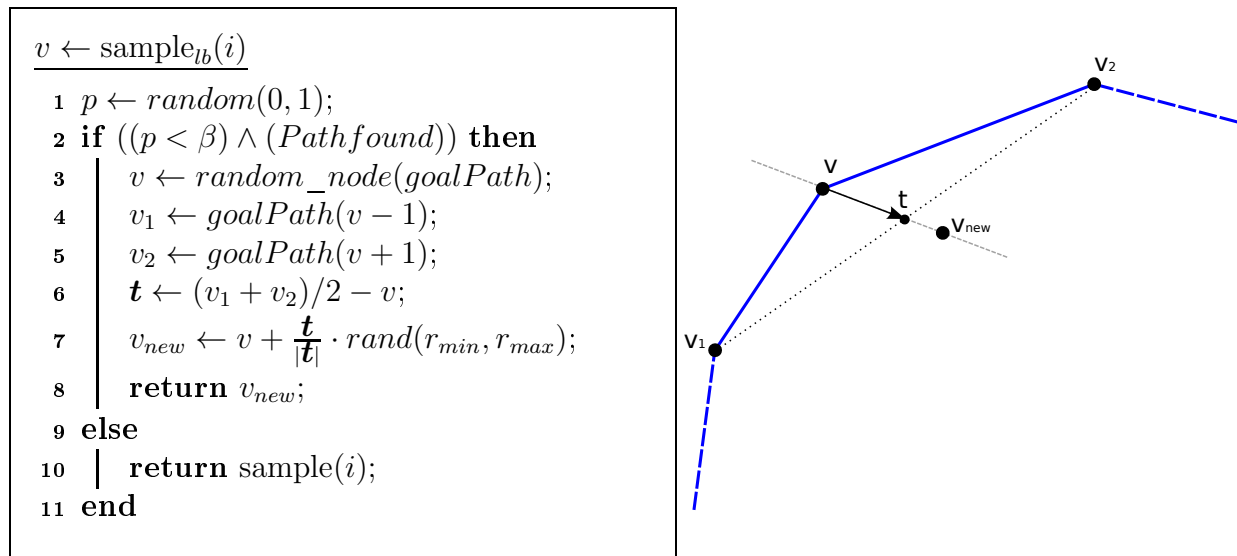
Abbildung 2.31: goal-bias sampling

Local Bias

Der *local-bias* [AS11] ist ebenso wie der *goal-bias* (Kapitel 2.8.3) eine Heuristik, welche das *Sampling* abändert, um dem Baumwachstum eine gewisse Richtung zu geben. Allerdings macht der *local-bias* erst Sinn, nachdem ein Zielpfad gefunden wurde. Das Ziel dieser Heuristik ist es, den bereits gefunden Pfad stärker zu optimieren, anstatt weiter in unbekannte Gebiete zu wachsen. Dieses Wachstum ins Unbekannte darf allerdings nicht vollständig ausgeschlossen werden, da sich durch dieses gegebenenfalls auch noch bessere Wege finden. Daher gibt es auch hier, ähnlich wie bei dem *goal-bias* einen Schwellwert β , welcher dafür sorgt, dass das lokale sampling nur zu einem gewissen Prozentsatz eingesetzt wird (siehe 2.32). Der in dieser Abbildung dargestellte Algorithmus, stammt von Akgun und Baris [AS11].

Läuft der *local-bias* allerdings auf einem Graphen, auf welchem zuvor der *goal-bias* (Kap. 2.8.3) angewandt wurde, so gibt es sehr viele Konfigurationen auf dem Pfad, welche genau mittig zwischen ihrem Vorgänger- und Nachfolgerknoten liegen. In diesem Fall wäre \mathbf{t} der Nullvektor und somit nicht normierbar. Um diesem zu umgehen, wird in einem solchen Fall statt \mathbf{t} ein zu $\mathbf{u} = v - v_1$ senkrechter Vektor mit der Länge 1 und zufälliger Orientierung konstruiert. Diese leicht abgeänderte Variante findet sich auch in der beigefügten Pfadplanungs-Bibliothek wieder.

Umgesetzt wird diese Heuristik, indem eine Zufallskonfiguration als Sample zurück gegeben wird, welche in Pfadnähe liegt, mit der Absicht diesen dadurch zu optimieren. Um ein *sample* in Pfadnähe zu erhalten, wird zunächst zufällig ein Knoten v aus dem bereits gefundenen Pfad ausgewählt (Abb. 2.32, Zeile 3), sowie sein Vorgänger v_1 und Nachfolger v_2 bestimmt (Zeilen 4-5). Da sich die Zustände im \mathbb{R}^n befinden, können die Knoten als Vektoren in diesem Raum vorgestellt werden

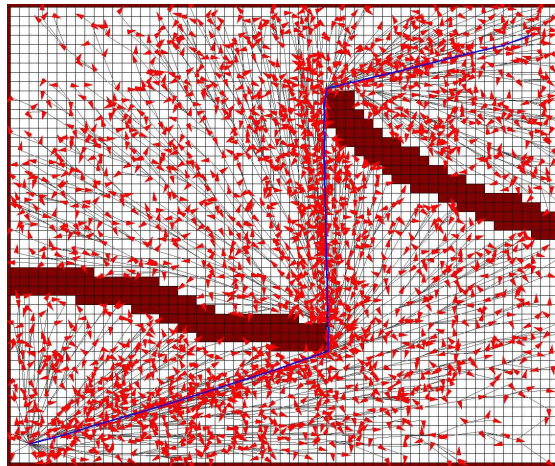
Abbildung 2.32: Das *local-bias* Sampling

und mittels einfacher Vektorarithmetik einen neuen zufälligen Knoten nahe v , zwischen v_1 und v_2 ermittelt werden. Dies wurde zum besseren Verständnis in der Skizze in Abbildung 2.32 visualisiert. Das Resultat dieser Erweiterung erkennt man in Abbildung 2.33, denn ein Großteil der Knoten liegen hier in Pfadnähe. Diese entstanden hauptsächlich, nachdem der erste gültige Pfad gefunden wurde, mit der Absicht diesen zu optimieren.

Sample Region

Da der *RRT* auch in unbegrenzten Terrain planen soll, galt es eine Region zu definieren, in welcher die neuen Zufallsknoten erlaubt werden. Diese Einschränkung ist nötig, um die *Samples* auf eine gewisse Region zu konzentrieren und somit schneller einen dichteren Baum zu erhalten und früher einen gültigen Pfad zu finden. Hierzu wird, in der, der Arbeit beigefügten Planungsbibliothek, eine Region um den Mittelpunkt zwischen der Startposition und der Zielposition definiert, welche einen Radius abhängig von der Distanz zwischen Start- und Zielposition hat. Da jedoch nicht sicher davon ausgegangen werden kann, dass der Zielpfad innerhalb dieser Region gefunden wird, wächst der Radius mit jeder Iteration, solange, bis ein gültiger Zielpfad gefunden wurde. Ab diesem Zeitpunkt ist der Radius immer genau die Hälfte der g-Kosten der Zielkonfiguration (= Länge des Pfades).

Im Folgenden wird bewiesen, dass jeder kürzere Pfad komplett innerhalb dieser Kugel zu finden ist. Geht man von dem Gegenteil aus, so gäbe es einen Punkt p , welcher außerhalb dieser Kugel läge (Abb. 2.34, Zeile 2.18) und trotzdem Teil

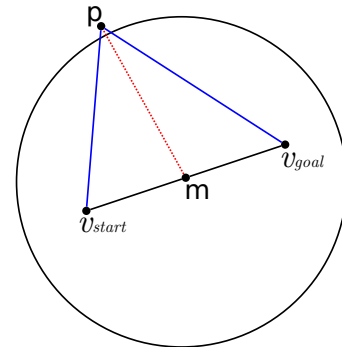
Abbildung 2.33: Ein *local-bias* Beispiel

$$\mathbf{m} = \frac{v_{start} + v_{goal}}{2} \quad (2.17)$$

$$\|\mathbf{p} - \mathbf{m}\| > \frac{g(v_{goal})}{2} \quad (2.18)$$

$$\|v_{start} - \mathbf{p}\| + \|\mathbf{p} - v_{goal}\| < g(v_{goal}) \quad (2.19)$$

$$\|v_{start} - \mathbf{p}\| + \|\mathbf{p} - v_{goal}\| < 2 \cdot \|\mathbf{p} - \mathbf{m}\| \quad (2.20)$$

Abbildung 2.34: Beweis zum *Sample-Region-Radius*

eines Pfades wäre, dessen Länge den doppelten Radius unterbietet. Der kürzeste Pfad wäre dabei die direkte Verbindung zwischen den drei Knoten (Zeile 2.19). Hiermit gelangt man zu der Aussage 2.20, welche niemals in Erfüllung gehen kann. Einfach zu erkennen ist dies an der Skizze in Abbildung 2.34. Wäre die Aussage 2.20 wahr, so gäbe es einen Fall, indem die blauen Linien zusammen kürzer wären, als zweimal die rote Linie, was jedoch niemals sein kann, womit die Aussage bewiesen wäre.

Node Rejection

Eine dritte Erweiterung des Sampling-Vorganges ist die *node-rejection* [AS11], welche ebenfalls erst eingesetzt werden kann, nachdem ein erster Pfad zum Ziel gefunden wurde. Ihr Ziel ist es, keine neuen Knoten zuzulassen, durch welche keine Verbesserung des aktuellen Pfades möglich ist. Auf diese Weise soll der

```

samplenr(i)
1 repeat
2   |  $v \leftarrow \text{sample}(i)$ ;
3 until
   ( $\|v - v_{init}\| + \|v_{goal} - v\| \leq g(v_{goal})$ );
4 return  $v$ ;

```

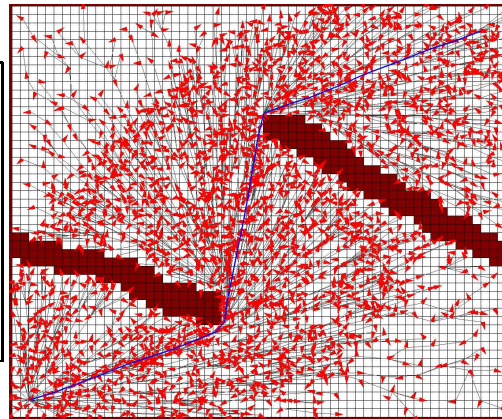


Abbildung 2.35: Das Node-Rejection sampling

Graph nicht unnötig groß werden. Abbildung 2.35 zeigt, dass keine Zufallskonfigurationen akzeptiert werden, deren minimalen Kosten von der Startkonfiguration v_{init} bis zu dem Sample v_{rand} und von v_{rand} zu der Zielkonfiguration v_{goal} bereits größer sind als die Kosten c_{best} des aktuell besten Zielpfades (Abb. 2.35). Die minimalen Kosten müssen dabei auf jeden Fall eine untere Schranke der tatsächlichen Kosten sein (beispielsweise die euklidische Distanz), ähnlich den Heuristiken beim A^* . Im Beispielbild in Abbildung 2.35 erkennt man, dass Anfangs noch alle Samples zugelassen wurden, und der Graph sich somit überall hin ausbreitet. Der Samplebereich wurde jedoch eingeschränkt, sobald das Ziel erreicht war.

Eine weitere Alternative dieser Heuristik besteht darin, in einer gewissen Frequenz über alle bereits bestehenden Knoten zu iterieren und zu testen, ob diese den aktuellen Pfad potentiell verbessern könnten oder sie ansonsten aus dem Baum zu entfernen. Dies ist besonders dann sinnvoll, wenn das Fahrzeug während der Planung fährt und somit große Graphbereiche überflüssig werden, wie es jedoch erst beim *Lifelong Planning RRT** (Kap. 2.9) auftreten kann. Dieser zweite Teil der Heuristik wurde jedoch noch nicht in der Planungsbibliothek umgesetzt.

Dynamic Domain RRT

Bei dem *Dynamic Domain RRT* [YJSL05] wird das *Sampling* in den Bereichen reduziert, in deren Richtung sich der Baum nicht weiter ausbreiten kann, da er dort sonst gegen Hindernisse wächst. LaValle et al. demonstrierte dieses Problem am Beispiel der sogenannten „bugtrap“ ([YJSL05]), wie sie in Abbildung 2.36 (a) dargestellt wird. Am eingezeichneten Voronoi Diagramm (rot) erkennt man, an welchem Knoten der Baum (blau) expandieren würde, gegeben der Stelle im Konfigurationsraum, an der der neue Zufallsknoten liegt (vgl. Kapitel 2.7.2). Hier erkennt man leicht, dass es nur einen sehr kleinen Bereich gibt, an den das nächste

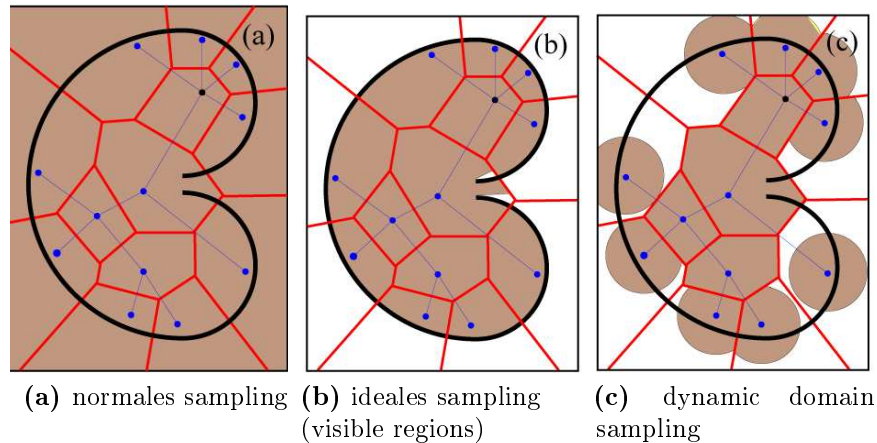


Abbildung 2.36: Sampling Bereiche in der „bugtrap“ [YJSL05] (blau: *RRT*-Graph, schwarz: Hindernis, rot: Voronoi-Grenzen)

Sample fallen müsste, damit der Baum aus der „Falle“ heraus wachsen könnte. Würde das Sample an irgendeiner anderen Stelle außerhalb der Falle liegen, so würde das nur dazu führen, dass der Baum gegen das Hindernis läuft. Stellt man sich nun vor, der Konfigurationsraum würde nach außen hin wachsen, so sinkt die Wahrscheinlichkeit innerhalb der *bugtrap* zu *samplen* noch weiter und somit auch die Wahrscheinlichkeit, dass der Baum aus der Sackgasse heraus wächst. Somit ergibt sich das Problem, dass es einen enormen Leistungsverlust des *RRT*-Algorithmus gibt, wenn viele **frontier nodes** ($\hat{=}$ Randknoten) gleichzeitig auch **boundary nodes** ($\hat{=}$ hindernisnahe Knoten) sind.

Die ideale Lösung wären hierfür die *visible regions*, zu sehen in Abbildung 2.36 (b). Die graubraun eingefärbte Region ist der Bereich, in denen die Samples erlaubt wären. Es würde also kein Sampling in nicht erreichbare Gebiete erlaubt, wodurch der starke Drang gegen Hindernisse zu laufen entfernt würde und trotzdem der sehr wichtige Drang des Wachsens in unbekannte Gebiete weiter unterstützt würde. Daher versucht der Dynamic Domain *RRT* diesen Idealfall anzunähern, mit sehr geringem Rechenaufwand und ohne die Vollständigkeit des *RRT* zu verlieren. Hierbei kann das Wachstum gegen die Hindernisse nicht komplett entfernt, aber trotzdem merklich verringert werden und der Bias in die unbekannt Gebiete bleibt trotzdem bestehen.

Erreicht wird dies, indem jedem Knoten ein Radius zugewiesen wird, welcher im Normalfall ∞ ist (Abb. 2.37, Zeile 8). Falls sich nun durch einen fehlgeschlagenen *extend*-Versuch herausstellt, dass ein Knoten ein *boundary*-Knoten ist, so wird dessen Radius auf einen festen Wert R gesetzt (Zeile Zeile 11). LaValle nutzte hier für R den zehnfachen Wert der Distanz, die ein *steer*-Schritt maximal schafft. Nun ist die einzige Änderung, welche noch an dem normalen *RRT*-Algorithmus


```

 $\mathcal{G} \leftarrow \text{DynamicDomain\_RRT}(v_{start})$ 
1  $\mathcal{G} \leftarrow \text{insertNode}(\emptyset, v_{start});$ 
2 for  $n = 1$  to  $N_{max}$  do
3   repeat
4      $v_{rand} \leftarrow \text{sample}(n);$ 
5      $v_{nearest} \leftarrow \text{nearest}(\mathcal{G}, v_{rand});$ 
6     until  $\text{distance}(v_{nearest}, v_{rand}) \leq \text{radius}(v_{nearest});$ 
7      $(v_{new}, u_{new}, \mathbf{x}_{new}) \leftarrow \text{steer}(v_{nearest}, v_{rand}, \Delta t);$ 
8     if  $\text{!isBlocked}(\mathbf{x}_{new})$  then
9        $\text{radius}(v_{new}) \leftarrow \infty;$ 
10       $\mathcal{G} \leftarrow \text{insertNode}(v_{nearest}, v_{new}, \mathcal{G});$ 
11     else  $\text{radius}(v_{nearest}) \leftarrow R;$ 
12 end
13 return  $\mathcal{G}$ 

```

Abbildung 2.37: Der *Dynamic Domain RRT* - Algorithmus

vorgenommen werden muss, die, dass ein neues Sample v_{rand} nur dann akzeptiert wird, wenn es innerhalb des Radius seines nächsten Nachbarn liegt (Zeile 5). Dies bewirkt, dass die Zufallskonfigurationen zwar immer noch im kompletten Konfigurationsraum gezogen werden, allerdings wird bei den meisten Punkten, bei denen die Erweiterung fehl schlagen würde, von vornherein auf die sehr kostenintensive Steuerfunktion (steer) und die Kollisionserkennung (!isBlocked) verzichtet. Abbildung 2.38 verdeutlicht noch einmal, inwiefern sich das *Sampling* im Vergleich zu den normalen *RRT* und den *visible regions* verändert. Die grau-braun eingefärbten Gebiete sind die, in denen neue Zufallsknoten v_{rand} erlaubt werden. Man erkennt, dass die Algorithmusänderung lediglich bei den problematischen boundary-Knoten eingreift. Dank des natürlichen Bias in unbekannte Gebiete tritt die Änderung vermehrt bei den *boundary*-Knoten auf, welche gleichzeitig auch *frontier*-Knoten sind.

LaValles Experimente mit dem bidirektionalen *Dynmaic Domain RRT-Connect* haben gezeigt, dass die beschriebene Erweiterung in Sonderfällen wie etwa in Abbildung 2.36 (a) sehr schnell eine Lösung findet, wobei der unveränderte *RRT-Connect*, je nach Größe des Umfeldes, vier mal oder 150 mal solange gebraucht hat einen Lösungsweg zu finden. Bei einem Versuch mit einem sehr großem Umfeld hat der unveränderte *RRT-Connect* selbst nach knapp einem Tag noch keine Lösung gefunden. Der *Dynmaic Domain RRT* fand jedoch bereits nach 1.6 sec einen gültigen Zielpfad [YJSL05]. Da ein solcher Sonderfall in der Realität jedoch kaum

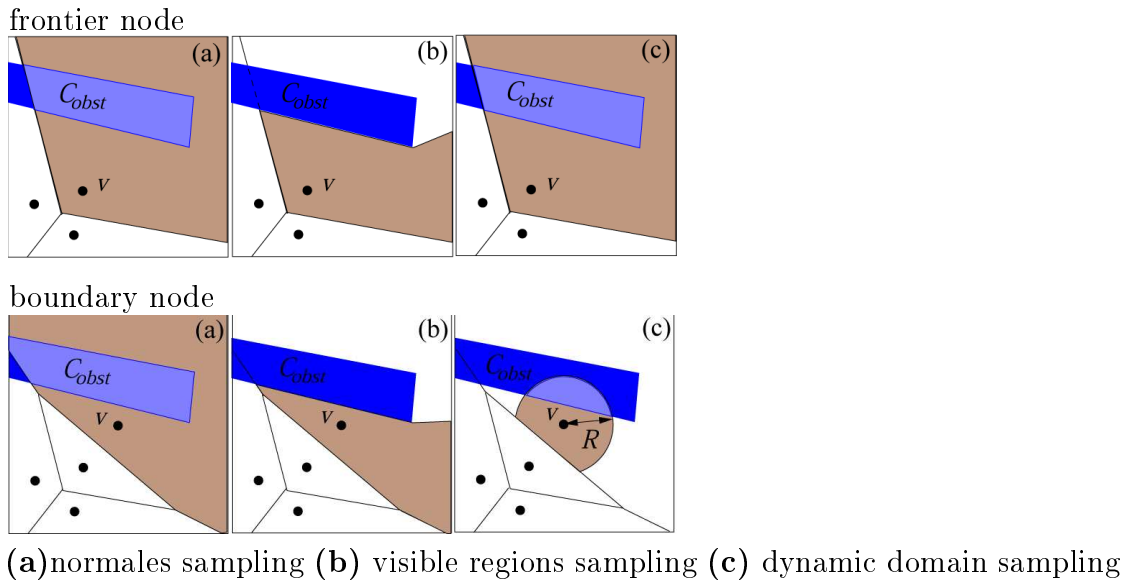


Abbildung 2.38: Sampling Bereich von v [YJSL05]

auftritt, wurden auch Experimente mit einem Pfadplaner mit vier Freiheitsgraden in einer sehr einfachen Umgebung ausgeführt. Auch in diesem Versuch war der *Dynamic Domain RRT* um etwa ein Drittel schneller als der herkömmliche *RRT-Connect* [YJSL05]. Es sollte jedoch beachtet werden, dass die Messergebnisse nicht von unabhängigen Dritten stammen, sondern von LaValle selbst. Eigene Messergebnisse finden sich hingegen in Kapitel 4.7.3.

Rekursives ReWire

Das folgende Kapitel erläutert ein Makel des *RRT**, welcher bei den Recherchen aufgefallen ist, und zeigt einen einfachen und einen erweiterten möglichen Ansatz zu seiner Behebung. Der *RRT** versucht den *RRT* so abzuändern, dass das gefundene Ergebnis dem Optimalen sehr nahe kommt. Hierbei soll das reWire die bereits bestehenden Pfade aufbrechen und so abändern, dass die einzelnen Knoten schneller angefahren werden, sobald dies möglich ist. Betrachtet man das *rewire* jedoch intensiver, so fällt auf, dass diese Veränderungen nur lokal innerhalb der direkten Umgebung eines neuen Knoten stattfinden, obwohl das Hinzufügen eines neuen Knotens auch noch Auswirkungen auf Knoten mit größerer Entfernung haben könnte. Offensichtlich wird dies am Beispiel in Abbildung 2.39 (a). Hierbei ist der Startknoten oben links und die Zielkonfiguration mit bereits gefundenem Pfad der blaue Kreis. Da die Lücke in dem grauen Hindernis sehr klein ist, wurde sie zunächst nicht gefunden und der Weg wurde komplett um das Hindernis herum geplant. Sobald aber der kürzere Weg durch das Hindernis hindurch entdeckt

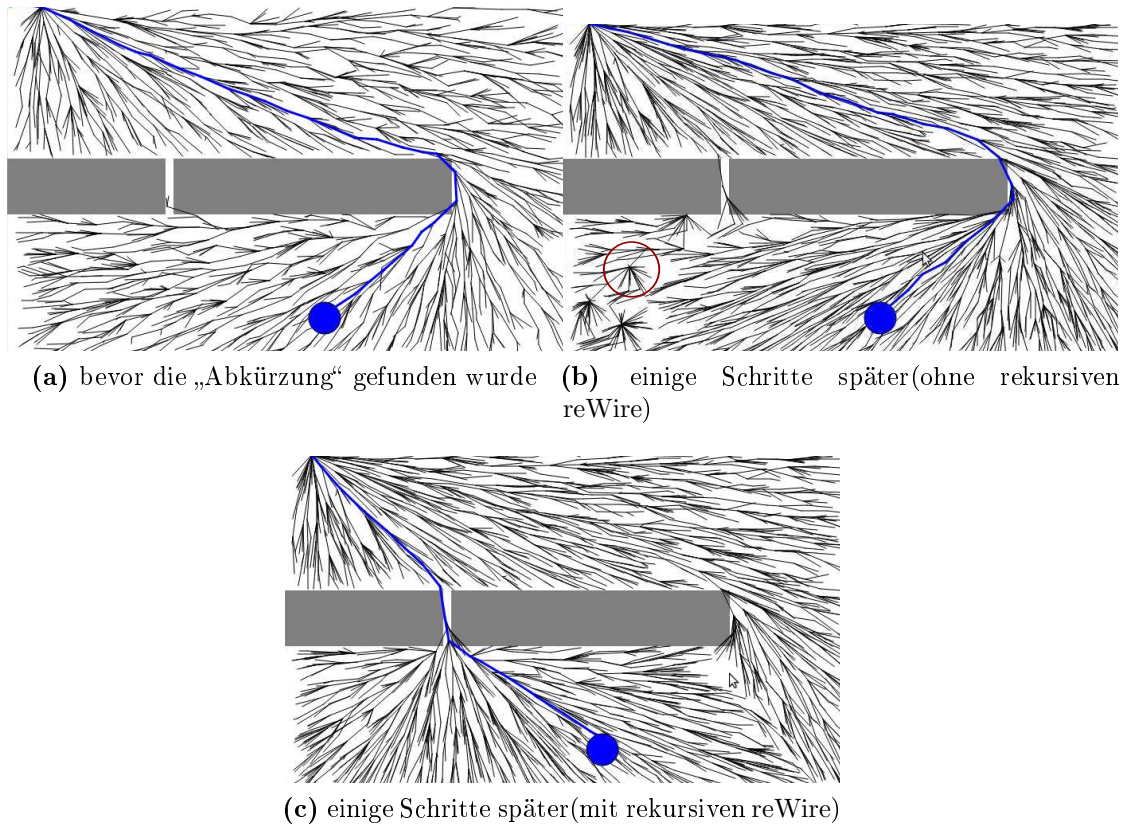


Abbildung 2.39: Motivation(a&b) und Ergebnis(c) des rekursiven reWire (grau: Hindernis, schwarz: RRT*-Graph, blau: Zielregion und Pfad)

```

reWirerec( $\mathcal{G}, V_{near}, v_{min}, v_{new}$ )
1  foreach  $v_{near} \in V_{near} \setminus \{v_{min}\}$  do
2     $(v', u', \mathbf{x}') \leftarrow \text{steer}(v_{new}, v_{near}, \Delta t)$ ;
3    if  $(\text{!isBlocked}(\mathbf{x}')) \wedge (v' = v_{near})$ 
4       $\wedge (c(v_{new}) + c(\mathbf{x}') < c(v_{near}))$  then
5         $\mathcal{G} \leftarrow \text{reConnect}(v_{new}, v_{near}, \mathcal{G})$ ;
6         $V'_{near} \leftarrow \text{near}(\mathcal{G}, v_{near}, r)$ ;
7        reWire( $\mathcal{G}, V'_{near}, v_{new}, v_{near}$ );
8      end
9  end
10 return  $\mathcal{G}$ 

```

Abbildung 2.40: Einfache rekursive reWire-Funktion

```

reWirerec( $\mathcal{G}, V_{near}, v_{new}$ )
1  foreach  $v_{near} \in V_{near}$  do
2     $(v', u', \mathbf{x}') \leftarrow \text{steer}(v_{new}, v_{near}, \Delta t)$ ;
3    if ( $\text{!isBlocked}(\mathbf{x}') \wedge (v' = v_{near})$ 
4       $\wedge (c(v_{new}) + c(\mathbf{x}') < c(v_{near}))$ ) then
5       $\mathcal{G} \leftarrow \text{reConnect}(v_{new}, v_{near}, \mathcal{G})$ ;
6       $V_{rewire} \leftarrow \text{push\_back}(V_{rewire}, v_{near})$ ;
7    end
8  end
9  while  $V_{rewire} \neq \emptyset$  do
10    $v \leftarrow \text{front}(V_{rewire})$ ;
11    $V'_{near} \leftarrow \text{near}(\mathcal{G}, v, r)$ ;
12    $\text{reWire}(\mathcal{G}, V'_{near}, v)$ ;
13 end
14 return  $\mathcal{G}$ 

```

Abbildung 2.41: Rekursive reWire-Funktion mittels Breitensuche

wurde, sieht der Baum im normalen RRT^* nach wenigen Iterationen so aus wie in Abbildung 2.39 (b). Wird ein neuer Knoten gesetzt, welcher Anschluss an den kürzeren neuen Pfad hat, wird dieser an den kurzen Pfad gebunden (durch `chooseParent`). Alle Knoten im engeren Umfeld nutzen nun diesen neuen Knoten durch das `reWire` als Vaterknoten, da dieser wesentlich geringere Kosten hat. Es ist gut zu erkennen, dass dieser Optimierungsschritt lediglich in dem kleinen V_{near} Umfeld bleibt (z.B. roter Kreis). Dabei wäre es wünschenswert, wenn alle die Knoten, deren Kosten durch das `reWire` gesunken sind, diese Information an ihre Nachbarknoten weitergeben würde, da diese dadurch eventuell ebenso durch ein Neuverbinden (`reConnect`) ihre Kosten senken könnten.

