Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Dahlem Center for Machine Learning and Robotics

Entwurf und Optimierung eines modularen RRT basierten Planers für autonome Autos

Linus Helfmann
Matrikelnummer: 4857593
linus.helfmann@fu-berlin.de

Betreuer:	Prof. Dr. Daniel Göhring
Eingereicht bei:	Prof. Dr. Daniel Göhring
Zweitgutachter:	Prof. Dr. Raúl Rojas

Berlin, 21. November 2018

Zusammenfassung

In dieser Arbeit wurde ein modularer RRT basierter Planer für autonome Autos konstruiert, auf dessen Basis verschiedene Performanceoptimierungen und Erweiterungen wie RRT* Smart und Informed RRT* analysiert wurden. Dabei gelang es, einfach Kurven mit festen Lenkeinstellungen zum schnellen Finden von Wegen zu nutzen, die dann mit Dubins Kurven optimiert wurden. Damit ist gezeigt, dass ein Kompromiss aus Geschwindigkeit und Qualität gefunden werden kann, der für autonome Autos nutzbar ist. Es verbleiben noch einige offene Punkte, die geklärt werden sollten, aber auch Potential für weitere Verbesserungen.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

21. November 2018

Linus Helfmann

Inhaltsverzeichnis

1 Einführung

Autonomes Fahren ist ein sehr aktives und auch gesellschaftlich beobachtetes Forschungsgebiet. Erste autonome Funktionen wie z.B. Einpark- und Spurhalteassistenten gibt es in vielen Autos, doch Autos, die komplett autonom fahren können, brauchen noch Jahre bis es sie verbreitet gibt.
Aktuell gibt es zwar einige Forschungsfahrzeuge, doch diese benötigen meist spezielles Kartenmaterial, können daher nicht überall fahren und müssen zusätzlich die ganze Zeit überwacht werden. Ein erfolgreiches autonomes Auto würde den kompletten Verkehr und damit die Mobilität der Gesellschaft verändern.
In dieser Arbeit konzentriere ich mich daher auf das Fahren ohne Kartenmaterial, konkreter auf das Planen von Wegen zwischen zwei Punkten auf einer Ebene mit Hindernissen.
Es gibt verschiedene Ansätze für das Finden eines Wegen durch einen unstrukturierten Raum, hier verwende ich RRT, da RRT sich sowohl schnell und gleichmäßig in einem Raum ausbreiten kann, aber gleichzeitig auch mit Einschränkungen in den Statusübergängen umgehen kann.

1.1 Verwandte Arbeiten

Diese Arbeit versucht hierzu Erkenntnisse aus den Arbeiten von David Gödicke [10] und Bernd Sahre [16] aufzugreifen und einen Überblick zu verschaffen. David Gödicke hat sich auf RRT^X[15] mit Reeds-Shepp Kurven konzentriert, kam aber zu dem Schluss dass diese zu kompliziert sind. Bernd Sahre hat versucht mit einfacheren Kurven zum Ziel zu kommen, dies gelang ihm aber ebenfalls nicht. Beide hatten aber Vorschläge gemacht, was man probieren könnte, um bestimmte Probleme wie die Performance oder das Finden von Wegen zu verbessern.

1.2 Ziel der Arbeit

In dieser Arbeit versuche ich ein paar dieser Vorschläge aufzugreifen und in einen modularen RRT-basierten Planer einzubauen. Die Modularität soll es dabei einfacher machen neue Erweiterungen einzubauen, damit diese besser unabhängig und auch im Zusammenspiel ausprobiert und analysiert werden können.
Da sowohl einfache Kurven wie auch komplexe Kurven ihre Vor- und Nachteile haben, versuche ich in dieser Arbeite einfache Kurven mit komplexen Kurven zu kombinieren um ihre Nachteile auszugleichen.

1.3 Aufbau der Arbeit

Zum Einstieg gibt es dafür einen kurzen Überblick über RRT in Abschnitt 2, dessen Vor- und Nachteile und welche Probleme allgemein und beim Anwenden auf Autos auftreten. Danach werden in Abschnitt 3 einige Erweiterungen von RRT vorgestellt, welche gewisse Eigenschaften von RRT's verbessern sollen und wie man sie für Autos anpassen muss. Im Abschnitt 4 wird ein Überblick über den modularen Aufbau des Planers gegeben und im Abschnitt 5 gibt es dann ein paar Experimente zum Vergleich der Ergebnisse, welche dann wiederum im Abschnitt 6 genauer beurteilt werden.