Eine einfache und verständliche Lösung dieses Gedankens findet sich in Abbildung 2.40. Hier wird das `reWire` rekursiv an jeden Knoten weitergegeben, welcher seine Kosten senken konnte. Dies ist jedoch nur ein erster Gedanke und ist in der Laufzeit noch zu verbessern. Eine Implementierung, welche die Breitensuche anstatt Tiefensuche für die `reWire` Weitergabe nutzt, ist in Abbildung 2.41 zu sehen. Dadurch, dass die Menge an Knoten V_{rewire} , welche noch `reWire`t werden müssen, wie eine *Queue* funktioniert, werden viele Mehrfachoptimierungen vermieden. Nach dieser Methode wird es auch in der Pfadplanungsbibliothek *ppLib* eingesetzt.

Das Ergebnis ist ein Baum wie er in Abbildung 2.39 (c) zu sehen ist, direkt nachdem die Abkürzung gefunden wurde. Untersucht man diesen Algorithmus genauer, so

erkennt man, dass dieser nicht nur in solchen extremen Situationen hilfreich ist. Selbst auf großen freien Flächen kommt das rekursive Optimieren oft zum Einsatz. In der Literatur lässt sich ein solcher Ansatz jedoch nicht finden. Dies könnte daran liegen, dass ohne diese Erweiterung das Hinzufügen eines Knotens eine fast konstante Zeit in Anspruch nimmt ($O(\rho)$, wobei ρ als Knotendichte zu verstehen ist). Dies ist durch die Rekursion nicht mehr gegeben. Daher lässt sich sehr viel schwerer abschätzen, wie viele Knoten pro Zeiteinheit hinzugefügt werden können. Des Weiteren werden alle Änderungen, welche durch das rekursive reWire entstehen, auch geschehen, wenn der normale *RRT** lange genug läuft. Jedoch ergibt sich durch das rekursive reWire eine neue positive Invariante: *Es ist zu jeder Zeit gegeben, dass zu jedem Knoten, der über die bereits bestehenden Knoten optimalste Pfad führt.* Des Weiteren liefert das rekursive reWire für den *Lifelong Planning RRT** (Kapitel 2.9) zusätzliche Vorteile.

Eine genaue Analyse über die zeitlichen Verzögerungen der Rekursion findet sich in Kapitel 4.7.2.

2.9 Lifelong Planning RRT*

Durch die einfache Handhabung und Variabilität des *RRT* und des *RRT**, bietet es sich an, diesen als Grundlage für einen *lifelong*-Planer zu nehmen. Im folgenden Kapitel wird der *Lifelong Planning RRT** (*LP-RRT**) beschrieben, eine Abwandlung des *RRT**, welcher sowohl mit *Änderungen im Umgebungsmodell* als auch mit einer *Veränderung der Startkonfiguration* umgehen kann. Somit erfüllt der *LP-RRT** alle Kriterien des *lifelong*, *replanning* und *anytime* Planens und ist geeignet für das effektive Planen in unbekannter und dynamischer Umgebung, obwohl er auf einen statischen *single-query* Pfadplaner basiert. Generell läuft der normale Expandierungsalgorithmus simultan zum *RRT** dauerhaft weiter, während Umgebungsänderungen, Zustandsänderungen, sowie Pfadabfragen auf Anfrage zur Laufzeit eingearbeitet werden, wie es in den folgenden Abschnitten genauer beschrieben wird. Ein Vorteil des *LP-RRT** ist, dass er einfache zweidimensionale holonome Pfade plant, um somit möglichst schnell und uneingeschränkt zu sein. Wird jedoch eine Pfad-Anfrage gestellt, so wird sichergestellt, dass ein geglätteter hindernisfreier Pfad ausgegeben wird, welcher alle fahrzeuggegebenen Einschränkungen einhält.

In einem Team des MIT zeigten Sertac Karamann und Emilio Frazzoli bereits 2007 in der *DARPA Urban Challenge*, dass der *RRT* sehr gut geeignet ist, in unbekannter, dynamischer Umgebung zu planen, indem sie mithilfe eines *closed loop-RRT* den vierten Platz einnahmen [KTK⁺08] [LHT⁺08]. Ihr Ansatz basiert jedoch auf dem einfachen *RRT*, wohingegen der *LP-RRT** auf dem *RRT** basiert.

2.9.1 Änderung des Fahrzeugzustandes

Um Änderungen der Startkonfiguration so einfach wie möglich einzubauen, wurde die Idee des Rückwärtsplanens vom *D* Lite* (Kapitel 2.4) übernommen. Das Versetzen des Baum-Ursprungs würde alle bereits berechneten Kostenwerte und einige Kanten ungültig machen. Das Verändern des Zielknotens (der Fahrzeugzustand) hat jedoch keinerlei Auswirkungen auf den restlichen bereits aufgebauten Graph, da der *RRT** in seiner Grundform eine uninformierte Suche ist. Lediglich der *goal-bias* (Kapitel 2.8.3) und der *local-bias* (Kapitel 2.8.3) benötigen die aktuelle Fahrzeugposition. Da aber davon ausgegangen werden kann, dass die fahrtbedingte Fahrzeugpositionsänderung nicht beliebig groß ist, bleibt es weiterhin sehr sinnvoll diese beiden Erweiterungen aktiv zu verwenden.

Verändert sich die Position des Fahrzeugs, so ist der neue Fahrzeugzustand nicht direkt mit dem Graphen verbunden. Allerdings ist die Wahrscheinlichkeit groß, dass der Graph trotzdem bis dorthin vorgedrungen ist, weshalb der neue Fahrzeugzustand dank dem *goal-bias* sehr schnell wieder mit dem restlichen Graphen verbunden sein. Sollte der neue Knoten bis zum Zeitpunkt der nächsten Pfadabfrage

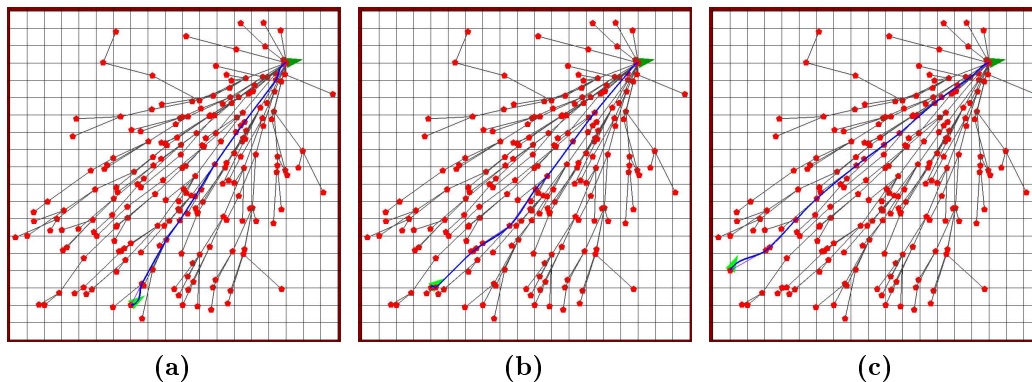


Abbildung 2.42: Die fahrzeugunabhängige Baumstruktur des $LP\text{-}RRT^*$ (rot-schwarz: Graph, hellgrün: Startpose, dunkelgrün: Zielpose, blau: Zielpfad)

noch nicht erreicht sein, beispielsweise, weil direkt nach der Positionsänderung die Pfadabfrage gestartet wurde, so wird beim *Backtracing* versucht diesen direkt mit einem extend-Schritt zu erreichen (Abb. 2.46 Zeilen 2.1 - 2.2). Diesen Vorgang kann man in Abbildung 2.42 sehen.

2.9.2 Änderung des Umgebungsmodells

Neue Hindernisse erscheinen

Da der Planer während der gesamten Fahrt vom Start bis zum Erreichen des Ziels durchgängig aktiv sein soll und nicht nach jeder Umgebungsänderung von Grund auf neu planen soll, muss er den bereits bestehenden Graphen an Umweltänderungen anpassen können. Wird ein Knoten durch ein neu entdecktes Hindernis unbefahrbar, so werden alle seine ein- und ausgehenden Kanten gelöscht (Abb. 2.45 Zeilen 8 & 10). Der Knoten selber bleibt bestehen, bekommt aber unendliche Kosten zugewiesen (Zeile 4), wodurch er bei der weiteren Expansion außer Acht gelassen wird. Hierbei werden bewusst nur Knoten auf Belegtheit getestet und nicht alle Kanten. Eine Überprüfung der Kanten wäre erstens viel aufwändiger, was den Algorithmus zu sehr verlangsamen würde. Zweitens sind die bis dato geplanten holonomen Kanten lediglich Annäherungen an den tatsächlich geglätteten Pfad, welcher erst im *Backtracing* berechnet wird. Sollte sich hierbei rausstellen, dass die Kante tatsächlich nicht befahrbar ist, so fängt dies das intelligente *Backtracing* ab, welches in Kapitel 2.9.3 beschrieben ist. Ebenso wie die belegten Knoten selbst, bekommen auch alle Nachfolgerknoten von diesen den g -Wert auf ∞ gesetzt (Zeile 4), da diese vorerst nicht zu erreichen sind. Hierdurch kann ein großer Bereich entstehen, welcher komplett als unerreichbar gilt, wie in Abbildung 2.43 (e) zu erkennen ist. Läuft der Algorithmus normal weiter, so wird irgendwann ein gültiger

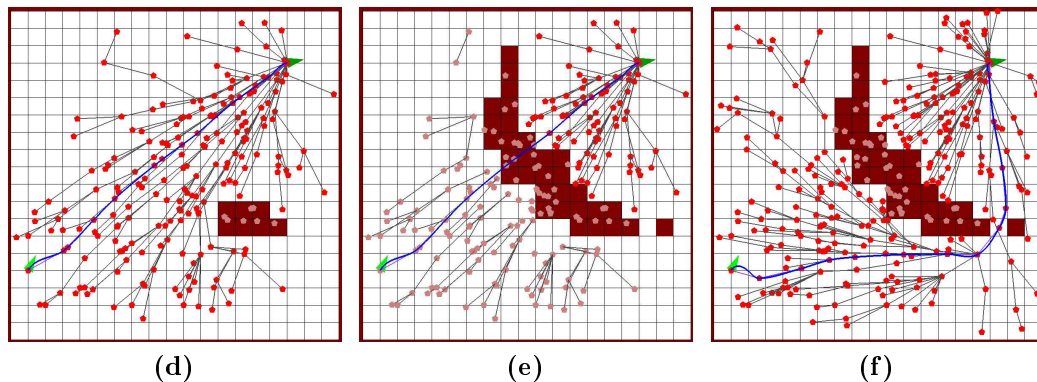


Abbildung 2.43: Neue Hindernisse erscheinen (rot-schwarz: Graph, hellgrün: Startpose, dunkelgrün: Zielpose, blau: Zielpfad, braun: Hindernisse)

Knoten diese Region erreichen und das rekursive reWire (Kapitel 2.8.3) sorgt dafür, dass die komplette Region sofort wieder gültige Kostenwerte erhält (siehe Abbildung 2.43 (f)). Da diese erste Verbindung jedoch nicht direkt die beste sein muss, kann es passieren, dass mehrerer reWire-„Wellen“ über dieses Gebiet laufen, was viel Aufwand mit sich bringt. Aus diesem Grund wird bei einem neu erschienenen Hinderniss versucht, dieses direkt zu umfahren, indem auf die direkten Nachfolgerknoten das chooseParent angewandt wird. War dies erfolgreich, so leitet das rekursive reWire die neuen Kosten an weitere Nachfolger weiter (Abb 2.45 Zeilen 6-7 & 13-17). Mit dieser Methode wird erreicht, dass Hindernisse möglichst sofort direkt umfahren werden, um häufige reWire-Wellen zu vermeiden. Zu erkennen ist dies in Abbildung 2.43 (d), welche direkt auf Abbildung 2.42 (c) folgt, ohne dass neue Samples fielen. Alternativ können auch alle unerreichbare Knoten sofort gelöscht werden, was Speicherplatz einspart, jedoch vor allem in dynamischen Umgebungen für eine ungleichmäßige Knotenverteilung sorgt. Hier wird deutlich, warum es trotz der relativ hohen Kosten sinnvoll ist, das rekursive reWire zu nutzen.

Hindernisse verschwinden

Lösen sich Hindernisse auf, so kann im Normalfall nur darauf gehofft werden, dass ein neues Sample in diesen Bereich fällt, wodurch der Graph in das neu erreichbare Gebiet vordringt. Ist dieses Gebiet jedoch vorher schon mal frei gewesen und wurde erst im späteren Verlauf blockiert, so können sich noch alte unerreichbare Zustände in diesem Gebiet befinden. Werden die zuvor blockierten Knoten wieder „befreit“, so wird auf diese sofort ein chooseParent und bei Erfolg ein reWire ausgeführt (Abb. 2.45, Zeilen 11-17). Hierdurch sollen die Knoten selbst wieder erreicht werden und es wird versucht den neuen Pfad zu nutzen, um andere Knoten schneller zu

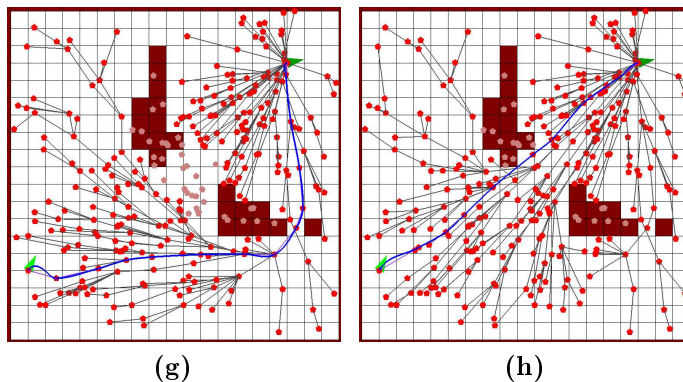


Abbildung 2.44: Hindernisse verschwinden

erreichen (Abb 2.44).

Sollte ein Teilgraph durch ein neues Hindernis un erreichbar geworden sein und dieser Teil keine Anknüpfung an den restlichen Graphen gefunden haben, so sorgt die Auflösung des Hindernisses sofort wieder zu einem vollständigen Erreichen des zuvor un erreichbaren Gebietes.

Diese Methode erklärt nun auch, warum alte Knoten, bei denen eine Hinderniskollision entdeckt wurde, nicht direkt gelöscht werden. Besonders in dynamischen Umgebungen ist dies sehr wichtig. Ansonsten würde zum Beispiel ein einfaches Auto, welches durch die Szene fährt, zur Unerreichbarkeit großer Gebiete führen.

2.9.3 Backtracing

Aus Effizienzgründen und zur leichteren Handhabung plant der *RRT* und seine Erweiterungen, welche im Rahmen dieser Arbeit implementiert wurden, lediglich im Zweidimensionalen und holonom. Der ermittelte Pfad wird beim *RRT* und *RRT** nachdem er gefunden wurde zwar noch geglättet, sodass er stetig differenzierbar ist, jedoch wird hierbei nicht mehr kontrolliert ob Hindernisse geschnitten werden. Genau dieses Problem löst der *LP-RRT**, indem er eine Kante, welche im geglätteten Verlauf mit einem Hindernis kollidieren würde oder nicht-holonome Bedingungen nicht einhält, gelöscht wird, sobald sie entdeckt wird (Abbildung 2.46, Zeile 1.7). Da alle Nachfolgeknoten der gelöschten Kante nun keinen Zugang zum Startknoten haben, werden ihre *g*-Werte auf ∞ gesetzt, um somit möglichst schnell im Verlauf des weiteren Planens wieder angefahren werden zu können (Zeile 6). Dies geschieht genauso wie bei der Entdeckung neuer Hindernisse (Kapitel 2.9.2), weshalb auch hier das rekursive reWire (Kapitel 2.8.3) einen klaren Vorteil darstellt. Da durch ein solches Vorgehen der Zielknoten, von welchem aus das *Backtracing* startet, nicht mehr zu erreichen ist, muss nach einem fehlgeschlagenem *Backtracing* so lange weiter geplant werden, bis dieser Knoten wieder einen gültigen *g*-Wert

```

processEnvironmentChanges( $C_{changed}$ )
1  $V_{update} \leftarrow \emptyset$ ;
2 foreach  $v \in V : vehicle(z(v)) \cap C_{changed} \neq \emptyset$  do
3   if isBlocked( $v$ ) then
4     foreach  $v' \in succOf(v) \cup \{v\}$  do  $g(v') \leftarrow \infty$  ;
5     foreach  $v_{child} : \{v, v_{child}\} \in \mathcal{E}$  do
6       if !isBlocked( $v_{child}$ ) then
7          $V_{update} \leftarrow V_{update} \cup \{v_{child}\}$ ;
8          $\mathcal{E} \leftarrow \mathcal{E} \setminus \{v, v_{child}\}$ ;
9       end
10       $\mathcal{E} \leftarrow \mathcal{E} \setminus \{v_{parent}, v\}$ ;
11   else  $V_{update} \leftarrow V_{update} \cup \{v\}$ ;
12 end
13 foreach  $v \in V_{update}$  do
14   if chooseParent( $v, near(v)$ ) then
15      $reWire(near(v), v)$  ;
16   end
17 end

```

Abbildung 2.45: Algorithmus zu Verarbeitung von Umgebungsänderungen

besitzt (Zeilen 2.1-5). Da in der *findPath*-Prozedur gegebenenfalls aktiv geplant wird, anstatt einfach ein *false* zurück zu geben, garantiert die Funktion immer einen gültige Pfad zu liefern. Das normale Planen läuft im Normalfall in einem separaten Thread, welcher für dieses *Backtracing* kurzzeitig angehalten wird, damit nicht zwei Algorithmen gleichzeitig den Graphen verändern.

Da diese Variante des Pfadfindens etwas aufwändiger ist, wird sie nur angewendet, wenn explizit nach diesem Pfad gefragt wird. Parallel läuft aber auch noch ein einfaches Knoten-*Backtracing*, welches gestartet wird, sobald sich die Zielknotenkosten verringern. Dieser dient dazu, einen annähernd gültigen Pfad darzustellen, welcher dem local-bias (Kapitel 2.8.3) und der Bestimmung der Sampleregion (Kapitel 2.8.3) dient.

```

1. bool ← backtrace(v)
  1 if  $g(v) = \infty$  then
  2   | return false ;
  3 while  $v \neq v_{start}$  do
  4   |  $\mathbf{x}_{(v,parent(v))} \leftarrow \text{steer}(v, parent(v));$ 
  5   | if isBlocked( $\mathbf{x}_{(v,parent(v))}$ ) then
  6     | foreach  $v' \in \text{succOf}(v)$  do  $g(v') \leftarrow \infty ;$ 
  7     |  $\mathcal{E} \leftarrow \mathcal{E} \setminus \{parent(v), v\};$ 
  8     | return false;
  9   |  $v \leftarrow parent(v);$ 
10 end
11 return true;

```

```

2. findPath()
  1 while !backtrace( $v_{goal}$ ) do
  2   | while  $g(v_{goal}) = \infty$  do
  3     | extend( $\mathcal{G}$ , sample());
  4   | end
  5 end

```

Abbildung 2.46: Der findPath - Algorithmus: intelligentes Backtracing

Kapitel 3

Pfadplanungsbibliothek

3.1 Kollisionsvermeidung

Eine einfache und schnelle Methode in der Kollisionsdetektion der Pfadplanung ist es, alle Hindernisse um die maximale Fahrzeugbreite zu erweitern, um dadurch das Fahrzeug als punktförmig annehmen zu können. Abbildung 3.1 zeigt ein beispielhaftes Szenario. Weil die Hindernisse immer nur um ein Vielfaches der Zellenbreite erweitert werden können, um in der Belegtheitskarte dargestellt werden zu können, stellt sich die Frage, ob optimistisch oder pessimistisch erweitert werden soll. Da bei der Trajektorien-Planung die Korrektheit des gefundenen Pfades wichtiger ist als die Vollständigkeit des Planers, sollte bei der Erweiterung der Hindernisse eher aufgerundet werden. Dies ist besonders dann von Bedeutung, wenn bei Eckpunkt-basierten Graph-Planern (z.B. *Field D**) ein Pfad, welcher direkt an Hindernissen entlang führt, noch gültig sein soll.

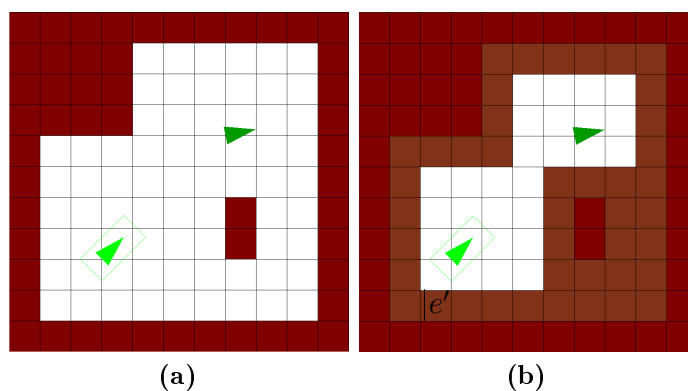


Abbildung 3.1: Erweiterung der Hindernisse um e'

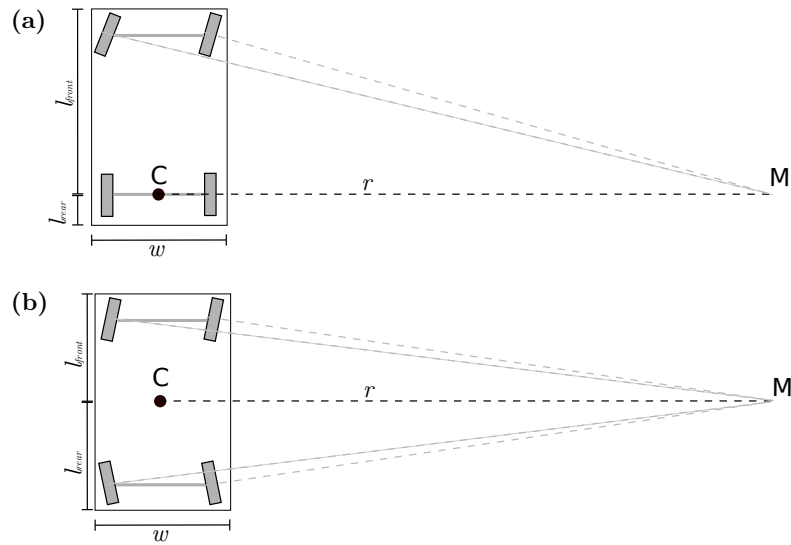


Abbildung 3.2: Fahrzeugmodell

Kollisionsvermeidung im $LP\text{-}RRT^*$ Beim $LP\text{-}RRT^*$ ist die Kollisionsdetektion im *Backtracing* unabhängig von der Detektion in der Expansion und wird lediglich bei einer Pfadabfrage aufgerufen. Daher ist es hierbei nicht so wichtig, auf eine schnelle Lösung zu achten und es kann mehr Wert auf die Vollständigkeit gelegt werden. Von daher wird in diesem Fall nicht die Hinderniserweiterung angewendet, sondern der exakte Fahrkorridor mit den Hindernissen verglichen, um somit keine Pfade auszuschließen, welche in Wahrheit kollisionsfrei sind.