2 RRT-basierte Algorithmen für Autonome Autos

2.1 Definition RRT

RRT steht für "Rapidly-Exploring Random Tree" und ist ein Algorithmus der 1998 vorgestellt wurde [14]. RRT ist ein sehr einfacher Algorithmus, der versucht eine schnelle und gleichmäßige Ausbreitung in einem n-dimensionalem Raum zu erreichen. RRT ist zwar etwas älter, wurde aber u.a. genau für solche Probleme mit eingeschränkten Statusübergängen geschaffen. So wird im Artikel explizit ein Auto mit einem 5-dimensionalen Status als Beispielanwendung genannt.

Variablen:
V: die Liste der Knoten im Graph
E: die Liste der Kanten als Knotenpaare
x: Knoten im Graph
Funktionen:
Sample: Generiert neue Position
Nearest(V,x): liefert den Knoten aus V der am nächsten zu x ist
ObstacleFree(x_1, x_2): prüft ob die Kante von x_1 nach x_2 keine Kollision hat
Steer($x_{nearest}, x_{rand}$): Berechnet von x_{rand} aus einen Knoten der näher an $x_{nearest}$ ist

Algorithm 1: RRT (basiert auf Pseudocode aus [12])

```
1  V ← {x_init}; E ← {}
2  for i=1,...,n do
3      x_rand ← Sample;
4      x_nearest ←Nearest(V,x_new);
5      x_new ← Steer(x_nearest, x_rand);
6      if ObstacleFree(x_new, x_nearest) then
7          V ← V ∪ {x_new};
8          E ← E ∪ {(x_nearest, x_new)};
9      end
10 end
```

2.2 Raum

Je nach Definition hat der Status eines Autos für die Pfadplanung verschieden viele Dimensionen. Die drei wichtigsten, auf die wir uns hier konzentrieren, sind die x- und y-Koordinate auf einer Ebene, sowie die Ausrichtung γ. Als weitere Eigenschaften könnte man z.B. die Geschwindigkeit und die aktuelle Lenkeinstellung nehmen. Da wir annehmen, dass Wege eher langsam abgefahren werden und damit die Lenkeinstellung auch schnell geändert werden kann, kann man diese aber erstmal vernachlässigen. Für das Planen von Pfaden für Autos verwenden wir also einen 3-dimensionalen Raum. Die Werte in den einzelnen Dimensionen können aber nicht alle gleichmäßig verteilt werden, da es Einschränkungen in den Statusübergängen gibt. Wenn man nur die x- und y- Koordinaten nimmt, so ist es problemlos möglich,

von einem Zustand A zu einem Zustand B zu gehen. Das Auto fährt einfach in gerader Linie von A nach B mit Richtung $\overrightarrow{AB}$ und Länge $|AB|$. Dafür müsste das Auto sich aber 360°auf der Stelle drehen können. Realistisch ist aber, dass das Auto während des Fahrens lenkt und einen Wendekreis hat. Für den Übergang von Zustand A nach Zustand B ist es dann nicht mehr trivial zu sagen, wie dies gelingt und wie aufwändig die Strecke ist. Man braucht also ein Konzept, das den Weg zwischen zwei Punkten beschreibt.

2.3 Kurven

Es gibt verschiedene Ansätze, die Positionen im Raum zu verbinden. Einige versuchen nur einfache Statusübergänge zu verwenden, sind dafür aber nicht ganz optimal. Andere versuchen den kürzesten Weg zu finden, sind dafür aber sehr komplex. Im folgenden werden dies Statusübergänge als Kurven bezeichnet.

2.3.1 Kurven mit festen Lenkeinstellungen

Der einfachste Ansatz ist, dass das Auto mit einer gegebenen Lenkeinstellung eine kurze Strecke fährt, man erhält also einen Kreisbogen oder eine Gerade. Dies war auch der Fokus der Bachelorarbeit von Bernd Sahre [16]. Diese Kurven lassen sich sehr einfach berechnen. Der Nachteil ist jedoch, dass man so nicht von jedem Punkt A zu jedem Punkt B kommen kann, sondern mehrere Kanten benötigt.
Punkte, die nicht erreichbar sind, muss man also verschieben.