3.1.1 Fahrzeugmodellierung und Ausschwenkmaße

Um die Breite der Hinderniserweiterung möglichst gering zu halten, ist es sinnvoll, das Fahrzeugmodell genauer zu betrachten. Das hier verwendete Fahrzeugmodell entspricht einem rechteckigen Fahrzeug, beispielsweise mit einer Ackermannlenkung, wie es in Abbildung 3.2 zu sehen ist. Wo genau sich die Lenkachsen befinden, ist hierbei weniger wichtig, als der Mittelpunkt C der starren Achse. Bei einer einfachen ein-Achs-Lenkung (Abb. 3.2 (a)) ist dieser Punkt der Mittelpunkt der Hinterachse. Bei einem abweichendem Lenkverhalten ist die starre Achse die, welche senkrecht zur Fahrtrichtung liegt und bei einer Kurvenfahrt genau auf den Kurvenmittelpunkt zeigt. Ein Beispiel mit zwei Lenkachsen wird in Abbildung 3.2 (b) gezeigt. Für die kollisionsfreie Navigation durch eine Umgebung ist es relevant einen Fahrkorridor zu bilden und diesen mit den Hindernissen abzugleichen. Für diesen Korridor ist die maximale Ausdehnung des Fahrzeuges, vom Mittelpunkt

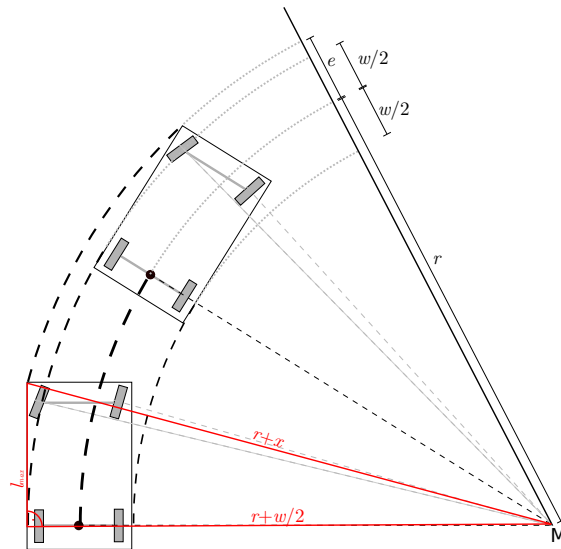


Abbildung 3.3: Fahrzeug schwenkt aus

C aus gesehen, die obere Schranke der halben Breite und somit auch die maximal nötige Ausdehnung der Hindernisse:

$$\bar{e} = \sqrt{\frac{w^2}{4} + l_{max}^2} \quad (3.1)$$

Dabei ist $2\bar{e}$ die maximal mögliche Breite des Korridors, w die Breite des Fahrzeugs und

$$l_{max} = \max(l_{front}, l_{back}) \quad (3.2)$$

ist die maximale Längs-Ausdehnung vom Mittelpunkt C aus. Der Wert \bar{e} liegt in den meisten Fällen weit entfernt von der tatsächlichen Korridorbreite, welche \bar{e} nur dann erreichen könnte, wenn sich das Fahrzeug auf der Stelle um den Mittelpunkt $C = M$ dreht. Um diese Überschätzung einzuschränken, sollte die Korridorausdehnung abhängig vom aktuellen Lenkwinkel berechnet werden. Betrachtet man Abbildung 3.3, so erkennt man, dass die Korridorausdehnung in Richtung des Kurvenmittelpunkts M nicht größer als $\frac{w}{2}$ wird. Nach Außen hin schwenkt das Fahrzeug jedoch um e aus. Dieser Wert berechnet sich ähnlich wie \bar{e} , doch ist der Drehmittelpunkt nicht gleich dem Fahrzeugmittelpunkt (vgl. Abbildung 3.3). Mit r als Kurvenradius ergibt sich:

$$e(r) = \sqrt{\left(r + \frac{w}{2}\right)^2 + l_{max}^2} - r \quad (3.3)$$

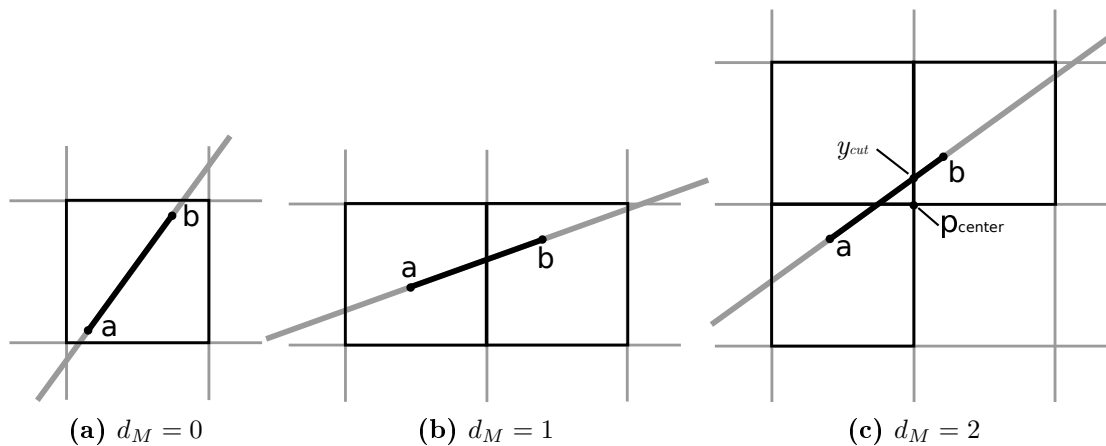


Abbildung 3.5: Linien Rasterisierung, drei verschiedene Fälle

weiterer Nachteil des ursprünglichen Bresenham's ist, dass er nicht alle geschnittenen Zellen im Ergebnisvektor zurück liefert. In Abbildung 3.4 würde er beispielsweise nur die dunkelgrauen Zellen ermitteln, für die Kollisionsvermeidung ist es jedoch notwendig alle geschnittenen Zellen zu bestimmen (alle Grauen Zellen).

Für den Algorithmus in dieser Form ist es notwendig, dass die Zellen quadratisch sind und alle die Breite 1.0 haben. Die grobe Bresenham-ähnliche Idee hinter dem Algorithmus besteht darin, in 1-er Schritten vom Startpunkt beginnend den Pfad entlang zu gehen und alle Zellen, die dabei 'getroffen' werden, im Ergebnisvektor zurückzugeben.

Es wird davon ausgegangen, dass a der letzte bearbeitete Punkt sei und b der nächste auf der Linie liegende Punkt ist, mit:

$$\|b - a\| = 1 \quad (3.4)$$

Von einem Punkt im \mathbb{R}^n ist die betreffende Zelle im \mathbb{Z}^n einfach durch Abrunden zu bestimmen, da die besagte Zellenbreite genau 1 ist. Daher ergibt sich für die Zellen z_a und z_b :

$$z_a \leftarrow [a] = \begin{pmatrix} [a.x] \\ [a.y] \end{pmatrix} \quad \text{und} \quad z_b \leftarrow [b] = \begin{pmatrix} [b.x] \\ [b.y] \end{pmatrix} \quad (3.5)$$

Da die Schrittweite genauso groß ist wie die Zellgröße, können nur drei verschiedene Fälle auftreten, wie die Zellen z_a und z_b zueinander liegen. Unterscheiden lassen sich diese Fälle einfach, indem die Manhattan Distanz (Norm 1) d_M zwischen den betroffenen Zellen ermittelt wird.

$$d_M = \|z_a - z_b\|_1 = \|z_a.x - z_b.x\| + \|z_a.y - z_b.y\| \quad (3.6)$$

Der einfachste Fall ist der, wenn $d_M = 0$: der Punkt \mathbf{b} liegt in der selben Zelle wie \mathbf{a} (zu sehen in 3.5 (a)). Hier muss nichts weiter getan werden, da die Zelle bereits durch Bearbeitung des Punktes \mathbf{a} ermittelt wurde. Es kann mit dem nächsten Iterationsschritt fortgefahren werden.

Ist $d_M = 1$, so bedeutet es, dass die neue Zelle \mathbf{z}_b ein direkter Nachbar der Vorgängerzelle \mathbf{z}_a ist. Da keine weitere Zelle dazwischen liegen kann, genügt es, die \mathbf{b} -umfassende Zelle zu dem Ergebnisvektor hinzuzufügen.

Etwas komplizierter wird es, wenn die Manhattan Distanz $d_M = 2$ ist. Da es sich hierbei um diagonale Nachbarn handelt, findet sich in der Regel noch eine dritte betroffene Zelle, welche zu bestimmen ist (siehe Abbildung 3.5 (c)). Hierzu wird getestet, ob die mittlere Zellenecke $\mathbf{p}_{\text{center}}$ zwischen \mathbf{z}_a und \mathbf{z}_b über oder unter der Strecke liegt. Es wird also ermittelt, wie $\mathbf{p}_{\text{center}} \cdot y$ zu y_{cut} steht, dem y -Wert der Strecke an der Stelle $\mathbf{p}_{\text{center}} \cdot x$. Hierzu muss die Steigung

$$\alpha = \frac{\mathbf{b} \cdot y - \mathbf{a} \cdot y}{\mathbf{b} \cdot x - \mathbf{a} \cdot x} \quad (3.7)$$

einmalig pro Linie vorberechnet werden. Es ergibt sich:

$$y_{\text{cut}} = \mathbf{a} \cdot y + \alpha \cdot (\mathbf{p}_{\text{center}} \cdot x - \mathbf{a} \cdot x), \quad (3.8)$$

Nun kann per Fallunterscheidung ermittelt werden, welche Zelle die gesuchte dritte Zelle \mathbf{z}_3 ist:

$$y_{\text{cut}} < \mathbf{p}_{\text{cross}} \cdot y \begin{cases} \mathbf{z}_a \cdot y < \mathbf{z}_b \cdot y \Rightarrow \mathbf{z}_3 = \begin{pmatrix} \mathbf{z}_b \cdot x \\ \mathbf{z}_a \cdot y \end{pmatrix} \\ \mathbf{z}_a \cdot y > \mathbf{z}_b \cdot y \Rightarrow \mathbf{z}_3 = \begin{pmatrix} \mathbf{z}_a \cdot x \\ \mathbf{z}_b \cdot y \end{pmatrix} \end{cases} \quad (3.9)$$

$$y_{\text{cut}} > \mathbf{p}_{\text{cross}} \cdot y \begin{cases} \mathbf{z}_a \cdot y < \mathbf{z}_b \cdot y \Rightarrow \mathbf{z}_3 = \begin{pmatrix} \mathbf{z}_a \cdot x \\ \mathbf{z}_b \cdot y \end{pmatrix} \\ \mathbf{z}_a \cdot y > \mathbf{z}_b \cdot y \Rightarrow \mathbf{z}_3 = \begin{pmatrix} \mathbf{z}_b \cdot x \\ \mathbf{z}_a \cdot y \end{pmatrix} \end{cases} \quad (3.10)$$

Abschließend wird sowohl \mathbf{z}_b also auch \mathbf{z}_3 zum Ergebnis hinzugefügt.

Dies wird solange fortgeführt, bis der Punkt \mathbf{b} hinter dem Zielpunkt liegt. In diesem Fall wird \mathbf{b} mit dem Zielpunkt gleichgesetzt und als letzter Kontrollpunkt genommen, bevor der Algorithmus abbricht. Dies kann ohne weiteres gemacht werden, da die Distanz zwischen \mathbf{a} und \mathbf{b} auch kleiner als 1 sein darf. Sie darf nur nicht größer sein, da sonst noch weitere Fälle auftreten könnten.

Der komplette Algorithmus ist im Anhang B.1 zu finden.

3.2 Glatte Pfadstücke

3.2.1 Einfache Kreisfahrt

Die vordefinierten Pfadstücke beim *hybrid A** basieren auf einfachen Kreisfahrten. Hierfür müssen Nachfolgezustände $z' = \{\mathbf{p}', \Theta'\} \in \mathcal{Z}$ berechnet werden, welche von einem Zustand $z = \{\mathbf{p}, \Theta\} \in \mathcal{Z}$ aus erreicht werden. Sie werden bestimmt, indem mit einem Radius r um den Mittelpunkt \mathbf{M} die Strecke *range* gefahren wird, wie in Abbildung 3.6 zu erkennen ist.

O.B.d.A kann für die Herleitung angenommen werden, dass der Startwinkel $\Theta = 0$ ist, wodurch sich Folgendes ergibt:

$$\frac{range}{2\pi r} = \frac{\alpha}{2\pi} \Rightarrow \alpha = \frac{range}{r} \quad (3.11)$$

$$\sin(\alpha) = \frac{d_x}{r} \Rightarrow d_x = \sin(\alpha) r \quad (3.12)$$

$$\cos(\alpha) = \frac{r - d_y}{r} \Rightarrow d_y = r - \cos(\alpha) r \quad (3.13)$$

$$\mathbf{d} = (d_x, d_y)^T \quad (3.14)$$

Zur schnelleren Anwendung im Algorithmus kann \mathbf{d} und α für $\Theta = 0$ im Voraus berechnet werden, wie oben gezeigt. Später genügt es \mathbf{d} um Θ zu drehen und Θ auf α zu addieren, um den neuen gesuchten Zustand $z' = \{\mathbf{p}', \Theta'\}$ zu erhalten:

$$\mathbf{p}' = \mathbf{p} + R_\theta \mathbf{d} \quad (3.15)$$

$$\Theta' = \Theta + \alpha \quad (3.16)$$

Die für den *hybrid A** benötigten verschiedenen Zustände, werden durch ein Variieren des Radius zwischen dem minimalen Kurvenradien $-r_{min}$ und $+r_{min}$ erlangt.

3.2.2 Glatte Kreisübergänge zwischen gegebenen Zuständen

Wenn der *RRT* nicht nur holonome Pfade plant, sondern mit glatten Pfadstücken expandiert, so wird eine Funktion benötigt, welche einen glatten Pfad zwischen zwei gegebenen Zuständen $z_a = (\mathbf{p}_a, \Theta)$ und $z_b = (\mathbf{p}_b, \Theta_b)$ liefert. Diese müssen auch an den Knotenpunkten stetig differenzierbar sein. Die Grundidee der folgenden Algorithmen ist dabei, den Zuständen nicht nur eine Position und Ausrichtung zuzuordnen, sondern zusätzlich noch einen Lenkwinkel. Der Lenkeinschlag wird so gewählt, dass das Fahrzeug mit dieser Konfiguration jeweils durch beide Posen fahren kann. In der Pose z_a wird der Lenkwinkel so gewählt, dass ein Fahrzeug mit dieser Konfiguration startend genau zur Position \mathbf{p}_b fährt, ohne jedoch die Ausrichtung der Pose z_b zu beachten. Im Knoten z_b wird der Lenkwinkel so gewählt,

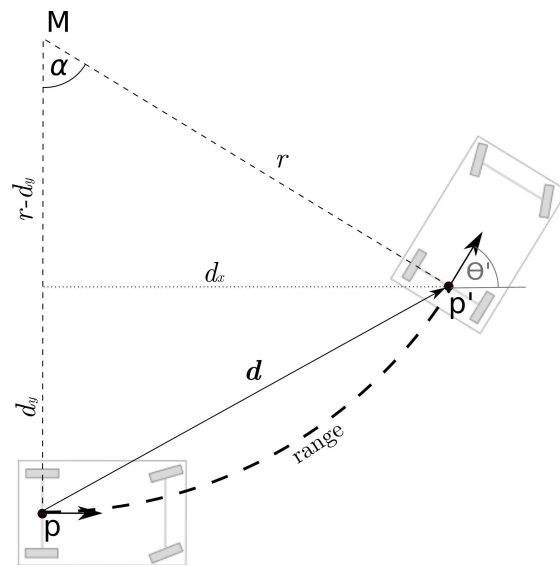


Abbildung 3.6: Einfache Kurvenfahrt

dass ein Fahrzeug mit dieser Einstellung von der Position \mathbf{p}_a aus kommend genau durch die Konfiguration von z_b fährt, ohne die Ausrichtung Θ von z_a zu beachten. Der gesuchte Pfad entsteht nun durch die Interpolation der beiden Teilstrecken zwischen den Punkten \mathbf{p}_a und \mathbf{p}_b . Im Folgenden wird dabei zwischen zwei verschiedenen Fällen unterschieden.

Circular-Linear Interpolation

Plant der *RRT* in zwei Dimensionen, so wird zwar die Position des zu erreichenden neuen Knotens vorgegeben, jedoch ist die Ausrichtung je nach Implementierung variabel. Bedenkt man die sich ausbreitende Natur des *RRT*, so ist es sinnvoll, die Orientierung des neuen Zustandes z_b so zu wählen, dass sie vom Vaterknoten z_a weg zeigt (siehe Abb. 3.7). Dies hat ebenfalls den Vorteil, dass der Lenkwinkel, welcher im Punkt \mathbf{p}_b gewählt wird, genau 0 ist, wodurch eines der zu interpolierenden Pfadstücke eine Gerade ist. Zu erkennen ist dies in Abbildung 3.7. Hier gehört der rote Kreisbogen zu dem Vorgängerknoten z_a und die grüne Strecke zu dem neuen Knoten z_b . Die blaue Linie ist der gesuchte Pfad, welcher durch die Interpolation der beiden Teilstücke entstanden ist.

Die grüne Strecke $\mathbf{p}_l(t)$ ist gegeben als Verbindungslinie zwischen \mathbf{p}_a und \mathbf{p}_b , wodurch sie automatisch die Bedingungen erfüllt, sowohl durch \mathbf{p}_a , als auch durch \mathbf{p}_b zu

mit dem ($\hat{=}$ Rechtskurve) oder gegen den Uhrzeigersinn ($\hat{=}$ Linkskurve) gefahren wird:

$$z_{cross} = \mathbf{d}_{ab \cdot x} \cdot \mathbf{d}_a \cdot y - \mathbf{d}_{ab \cdot y} \cdot \mathbf{d}_a \cdot x \quad (3.21)$$

Daraus ergibt sich die Fahrtrichtung:

$$\text{Fahrtrichtung} = \begin{cases} \text{geradeaus} & \text{wenn } z_{cross} = 0 \Leftrightarrow \cos \alpha = 0 \\ \text{Rechtskurve} & \text{wenn } z_{cross} > 0 \\ \text{Linkskurve} & \text{wenn } z_{cross} < 0 \end{cases} \quad (3.22)$$

Womit die Richtung, in welche der Kreismittelpunkt liegt, bestimmt wäre:

$$\mathbf{d}_{aM} = \begin{pmatrix} \sin \Theta \\ -\cos \Theta \end{pmatrix} \cdot \text{sign}(z_{cross}) \quad (3.23)$$

Gesucht ist noch der Radius des Kreises. In der Abbildung erkennt man:

$$\sin \alpha = \frac{\|\mathbf{d}_{ab}\|/2}{r} \quad (3.24)$$

$$\Rightarrow r = \frac{\|\mathbf{d}_{ab}\|}{2 \sin \alpha} \quad (3.25)$$

Damit ergibt sich für den Mittelpunkt M:

$$\mathbf{M} = \mathbf{p}_a + r \cdot \mathbf{d}_{aM} \quad (3.26)$$

Somit ergibt sich für den roten Kreisbogen Folgendes:

$$\beta(t) = \Theta - \text{sign}(z_{cross}) \cdot \frac{\Pi}{2} + t \cdot 2\alpha \cdot \text{sign}(z_{cross}) \quad (3.27)$$

$$\mathbf{p}_c(t) = \mathbf{M} + r \cdot \begin{pmatrix} \cos \beta(t) \\ \sin \beta(t) \end{pmatrix} \quad (3.28)$$

Für den gesuchten interpolierten Pfad ergibt sich nun:

$$\mathbf{p}(t) = (1-t)\mathbf{p}_c(t) + t\mathbf{p}_l(t) \quad \text{mit } t \in [0..1] \quad (3.29)$$

Circular-Circular Interpolation

Das oben beschriebene Szenario ist nur dann möglich, wenn die Ausrichtung des zweiten Knotens z_b frei wählbar ist. Soll diese Interpolation jedoch auch für den *RRT** genutzt werden, so finden sich durch das reWire Situationen, in denen die Orientierung des Zielknotens bereits gegeben ist. In diesem Fall müssen zwei

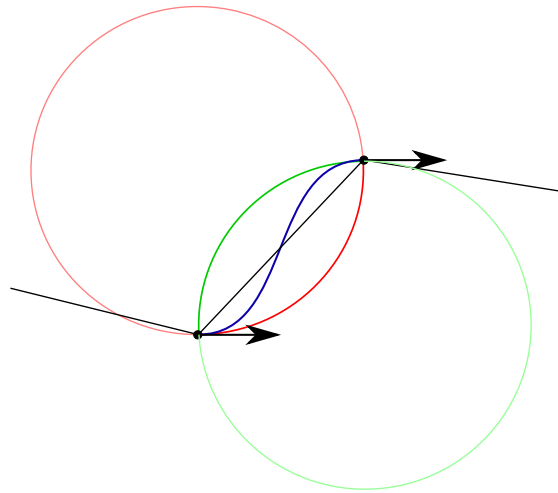


Abbildung 3.8: Circular-Circular Interpolation

Kreisbögen nach oben beschriebenem Vorgehen konstruiert werden, zwischen denen der Zielpfad interpoliert wird (vgl. Abbildung 3.8). Die Konstruktion des zweiten Kreises erfolgt analog zu der des ersten Kreises, lediglich die Ausrichtung \mathbf{d}_b wird zur Berechnung um 180° gedreht. Die Interpolation sieht nun wie folgt aus:

$$\beta_a(t) = \Theta_a - \text{sign}(z_{\text{cross}_a}) \cdot \frac{\Pi}{2} + (1-t) \cdot 2\alpha_a \cdot \text{sign}(z_{\text{cross}_a}) \quad (3.30)$$

$$\beta_b(t) = \Theta_b + \text{sign}(z_{\text{cross}_b}) \cdot \frac{\Pi}{2} + (1-t) \cdot 2\alpha_b \cdot \text{sign}(z_{\text{cross}_b}) \quad (3.31)$$

$$\mathbf{p}_{c_a}(t) = \mathbf{M}_a + r_a \cdot \begin{pmatrix} \cos \beta_a(t) \\ \sin \beta_a(t) \end{pmatrix} \quad (3.32)$$

$$\mathbf{p}_{c_b}(t) = \mathbf{M}_b + r_b \cdot \begin{pmatrix} \cos \beta_b(t) \\ \sin \beta_b(t) \end{pmatrix} \quad (3.33)$$

$$\mathbf{p}(t) = (1-t) \cdot \mathbf{p}_{c_a}(t) + t \cdot \mathbf{p}_{c_b}(t) \text{ mit } t \in [0..1] \quad (3.34)$$

Diese Berechnungen sind zwar etwas aufwändiger, jedoch einfach zu verstehen und daher auch bei Bedarf leicht abzuändern. Des Weiteren dienen sie als Grundlage für eine Pfadglättung, welche in Kapitel 3.3.1 vorgestellt wird.

3.3 Pfadglättung

Die meisten in dieser Arbeit behandelten Algorithmen planen einfache holonome Pfade. Sollen diese jedoch von einem nicht holonomen Fahrzeug ausgeführt werden, so ist es erforderlich die gefundenen Pfade so zu glätten, sodass sie stetig differenzierbar sind. Im Folgenden wird davon ausgegangen, dass der ermittelte Plan lediglich zwei Dimensionen hat, wodurch die Ausrichtung der Knotenpunkte noch frei wählbar ist. Im Rahmen dieser Arbeit wurden hierzu zwei verschiedenen Verfahren implementiert: erstens ein eigener Ansatz, basierend auf der Kreis-Interpolation aus Kapitel 3.2.2, und zweitens die bewährten Catmull-Rom Splines [CR74] zum Vergleichen. Es wurden bewusst Glättungs-Algorithmen gewählt, welche einen Pfad erzeugen, der durch alle Knotenpunkte führt, anstatt an diesen vorbei, um somit eventuell einen kürzeren und glatteren Pfad zu finden, wie beispielsweise bei den B-Splines. Dies ist notwendig, weil die Planungsalgorithmen einen möglichst optimalen Pfad planen, welcher an Hindernissen knapp vorbei führt. Jedes Abkürzen einer Kurve würde somit zu einer Kollision mit dem Hindernis führen, wie in Abbildung 3.9 verdeutlicht wurde.

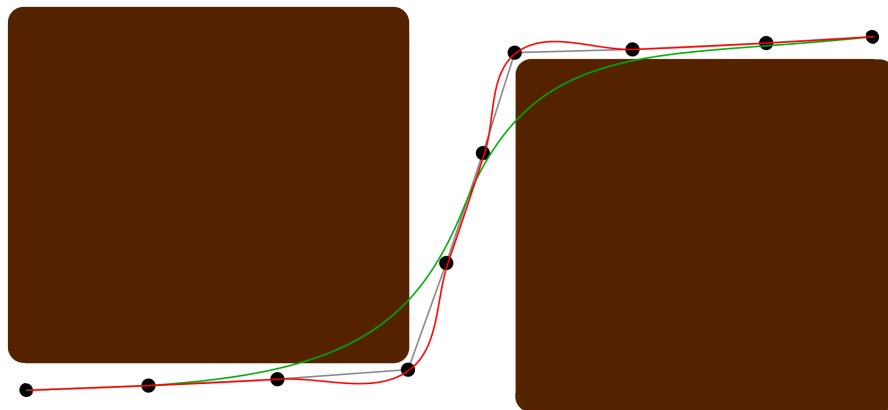


Abbildung 3.9: B-Splines (grün) vs. Catmull-Rom Splines (rot) (braun: Hindernisse, schwarz: ursprünglicher Pfad mit Kontrollpunkten)

3.3.1 Kreis-Interpolation am Pfad

Motiviert durch die Kreis-Interpolation aus Kapitel 3.2.2 wurde im Kontext dieser Arbeit ein Glättungsalgorithmus entwickelt, welcher einen stetig differenzierbaren Pfad durch gegebene Kontrollpunkte konstruiert. Bei einem Pfad, bestehend aus einer Reihe von Kontrollpunkten, bekommt dabei jeder dieser Punkte einen Kreis zugewiesen, welcher sowohl durch diesen Punkt, als auch durch seinen Vorgänger- und Nachfolgerpunkt geht. Anschließend wird ein Pfad mittels Interpolation dieser

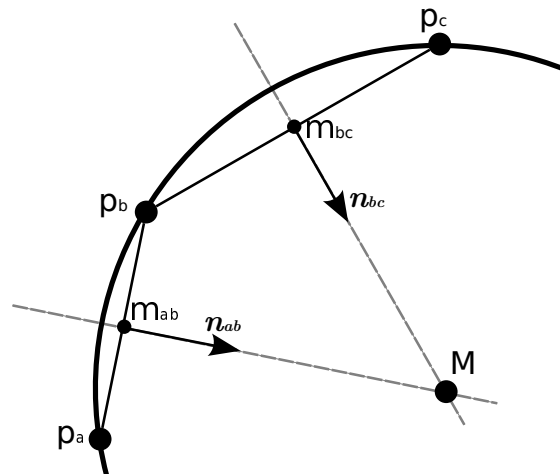


Abbildung 3.10: Kreis-Konstruktion für die Pfad-Interpolation pt. 1

Kreise ermittelt, sodass der Pfad in jedem Kontrollpunkt genau die entsprechende Kreisbahn annimmt. Durch dieses Vorgehen wird jedem Knotenpunkt nicht nur eine Orientierung, sondern auch ein Lenkwinkel zugewiesen, welcher bestmöglich zu den benachbarten Punkten passt. Lediglich der Start- und der Zielknoten brauchen eine vordefinierte Ausrichtung. Deren zugehörigen Kreise werden genau so ermittelt, wie in Kapitel 3.2.2 beschrieben.

Dadurch, dass jedem Knoten ein Kreis durch Vorgänger und Nachfolger zugewiesen ist, besitzt jede Kante des Pfades zwei Kreisbögen, aus denen der gesuchte glatte Pfad durch Interpolation entsteht.