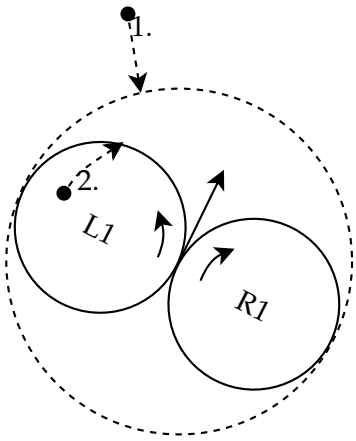

Abbildung 1: Verschieben der neuen Knoten

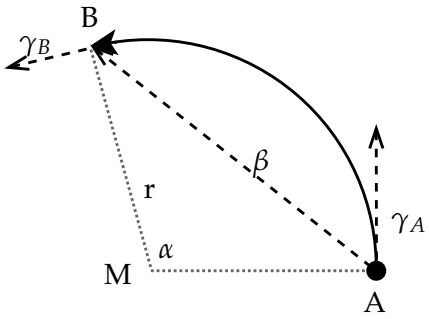

Abbildung 2: Berechnung der Kurvenparameter

Wie man in Abbildung 1 sehen kann, wird als erstes der neue Punkt in den Radius der maximalen Kurvenlänge geschoben. Danach wird geprüft, ob der Knoten in einem der Wendekreise liegt, falls ja, so wird er aus ihm rausgeschoben.
Danach kann man die Kurve berechnen:
Gegeben sind die zwei Positionen $A = (x_A, y_A, \gamma_A)$ und $B = (x_B, y_B, \gamma_B)$.
Wir berechnen den Winkel $\beta = \sphericalangle(\overrightarrow{AB})$ zwischen den beiden Positionen.
Es gilt: Die Differenz von γ_A zu β entspricht der Differenz von β zu γ_B.

Wir definieren $\alpha = 2 * (\beta - \gamma_A)$, dies ist gleichzeit der Winkel des Kreisbogens. Damit können wir dann den gewünschten Winkel am Ende der Kurve berechnen: $\gamma_B = \alpha + \gamma_A = 2 * \beta - \gamma_A$
Wenn γ_B auf diesen Wert gesetzt wird, dann kann ein Kreisbogen zu dieser Position gezogen werden. Man muss aber noch den Radius berechnen.
Falls $\alpha = 0$, ist dies natürlich nicht möglich, da der Knoten direkt vor uns liegt und wir daher eine Gerade haben.
Ansonsten nimmt man das Dreieck $\triangle ABM$ mit M Mittelpunkt des Kreises.
Mit Cosinussatz gilt: $|\overline{AB}|^2 = 2 * r^2 - r^2 * cos(\alpha)$
Daraus folgt Radius $r = \frac{|\overline{AB}|}{\sqrt{(2-cos(\alpha))}}$
Die Länge der Kurve ist dann einfach $\alpha * r$, für α in Bogenmaß.

2.3.2 Dubins Kurven

Dubins Kurven sind kompliziertere Kurven. Sie bestehen aus einer Abfolge von 3 Teilkurven mit festen Lenkeinstellungen und können zwischen jeden zwei Positionen einen Weg finden. Der erste und letzte Teil ist dabei immer eine Rechts- oder Linkskurve, der mittlere Teil kann auch eine Gerade sein. Dabei ist der Radius aller Kurven fest und möglichst klein, um einen möglichst kurzen Weg zu finden.
Es gibt damit 6 Typen: LRL, LSL, LSR, RLR, RSR und RSL, wobei R für Rechts, L für Links und S für Gerade (straight) steht.
Die Berechnung ist dabei noch relativ einfach, man nimmt die Wendekreise der Start- und Endposition und probiert jeden Wendekreis von der Startposition mit jedem Wendekreis der Endposition zu verbinden [9]. Zum Verbinden benutzt man dabei eine Gerade, die tangential zu beiden Kreisen ist und in Fahrrichtung vom Startwendekreis abgehen und in Fahrtrichtung am Endwendekreis ankommt, wie in der Skizze 3 veranschaulicht.

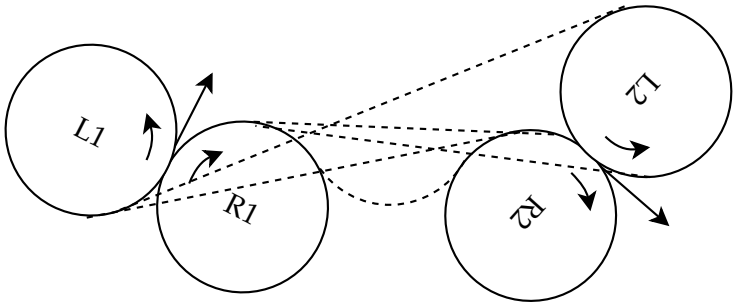

Abbildung 3: Dubins Kurven

Die Länge der Kurve ist dann der Kreisbogenabschnitt auf dem Startwendekreis, die Gerade und der Kreisbogenabschnitt auf dem Endwendekreis. Die Typen LSL und RSR funktionieren dabei immer. Bei LSR und RSL dürfen die Wendekreise dabei nicht überlappen, da sonst keine Verbindungsgerade gefunden werden kann.
Die letzten zwei Typen von Dubins Kurven sind RLR und LRL, wo statt einer Geraden

ein weiterer Kreisbogen eingesetzt wird. Dies funktioniert nur, wenn der Abstand der Mittelpunkte der Wendekreise maximal das vierfache des Wendekreisradius ist, da sonst der mittlere Kreisbogen nicht lang genug ist um den Übergang zwischen den beiden Kreisen zu bilden. Der Mittelpunkt des mittleren Kreises wird dabei über das gleichschenklige Dreieck aus den 3 Kreismittelpunkten berechnet, die Basis ist dabei die Verbindungslinie der zwei Wendekreise und die Schenkel des Dreiecks sind der doppelte Wendekreisradius.

2.3.3 Weitere Kurven

Es gibt noch viele weitere Kurven, u.a. Reeds-Shepp Kurven, welche David Gödicke in seiner Bachelorarbeit [10] verwendet hat. Reeds-Shepp Kurven bestehen ähnlich wie Dubins Kurven auch aus dem Zusammenfügen von mehreren Teilkurven, jedoch sind es 3 bis 5 Teilkurven, wobei auch Richtungswechsel zwischen den Teilkurven stattfinden. Am Ende kommen 48 verschiedene Typen zustande, was die Implementierung aufwändiger macht.
Oder man könnte sich von der festen Lenkeinstellung während des Fahrens lösen und Kurven verwenden, bei denen sich die Lenkeinstellung während des Fahrens ändert. Diese können meist auch schneller abgefahren werden, da man nicht zum Ändern der Lenkeinstellung anhalten muss, wie man es bei Dubins und Reeds-Shepp Kurven theoretisch müsste. Dafür sind sie jedoch noch aufwändiger zu berechnen und auch die Kollisionsberechnungen werden schwieriger.

2.4 Performance

Wie bereits angesprochen ist gute Performance ein Ziel dieser Arbeit. RRT und besonders RRT* (siehe Kapitel 3.1) sind stark vom Finden von Nachbarknoten abhängig und wenn man bedenkt, dass die Abstandsberechnung mit dem ziehen einer Wurzel nicht gerade günstig ist, dann macht dies schnell einen Großteil der Laufzeit aus.
So muss man:
1. beim Einfügen des Knotens den nächsten Nachbarn finden.
2. bei RRT* mehrfach die Nachbarn im Rewireradius r finden.
Der naive Ansatz wäre das Speichern aller Knoten in einer Liste. Zum Finden der Nachbarn würde man dann alle Knoten durchgehen und jeweils den Abstand berechnen. Das gäbe für den 1. Fall eine Performance von $O(n)$ pro neuem Knoten, also $O(n^2)$ insgesamt. Für 2. hätte man $O(n * log(n))$ pro Knoten bzw. $O(n^2 * log(n))$ insgesamt, wobei $log(n)$ die zusätzlichen Knoten sind, von denen man die Vorgänger auch neu berechnen muss.

2.4.1 Verbesserungsansatz Caching

Der erste Ansatz ist das Zwischenspeichern der Abfragen, denn im 2. Fall wird die selbe Abfrage, über die Laufzeit des Programmes, mehrfach gestellt. Man speichert also an jedem Knoten seine Nachbarn. Dies funktioniert natürlich nur unter der Annahme, dass Knoten sich nicht mehr bewegen können und der Rewireradius r konstant ist.
Wenn nun ein neuer Knoten eingefügt wurde und an seinem endgültigen Platz ist,

berechnet man alle seine Nachbarn und speichert diese als Liste an ihm. Zusätzlich trägt man den neuen Knoten bei seinen Nachbarn ebenfalls als Nachbar ein.
Dies sorgt beim 2. Fall für eine Performance in $O(n + log(n)) = O(n)$ pro neuem Knoten bzw. $O(n^2)$ für alle Knoten, da man nur für den neuen Knoten alle Nachbarn berechnen muss.