Vorgehen

Zu jedem Kontrollpunkt p_b aus der Reihe (außer Start- und Zielpunkt) wird ein Kreis mit Mittelpunkt M und Radius r konstruiert, welcher sowohl durch p_b , als auch den zugehörigen Vorgängerpunkt p_a und den Nachfolgebepunkt p_c geht (siehe Abbildung 3.10). Hierzu werden die zwei Mittelsenkrechten n_{ab} und n_{bc} der Verbindungsstrecken zwischen den einzelnen Punkten p_a , p_b und p_c ermittelt. Deren Schnittpunkt, welcher durch Gleichsetzung der Geradengleichungen bestimmt wird, ist der Kreismittelpunkt M . Die Ortsvektoren der Mittelsenkrechten sind die Mittelpunkte der jeweils benachbarten Punkte:

$$m_{ab} = \frac{p_a + p_b}{2} \quad (3.35)$$

$$m_{bc} = \frac{p_b + p_c}{2} \quad (3.36)$$

Die Richtungsvektoren der Mittelsenkrechten, werden durch eine 90° -Rotation der Verbindungsstrecken bestimmt:

$$\mathbf{R}_{\frac{\pi}{2}} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad (3.37)$$

$$\mathbf{n}_{ab} = \mathbf{R}_{\frac{\pi}{2}} \cdot (\mathbf{p}_b - \mathbf{p}_a) \quad (3.38)$$

$$\mathbf{n}_{bc} = \mathbf{R}_{\frac{\pi}{2}} \cdot (\mathbf{p}_c - \mathbf{p}_b) \quad (3.39)$$

Nun lässt sich der Schnittpunkt der beiden Geraden mithilfe eines linearen Gleichungssystems ermitteln:

$$\mathbf{M} = m_{ab} + \alpha \cdot \mathbf{n}_{ab} = m_{bc} + \beta \cdot \mathbf{n}_{bc} \quad (3.40)$$

$$\begin{cases} m_{ab} \cdot x + \alpha \cdot \mathbf{n}_{ab} \cdot x = m_{bc} \cdot x + \beta \cdot \mathbf{n}_{bc} \cdot x & (1) \\ m_{ab} \cdot y + \alpha \cdot \mathbf{n}_{ab} \cdot y = m_{bc} \cdot y + \beta \cdot \mathbf{n}_{bc} \cdot y & (2) \end{cases} \quad (3.41)$$

$$(1) \Rightarrow \beta = \frac{m_{ab} \cdot x + \alpha \cdot \mathbf{n}_{ab} \cdot x - m_{bc} \cdot x}{\mathbf{n}_{bc} \cdot x} \quad (3.42)$$

$$\beta \text{ in } (2) \Rightarrow m_{ab} \cdot y + \alpha \cdot \mathbf{n}_{ab} \cdot y = m_{bc} \cdot y + (m_{ab} \cdot x + \alpha \cdot \mathbf{n}_{ab} \cdot x - m_{bc} \cdot x) \cdot \frac{\mathbf{n}_{bc} \cdot y}{\mathbf{n}_{bc} \cdot x} \quad (3.43)$$

$$\Rightarrow \alpha \cdot (\mathbf{n}_{ab} \cdot y - \mathbf{n}_{ab} \cdot x \cdot \frac{\mathbf{n}_{bc} \cdot y}{\mathbf{n}_{bc} \cdot x}) = m_{bc} \cdot y - m_{ab} \cdot y + (m_{ab} \cdot x - m_{bc} \cdot x) \cdot \frac{\mathbf{n}_{bc} \cdot y}{\mathbf{n}_{bc} \cdot x} \quad (3.44)$$

$$\Rightarrow \alpha = \frac{m_{bc} \cdot y - m_{ab} \cdot y + (m_{ab} \cdot x - m_{bc} \cdot x) \cdot \frac{\mathbf{n}_{bc} \cdot y}{\mathbf{n}_{bc} \cdot x}}{(\mathbf{n}_{ab} \cdot y - \mathbf{n}_{ab} \cdot x \cdot \frac{\mathbf{n}_{bc} \cdot y}{\mathbf{n}_{bc} \cdot x})} \quad (3.45)$$

Ist $\mathbf{n}_{bc} \cdot x = 0$, so sollte dies rechtzeitig erkannt werden und der Term vereinfacht sich zu:

$$\alpha = \frac{m_{ab} \cdot x - m_{bc} \cdot x}{-\mathbf{n}_{ab} \cdot x} \quad (3.46)$$

Damit sind für die Bestimmung des Mittelpunkts und des Radius alle Werte bekannt:

$$\mathbf{M} = m_{ab} + \alpha \cdot \mathbf{n}_{ab} \quad (3.47)$$

$$r = \|\mathbf{p}_a - \mathbf{M}\| \quad (3.48)$$

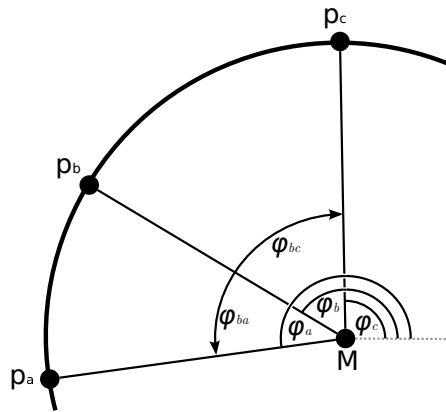


Abbildung 3.11: Kreis-Konstruktion für die Pfad-Interpolation pt. 2

Der dem Punkt p_b zugeordnete Kreis ist somit vollständig bestimmt. Anschließend müssen noch die Winkel sowie die Drehrichtung zur Bestimmung der Kreisbögen definiert werden.

$$\varphi_a = \arctan\left(\frac{p_a.y - M.y}{p_a.x - M.x}\right), \quad \text{analog } \varphi_b, \varphi_c \quad (3.49)$$

Zur Bestimmung der Fahrtrichtung (Drehrichtung) wird getestet ob p_b links oder rechts des Verbindungsvektors von p_a nach p_c liegt, indem der z -Wert z_{cross} des Kreuzproduktes $(p_c - p_a) \times (p_b - p_a)$ kontrolliert wird, wie bereits aus Kapitel 3.2.2 bekannt.

$$z_{cross} = (p_c.x - p_a.x) \cdot (p_b.y - p_a.y) - (p_c.y - p_a.y) \cdot (p_b.x - p_a.x) \quad (3.50)$$

$$\text{Fahrtrichtung} = \begin{cases} \text{Rechtskurve} & \text{wenn } z_{cross} < 0 \\ \text{Linkscurve} & \text{sonst} \end{cases} \quad (3.51)$$

Mit diesem Wissen lassen sich die Kreisbögen-Winkel φ_{ba} und φ_{bc} bestimmen, nachdem dafür gesorgt wurde, dass die Winkel nicht über die $360^\circ \rightarrow 0^\circ$ -Grenze geraten.

```

1 if  $z_{cross} < 0$  then
2   | if  $\varphi_b > \varphi_c$  then  $\varphi_c \leftarrow \varphi_c + 2\pi$ ;
3   | if  $\varphi_b < \varphi_a$  then  $\varphi_a \leftarrow \varphi_a - 2\pi$ ;
4 else
5   | if  $\varphi_b < \varphi_c$  then  $\varphi_c \leftarrow \varphi_c - 2\pi$ ;
6   | if  $\varphi_b > \varphi_a$  then  $\varphi_a \leftarrow \varphi_a + 2\pi$ ;
7 end

```

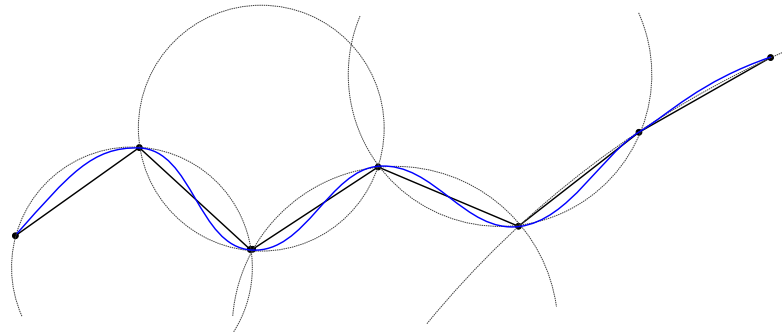


Abbildung 3.12: Interpolierter Pfad mit Interpolationskreisen pro Kontrollpunkt

$$\varphi_{ba} = \varphi_a - \varphi_b \quad (3.52)$$

$$\varphi_{bc} = \varphi_c - \varphi_b \quad (3.53)$$

Damit sind alle Parameter der Kreisbögen zwischen den Punkten gegeben:

$$\mathbf{p}_{b_a}(t) = \mathbf{M} + r \cdot \begin{pmatrix} \cos(\varphi_b + t \cdot \varphi_{ba}) \\ \sin(\varphi_b + t \cdot \varphi_{ba}) \end{pmatrix} \quad (3.54)$$

$$\mathbf{p}_{b_c}(t) = \mathbf{M} + r \cdot \begin{pmatrix} \cos(\varphi_b + t \cdot \varphi_{bc}) \\ \sin(\varphi_b + t \cdot \varphi_{bc}) \end{pmatrix} \quad (3.55)$$

Wird dies für alle Knoten bestimmt, so existieren für jede Kante zwischen zwei Knoten zwei Kreisbögen zwischen denen interpoliert werden kann, um das gesuchte Pfadstück zu erhalten:

$$\mathbf{p}_{ab}(t) = (1 - t) \cdot \mathbf{p}_{a_b}(t) + t \cdot \mathbf{p}_{b_a}(1 - t) \quad (3.56)$$

$$\text{mit } t \in [0..1] \quad (3.57)$$

In Abbildung 3.12 erkennt man einen Ausschnitt eines interpolierten Pfads zusammen mit den jeweiligen Kreisen der Kontrollpunkte.

3.3.2 Vergleich mit Catmull-Rom Splines

Die Catmull-Rom Splines [CR74] sind eine bekannte Variante der Kubisch Hermiteschen Splines. Jedem Kontrollpunkt \mathbf{p}_b wird ein weiterer Vektor \mathbf{d}_b zugewiesen, welcher die halbe Strecke vom Vorgängerpunkt \mathbf{p}_a zum Nachfolgerpunkt \mathbf{p}_c darstellt:

$$\mathbf{d}_b = \frac{\mathbf{c} - \mathbf{a}}{2} \quad (3.58)$$

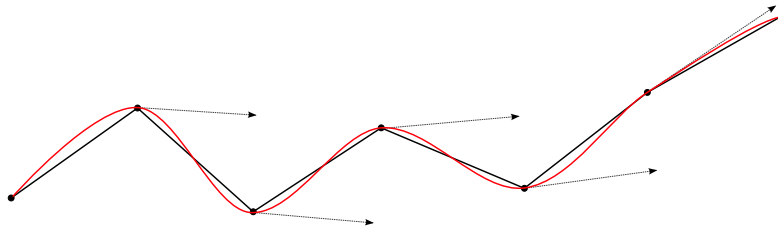


Abbildung 3.13: Catmull-Rom Spline mit Interpolationsvektoren pro Kontrollpunkt

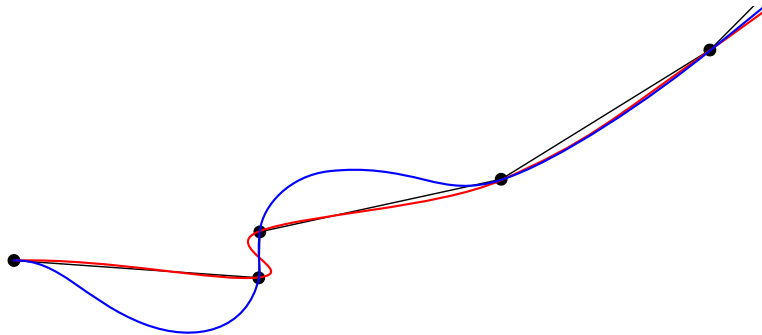


Abbildung 3.14: Kreisinterpolation (blau) vs. Catmull-Rom Spline (rot)

Da dies beim Start- und Zielknoten nicht möglich ist, wird stattdessen ein Vektor definiert, entlang der gegebenen Ausrichtung der Knoten mit der gleichen Länge, wie der Vektor des Nachbarknoten. In Abbildung 3.13 sind diese Vektoren als graue Pfeile zu erkennen. Der Spline zwischen zwei Knoten \mathbf{p}_a und \mathbf{p}_b mit den Vektoren \mathbf{d}_a und \mathbf{d}_b berechnet sich mit folgendem Polynom:

$$\begin{aligned}
 \mathbf{p}_{ab}(t) = & (2t^3 - 3t^2 + 1)\mathbf{p}_a \\
 & + (-2t^3 + 3t^2)\mathbf{p}_b \\
 & + (t^3 - 2t^2 + t)\mathbf{d}_a \\
 & + (t^3 - t^2)\mathbf{d}_b
 \end{aligned} \tag{3.59}$$

Vergleicht man die Catmull-Rom Splines mit dem Pfad der Kreis-Interpolation aus Kapitel 3.3.1, so sehen diese in den meisten Fällen sehr ähnlich aus, wie beispielsweise in den Abbildungen 3.12 und 3.13. Betrachtet man jedoch Situationen wie eine in Abbildung 3.14 zu sehen ist, so erkennt man gravierende Unterschiede. Bei den Catmull-Rom Splines werden die Orientierungen der Knoten lediglich von der Lage der Vorgänger- und Nachfolgerknoten zueinander bestimmt, unabhängig davon wie der mittlere Knoten zu diesen liegt. Bei der Kreis-Interpolation richtet sich die Orientierung jedoch automatisch stärker an dem näher liegenden Nachbarknoten. Hierdurch lassen sich viele unnötige scharfe Kurven vermeiden und der Pfad wird generell etwas geschmeidiger (vgl. Abbildung 3.14). Allerdings kann es,

je nachdem wie die Knoten zueinander stehen, zur Konstruktion von sehr großen Kreisen führen, wodurch der Pfad unnötig weit ausschwenkt, wie ebenfalls in der Grafik zu sehen ist. Hier ist zu überlegen, ob man in dem Algorithmus nur Kreise bis zu einem maximalen Radius zulässt. Diese oder ähnliche Überlegungen konnten im Rahmen dieser Arbeit leider nicht mehr weiter verfolgt werden.

In Kapitel 4.9 findet sich ein Vergleich der Laufzeiten zwischen den beiden Glättungsverfahren.

3.4 Pfadplanungsbibliothek ppLib

Teil dieser Masterarbeit ist die Pfadplanungsbibliothek *ppLib*, welche alle hier beschriebenen Algorithmen beinhaltet. Die Bibliothek ist aus vier Grundbausteinen aufgebaut, welche in Abbildung 3.15 dargestellt werden. Die Hauptklasse ist hierbei der *Planner*, welcher den Planungsalgorithmus ausführt und als Hauptschnittstelle nach außen fungiert. In der *Vehicle*-Klasse finden sich alle fahrzeugabhängigen Parameter und Funktionen, wie beispielsweise die vordefinierten Pfadstücke des *hybrid A** (Kap. 2.6). Hierdurch lässt sich die Bibliothek leicht für mehrere Fahrzeuge konfigurieren. Als Schnittstelle für die Umgebungskarten dient die *DynamicGrid*-Klasse. Sie implementiert ein zweidimensionales Feld aus Zellen, welches in alle Richtungen unbegrenzt ist. Diese Unbegrenztheit ist besonders wichtig für *lifelong*-Planner, welche während des Planens die Karte erweitern. Diese Erweiterung geschieht über die *extend*-Funktion, über die der aktuellste Umgebungs-Scan übergeben und somit an das bereits existierende Grid angeschlossen wird. Der aktuelle Scan muss dabei ebenfalls in ein *DynamicGrid*-exportiert worden sein und den selben Ursprung haben wie das globale Grid. Das bedeutet, dass ein *Scan-Matching* separat durchgeführt werden muss. Das Zusammenschließen der Karten geschieht jedoch in der *ppLib*, um somit die Veränderungen in der Karte gleich so aufzuarbeiten, dass die Planungsalgorithmen diese nutzen können. Die vierte Klasse ist der *PathSmoother*, welcher einerseits die Pfadglättung durchführt und andererseits die Kreis-Interpolations-Schritte aus Kapitel 3.2.2 berechnet. Die *Planner*-Klasse ist eine abstrakte Klasse, welche selbst nur einige Algorithmusübergreifende Funktionen implementiert. Die Algorithmen selber finden sich in den ererbenden Klassen aus Abbildung 3.17. In dieser Arbeit bauen viele Algorithmen aufeinander auf, was in der Bibliothek mittels Vererbung umgesetzt wurde, um somit keinen duplizierten Code zu haben. Hierzu war eine Templateisierung der Knoten notwendig (siehe Abbildung 3.16). Die in Abbildung 3.17 angegebenen Template-Klassen sind dabei lediglich die Klassen von Knoten, welche mindestens benötigt werden. Eine zugehörige *readme*-Datei mit Nutzungsanleitung findet sich im Anhang D.

Ebenfalls Teil dieser Arbeit ist eine graphische Benutzeroberfläche zum Testen und Evaluieren von Planungsalgorithmen. Sie beinhaltet einen Karten-Editor, zu jedem Planungsalgorithmus eigene *render*-Klassen und Simulationen für Fahrzeug und Umgebungs-scanner. Ein Screenshot ist in Abbildung 3.18 zu sehen. Es wird im Rahmen dieser Ausarbeiten jedoch nicht weiter auf sei eingegangen.

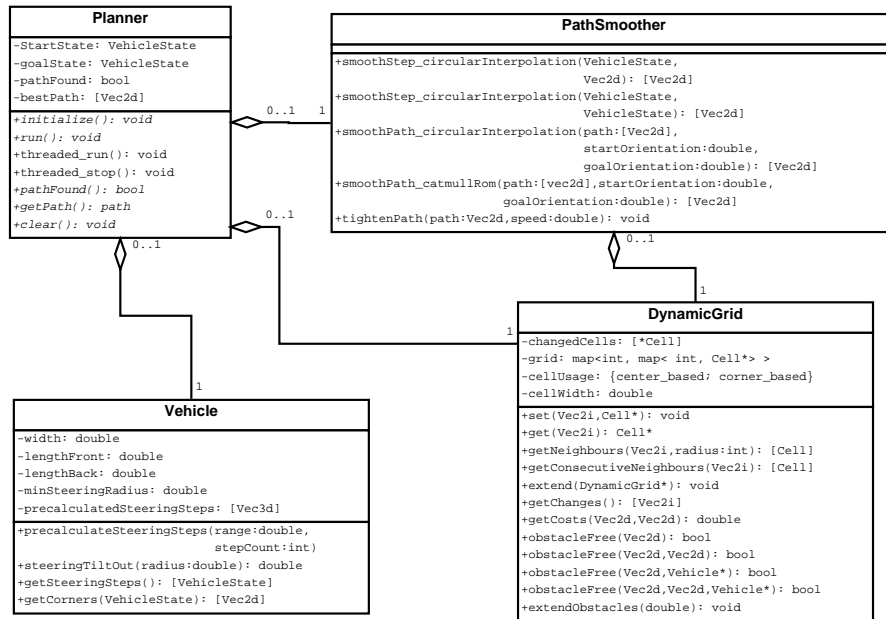


Abbildung 3.15: Planungsbibliothek Grundgerüst

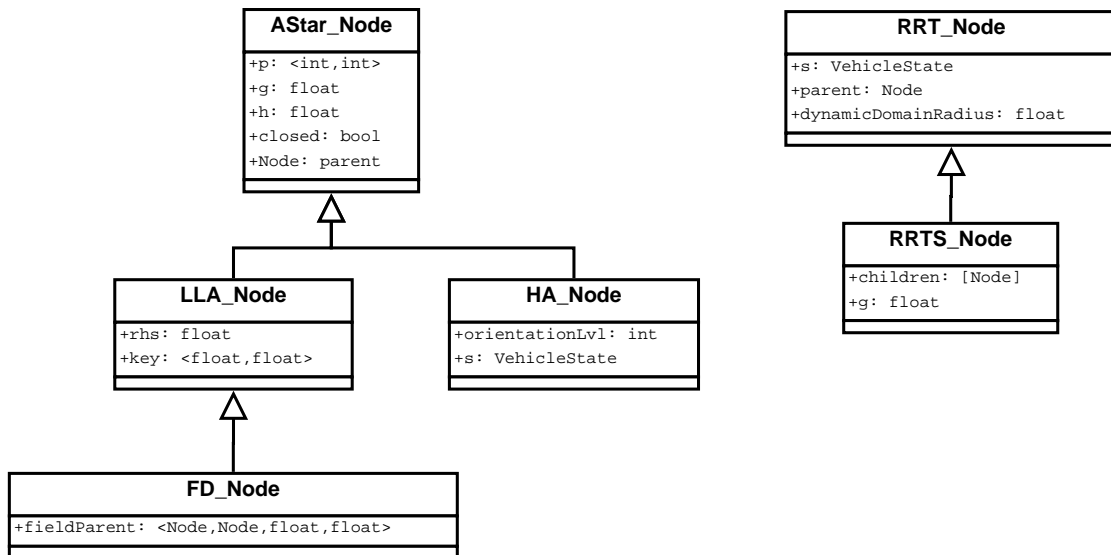


Abbildung 3.16: Knoten der Planungsalgorithmengraphen

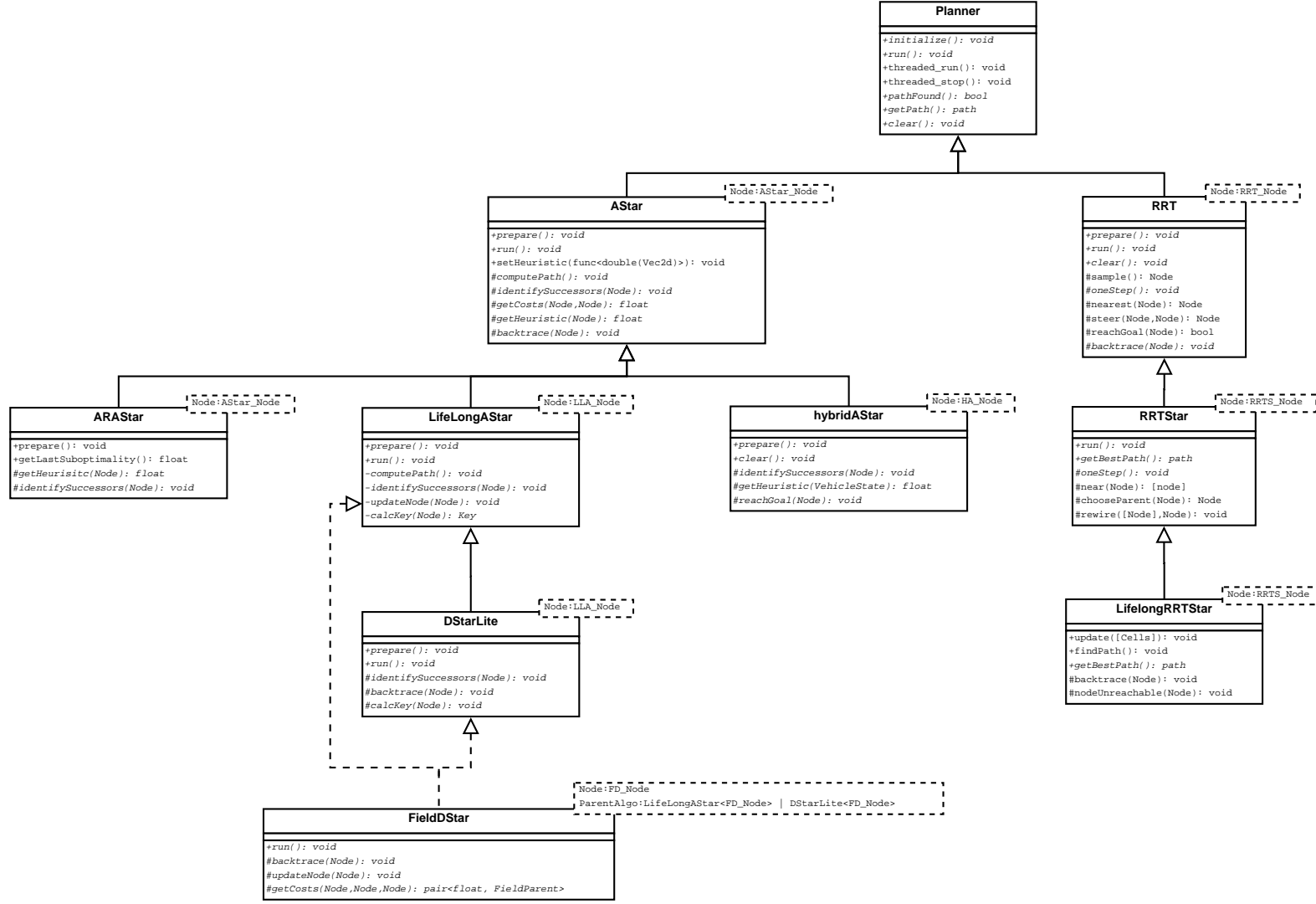


Abbildung 3.17: Planungsalgorithmen

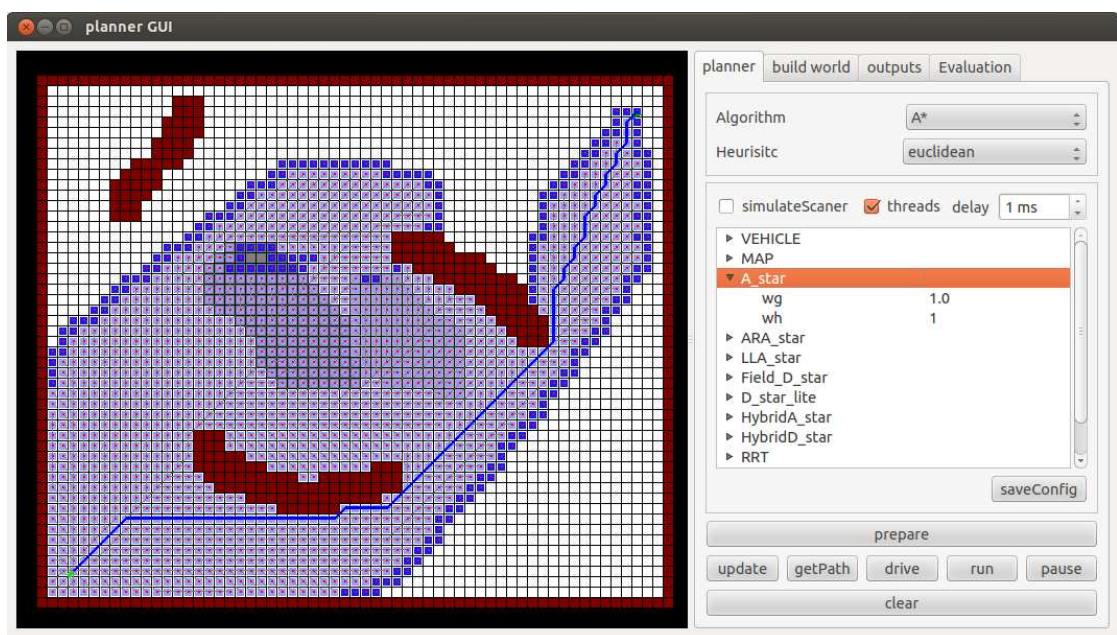


Abbildung 3.18: Graphische Benutzeroberfläche

Kapitel 4

Evaluation

4.1 Vergleich ARA^* mit A^*

Der folgende Abschnitt befasst sich mit dem Vergleich des ARA^* (Kap. 2.1) mit dem A^* (Kap. 2.2), um zu ermitteln wie groß der tatsächliche Gewinn des *anytime*-Algorithmus ist. Getestet wurde hierzu auf einer so entworfenen Karte (Anhang: C.2), dass je nach Heuristikgewicht ein unterschiedlicher Pfad gefunden wird. Ein ARA^* wurde mit $\epsilon_{max} = 2.5$ und $\Delta\epsilon = 0.2$ gestartet. Parallel dazu wurde in jeder Iteration des ARA^* ein A^* in der gleichen Umgebung laufen gelassen, mit gleichem Heuristikgewicht wie der ARA^* in dieser Runde: $w_h = \epsilon$.

Der komplette Ablauf des Experiments ist dabei in Abbildung 4.4 zu sehen. Interessant ist die Beobachtung des gefundenen Pfades (blaue Linie) und der in dieser Runde bearbeiteten Knoten (hellblau). Es wird gewahr, dass der A^* in jeder Runde alle Knoten neu berechnen muss, der ARA^* kann hingegen auf einige Berechnungen verzichten, da er das Wissen der Vorrunde nutzt.