2.4.2 Verbesserungsansatz Datenstruktur

Ein weiterer Ansatz ist es, die Datenstruktur, in der die Knoten gespeichert werden, anzupassen. Diesen Vorschlag hatte auch Bernd Sahre [16, Kapitel 2.6] erwähnt.
Da man meist nur lokale Knoten finden will, wäre es hilfreich, dies in der Datenstruktur abzubilden. Der einfachste Ansatz ist dabei, die 2D-Ebene in quadratische Boxen mit Kantenlänge k aufzuteilen und nur die Knoten in nahen Boxen zu prüfen. k sollte dabei in Zusammenhang mit dem Rewireradius r gewählt werden. Man will nicht zu große Boxen haben, so dass man zu viele unnötige Knoten prüfen muss. Sie sollten aber auch nicht zu klein sein, da man dann zu viel Aufwand mit der Behandlung der Boxen hat.
Zum Suchen des nächsten Nachbarn fängt man dann in der entsprechenden Box an und geht in Ringen langsam nach außen, bis man einen Knoten gefunden hat. Danach prüft man noch, dass es keine Box mehr geben kann, in der sich Knoten befinden können, die näher sind. Dies können anfangs viele leere Boxen sein, welche man aber schnell überspringen kann. Wenn der Graph voller wird, so muss man dann nur noch wenige bzw. eine konstante Anzahl von Boxen prüfen. Mit Knotendichte d Knoten pro Feld liegt die Performance von 1. in $O(d)$ pro neuem Knoten bzw. $O(d * n)$ insgesamt. Für das Finden aller Nachbarn im Radius r ist es ebenfalls abhängig von der Knotendichte, da mit festem k eine konstante Anzahl von Feldern zu prüfen ist. Wir liegen also in $O(d + log(n))$ pro Knoten bzw. $O((d + log(n)) * n)$ insgesamt unter Annahme das Caching gleichzeitig aktiviert ist.

2.5 Wurzelposition und dynamisches Planen

Eine weitere Designentscheidung, die man treffen muss, ist die Wurzelposition. RRTs haben einen fixierten Wurzelknoten, von dem aus der Baum aufgebaut ist und zu dem alle Knoten zurückgehen. Optimierungen bei RRT* und anderen Algorithmen laufen immer zu diesem Wurzelknoten hin.
Wenn man dynamisch planen und Teile des Graphens wiederverwenden will, so muss man sich Gedanken darüber machen, wo man den Graphen verankert.

2.5.1 Fixierte Autoposition

Die Standardvariante ist, die Autoposition als Wurzel für den Baum zu verwenden. Dies ermöglicht das gleichzeitige Berechnen von Wegen zu mehreren möglichen Zielen. Wenn z.B. das Auto steht und mehrere Parklücken gefunden wurden, kann man so auswählen, ohne neu zu berechnen. Oder wenn sich das Ziel plötzlich ändert während des Fahrens, kann man auch schnell reagieren.
Wenn sich das Auto bewegt, muss man aber den kompletten Teilbaum der am letzten

Knoten hing wegwerfen, oder ihn zumindest umhängen. Bei Bäumen ist es theoretisch jederzeit möglich, die Wurzel auf einen anderen Knoten umzulegen. Es muss nur die Kante zum Vorgänger bei allen Knoten umgedreht werden, die noch nicht im Teilbaum unter der neuen Wurzel hingen.
Es kommt aber hinzu, dass man die Kosten bei all diesen Knoten neu berechnen muss und diese Knoten schlecht von der neuen Autoposition erreichbar sind, weshalb das Verwerfen eventuell einfacher ist. Bei den Knoten die bereits unter der neuen Wurzel hängen, könnte man die Kosten auch einfach beibehalten, da die Kosten am Start nicht unbedingt Null sein müssen, es reicht wenn sie konstant sind.
Ein Nachteil ist aber, dass der Baum bei Lenkfehlern nicht so flexibel ist. Wenn das Auto ungenau lenkt und nicht auf einer geplanten Position landet, dann muss die Position des Autos als neuer Knoten in den Baum eingefügt werden. Damit man aber von dieser Position zum Ziel kommt, müsste man erst zurück zur Wurzel fahren, denn vom aktuellen Knoten existiert kein anderer Weg. Natürlich kann man auch einen Knoten auf dem Weg zwischen Wurzel und Ziel nehmen, der besser erreichbar ist, aber das funktioniert eventuell nicht immer.