In 4.1 sieht man die Anzahl der Knoten des jeweiligen Graphen. Beim ARA^* ist hierbei die Gesamtanzahl (hellblau) von der Anzahl der in dieser Runde neu berechneten Knoten (dunkelblau) zu unterscheiden. Die Gesamtanzahl korreliert hierbei mit dem Speicherbedarf, die Anzahl der berechneten Knoten mit der Laufzeit. Die Grafik verdeutlicht, dass beide Algorithmen gleich starten, da der erste ARA^* -Durchlauf identisch einem A^* -Durchlauf ist. In 4.1 erkennt man überraschend, dass der ARA^* -Graph teilweise kleiner ist, als der von A^* . Dies kann vorkommen, da bereits ein Pfad aus vorherigen Runden bekannt ist und dessen Kosten als obere Grenze der bearbeiteten Knoten genommen wird. Das bedeutet, dass alle Knoten deren f-Wert größer ist als diese Kosten, nicht mehr bearbeitet werden. Der A^* hat hingegen gegebenenfalls zu diesem Zeitpunkt noch keinen Zielpfad gefunden, wodurch diese obere Schranke wegfällt und der Graph somit etwas breiter wird.

In der Abbildung 4.2 wird jedoch ein Nachteil dieses vorzeitigen Abbrechens des

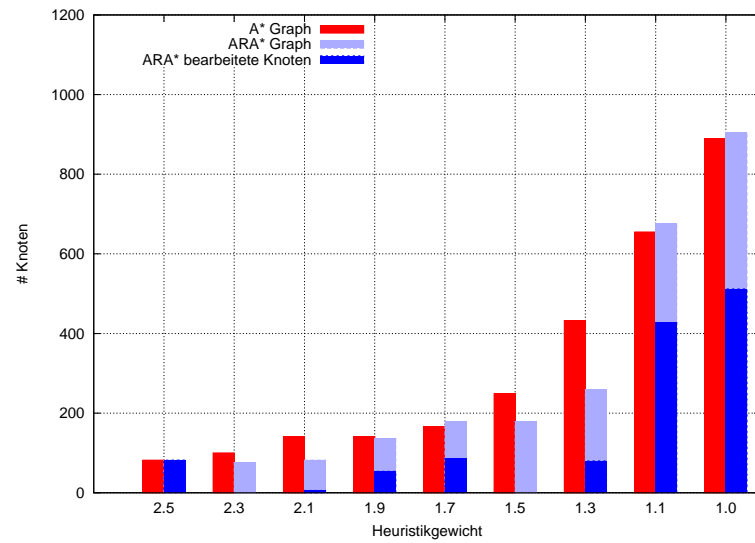


Abbildung 4.1: Der direkte Vergleich der Graphgröße zwischen A^* und ARA^*

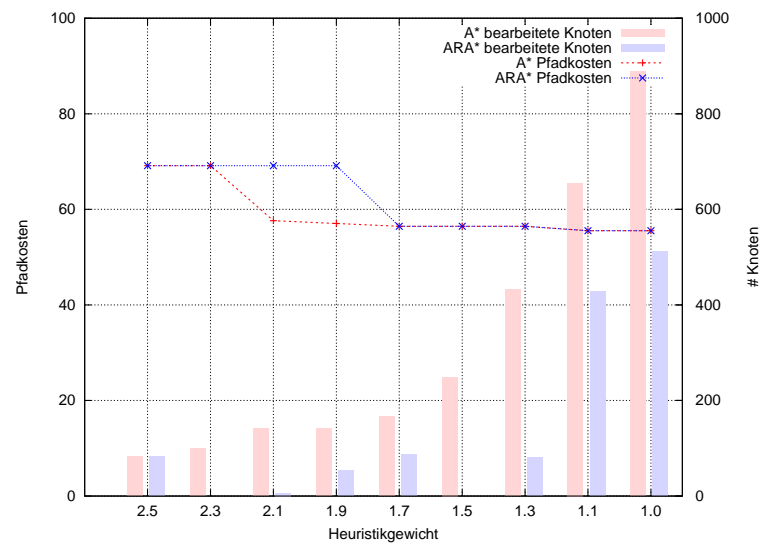


Abbildung 4.2: Der direkte Vergleich der Pfadlänge zwischen A^* und ARA^*

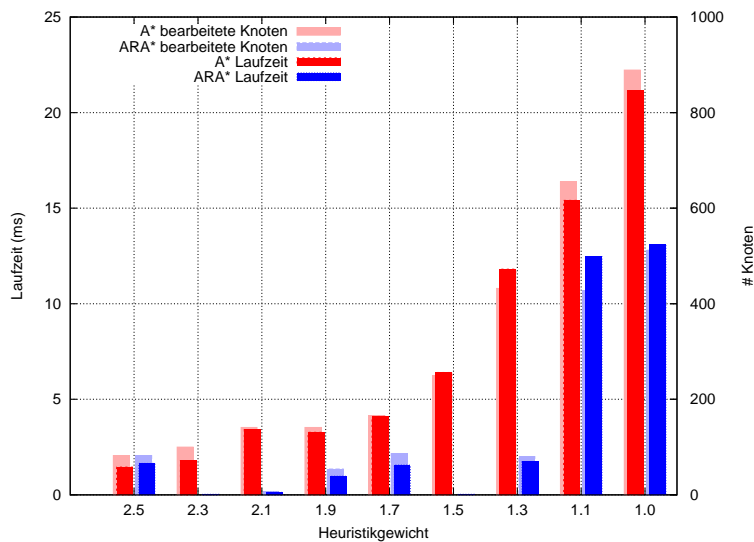


Abbildung 4.3: Der direkte Laufzeitvergleich zwischen A^* und ARA^*

ARA^* deutlich. Diese Grafik visualisiert ist die Länge des ermittelten Pfades der jeweiligen Runde. Überraschenderweise ist der Pfad des A^* in einigen Iterationen kürzer, als der des ARA^* . Dies ist dem Zustand zu verdanken, dass die Heuristik durch die Gewichtung mittels ϵ und w_h nicht mehr konsistent ist. Der ARA^* stoppt das Planen, sobald der kleinste f-Wert ($f = g + h$) größer ist als die Kosten des aktuellen Zielpfades. Er nimmt an, dass der Pfad selbst mit dem besten Knoten nicht mehr zu optimieren ist. Durch die Gewichtung ist die Heuristik jedoch nicht mehr zwangsläufig unterschätzend, wodurch auch der f-Wert überschätzend sein kann. Dies wiederum bedeutet, dass doch ein Zielpfad über den gegebenen Knoten gefunden werden kann, mit kleineren Kosten als dem entsprechenden f-Wert. Daher stoppt der ARA^* unter dieser falschen Annahme je nach Situation zu früh, wohingegen der A^* weiter plant, da ihm die Pfadkosten der Vorrunde als obere Schranke fehlen. Folglich kann es vorkommen, dass der A^* trotz gleicher Heuristik einen kürzeren Pfad findet. Dies zeigt, dass das Nutzen des Wissens der Vorrunde nicht immer nur Vorteile bringt.

In Abbildung 4.3 ist die Laufzeit der beiden Algorithmen gegenübergestellt. Legt man die Anzahl der in dieser Runde berechneten Knoten hinter die Grafik (blasse Balken), so ist zu erkennen, dass die Laufzeit pro Knoten konstant und bei beiden Algorithmen abgesehen von kleinen Schwankungen annähernd gleich ist. In diesem Fall benötigt der ARA^* im Mittel 0.0221 ms pro Knoten und der A^* 0.0231 ms.

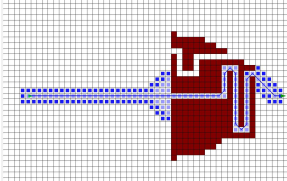
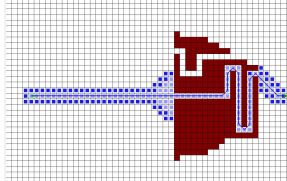
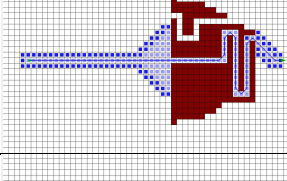
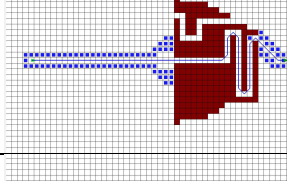
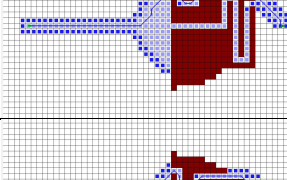
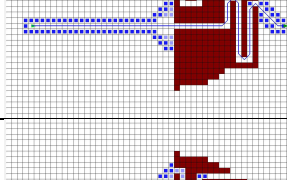
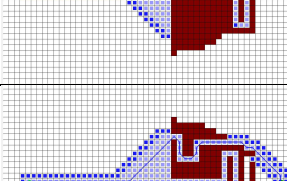
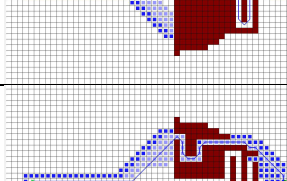
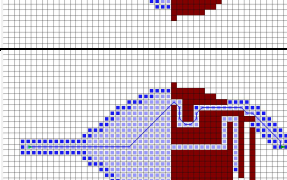
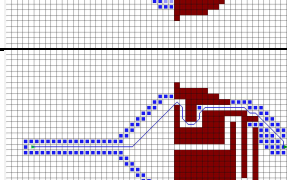
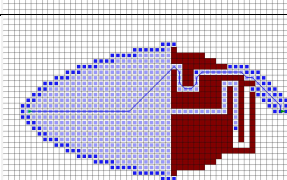
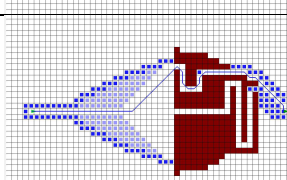
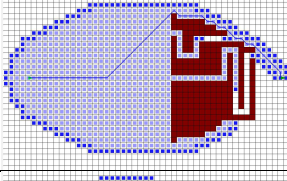
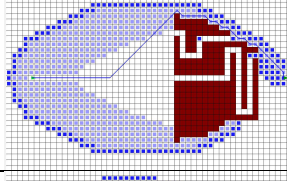
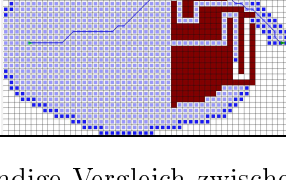
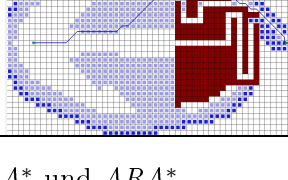


			A^*	ARA^*
$w_h, \epsilon = 2.5$	A^*	ARA^*		
# Knoten	82	82		
Pfadlänge	69.1421	69.1421		
$w_h, \epsilon = 2.3$	A^*	ARA^*		
# Knoten	100	0		
Pfadlänge	69.1421	69.1421		
$w_h, \epsilon = 2.1$	A^*	ARA^*		
# Knoten	141	6		
Pfadlänge	57.6274	69.1421		
$w_h, \epsilon = 1.9$	A^*	ARA^*		
# Knoten	141	54		
Pfadlänge	57.0416	69.1421		
$w_h, \epsilon = 1.7$	A^*	ARA^*		
# Knoten	166	86		
Pfadlänge	56.4558	56.4558		
$w_h, \epsilon = 1.5$	A^*	ARA^*		
# Knoten	249	0		
Pfadlänge	56.4558	56.4558		
$w_h, \epsilon = 1.3$	A^*	ARA^*		
# Knoten	432	80		
Pfadlänge	56.4558	56.4558		
$w_h, \epsilon = 1.1$	A^*	ARA^*		
# Knoten	655	427		
Pfadlänge	55.5269	55.5269		
$w_h, \epsilon = 1.0$	A^*	ARA^*		
# Knoten	899	512		
Pfadlänge	55.5269	55.5269		

Abbildung 4.4: Der vollständige Vergleich zwischen A^* und ARA^*

4.2 A^* , LPA^* und D^* Lite Laufzeitmessung

In recht großem Terrain von 200×200 Zellen (Anhang C.5), wurden der A^* und LPA^* 100 mal ausgeführt und deren Laufzeiten gemittelt. Durch das etwas abgeänderte Abbruchkriterium des LPA^* ist der Graph hier etwas größer, als der des A^* . Der hierbei ermittelte kürzeste Pfad ist bei beiden Algorithmen der gleiche (Länge: 490.132; Kosten: 524.603). Betrachtet man jedoch die Laufzeit, welche die Algorithmen pro Knoten benötigen, so zeigt sich, dass der LPA^* fast acht mal so lange braucht wie der ursprüngliche A^* .

Dieser enorme Unterschied zwischen A^* und LPA^* lässt sich dadurch ausgleichen, dass der LPA^* bei einer Reihe von Berechnungen, den vorherigen Graphen als Vorwissen nutzen kann, und somit weniger Knoten neu berechnen muss. Diese Einsparung an Berechnungen lässt sich jedoch schlecht evaluieren, da es sehr umgebungsabhängig ist und sich somit keine allgemeine Aussage machen lässt.

Wie bei dem LPA^* wurde auch für den D^* Lite die Zeit gemessen. In der gleichen Umgebung (Anhang C.5) wurde auch der D^* Lite 100 mal gestartet. Der D^* Lite lief jedoch rückwärts, damit die Graphgröße mit dem A^* und LPA^* vergleichbar ist. Wie zu erwarten unterscheiden sich die Zeiten des D^* Lite und des LPA^* kaum, da sie der selben Expansionsstrategie folgen.

Zu beachten ist, dass die Zeit pro Knoten in Abbildung 4.5 berechnet wurden, indem die Laufzeit durch die Anzahl der Knoten geteilt wurde. Somit werden auch die Vor- und Nachbereitsungszeiten, wie zum Beispiel das Zurückverfolgen des Pfades, mit einberechnet. Diese sind jedoch bei einer genügend großen Anzahl an Knoten im Graph verschwindend gering und konnten daher hier ignoriert werden.

	Laufzeit	# Knoten	Zeit / Knoten
A^*	344.84 ms	26484	0.0130 ms
LPA^*	2698.10 ms	26520	0.1017 ms
D^* Lite	2721.98 ms	26520	0.1026 ms

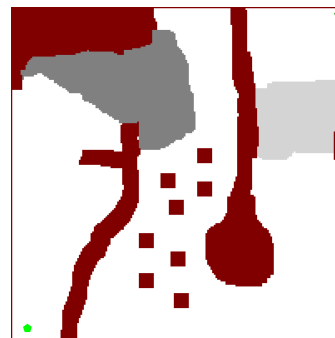


Abbildung 4.5: Laufzeitvergleich zwischen A^* , LPA^* und D^* Lite

4.3 Graphgrößenvergleich zwischen repeated A^* und D^* Lite

Wie bereits beschrieben, ist es schwer die tatsächliche Laufzeit zwischen dem A^* und dem D^* Lite zu vergleichen. Die Laufzeit pro Knoten ist beim D^* Lite (ähnliche wie beim LPA^*) deutlich höher als beim normalen A^* , allerdings ist diese nicht so sehr aussagekräftig, da die Anzahl der berechneten Knoten meistens geringer ist. Dies variiert jedoch abhängig von der Umgebung sehr stark. Im Folgenden wurde dies trotzdem an einer dynamischen Szene beispielhaft verglichen, deren Ablauf in Abbildung C.6 veranschaulicht wird. In der Umgebung, welche in 4.5 zu sehen ist, wurde eine Fahrt in 11 Schritten vom Startpunkt bis zum Ziel simuliert. Es wurde eine beschränkte Sichtweite des Vehikels simuliert, sodass in jeder Runde neue Hindernisse entdeckt wurden. Ebenfalls bewegten sich andere Hindernisse, wodurch vormals blockierte Gebiete wieder frei wurden. Zum Vergleichen wurde einmal der D^* Lite mit dem Vorwissen aus den Vorrunden gestartet und parallel von der gleichen Position aus der A^* . Dieser lief jedoch rückwärts, um somit die Graphen besser vergleichen zu können

Die Ergebnisse dazu werden in Abbildung 4.6 aufgezeigt. Die roten Balken zeigen die Größe des jedes mal neu gestarteten A^* -Graphen an, die blauen Balken die Größe des Graphen, welcher mit dem D^* Lite aufgebaut wurde. Hierbei ist der dunkle Bereich, genau die Anzahl der Neuberechnungen. Wie bereits beschrieben, sind die Graph-Größen beim initialen Durchgang gleich.

Fazit ist, dass der D^* Lite hier in der Regel, trotz einer sehr großen Schrittweite

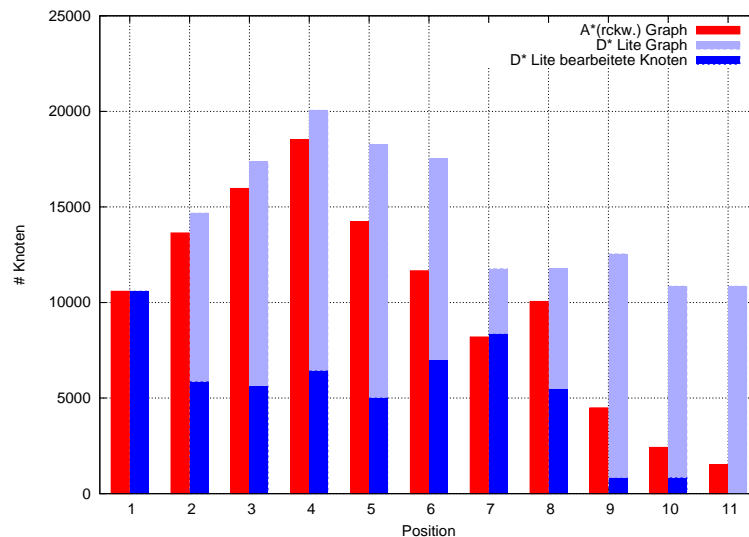


Abbildung 4.6: Größenvergleich des Graphen des D^* Lite mit dem A^* -Graphen

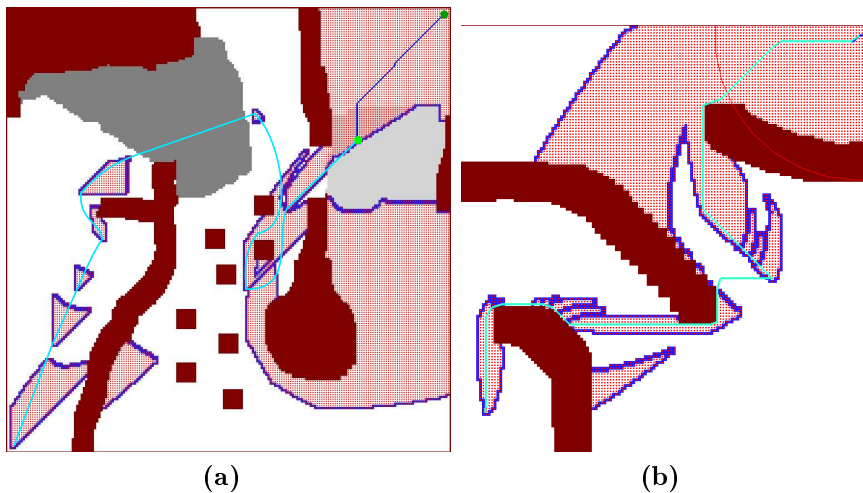


Abbildung 4.7: Graph-Reste (rot: Knoten im Graph; dunkelblau: *OpenList*) bei einer Fahrt mittels $D^* Lite$. Die dunkelblaue Strecke zeigt den geplanten, noch zu fahrenden Weg an, die hellblaue Strecke den bereits zurückgelegten Weg

und somit großen Änderungen in der Umgebung, sehr viel weniger Knoten neu berechnen muss als der A^* . Jedoch kann es in gewissen Situationen auch passieren, dass der $D^* Lite$ sogar mehr berechnen muss, als der A^* (Abbildung 4.6 an der Position 7). Dies ist darauf zurückzuführen, dass der $D^* Lite$ erst den alten Graphen zurück und dann die neuen Teile wieder neu aufbauen muss, somit viele Knoten zweimal „angepackt“ werden, wie bereits in Kapitel ?? beschrieben. Im Gegensatz dazu kann es jedoch auch passieren, dass der $D^* Lite$ gar nicht expandieren muss, wenn das hinzugekommene Weltwissen den bis dahin kürzesten Pfad nicht ändert und die neue Fahrzeugposition innerhalb des alten Graphen liegt. Der normale A^* muss hingegen den kompletten Graph neu aufbauen. Ein solche Situation findet sich an der Position 11.

Betrachtet man die Gesamtgröße des $D^* Lite$ -Graphen und nicht nur die Anzahl der Neuberechnungen, so fällt ins Auge, dass diese stets größer gleich der Größe des A^* -Graphen ist. Dies liegt daran, dass der $D^* Lite$ nicht immer alle veralteten Knoten abbaut. Er macht dies nur so lange, wie es relevant für den aktuellen Pfad ist. Sobald die geringsten f -Werte größer sind als die Zielpfadkosten, bricht er ab, da kein neuer Knoten den Zielpfad noch optimieren kann. Dies führt dazu, dass oft „Reste“ des Vorgänger-Graphen bleiben, welche in Abbildung 4.7 (a) zu erkennen sind. In (b) sind Reste zu sehen, welche mit einer kleineren Schrittfolge in einer anderen Umgebung geblieben sind. Soll der Algorithmus sehr lange Strecken in großen Umgebungen fahren, so ist zu überlegen den Algorithmus so zu erweitern, dass diese Reste gelöscht werden, da sie lediglich den Speicher füllen, jedoch keinerlei brauchbare Informationen enthalten.

4.4 Field D^* Evaluation

4.4.1 Pfadkosteneinsparung des *Field D^**

Für eine Evaluation des *Field D^** interessiert besonders der direkte Vergleich mit dem unveränderten *D^* Lite* hinsichtlich der eingesparten Pfadkosten. Hierzu wird gezeigt, wie viel Wegstrecke in einem Grid mit Achter-Nachbarschaft maximal eingespart werden kann. Außerdem wurde eine Testreihe gestartet, um zu ermitteln wie groß der Gewinn in der Praxis in etwa ist. Die Testreihe fand in dem gleichen Szenario statt, welches schon aus Kapitel 4.3 bekannt ist (Anhang C.6). Auf der x-Achse in Abbildung 4.8 sind die einzelnen Positionen im Szenario aufgelistet und auf der y-Achse sind die ermittelten Pfadkosten des *D^* Lite* und des *Field D^** abgetragen. Gemittelt über alle Positionen ist hierbei der Pfad des *D^* Lite* 5.43% länger als der interpolierte Pfad des *Field D^** .

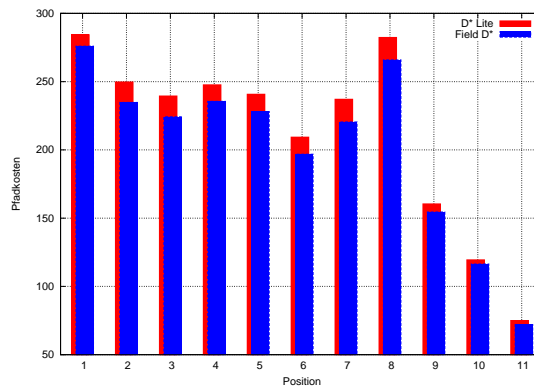


Abbildung 4.8: Vergleich der Pfadlängen zwischen *D^* Lite* und *Field D^** in der Szene aus C.6

Maximale Einsparung

Im Folgenden soll bestimmt werden, um wie viel kürzer der interpolierte Pfad maximal werden kann, im Vergleich zu einem Knoten-gebundenen Pfad, welcher durch den normalen A^* oder *D^* Lite* bestimmt wurde. Hierzu lässt sich ein Pfad aufteilen in Teilstücke zwischen Knoten, welche sowohl vom interpolierten Pfad als auch vom normalen Pfad geschnitten werden. In Abbildung 4.9 (a) sind dies die Knotenpunkte \mathbf{p}_s und \mathbf{p}_z . Die Dreiecksungleichung besagt:

$$\mathbf{s}_F \leq \mathbf{s}_1 + \mathbf{s}_2 \quad (4.1)$$

Hierbei ist \mathbf{s}_F die Länge der mittels Interpolation gefundenen direkten Verbindung der Knoten und $\mathbf{s}_1 + \mathbf{s}_2$ die Länge des Pfades entlang der Knoten-Nachbarschaften.

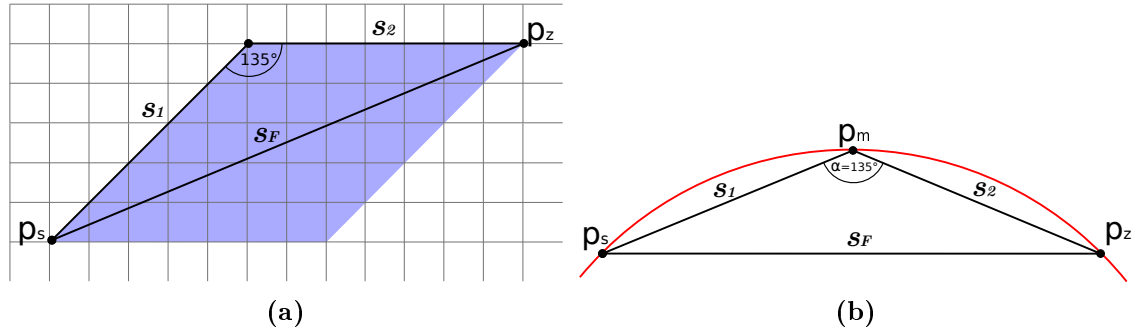


Abbildung 4.9: Maximale Pfadverkürzung durch die Field-Kostenrechnung

Außer dem aus \mathbf{s}_1 und \mathbf{s}_2 bestehenden Pfad, gibt es noch weitere kürzeste Verbindungen zwischen \mathbf{p}_s und \mathbf{p}_z , welche nur entlang der Nachbarschaften führen. Diese befinden sich alle vollständig innerhalb des blau markierten Bereichs in Abbildung 4.9 (a) und haben exakt die gleiche Länge wie \mathbf{s}_1 und \mathbf{s}_2 zusammen. Da garantiert einer dieser kürzesten Pfade gültig ist, wenn auch \mathbf{s}_F nicht blockiert wird, kann hier o.B.d.A. mit $\mathbf{s}_1 + \mathbf{s}_2$ als kürzester knotengebundener Pfad ausgegangen werden. Durch die Achter-Nachbarschaft ist gegeben, dass der Winkel α zwischen beiden Teilstrecken zu \mathbf{s}_1 und \mathbf{s}_2 immer 135° ist. Dies bedeutet, dass sich zu jedem Teilstück \mathbf{s}_F des Interpolierten Pfades zwischen zwei Knotenpunkten \mathbf{p}_s und \mathbf{p}_z , ein Punkt \mathbf{p}_m findet, über welchen der kürzeste nachbarschaftsgebundene Pfad geht. Dabei kann \mathbf{p}_m ein beliebiger Punkt auf dem in Abbildung 4.9 (b) rot eingezeichneten Kreis sein, da hier der Winkel α stets 135° bleibt. Ohne Beweis, aber dank der Skizze leicht nachzuvollziehen, ist die Strecke $\|\mathbf{s}_1\| + \|\mathbf{s}_2\|$ genau dann am längsten, wenn \mathbf{p}_m genau mittig zwischen \mathbf{p}_s und \mathbf{p}_z liegt und somit $\|\mathbf{s}_1\| = \|\mathbf{s}_2\|$ gilt. Es lässt sich das Dreieck $\triangle \mathbf{p}_s \mathbf{p}_m \mathbf{p}_z$ in zwei symmetrische Dreiecke aufteilen und es ergibt sich als obere Schranke Folgendes:

$$\sin\left(\frac{\alpha}{2}\right) = \frac{\|\mathbf{s}_F\|/2}{\|\mathbf{s}_1\|} \quad (4.2)$$

$$\|\mathbf{s}_1\| = \frac{\|\mathbf{s}_F\|}{2 \cdot \sin\left(\frac{\alpha}{2}\right)} \quad (4.3)$$

$$\|\mathbf{s}_1\| + \|\mathbf{s}_2\| = 2 \cdot \|\mathbf{s}_1\| = \frac{\|\mathbf{s}_F\|}{\sin\left(\frac{\alpha}{2}\right)} = \frac{\|\mathbf{s}_F\|}{\sin\left(\frac{135^\circ}{2}\right)} \quad (4.4)$$

$$\|\mathbf{s}_1\| + \|\mathbf{s}_2\| = \frac{1}{\sin\left(\frac{135^\circ}{2}\right)} \cdot \|\mathbf{s}_F\| = 1.0823922 \cdot \|\mathbf{s}_F\| \quad (4.5)$$

Das führt zu der Erkenntnis, dass der den Knotenpunkte entlang führende Pfad maximal 8.24% länger ist, als der kürzeste direkte Pfad. Dies gilt nicht nur für

einzelne Teilstücke, sondern lässt sich auch als obere Schranke für den kompletten Pfad sehen.