2.5.2 Fixiertes Ziel

Die Alternative ist das Fixieren des Ziels, dies ermöglicht eine flexible Start- und damit eine flexible Autoposition. Wenn sich das Auto bewegt, bleibt der Graph valide. Selbst ein Abkommen vom Weg ist kein Problem, da wir von jedem Knoten einen Weg zum Ziel kennen, für die neue Autoposition muss also bloß ein neuer Knoten eingefügt werden und es kann weitergehen.
Der Nachteil ist natürlich, dass man das Ziel nun nicht mehr so einfach ändern kann, aber da die Zieländerung meist eine größere Änderung ist, wäre hier eine verzögerte Reaktion des Algorithmus akzeptabel.

2.5.3 Weitere Ansätze

Ein weiterer Ansatz ist es, zwei Bäume zu verwenden und sowohl die Position des Auto als auch das Ziel zu fixieren, das macht z.B. RRT-Connect [13]. Dies würde den Algorithmus aber noch statischer machen und nicht dynamischer, wie es gewünscht ist.
Ein letzter, eher theoretischer Ansatz ist, eine anderen Knoten als Wurzel zu fixieren. Dies würde ein Verschieben von Start und Ziel erlauben, hätte aber das Problem, dass man eine optimale Position für die Wurzel finden muss. Denn durch diesen Knoten muss man immer fahren. Wenn man aber diese optimale Position berechnen kann, dann weiß man schon soviel, dass man auch gleich den Weg berechnen kann. Und wenn die Position durch die Umgebung gegeben ist, z.B. eine Tordurchfahrt oder andere bauliche Einschränkung, dann kann man auch gleich diese Position als Ziel setzen und parallel einen zweiten RRT berechnen, der vom Ziel des ersten RRT zum ursprünglichen Ziel fährt.

2.5.4 Fazit

Die Entscheidung ist also klar, wenn das Ziel wirklich fest ist, was es meistens ist, dann legt man dahin die Wurzel, da man nicht vor hat es zu ändern.
Wenn das Ziel unbekannt ist, dann kann man die Wurzel auch nicht dahin legen und muss sie also auf die aktuelle Position legen.
Für die Implementierung wurde also immer das Ziel als Wurzel verwendet, da das Ziel fest vorgegeben ist.

3 Analyse von RRT-Erweiterungen

3.1 RRT*

RRT* ist eine der bekanntesten Erweiterungen von RRT, sie versucht bei neu hinzugefügten Knoten den Vorgängerknoten zu optimieren [12].
RRT nimmt immer den nächsten Knoten, dieser ist aber nicht unbedingt der optimale. RRT* führt daher die neue Funktion rewire hinzu. Rewire wird nach dem Hinzufügen des Knotens zum Graph auf den neuen Knoten ausgeführt und versucht einen besseren Vorgänger zu finden.

Funktionen:
Near(V,x,r): liefert die Knoten aus V, deren Abstand zum Knoten x kleiner als r ist
Cost(x): Kosten des Knotens
Cost(x_1, x_2): Kosten der Kante zwischen x_1 und x_2
Parent(E,x): bestimmt Vorgänger von x

Algorithm 2: rewire(V,E,x,isnew,r)

```
1  if isnew then
2      x_parent ← Parent(E,x);
3      foreach x_near ∈ Near(V,x,r) do
4          if Cost(x_near) + Cost(x_near, x) < Cost(x) ∧ ObstacleFree(x_near, x) then
5              x_parent = x_near;
6          end
7      end
8      E ← (E \ {(Parent(E,x),x)}) ∪ {(x_parent, x)};
9  end
10 foreach x_near ∈ Near(V,x,r) do
11     if Cost(x) + Cost(x, x_near) < Cost(x_near) ∧ ObstacleFree(x, x_near) then
12         E ← (E \ {(Parent(E,x_near), x_near)}) ∪ {(x, x_near)};
13         rewire(V,E,x_near,false,r);
14     end
15 end
```

In der Funktion rewire wird für den neuen Knoten der Nachbar mit Abstand kleiner als r als Vorgänger gewählt, mit dem der neue Knoten die geringsten Kosten hat. Danach wird geprüft, ob es einen Nachbarknoten mit Abstand kleiner als r gibt, für den der neue Knoten ein besserer Vorgänger ist. Falls ja, so ersetzt der neue Knoten den

Vorgänger des Nachbarknoten und rewire wird auf den Nachbarknoten aufgerufen.

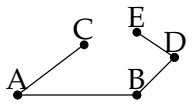

Abbildung 4: ohne RRT*

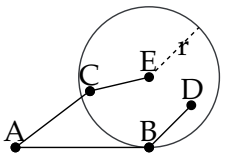

Abbildung 5: mit RRT*

Im Abbildung 4 sieht man den neu hinzugefügten Knoten E. Der nächstgelegene Knoten zu E ist D, also wird D als Vorgänger gewählt. In Abbildung 5 sieht man das rewiring. Ausgehenden von E werden alle Nachbarn innerhalb des Kreises geprüft. C ist der Nachbar, wo die Summe der Kosten des Knotens und der Kosten der Kante zum Knoten am geringsten ist, deshalb wird C als neuer Vorgänger gewählt.
Mit RRT* werden also die Kanten zur Wurzel hin optimiert. Mit dieser Erweiterung findet RRT asymptotisch den optimalen Weg, daher ist diese Erweiterung essentiell. Dadurch, dass aber nicht alle Knoten mit einer Kurve verbunden werden können, muss man darauf achten, auch zu prüfen ob die Kurve möglich ist.

3.2 Informed RRT*

Mit Informed RRT* wird versucht den Raum, in dem neue Positionen gesampled werden, zu reduzieren [8]. Dies soll die Konvergenz zum kürzesten Weg beschleunigen, da keine Positionen mehr gesampled werden, die sowieso zu abgelegen liegen, um diesen zu verbessern.

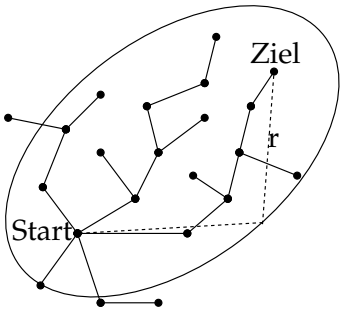

Abbildung 6: Ellipse von Informed RRT*

Sobald der erste Weg gefunden wurde, liegen alle gleichlangen und kürzeren Wege innerhalb einer Ellipse mit Brennpunkten Start und Ziel sowie den Kosten der Strecke als Radius, wie in Abbildung 6 dargestellt. Man muss also nur noch in dieser Ellipse planen, denn alle Wege, die über Knoten außerhalb der Ellipse gehen, sind länger als der kürzeste gefundene Weg.
Da diese Erweiterung nur das Samplingverhalten verändert, kann sie auch ohne Anpassung mit Kurven verwendet werden.

3.3 RRT*-Smart

RRT*-Smart ist eine Erweiterung, die versucht gefundene Wege zu optimieren, indem sie neue Positionen berechnet, die den Weg verbessern [11].
Sobald der erste Weg gefunden wurde, wird in regelmäßigen Abständen, statt einer zufälligen eine berechnete Position verwendet. Dazu nehmen wir zwei Knoten im kürzesten Weg, zwischen denen ein weiterer Knoten liegt. Zwischen diesen beiden Knoten berechnen wir die kürzeste Kante. Ist die Kante kürzer als die Summe der beiden Kanten, so generieren wir einen Punkt auf dieser Kante.
Für unser Problem funktioniert das nicht immer, da wir nicht einfach mit der Dreiecksungleichung arbeiten können. Es gibt nicht zwischen jeden zwei beliebigen Knoten einen Weg, deshalb müssen wir ein paar Änderungen durchführen.
1. Wenn wir mit festen Lenkeinstellungen fahren, verwenden wir zur Berechnung der Abkürzung statt festen Lenkeinstellungen komplexere Kurven.
2. Damit die Wege einfacher und schneller optimiert werden, nehmen wir nicht nur 2 Knoten mit einem Abstand von zwei Kanten, sondern Knoten welche einen Abstand von 3 oder 4 Kanten besitzen.