4.4.2 Mehrfachoptimierung durch das Zwei-Eltern Dilemma

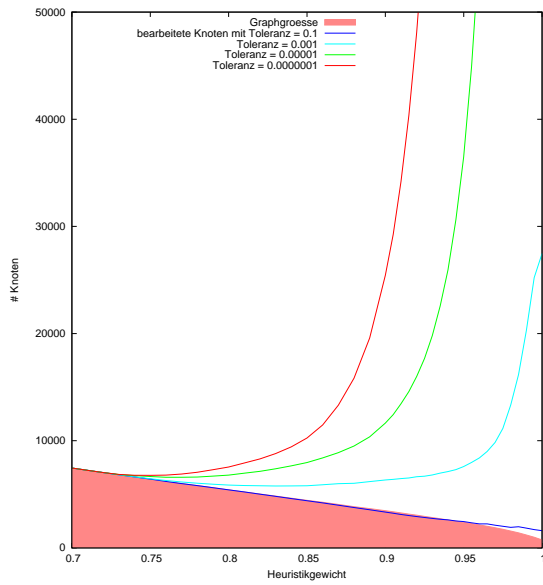
In Kapitel 2.5.4 wurde bereits ausgiebig über das Problem der Mehrfachaktualisierungen der einzelnen Knoten im *Field D** informiert. Im folgenden Abschnitt wird gemessen, wie ausgeprägt die Folgeerscheinungen des Problems tatsächlich sind und wie gut die Methoden der Heuristik-Reduzierung und der Toleranz-Einführung tatsächlich helfen.

Hierzu wurde der *Field D** mehrfach auf einer komplett freien Karte ausgeführt, wobei der Start- und Zielpunkt in zwei gegenüberliegenden Ecken liegen. Nun wurde abhängig von der Toleranz und vom Heuristikgewicht die Anzahl der Knoten Neuberechnungen gezählt. Die Ergebnisse werden in Abbildung 4.10 gezeigt.

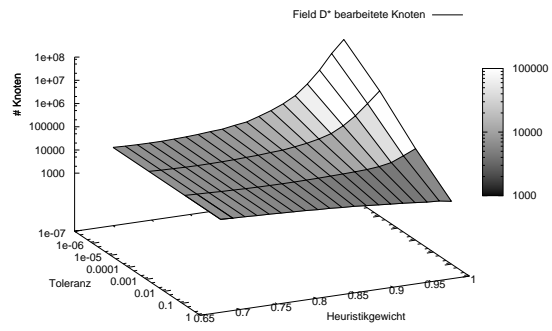
Wie bereits bekannt, sorgt eine Reduzierung des Heuristikgewichts für eine Vergrößerung des Graphen, sichtbar an dem rot markierten Bereich in (a). Dieser Bereich steht dabei auch für den Speicherbedarf, wohingegen die einzelnen Linien die Laufzeit verdeutlichen. Der Bereich zwischen dem roten Gebiet und der entsprechenden Linie, zeigt hierbei die Anzahl der ungewollten Mehrfachberechnungen auf. Ein normaler *D* Lite* ohne Toleranz würde hierbei eine Linie entlang der oberen Kante des roten Gebietes bilden. Zu erkennen ist, dass es durch die Einführung der Toleranz annähernd möglich ist, dieses Optimum zu erreichen, wenn dieser groß genug gewählt ist, was jedoch zu kleinen Einbußen in der Optimalität des Pfades führt.

Je kleiner das Heuristikgewicht ist, desto geringer kann auch die Toleranz gewählt werden, was jedoch zu einem größeren Graphen und somit zu einem höheren Speicherbedarf führt.

Generell lässt sich schlussfolgern, dass die Auswirkungen nicht vernachlässigbar sind. Bevor der Algorithmus in der Praxis eingesetzt wird, wäre es notwendig eine komplette Lösung des Problems zu finden, anstatt nur die Hilfsmittel zur Symptomreduzierung anzuwenden.



(a)



(b)

Abbildung 4.10: Mehrfachberechnung der Knoten bei der Field-Heuristik

4.5 Vergleich hybrider A^* mit A^*

Der *hybrid A^** hat im Vergleich zum A^* den Vorteil, dass er direkt ausführbare Pfade bestimmt, welche nicht-holonome Einschränkungen einhalten, aber dennoch ist ein zeitlicher Vergleich der beiden Algorithmen interessant. In der Karte, welche im Anhang C.1 zu sehen ist, wurde der *hybrid A^** mit fünf verschiedenen vordefinierten Pfadstücken, einer Schrittweite von 4 Zellen und einer Winkelauflösung von $\frac{2\pi}{100}$ gestartet. Die Messergebnisse, welche aus 100 Testrunden gemittelt wurden, sind in Abbildung 4.11 nachzulesen. Dadurch, dass der *hybrid A^** im Vergleich zum A^* in drei Dimensionen plant, benötigt er wesentlich mehr Knoten, um das Ziel zu finden, woraus auch in erster Linie die wesentlich höhere Laufzeit resultiert. Für einen aussagekräftigeren Vergleich würde ein A^* benötigt, welcher ebenfalls in drei Dimensionen plant. Ein solcher ist jedoch nicht Teil dieser Arbeit und stand daher nicht zur Verfügung.

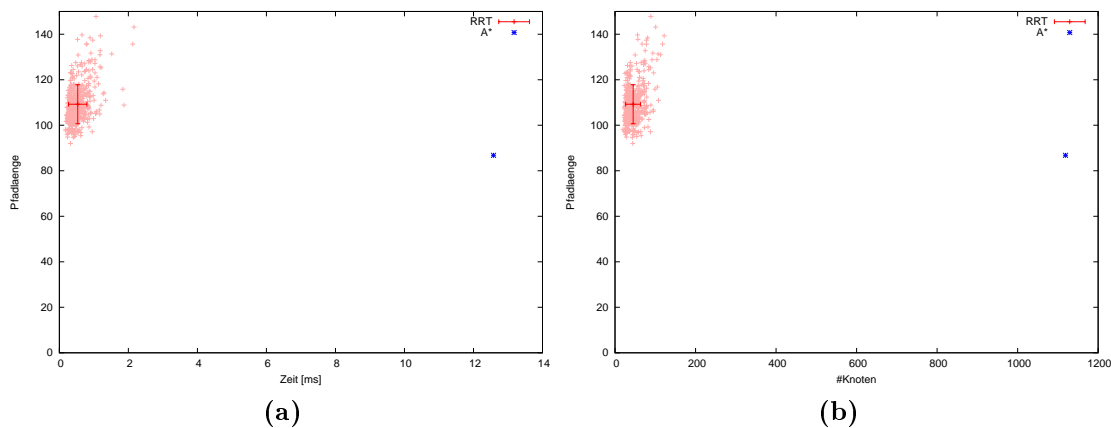
Unabhängig von den Dimensionen ist jedoch die Pfadlänge, welche beim *hybrid A^** geringer ausfällt als beim A^* . Da in dieser Testsituation lediglich fünf vordefinierte Pfadstücke genutzt wurden, wäre die Pfadlänge noch weiter zu optimieren, indem diese Anzahl erhöht wird.

	Laufzeit	# Knoten	Zeit pro Knoten	Pfadkosten
A^*	12.58 ms	1119	0.0112 ms	86.77
<i>hybrid A^*</i>	230.85 ms	8610	0.0268 ms	82.75

Abbildung 4.11: Messergebnisse eines Vergleichs von A^* und *hybrid A^**

4.6 Vergleich RRT mit A*

Der *Rapidly-exploring Random Tree* (Kapitel 2.7) ist im Gegensatz zu den bisher behandelten Algorithmen ein probabilistischer Algorithmus. Dies hat zur Folge, dass er bei mehreren Durchläufen jedes Mal einen anderen Pfad findet. Aus diesem Grund muss in der Evaluation bei den Messungen der Graphgrößen, der Pfadkosten oder der Laufzeit, nicht nur ein Mittelwert betrachtet werden, sondern auch die Varianz bzw. die Standardabweichung.

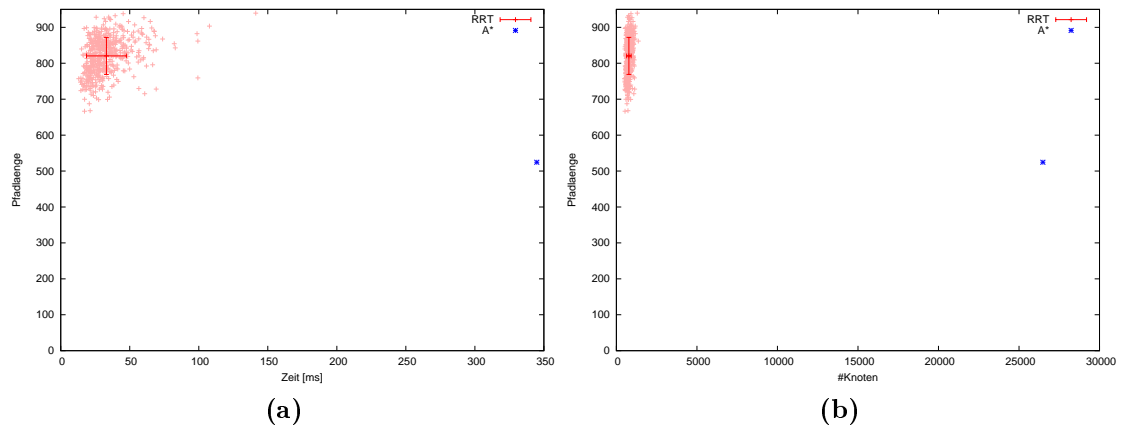


Karte: [C.1]	<i>RRT</i>		<i>A*</i>
	Mittelwert	Standardabweichung	
Zeit [ms]	0.532	0.267	12.581
Pfadkosten	109.24	8.58	86.77
Graphgröße [Knoten]	45.05	18.6	1119

(c)

Abbildung 4.12: Vergleich des *RRT* und *A** in der Umgebung C.1

Im folgenden Experiment wurde der *RRT* mit einer Schrittweite von 5 Zellbreiten und einem *goal bias* von 0.5 mit dem *A** verglichen. Der Mittelwert und die Standardabweichung entstanden dabei aus 500 Messungen. Das Experiment wurde sowohl auf einer kleinen Karte, mit 60×40 Zellen (Anhang: C.1), durchgeführt, als auch auf einer großen Karte mit 200×200 Zellen (Anhang: C.5). Die Ergebnisse sind in den Abbildungen 4.12 und 4.13 zu sehen. In (a) ist jeweils die Laufzeit und die Länge der gefundenen Pfade abgebildet, in (b) die Anzahl der Knoten, die der Graph bei der Zielfindung hat. Hierbei zeigen die roten Kreuze die Mittelwerte und Standardabweichungen und die hellroten Punkte sind die Einzelmessungen. Die genauen Messwerte sind dabei in den Tabellen in (c) zu sehen.



Karte: [C.5]	<i>RRT</i>		<i>A*</i>
	Mittelwert	Standardabweichung	
Zeit [ms]	33.07	14.47	344.84
Pfadkosten	820.44	51.57	524.6
Graphgröße [Knoten]	778.34	139.22	26484

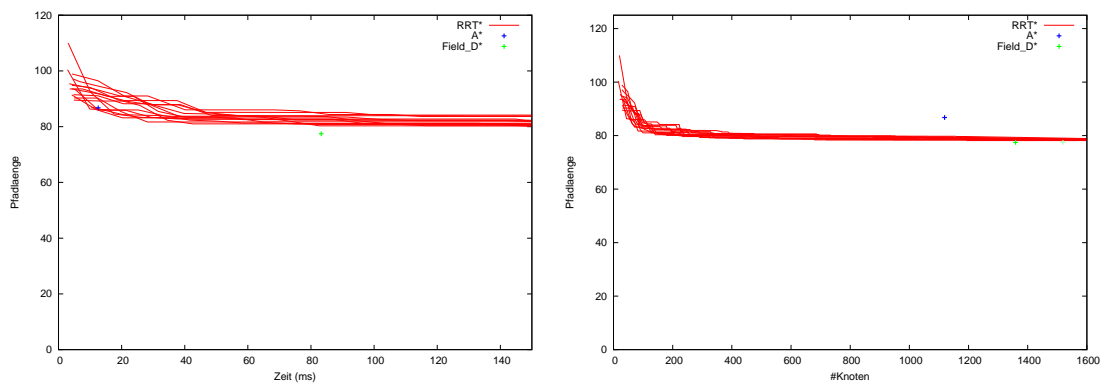
(c)

Abbildung 4.13: Vergleich des *RRT* und *A** in der Umgebung C.5

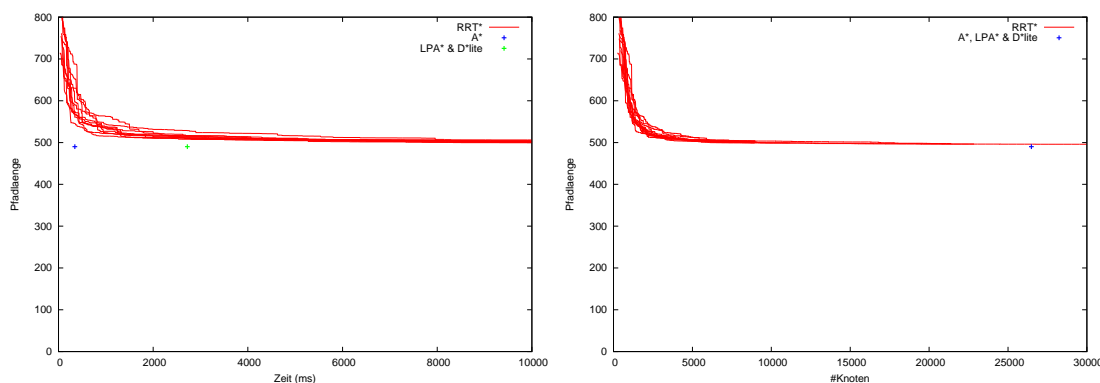
Mit dem Vergleich wird deutlich, dass der *RRT* zwar längere Pfade findet, dies jedoch in sehr viel kürzerer Zeit und mit sehr viel weniger Speicherbedarf. Je größer die Karte ist, desto ausgeprägter ist der Unterschied zwischen den Graphgrößen der beiden Algorithmen, obwohl bei beiden Experimenten mit der gleichen maximalen Schrittweite von 5-Terrainzellen gearbeitet wurde. Bei großen freien Karten macht es sogar Sinn noch größere Schrittweiten zu wählen, oder diese abhängig von der Belegtheit der Karte zu machen, wodurch sich noch mehr Knoten und damit auch mehr Zeit einsparen lässt. Ist bei der Planung die Optimalität nicht so wichtig wie die Laufzeit, so bietet der *RRT* eine gute Alternative zum *A**.

4.7 *RRT** Evaluation

Soll der *RRT** (Kapitel 2.8) evaluiert werden, so genügt es nicht, den Zustand des Planers bei der Zielfindung zu betrachten. Er muss besonders in der Zeit danach beobachtet werden, da der Zielpfad hier noch weiter optimiert wird. Im folgenden Experiment, wurde der *RRT** sowohl mit dem *A** als auch mit dem *Field D** verglichen, da dieser den kürzesten Pfad anzeigt. Die Experimente wurden auf den gleichen Karten durchgeführt wie in der *RRT*-Evaluation, diesmal jedoch mit einer größeren Schrittweite von maximal 20 Terrainzellen. Da die Messungen sehr stark von der Umgebung abhängen, in der das Experiment abläuft, erlangt man hier keinen Mehrwert dadurch das Experiment sehr häufig durchzuführen, um möglichst genaue Mittelwerte und Standardabweichungen zu erhalten, da diese bei jeder Karte anders wären. Aus diesem Grund wurden hier jeweils nur circa 10 Messungen vorgenommen, da diese genügen um eine aussagekräftige Tendenz anzuzeigen.



(a) Vergleich der Algorithmen in der Umgebung C.1



(b) Vergleich der Algorithmen in der Umgebung C.5

Abbildung 4.14: Evaluationsergebnisse des *RRT**

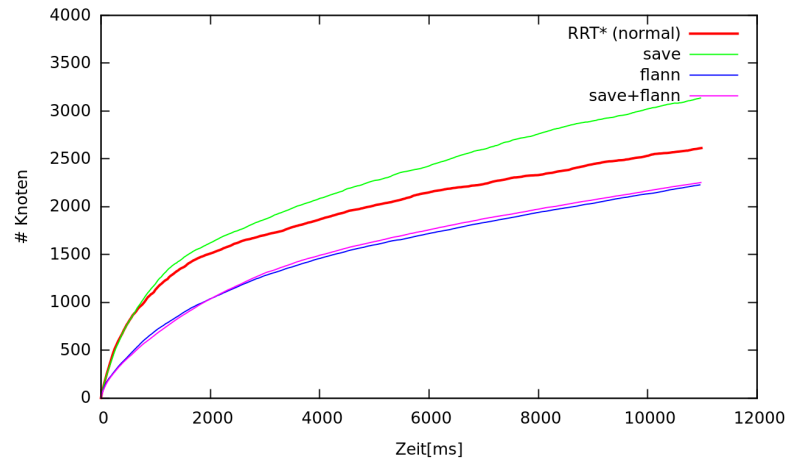


Abbildung 4.15: Geschwindigkeitsmessung der Graphausbreitung beim RRT^* mit verschiedenen Erweiterungen

In den Abbildungen 4.14 und ?? sind die Messungen der Experimente zu sehen. Die roten Linien sind die Messungen des RRT^* . Man sieht, dass der Algorithmus sehr schnell einen gültigen Pfad findet, welcher anfangs noch recht suboptimal ist. Läuft der Algorithmus jedoch weiter, so wird dieser Pfad stetig verbessert und nähert sich dem optimalen Pfad (vgl. $Field D^*$) an. Für den zeitlichen Vergleich wurde hier der A^* gewählt, da dieser in seiner Grundform am schnellsten ist. Zu erkennen ist, dass der RRT^* selbst hier zeitlich mithalten kann, und diesen in der Graphgröße deutlich unterbietet. Es sind jedoch Ausnahmen, wenn der RRT^* in der gleichen Zeit einen kürzeren Pfad gefunden hat als der A^* .

4.7.1 FLANN im RRT^*

Zeitlich die größte Verzögerung im RRT^* bildet das sehr häufige Suchen eines nächsten Nachbarn oder aller Nachbarn in einem gewissen Radius um einen Knotenpunkt (Radius-Suche). Da dieses für jeden Knotenpunkt, welcher neu hinzugefügt wird, mehrfach durchgeführt werden muss, wäre es ein großer Vorteil, wenn dies schneller ginge. Ein vielversprechendes Werkzeug hierfür schien $FLANN$ (*Fast Library for Approximate Nearest Neighbors* [Fla14]) zu sein, welches sowohl eine schnelle nächster-Nachbar-Suche als auch eine Radius-Suche anbietet.

Nach der Implementierung wurde die Geschwindigkeit des RRT^* beurteilt, indem gemessen wurde, wie schnell der Graph wächst (siehe Abbildung 4.15). Überraschenderweise erhielt der Algorithmus durch die $FLANN$ -Bibliothek keinen Vorteil, sondern einen konstanten Geschwindigkeits-Verlust. $FLANN$ verspricht zwar sehr schnell bei vielen Anfragen auf einer großen statischen Menge von Punkten zu sein.

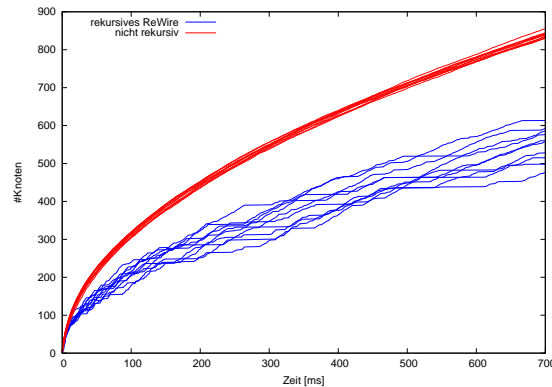


Abbildung 4.16: Vergleich der Expandierungsgeschwindigkeit des *RRT** mit und ohne rekursiven reWire

Wächst diese Menge jedoch stetig, sodass fast bei jeder neuen Abfrage auch eine leicht veränderte Datenmenge vorliegt, so schwindet der Vorteil der Bibliothek. Aufgrund des Versagens von *FLANN* wurde eine eigene Alternative entwickelt, die den Algorithmus etwas beschleunigt. Hierbei speichert jeder Knoten eine Referenz zu allen bisher entdeckten Knotenpunkten innerhalb des Radius ab, um Mehrfachberechnungen in wiederholten Radiussuchen vermeiden zu können. Der Nachteil dieser Methode ist eine geringe Speicherbedarfserhöhung abhängig von der Knotendichte des Graphen. Der deutliche Geschwindigkeitsgewinn ist in Abbildung 4.15 zu erkennen, in welcher dieser Methode mit „save“ bezeichnet wird.

4.7.2 Rekursives ReWire im *RRT**

In Kapitel 2.8.3 wurde eine rekursive Version der reWire-Funktion vorgestellt. Die rekursive Weitergabe der reWire-Funktion führt jedoch, neben den Vorteilen die sie bringt, auch zu einer Erhöhung des Aufwandes. Wie groß dieser tatsächlich ist, wurde in einem Experiment in der Umgebung C.1 ermittelt. Hier wurde der *RRT** sowohl mit, als auch ohne dem rekursiven reWire ausgeführt.

In den Abbildungen 4.16 ist dabei lediglich die Expandierungs-Geschwindigkeit der einzelnen Durchläufe abgebildet. Hier ist sehr deutlich die einheitliche logarithmische Expandierung des *RRT** zu erkennen, wenn kein rekursives reWire genutzt wird. Ist dieses jedoch aktiv, so verlangsamt sich die Ausbreitung enorm und hat eine deutlich höhere Varianz als vorher.

Wie sich dies auf die Pfadfindung auswirkt wird in Abbildung 4.17 verdeutlicht. Die Optimierung des Pfades findet deutlich langsamer statt, wie in (a) zu sehen ist. Betrachtet man jedoch die Pfadlänge abhängig von der Größe des Graphen in (b), so sieht man, dass das rekursive reWire hier einen Vorteil bringt. Der unsaubere Verlauf der roten Linie liegt hier an der geringen Anzahl von Messdaten.

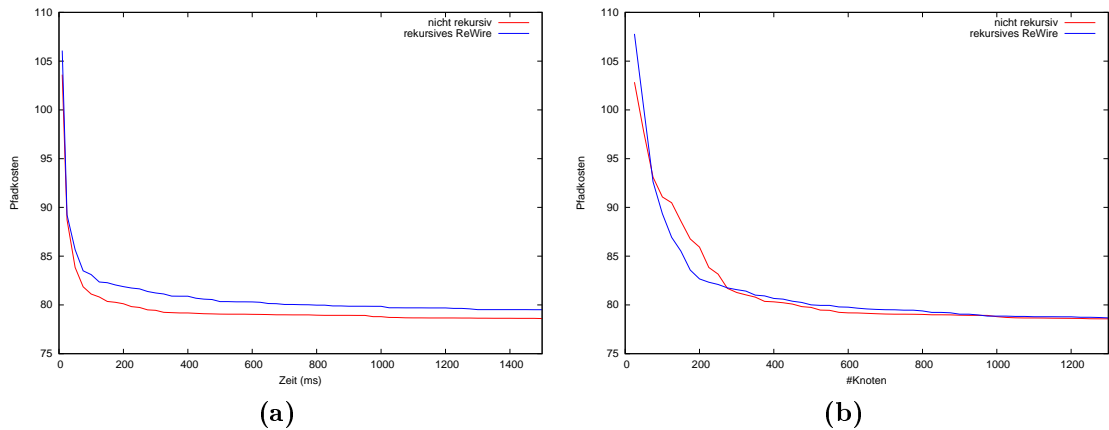


Abbildung 4.17: Vergleichsmessungen des RRT^* mit und ohne rekursiven reWire

Generell lässt sich jedoch sagen, dass das rekursive reWire in dieser Form im normalen RRT^* nicht unbedingt zu empfehlen ist. Im *Lifelong Planning RRT^** macht dies wiederum trotzdem Sinn, wie in Kapitel 2.9 beschrieben wurde.

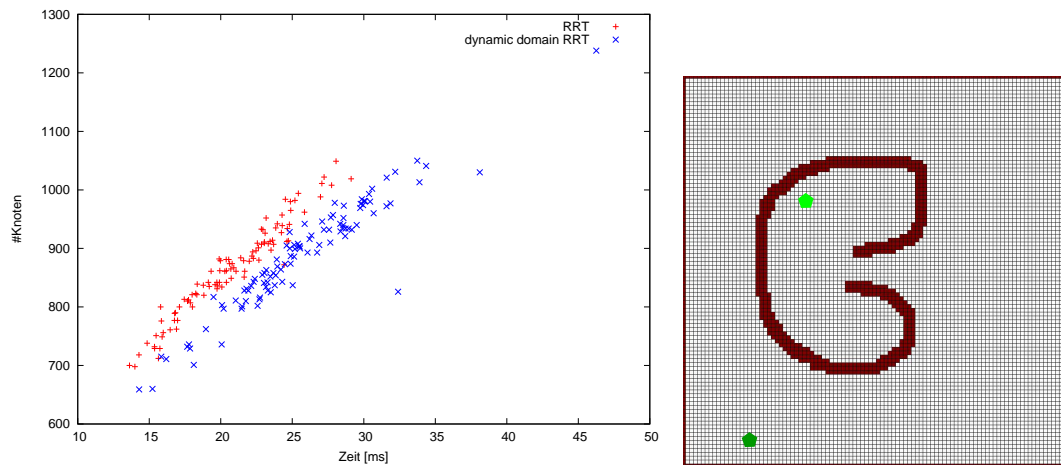
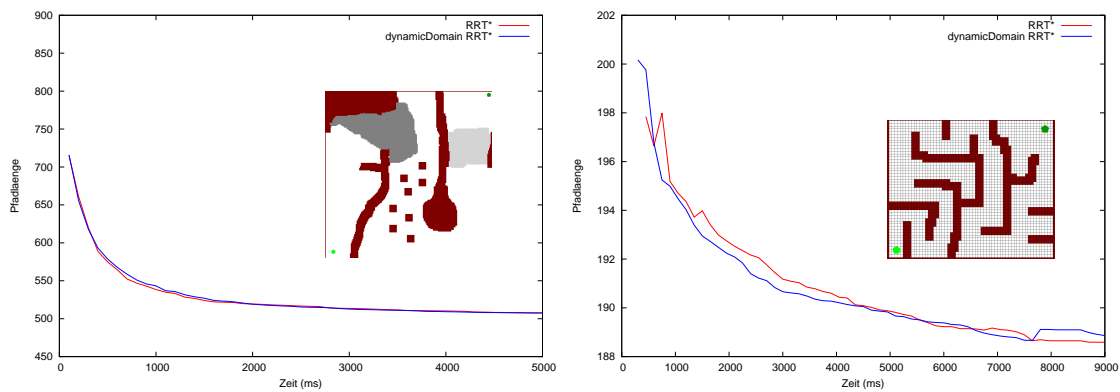
4.7.3 Dynamic Domain RRT Evaluation

Die folgenden Experimente sollen den Vorteil des *Dynamic Domain RRT* messen, welcher aus Kapitel 2.8.3 bekannt ist. Hierzu wurde der Algorithmus auf einer ähnlichen *bugtrap*-Karte durchgeführt, welche auch die Entwickler der Erweiterung zur Motivation nutzen. Sie ist in Abbildung 4.18 neben den Ergebnissen abgebildet. Der genutzte RRT besaß eine maximale Schrittweite von 5 Terrainzellen und einen *goal-bias* von 0,1. Der Radius R der *dynamic domain* Erweiterung betrug 15 Zellbreiten, also 3 Schrittweiten. Es wurden folgende Mittelwerte gemessen:

	RRT	dynamic domain RRT
Gesamtzeit [ms]	25.6	20.83
Zeit pro Knoten [ms]	0.02861	0.023888

In Abbildung 4.18 sind die einzelnen Messpunkte zu sehen, welche den Vorteil des *dynamic domain RRT* noch deutlicher visualisieren.

Es wurden noch weitere Experimente durchgeführt, in welchen die *dynamic domain* Erweiterung auf dem RRT^* angewandt wurde (siehe Abbildung 4.19). Die Unstetigkeiten in den Kurven entstehen hierbei durch die unterschiedlichen Startpunkte der Einzelmessungen, da das Ziel immer zu unterschiedlichen Zeitpunkten das erste Mal entdeckt wird.

Abbildung 4.18: Dynamic Domain *RRT* Zeitmessungen in der „bugtrap“Abbildung 4.19: Dynamic Domain *RRT** Zeitmessungen

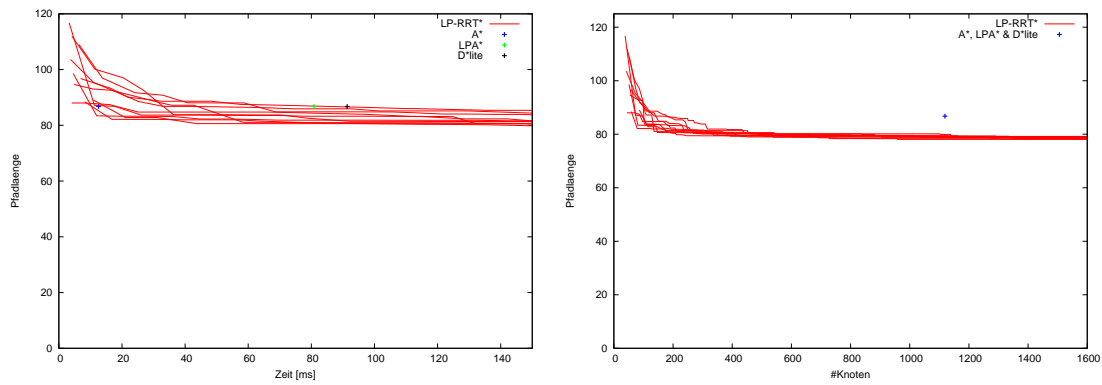
Generell lässt sich sagen, dass der *Dynamic Domain RRT* in sehr komplexen Umgebungen mit vielen Hindernissen spürbare Vorteile bringt. In einfacheren Umgebungen ist kaum ein Unterschied zu erkennen. Da durch die Erweiterung jedoch auch kein Nachteil entsteht und sie sehr einfach zu implementieren ist, bleibt es dennoch empfehlenswert.

4.8 Statische LP-RRT* Evaluation

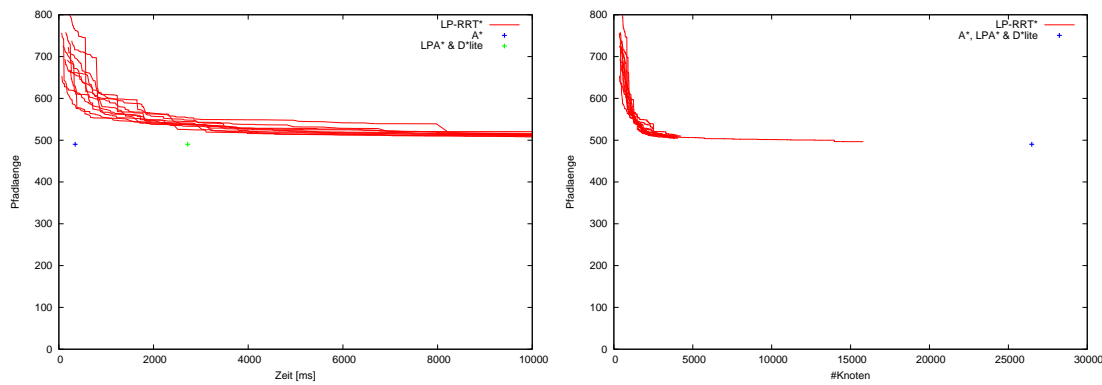
Im Folgenden wird der *Lifelong Planning RRT** (Kapitel 2.9) auf die gleiche Weise vermessen wie der *RRT** im vorherigen Kapitel. Der *LP-RRT** wurde hier mit einer maximalen Schrittweite von 10 Terrainzellen und einem *goal bias* von 0.3 gestartet. Im Gegensatz zu der Evaluation des *RRT** wurde hier das rekursive reWire genutzt. Die gemessenen Ergebnisse sind in Abbildung 4.20 zu sehen. In der zweiten Umgebung wurden die meisten Messungen nach 20 Sekunden und eine Messung nach 6 Minuten abgebrochen, da kein weiterer Informationsgewinn stattfand, weshalb die Linien in der Abbildung nicht ganz durchgängig sind.

Es ist zu erkennen, dass der *LP-RRT** in großen Umgebungen leider etwas langsamer ist als der *D* Lite*. Dies liegt daran, dass der Graph sehr groß wird, wodurch das *chooseParent* und das reWire auf Grund der Nachbarsuchen entsprechend lange brauchen. Besonders da hier das rekursive reWire zum Einsatz kommt, sind die Auswirkungen stärker als bei dem normalen *RRT**. Fände sich eine Möglichkeit, die Nachbarsuche zu beschleunigen, so würde das dem *LP-RRT** einen sehr großen Vorteil bringen.

Die hier gezeigten Daten vergleichen den *LP-RRT** nur in einer statischen Weise mit dem *D* Lite*. Interessant wäre ein Vergleich in einer dynamischen Umgebung, was sich jedoch generell nicht durchführen lässt, da hier der Verlauf des *LP-RRT** von sehr vielen Parametern abhängt.



(a) Vergleich der Algorithmen in der Umgebung C.1



(b) Vergleich der Algorithmen in der Umgebung C.5

Abbildung 4.20: Evaluationsergebnisse des *Lifelong Planning RRT**

4.9 Laufzeitmessung der Kreis Interpolation

Abschließend wird die Zeit, welche die Kreis Interpolation aus Kapitel 3.3.1 zur Glättung eines Pfades benötigt, mit der der Catmull-Rom Splines verglichen. Hierzu wurden 500 verschiedene Pfade mit durchschnittlich 300 Kontrollpunkten von beiden Algorithmen geglättet. Die folgende Tabelle zeigt die dabei gemessenen Laufzeiten:

	Laufzeit pro Kontrollpunkt	Standardabweichung
Catmull-Rom Splines	641.7 ns	96.1 ns (15%)
Kreis-Interpolation	4229.1 ns	180.7 ns (4.3%)

Hieraus ergibt sich, dass die Kreis-Interpolation fast 6,6 mal so lange braucht wie die Catmull-Rom Splines. Dies ist zwar ein sehr hoher Faktor, jedoch hat die Interpolation den Vorteil, dass sie einfach zu verstehen ist und sich somit einfach erweitern und anpassen lässt. Außerdem ergibt sie einen anderen geschmeidigeren Verlauf, welcher zu empfehlen ist, wenn genügend Zeit zur Berechnung der Glättung vorhanden ist.

Kapitel 5

Zusammenfassung

Diese Masterarbeit befasste sich mit der Untersuchung und dem Vergleich verschiedener Pfadplanungsalgorithmen. Abschließend ist für jeden nochmal zusammengefasst, welche Vor- und Nachteile diese bieten.

A^* Der A^* (Kapitel 2.1) bietet eine gute Grundlage für Planungsalgorithmen. Er ist jedoch im eigentlichen Sinne ein Graph-Suchalgorithmus, weshalb das Umfeld auch als Graph dargestellt werden muss. Da die Umgebung in der Robotik sehr oft als Grid (Feld aus einzelnen quadratischen Zellen) dargestellt wird, eignet sich hier der A^* sehr gut. In der Graph-Suche ist der A^* zwar optimal und optimal effizient, wie gezeigt wurde ist dies jedoch nicht mehr zwangsläufig für die Pfadsuche auf freien Flächen gegeben. In dieser Arbeit wurde der A^* als zweidimensionaler Planer vorgestellt, was bei nicht-holonomen Fahrzeugen eine abschließende Glättung des Pfades verlangt. Da die Pfadglättung erst im Nachhinein durchgeführt wird und beim Planen selbst nicht beachtet wird, kann die Korrektheit des Pfadplaners nicht mehr garantiert werden. Außerdem kann durch die Diskretisierung der Umgebung in einen Graphen, die Vollständigkeit verloren gehen. Dies spielt jedoch nur in Ausnahmefällen und in sehr engen Umgebungen eine Rolle, weshalb der A^* sehr häufig in der Pfadplanung eingesetzt wird. Als mehrfach aufgerufener *repeated* A^* ist er sogar in seiner Grundform für das Planen in unbekanntem dynamischen Umgebungen einsetzbar. Hier eignen sich abgeänderte *lifelong*-Planer jedoch besser, von denen einige in dieser Arbeit vorgestellt wurden, weil sie durch die Nutzung von Vorwissen die Effizienz steigern können.

Anytime Repairing A^* Der *Anytime Repairing A^** aus Kapitel 2.2 ist eine *anytime*-Abwandlung des A^* . Dies bedeutet, dass er sehr schnell einen gültigen Pfad findet, welcher jedoch nicht zwangsläufig optimal ist. In weiteren Schritten optimiert er diesen Pfad jedoch inkrementell. Trotz der inkrementellen Planung, kann der ARA^* nicht mit Änderungen in der Umgebung umgehen. Er ist von

daher nur für statische, vollkommen bekannte Umgebungen geeignet. Der von den Entwicklern genannte Vorteil das ARA^* ist der, dass das Fahrzeug früher losfahren kann, als bei anderen Algorithmen, und trotzdem nach einer gewissen Zeit den optimalen Pfad findet. Allerdings weiß der Algorithmus nicht mit einer Änderung der Startposition umzugehen, daher stellt sich die Frage, wie sinnvoll der praktische Nutzen dieses Algorithmus wirklich ist. Führt das Fahrzeug nämlich den zuerst gefundenen suboptimalen Pfad aus, so darf sich dieser in den weiteren Iterationen nicht zu sehr in Fahrzeugnähe ändern, sonst könnte es passieren, dass sich dieses nicht mehr auf dem Pfad befindet. Daher sollte darauf geachtet werden, in was für Situation der ARA^* eingesetzt wird.

Lifelong Planning A^* Der *Lifelong Planning A^** (Kapitel 2.3) dient in dieser Arbeit weniger als eigenständiger Pfadplaner, viel mehr als Vorbereitung für den $D^* Lite$. Der LPA^* setzt eine optimal effiziente Methode ein, Änderungen des Umgebungsmodell in den bereits expandierten Graphen ein zu bauen. Da hierzu nicht nur Teilgraphen neu 'aufgebaut', sondern andere Gebiete 'abgebaut' werden müssen, kann dies in Ausnahmesituation dazu führen, dass dies aufwändiger wird, als eine vollständige Neuplanung. In einer autonomen Fahrt, mit hoher Aktualisierungsrate des Umgebungsmodells, finden sich jedoch meist nur kleine Änderungen in der Karte. Hier bietet das inkrementelle Planen besonders bei großen Umgebungen eine enorme Effizienzsteigerung. Die Evaluation zeigt zwar, dass die Berechnungskosten pro Knoten im Graphen höher sind als beim normalen A^* , allerdings nicht so hoch, dass sie den Gewinn aufheben. Da der LPA^* zwar Umgebungsänderungen, jedoch keine Fahrzeugzustandsänderungen verarbeiten kann, ist auch dieser noch nicht geeignet für die inkrementelle Pfadplanung in unbekannter dynamischer Umgebung während der Fahrt.

$D^* Lite$ Der $D^* Lite$ (Kapitel 2.4) baut auf den LPA^* auf, und erweitert ihn auf eine solche Weise, dass auch eine Positionsänderung des Fahrzeuges so berücksichtigt werden kann, dass der bereits expandierte Graph nicht ungültig wird. Er ist ebenso wie der normale A^* immer noch optimal und optimal effizient, gegeben dem unterliegenden Graphen und der genutzten Heuristik. Somit bietet sich der $D^* Lite$ als gute Lösung für einen inkrementellen Pfadplaner in dynamischer unbekannter Umgebung an. Er führt lediglich die Nachteile der Graph-Gebundenheit mit sich, welche bereits vom normalen A^* bekannt sind. Da diese Nachteile allerdings recht gravierend sind, wenn der Algorithmus in autonomen Fahrt eingesetzt werden soll, gilt es diese mit den folgenden Erweiterungen zu beheben.

Field D^* Der *Field D^** aus Kapitel ?? behebt den Nachteil des A^* , bei der Kostenberechnung lediglich die Knotenmittelpunkte der direkten Nachbarknoten

zu beachten, wodurch der ermittelte Pfad meistens nicht optimal ist. Mit dem *Field D** wird der Pfad intuitiver, geschmeidiger und lässt sich um bis zu 8.2% verkürzen. Für die praktische Umsetzung war jedoch ein eigener Backtrace-Algorithmus zu entwickeln. Dieser eigentlich gute Ansatz, die Kosten mittels Interpolation zwischen den Nachbarknoten zu berechnen, kann in der Form, wie er von den Entwicklern dargestellt wird, jedoch nicht angewandt werden. Dies führt zu einer mit der Graphgröße exponentiell ansteigenden Mehrfachoptimierung der Knoten. Dementsprechend beschäftigt sich der Algorithmus nach einer gewissen Zeit fast ausschließlich mit der Optimierung und kaum noch mit der Expandierung des Graphen. Es wurde gezeigt, dass eine leichtere Gewichtung der Heuristik oder die Einführung einer Toleranz in der Kostenberechnung Abhilfe verschafft, das Problem jedoch nicht vollständig behebt. Bevor der Algorithmus in der Praxis genutzt wird, sollte hierfür eine Lösung gefunden werden.

hybrid A* Der *hybrid A** (Kapitel 2.6) plant ähnlich wie der normale *A**, jedoch nutzt er vordefinierte Pfadstücke als einzelne Schritte von einem Knoten zum nächsten. Damit ist es möglich, direkt alle nicht-holonomen Einschränkungen des Fahrzeugs zu berücksichtigen. Allerdings zeigte sich, dass dieser Planer weder vollständig noch optimal ist. Außerdem ist es hier nicht möglich, in lediglich zwei Dimensionen zu planen, was jedoch einen enormen Zeitgewinn mit sich bringen würde. Der große Vorteil ist jedoch, dass dieser Algorithmus im Gegensatz zu den anderen *A**-basierten Pfadplanern, die Korrektheit des gefundenen Pfades garantieren kann. Des Weiteren zeigten die Entwickler des *hybrid A** einen Pfadglättungs-Algorithmus, welcher in Verbindung mit dem *hybrid A** sehr gute Ergebnisse liefert. Dieser konnte im Rahmen dieser Arbeit leider nicht mehr umgesetzt werden. Ein Versuch, den *hybrid A** mit dem *D* Lite* zu verbinden, scheiterte, da dieses zu erheblichen Mehrkosten in der Planung führt, welche den Algorithmus zu sehr verlangsamen.

Rapidly-exploring Random Tree Diese Arbeit hat gezeigt, dass der probabilistische Pfadplaner *Rapidly-exploring Random Tree* (Kapitel 2.7) eine sehr gute Alternative zum *A** ist. Lediglich in sehr komplexen Umgebungen ist er nicht so Effektiv, da ihm das Systematische Vorgehen den Zielpunkt zu erreichen fehlt. Die Evaluation hat jedoch ergeben, dass er im Normalfall schneller ist als der *A** und sehr viel weniger Knotenpunkte benötigt. Besonders durch die variable Schrittweite, gewinnt der *RRT* an Effizienz. Einziger großer Nachteil ist, dass der nicht den kürzesten, sondern einen beliebigen Pfad findet. Außerdem ist er nur probabilistisch vollständig. Da der *RRT* von der Algorithmuseite keine direkten Ansprüche an die Steuerfunktion hat, eignet er sich einerseits für eine schnelle holonome Planung, aber genauso gut auch für komplexere Planungen, welche

direkt die nicht-holonomen Fahrzeugeinschränkungen beachten. Ebenso erlaubt er eine beliebig genaue Kollisionsdetektion, sodass man auch hier die Wahl zwischen Geschwindigkeit oder Genauigkeit hat. Dank dieser vielen Freiheiten, und weil er so leicht zu verstehen ist, ist es sehr einfach eigene Erweiterungen zu entwerfen oder ihn beliebig anzupassen. Daher finden sich auch viele Erweiterungen und Heuristiken, von denen einige in dieser Arbeit vorgestellt werden. Lohnenswert wäre ein tieferer Einblick in interessante Erweiterungen, wie beispielsweise den bidirektionalen-*RRT* oder einer umgebungsabhängigen Schrittweite. Wie ein Team des MIT zeigte, eignet sich der *RRT* sogar in seiner Grundform dazu, in dynamischer und unbekannter Umgebung zu planen. Im Kontext dieser Arbeit wurde dies jedoch auf Basis des *RRT** erreicht, um so kürzere Pfade zu finden.

RRT* Der *RRT**, welcher in Kapitel 2.8 behandelt wurde, erweitert den *RRT* so, dass er asymptotisch optimal wird. Dies bedeutet, dass der Algorithmus auch nach der Zielfindung weiter läuft, um den ermittelten Pfad zu optimieren. Das größte Problem des *RRT** ist, dass er sehr häufig eine Nächster-Nachbar-Suche im Zustandsraum des Fahrzeuges durchführen muss, was bei großen Graphen zur zeitlichen Beeinträchtigung führt. Ein Versuch, dies mithilfe der FLANN-Bibliothek zu beschleunigen, erwies sich als weitere Verlangsamung, da es sich hier um eine sehr schnell wachsende Datenmenge handelt. Das im Rahmen dieser Arbeit entwickelte rekursive ReWire, welches eine verbesserte globale Informationsweitergabe darstellt, welche vorher nur lokal stattfand, erwies sich in der Evaluation leider als sehr langsam. Unter anderem ist dies auf die Nachbarschaftssuche zurückzuführen. Würde sich hier eine schnellere Lösung finden, so könnte auch das rekursive ReWire bessere Ergebnisse liefern. Die Evaluation hat gezeigt, dass der *RRT** in einfacheren Umgebungen zeitlich sehr gut mit dem bewährten *A** mithalten kann, lediglich in großen Umgebungen schafft er dies nicht ganz. Hinsichtlich des Speicherbedarfs ist er jedoch sehr viel effizienter. Vergleicht man die Zeiten mit dem *D* Lite* oder sogar dem *Field D**, so zeigt der *RRT** selbst in größeren Umgebungen gute Ergebnisse. Daher bestand die Motivation einen *lifelong*-Planer auf der Basis des *RRT** zu entwerfen: der *Lifelong Planning RRT** .

Lifelong Planning RRT* Der *Lifelong Planning RRT**, welcher in Kapitel 2.9 vorgestellt wird, wandelt den *RRT** so ab, dass Änderungen im Umgebungsmodell in dem bereits bestehenden Plan eingearbeitet werden können. Ebenso ist es möglich eine Änderung des Fahrzeugzustandes zu verarbeiten, womit sich der Algorithmus als Pfadplaner für mobile Systeme in einer dynamischen unbekanntem Umgebung eignet. Im Gegensatz zum *RRT** ist es hier allerdings, trotz der zeitlichen Verzögerung, sinnvoll, das rekursive ReWire anzuwenden. Gebiete, welche durch neue Hindernisse, den Anschluss zum restlichen Graphen verloren haben, können auf

diese Weise schneller wieder vollständig abgedeckt werden. Die Entwicklung des *LP-RRT** zeigte, dass sich der *RRT** sehr gut als Grundlage für einen dynamischen Planungsalgorithmus eignet, welcher einfach an neue Ideen oder spezielle Gegebenheiten anzupassen ist.

Kollisionsvermeidung In dieser Arbeit wurde die Kollisionsvermeidung in Kapitel 3.1 hauptsächlich über die Erweiterung der Hindernisse durchgeführt. Hier wurde gezeigt, wie sich das Ausmaß des Ausschwenkens abhängig vom Kurvenradius berechnet, um somit die Ausdehnung der Hindernisse möglichst gering halten zu können. Außerdem wurde im Rahmen dieser Arbeit ein Algorithmus entwickelt, welcher alle von einer beliebigen Strecke geschnittenen Terrainzellen schnell bestimmt, um mit diesen die Kollisionsdetektion und Kostenberechnung durchzuführen.

Pfadglättung Für die Pfadglättung (Kapitel 3.3), wurde ebenfalls eine eigene Idee umgesetzt, welche den stetig differenzierbaren Pfad durch die Interpolation von Kreisbögen ermittelt. Die Grundlagen hierzu wurden bereits in Kapitel 3.2 gelegt, in dem glatte Pfadstücke bestimmt wurden, welche im *RRT* und *RRT** einsetzbar sind. Die neue Pfadglättung ist zwar im Vergleich zu den Catmull-Rom Splines etwas langsamer, allerdings ist der ermittelte Pfad geschmeidiger und intuitiver. Außerdem ist der Algorithmus einfach zu verstehen, wodurch es leicht möglich ist diesen noch zu erweitern, verbessern oder anzupassen. Im Rahmen des *Lifelong Planning RRT** wurde außerdem gezeigt, wie sich die Pfadglättung durchführen lässt, sodass ein kollisionsfreier glatter Pfad ermittelt wird, obwohl der Planer aus Effizienzgründen mit einfachen holonomen Pfaden expandiert.

In dieser Arbeit wurde gezeigt, dass die Forschung im Bereich der dynamischen Pfadplaner noch bei weitem nicht abgeschlossen ist. Da dieses Themengebiet überaus komplex ist, konnten hier nicht alle Teilbereiche ausreichend vertieft werden. So wären beispielsweise weitere Forschungen im Gebiet der parallelen Planung auf mehreren Threads oder auf der GPU interessant, was einen enormen Geschwindigkeitsvorteil bringen würde. Ebenso wäre eine gründlichere Behandlung der Pfadglättung, welche nicht-holonome Einschränkungen und Kollisionsvermeidung berücksichtigt, wünschenswert gewesen. Bezüglich der Pfadplanung konnte jedoch eine beträchtliche Anzahl an verschiedenen Algorithmen getestet und evaluiert werden. Daraus ging hervor, dass sowohl der *A** als auch der *Rapidly-exploring Random Tree* sehr gute statische Planungsalgorithmen sind, mit jeweils eigenen Vor- und Nachteilen. Außerdem eignen sich beide gut als Grundlage für Pfadplaner mobiler Systeme in dynamischer unbekannter Umgebung.

Anhang A

Field-Backtrace Algorithmus

$path \leftarrow fieldBacktrace(v_{goal})$

```

1  $v_a \leftarrow v_{goal}$ ;
2  $\mathbf{p} \leftarrow p(v_{goal})$ ;
3  $path.push\_back(\mathbf{p})$ ;
4  $\mathbf{p1}_a \leftarrow p(parent1(v_a))$ ;
5  $\mathbf{p2}_a \leftarrow p(parent2(v_a))$ ;
6 if  $i(v_a) > 0$  then
7   |  $path.push\_back(\mathbf{p} + (\mathbf{p1}_a - \mathbf{p}) * i(v_a))$ ;
8   |  $\mathbf{p} \leftarrow \mathbf{p1}_a + (\mathbf{p2}_a - \mathbf{p1}_a) * j(v_a)$ ;
9   |  $path.push\_back(\mathbf{p})$ ;
10  $v_b \leftarrow parent2(v_a)$ ;
11  $v_a \leftarrow parent1(v_a)$ ;
12 while  $(v_a \neq v_{start}) \wedge (v_b \neq v_{start})$  do
13   |  $\mathbf{s}_p \leftarrow estimateS(\mathbf{p}, v_a, v_b)$ ;
14   |  $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{s}_p$ ;
15   |  $\{v_a, v_b\} \leftarrow findNeighbours(\mathbf{p})$ ;
16   |  $path.push\_back(\mathbf{p})$ ;
17 end

```

Abbildung A.1: Field-Backtrace - pt.1


```

 $s_p \leftarrow \text{estimateS}(p, v_a, v_b)$ 
1   $p1_a \leftarrow p(\text{parent1}(v_a));$ 
2   $p2_a \leftarrow p(\text{parent2}(v_a));$ 
3   $t_a \leftarrow p1_a + ((p2_a - p1_a) \cdot j(v_a));$ 
4   $p1_b \leftarrow p(\text{parent1}(v_b));$ 
5   $p2_b \leftarrow p(\text{parent2}(v));$ 
6   $t_b \leftarrow p1_b + ((p2_b - p1_b) \cdot j(v_b));$ 
7   $d \leftarrow \|(p - p(v_a))\|;$ 
8   $s_p \leftarrow (1 - d)(t_a - p(v_a)) + d(t_b - p(v_b));$ 
9  if  $p(v_a).x = p(v_b).x$  then
10 |
11 |   if  $s_p.x = 0$  then
12 | |    $s_p \leftarrow t_a - p;$ 
13 | |   else
14 | | |    $s_p \leftarrow \frac{s_p}{|s_p.x|};$ 
15 | | |   if  $|s_p.y| > 1$  then
16 | | | |   if  $s_p.y > 0$  then
17 | | | | |    $s_p \leftarrow s_p * \frac{\max(p(v_a).y, p(v_b).y) - p.y}{s_p.y};$ 
18 | | | | |   else
19 | | | | |    $s_p \leftarrow s_p * \frac{\min(p(v_a).y, p(v_b).y) - p.y}{s_p.y};$ 
20 | | | |   end
21 | | |   end
22 | |   end
23 | else
24 |
25 |   if  $s_p.y = 0$  then
26 | |    $s_p \leftarrow t_a - p;$ 
27 | |   else
28 | | |    $s_p \leftarrow \frac{s_p}{|s_p.y|};$ 
29 | | |   if  $|s_p.x| > 1$  then
30 | | | |   if  $s_p.x > 0$  then
31 | | | | |    $s_p \leftarrow s_p * \frac{\max(p(v_a).x, p(v_b).x) - p.x}{s_p.x};$ 
32 | | | | |   else
33 | | | | |    $s_p \leftarrow s_p * \frac{\min(p(v_a).x, p(v_b).x) - p.x}{s_p.x};$ 
34 | | | |   end
35 | | |   end
36 | |   end
37 | end
38 return  $s_p;$ 

```

▷ same x-level

▷ same y-level

Abbildung A.2: Field-Backtrace - pt.2

$\{v_a, v_b\} \leftarrow \text{findNeighbours}(\mathbf{p})$

```

1 if  $|\mathbf{p}.x - \lfloor \mathbf{p}.x + 0.5 \rfloor| < |\mathbf{p}.y - \lfloor \mathbf{p}.y + 0.5 \rfloor|$  then
2    $v_a \leftarrow \text{getNode}(\mathcal{G}, \mathbf{p}.x, \lfloor \mathbf{p}.y \rfloor);$ 
3    $v_b \leftarrow \text{getNode}(\mathcal{G}, \mathbf{p}.x, \lceil \mathbf{p}.y \rceil);$ 
4 else
5    $v_a \leftarrow \text{getNode}(\mathcal{G}, \lfloor \mathbf{p}.x \rfloor, \mathbf{p}.y);$ 
6    $v_b \leftarrow \text{getNode}(\mathcal{G}, \lceil \mathbf{p}.x \rceil, \mathbf{p}.y);$ 
7 return  $\{v_a, v_b\};$ 

```

Abbildung A.3: Field-Backtrace - pt.3

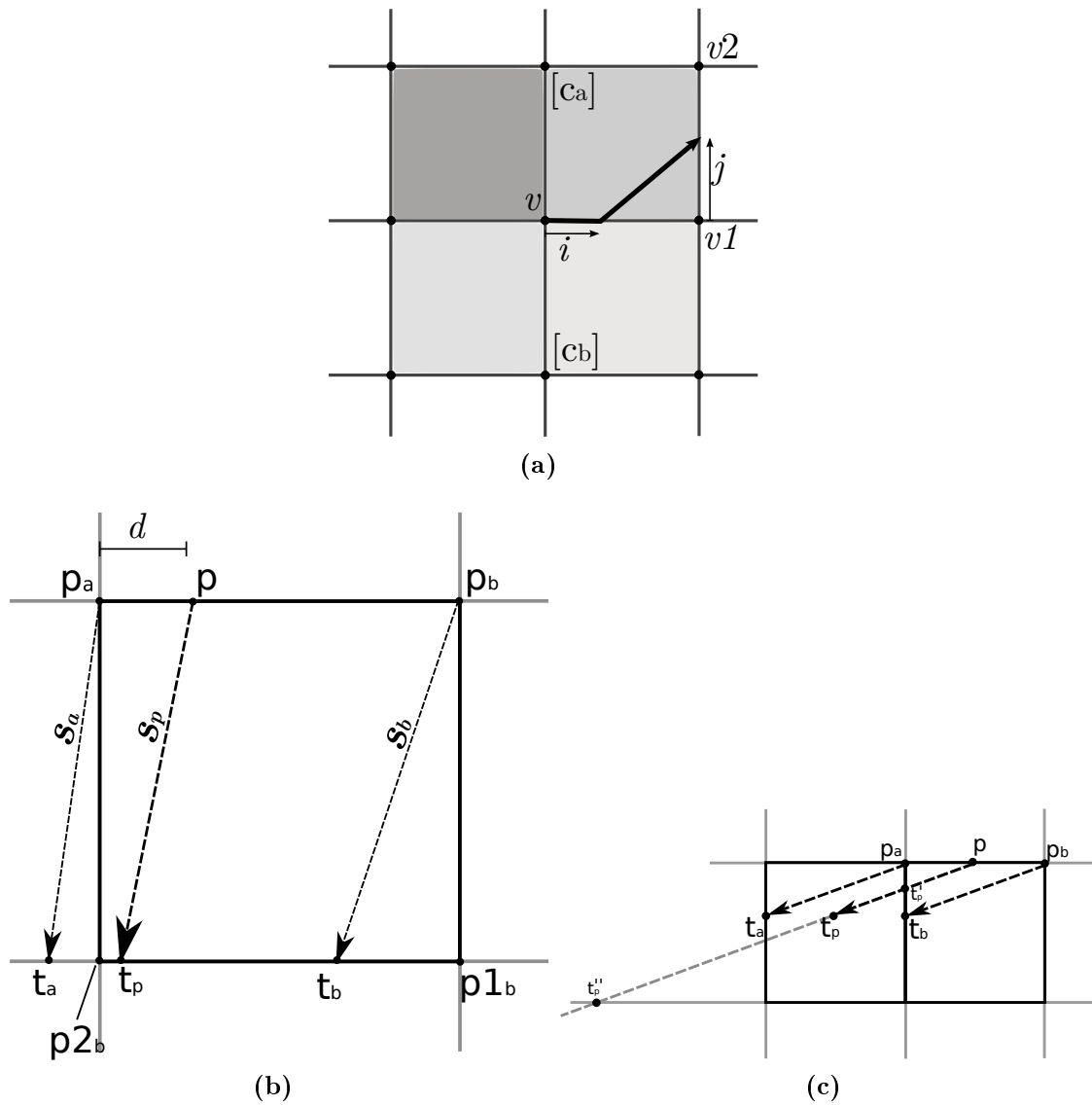


Abbildung A.4: Skizze zur Verdeutlichung der Variablen im *Field D**-Algorithmus

Anhang B

Linien-Rasterisierungs Algorithmus

$P \leftarrow \text{rasterizeLine}(\mathbf{p}_{\text{start}}, \mathbf{p}_{\text{end}})$

```

1  $\mathbf{d} \leftarrow \frac{\mathbf{p}_{\text{end}} - \mathbf{p}_{\text{start}}}{\|\mathbf{p}_{\text{end}} - \mathbf{p}_{\text{start}}\|};$ 
2  $\mathbf{p}_a \leftarrow \mathbf{p}_{\text{start}};$ 
3  $\mathbf{p}_b \leftarrow \mathbf{p}_{\text{start}};$ 
4  $P \leftarrow \{\lfloor \mathbf{p}_a \rfloor\};$ 
5  $i \leftarrow 1;$ 
6 while  $i < \|\mathbf{p}_{\text{end}} - \mathbf{p}_{\text{start}}\| + 1$  do
7   if  $i \geq \|\mathbf{p}_{\text{end}} - \mathbf{p}_{\text{start}}\|$  then
8      $\mathbf{p}_b \leftarrow \mathbf{p}_{\text{end}};$ 
9   else
10     $\mathbf{p}_b \leftarrow \mathbf{p}_b + \mathbf{d};$ 
11     $d_M = \|\lfloor \mathbf{p}_b \rfloor - \lfloor \mathbf{p}_a \rfloor\|_1;$ 
12    if  $d_M = 2$  then
13       $\mathbf{p}_{\text{center}} = \left( \max(\lfloor \mathbf{p}_a.x \rfloor, \lfloor \mathbf{p}_b.x \rfloor), \max(\lfloor \mathbf{p}_a.y \rfloor, \lfloor \mathbf{p}_b.y \rfloor) \right)^T;$ 
14       $y_{\text{cut}} = \mathbf{p}_{\text{start}}.y + \frac{\mathbf{d}.y}{\mathbf{d}.x} \cdot (\mathbf{p}_{\text{center}}.x - \mathbf{p}_{\text{start}}.x);$ 
15      if  $y_{\text{cut}} < \mathbf{p}_{\text{center}}.y$  then
16        if  $\mathbf{p}_a.y < \mathbf{p}_b.y$  then
17           $P \leftarrow P \cup \{(\lfloor \mathbf{p}_b.x \rfloor, \lfloor \mathbf{p}_a.y \rfloor)^T\};$ 
18        else
19           $P \leftarrow P \cup \{(\lfloor \mathbf{p}_a.x \rfloor, \lfloor \mathbf{p}_b.y \rfloor)^T\};$ 
20        end
21      else if  $y_{\text{cut}} > \mathbf{p}_{\text{center}}.y$  then
22        if  $\mathbf{p}_a.y < \mathbf{p}_b.y$  then
23           $P \leftarrow P \cup \{(\lfloor \mathbf{p}_a.x \rfloor, \lfloor \mathbf{p}_b.y \rfloor)^T\};$ 
24        else
25           $P \leftarrow P \cup \{(\lfloor \mathbf{p}_b.x \rfloor, \lfloor \mathbf{p}_a.y \rfloor)^T\};$ 
26        end
27      end
28    end
29    if  $d_M \geq 1$  then
30       $P \leftarrow P \cup \{\lfloor \mathbf{p}_b \rfloor\};$ 
31     $\mathbf{p}_a \leftarrow \mathbf{p}_b;$ 
32     $i \leftarrow i + 1;$ 
33 end
34 return  $P$ 

```

Abbildung B.1: Linien-Rasterisierungs-Algorithmus

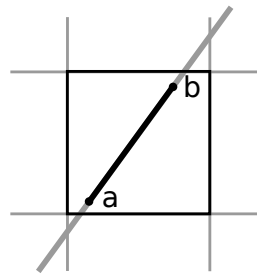
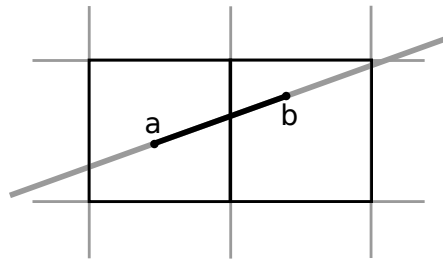
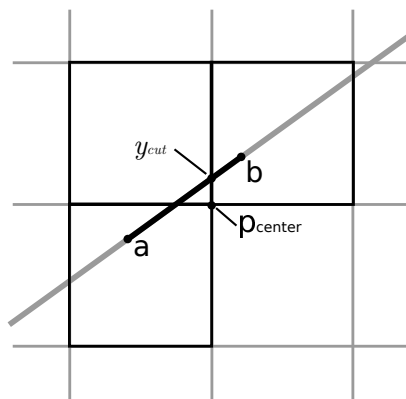
(a) $d_M = 0$ (b) $d_M = 1$ (c) $d_M = 2$

Abbildung B.2: Skizze zur Verdeutlichung der Variablen im Linien-Rasterisierungs-Algorithmus

Anhang C

Evaluationsumgebungen

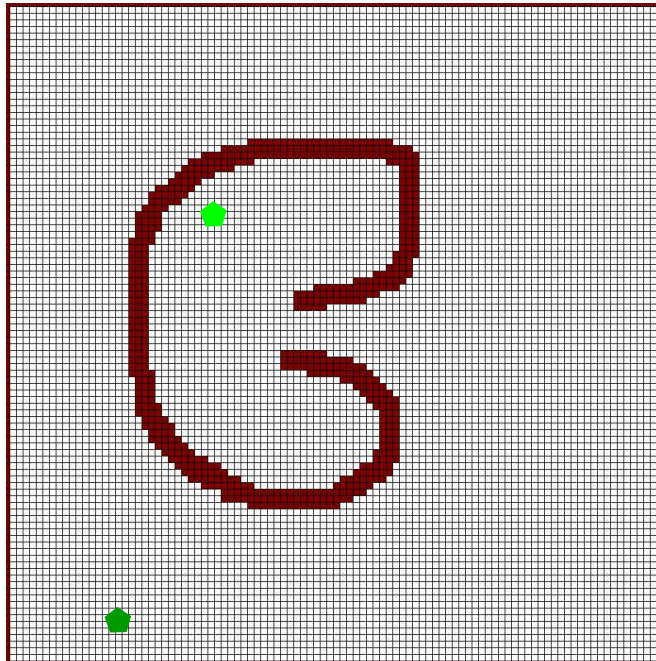


Abbildung C.4: Karte 4: 100×100 Zellen

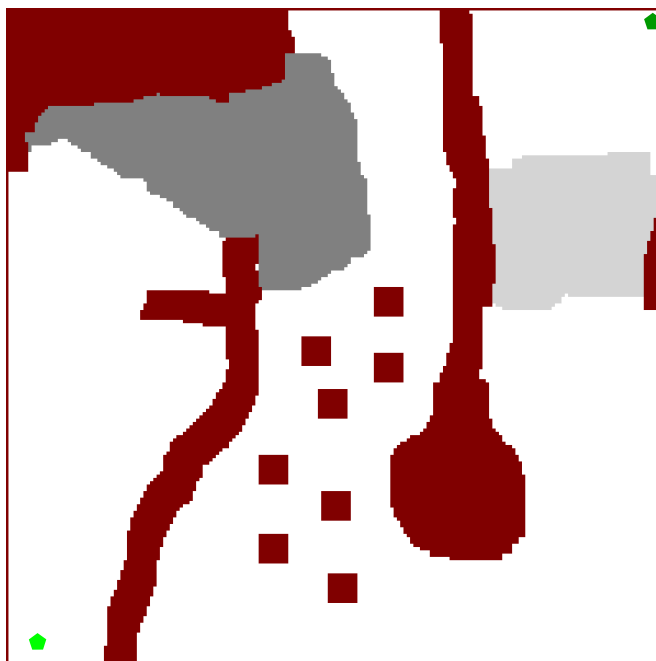


Abbildung C.5: Karte 3, 200×200 Zellen

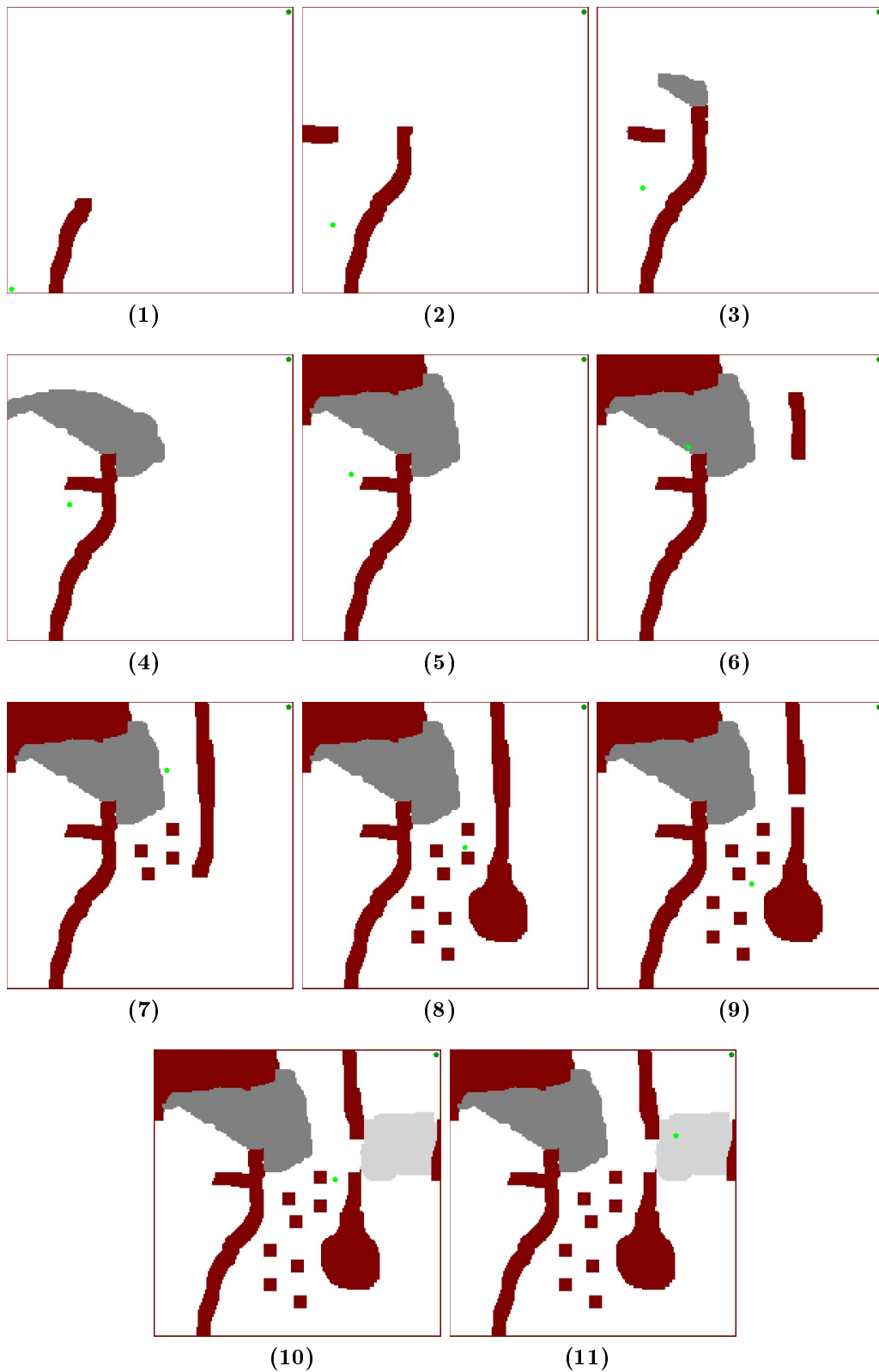


Abbildung C.6: Die simulierte Fahrt mit begrenzter Sichtweite durch eine dynamische Umgebung zur Evaluation des $D^* Lite$ mit 200×200 Zellen. (hellgrüner Punkt: aktuelle Position; dunkelgrüner Punkt: Zielposition)

Anhang D

ppLib - readme

| ppLib |

a PathPlanning-Library

by Benedikt Jöbgen
[bjoebgen@uni-koblenz.de]
January 2015

providing som Pathplanning Algorithms:

- A*
- Anytime Repairing A* (ARA*)
- hybrid A*
- Lifelong Planning A* (LPA*)
- D* Lite
- Field D*
- Field Lifelong Planning A*
- Rapidly-exploring Random Tree (RRT)
- RRT*
- Lifelong Planning RRT* (LL-RRT*)

For more details see the corresponding Master's thesis.

depends on:

- Eigen library
version 3.2.0
<http://eigen.tuxfamily.org>

How to Build:

- cmake .
- make

How to use:

```
1) the simple way: (it will include all Planning Algorithm)
   (for the other Algorithms see ppLib.h)

#include "/ppLib/ppLib.h"

...

// create a vehicle
Vehicle* robot = new Vehicle(halfWidth, lengthFront, lengthRear, minSteeringRadius);

// create a grid
DynamicGrid* grid = new DynamicGrid();
grid->setCellSize(cellSize);
```

```

// fill this grid
grid->set(x1,y1, new DynamicGrid::Cell(INFINITE,DynamicGrid::Cell::REAL_BLOCKED));
grid->set(x2,y2, new DynamicGrid::Cell(costs,DynamicGrid::Cell::NOT_BLOCKED));

// create Start and Goal Konfiguration
Vehicle::State startState(startPosition, startOrientation);
Vehicle::State goalState(goalPosition, goalOrientation);

// create a Planner
Planner* astar = pplib.getAstar(grid, robot, smoothPath,
                               wg, wh,
                               pplib.getHeuristicFunction_euclidean(goalPosition));

// tell him about start and goal
astar->setStart(startState);
astar->setGoal(goalState);

// extend Obstacles for kollision-avoidance (needs to be after getting the planner)
grid->extendObstacles(robot->maxSteeringTiltOut());

// preparing the Planner
astar->prepare();

// running the planner
astar->run();

// get the resulting path
std::vector<Vector2d> path = astar->getBestPath();

```

2) running at a seperate Thread:

```

...

// preparing the Planner
astar->prepare();

// running the planner at his own thread
astar->threaded_run();

//... waiting

// get the resulting path
if(astar->gotPath())
    std::vector<Vector2d> path = astar->getBestPath();

// or if it lasts too long (you can rerun it later)
astar->stop();

```

3) a more complex way: (includes just the one planning Algorithm) (for the other Algorithms see pplib.cpp)

```

#include "/ppLib/algorithms/astar.h"

...

// create a Planner
grid->setNodeUsage(DynamicGrid::CENTER_BASED);
AStar<AStar_Node>* aStar= new AStar<AStar_Node>();
aStar->initAStar(wg, wh,
               [&](double x, double y){return (Vector2d(x,y)-goalPosition).norm();});
aStar->initPlanner(grid, vehicle, smoothPath);

...

```

4) Anytime Replanning A* Iterations

```

// running the planner & get Path
while(arastar->getLastSuboptimality() > 1.0) {
    arastar->run();
    std::vector<Vector2d> path = arastar->getBestPath();
    ...
}

```

```
5) add some dynamic

// change environment
grid->extend(currentScanGrid);

// change Startposition
planner->setStart(newStartState);

- LPA*, D*Lite, FieldD* :

    // integrate Environmentchanges
    planner->run();

LL-RRT* :

    // integrate Environmentchanges (optional while running)
    planner->update();      or      ->threaded_update();

    // get smooth Path
    planner->getBestPath();
```

Literaturverzeichnis

- [AS11] AKGUN, Baris ; STILMAN, Mike: Sampling heuristics for optimal motion planning in high dimensions. In: *IROS*, IEEE, 2011
- [Bre65] BRESENHAM, J. E.: Algorithm for Computer Control of a Digital Plotter. In: *IBM System Journal* 4 (1965), Nr. 1, S. 25–30
- [CR74] CATMULL, E. ; ROM, R.: A Class of Local Interpolating Splines. In: *Computer aided geometric design. Academic Press.* (1974), S. 317–326
- [DTMD10] DOLGOV, Dmitri ; THRUN, Sebastian ; MONTEMERLO, Michael ; DIEBEL, James: Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments. In: *I. J. Robot Res.* 29 (2010), Nr. 5, S. 485–501
- [Fla14] *FLANN - Fast Library for Approximate Nearest Neighbors.* <http://www.cs.ubc.ca/research/flann/>. Version: September 2014
- [FS] FERGUSON, Dave ; STENTZ, Anthony: The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-uniform Cost Environments. – Forschungsbericht
- [FS05] FERGUSON, Dave ; STENTZ, Anthony: Field D*: An Interpolation-Based Path Planner and Replanner. In: *Proceedings of the International Symposium on Robotics Research (ISRR)*, 2005, S. 1926–1931
- [HNB68] HART, Peter E. ; NILSON, Nils ; BERTRAM, R.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE* (1968)
- [KF10] KARAMAN, Sertac ; FRAZZOLI, Emilio: Incremental Sampling-based Algorithms for Optimal Motion Planning. In: *Robotics: Science and Systems VI, Universidad de Zaragoza*, 2010

- [KF11] KARAMAN, Sertac ; FRAZZOLI, Emilio: Sampling-based algorithms for optimal motion planning. In: *I. J. Robot Res.* 30 (2011), Nr. 7, S. 846–894
- [KL02] KOENIG, Sven ; LIKHACHEV, Maxim: D*Lite. In: *AAAI/IAAI*, AAAI Press / The MIT Press, 2002, S. 476–483
- [KL05] KOENIG, Sven ; LIKHACHEV, Maxim: Fast replanning for navigation in unknown terrain. In: *IEEE Transactions on Robotics* 21 (2005), Nr. 3, S. 354–363
- [KLF04] KOENIG, Sven ; LIKHACHEV, Maxim ; FURCY, David: Lifelong Planning A*. In: *Artificial Intelligence Journal* 155 (2004), S. 93–146
- [KSLO96] KAVRAKI, Lydia E. ; SVESTKA, Petr ; LATOMBE, Jean-Claude ; OVERMARS, Mark H.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In: *IEEE T. Robotics and Automation* 12 (1996), Nr. 4, S. 566–580
- [KTK⁺08] KUWATA, Yoshiaki ; TEO, Justin ; KARAMAN, Sertac ; FIORE, Gaston ; FRAZZOLI, Emilio ; HOW, Jonathan P.: Motion Planning in Complex Environments using Closed-loop Prediction. In: *AIAA Guidance, Navigation, and Control Conference*, 2008
- [KWP⁺11] KARAMAN, Sertac ; WALTER, Matthew R. ; PEREZ, Alejandro ; FRAZZOLI, Emilio ; TELLER, Seth J.: Anytime Motion Planning using the RRT*. In: *IEEE International Conference on Robotics and Automation, ICRA*, 2011, S. 1478–1483
- [LaV98] LAVALLE, Steven M.: Rapidly-Exploring Random Trees: A new Tool for Path Planning. In: *Department of Computer Science* (1998)
- [LaV06] LAVALLE, Steven M.: *Planning Algorithms*. Cambridge, U.K. : Cambridge University Press, 2006. – ISBN 978-0521862059. – Available at <http://planning.cs.uiuc.edu/>
- [LaV11] LAVALLE, Steven M.: Motion planning: The Essentials. In: *IEEE Robotics and Automation Society Magazine*, 2011, S. 79–89
- [LGT03] LIKHACHEV, Maxim ; GORDON, Geoff ; THRUN, Sebastian: ARA*: Formal Analysis. School of Computer Science, Carnegie Mellon University, 2003. – Forschungsbericht

- [LGT04] LIKHACHEV, Maxim ; GORDON, Geoff ; THRUN, Sebastian: ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In: *Advances in Neural Information Processing Systems 16*, MIT Press, 2004
- [LHT⁺08] LEONARD, John J. ; HOW, Jonathan P. ; TELLER, Seth J. ; BERGER, Mitch ; CAMPBELL, Stefan ; FIORE, Gaston A. ; FLETCHER, Luke ; FRAZZOLI, Emilio ; HUANG, Albert S. ; KARAMAN, Sertac ; KOCH, Olivier ; KUWATA, Yoshiaki ; MOORE, David ; OLSON, Edwin ; PETERS, Steve ; TEO, Justin ; TRUAX, Robert ; WALTER, Matthew ; BARRETT, David ; EPSTEIN, Alexander ; MAHELONI, Keoni ; MOYER, Katy ; JONES, Troy ; BUCKLEY, Ryan ; ANTONE, Matthew E. ; GALEJS, Robert ; KRISHNAMURTHY, Siddhartha ; WILLIAMS, Jonathan: A Perception Driven Autonomous Urban Vehicle. In: *Journal of Field Robotics* 25 (2008), Nr. 10
- [Pel11] PELLENZ, Johannes: *Aktive Sensorik für autonome mobile Systeme*. Koblenz, Germany, University of Koblenz-Landau, Diss., 2011. – PhD defence: 3 March 2011
- [Poh70] POHL, Ira: Heuristic search viewed as path finding in a graph. In: *Artificial Intelligence* 1 (1970), Nr. 3-4, S. 193–204
- [Ste95] STENTZ, Anthony: The Focussed D* Algorithm for Real-Time Replanning. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, Morgan Kaufmann, 1995, S. 1652–1659
- [YJSL05] YERSHOVA, Anna ; JAILLET, Leonard ; SIMÃO, Thierry ; LAVALLE, Steven M.: Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain. In: *ICRA, IEEE*, 2005, S. 3856–3861
- [ZH02] ZHOU, Rong ; HANSEN, Eric A.: Multiple Sequence Alignment Using Anytime A*. In: *AAAI/IAAI*, AAAI Press / The MIT Press, 2002, S. 975–977