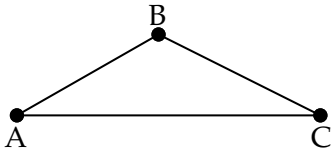

Abbildung 7: RRT* Smart mit Geraden

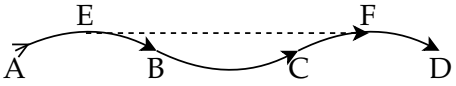

Abbildung 8: RRT* Smart mit Kurven

In Abbildung 8 sieht man, dass es nicht möglich ist, einen besseren Weg von A nach C zu finden als den vorhandenen. Von A nach D ist es aber möglich entlang der gestrichelten Linie zu fahren.
RRT erzeugt meist Zickzackwege, besonders mit Kurven erhält man einen andauernden Wechsel zwischen Rechts- zu Linkskurven, der vermeidbar ist. RRT*-Smart begradigt diese Wege und macht sie damit besser abfahrbar. Damit ist RRT*-Smart gut geeignet um gefundene Wege schnell zu optimieren und die Abfahrbarkeit zu verbessern.

3.4 RRT* Fixed Nodes

Bei RRT* FN (Fixed Nodes) geht es darum, dass man die Anzahl der Knoten beschränkt [4]. Der Graph wird zwar mit steigender Anzahl der Knoten besser, die Ressourcenanforderungen steigen aber auch, was in Echtzeitsystemen ein Problem ist. Bei RRT*FN probiert man Knoten mit wenig oder keinem Nutzen zu entfernen, so dass der Graph nicht zu groß wird. Dies wäre ebenfalls eine gute Erweiterung für RRT, da bei einer Optimierung immer Knoten übrig bleiben, die durch Bessere ersetzt wurden und nicht mehr benötigt werden. Aus Zeitgründen wurde die Erweiterung aber nicht umgesetzt.

3.5 Bessere Auswahl des Vorgängerknoten

Da es Einschränkungen bezüglich der Winkel der Knoten gibt, die durch eine Kurve verknüpft werden sollen, macht es nicht immer Sinn einfach nur den nächsten Knoten zu wählen. Bei Verwendung von Dubins Kurven führt das zu unnötig langen Kanten und einem unübersichtlichen Graphen. Denn selbst wenn zwei Knoten nur 0,5 m entfernt sind, aber genau in entgegengesetzte Richtung schauen, kann es sein, dass man mehrere Meter fahren muss um zu wenden. Ein Ansatz ist also den Elternknoten nicht nur nach Nähe auszuwählen, sondern auch ob man von ihm mit geringen Kosten zum aktuellen Ort kommt. Oder man den aktuellen Ort damit besser erreichbar macht.
Dies führt bei Kurven mit festen Lenkeinstellungen dazu, dass nicht so viele Knoten verschoben werden müssen, die dann am minimalen Wendekreis liegen und zu übereinanderliegenden Kurven führen. Das heißt wir haben viele Knoten, die zu ähnlich sind und keinen Mehrwert bieten, dies wollen wir ändern.
Als Kurzbezeichnung wird diese Erweiterung in dieser Arbeit auch als RRT* BestAncestor bezeichnet.

3.6 Verbindung des Ziels mit dem Graphen

RRT* mit Kurven mit festen Lenkeinstellungen sind nicht immer in der Lage einen Weg zu finden, denn dazu muss der letzte Knoten genau richtig liegen, um zum Ziel eine Kurve bauen zu können. Deshalb prüfen wir, ob der Knoten nah genug am Ziel ist, z.B. weniger als 1 m entfernt. Falls ja, dann probiert man mit einer komplexeren Kurve die Verbindung aufzubauen. Falls dies funktioniert, zerlegen man die komplexe Kurve in einfachere Kurven und fügen die neuen Knoten hinzu.
Dies erhöht die Wahrscheinlichkeit, dass ein Weg gefunden wird und ist damit hilfreich, besonders da andere Erweiterungen stark vom frühen Finden eines Weges abhängig sind. Als Kurzbezeichnung wird diese Erweiterung in dieser Arbeit auch als RRT* Hybrid bezeichnet, da sowohl einfach als auch komplexe Kurven verwendet werden.

4 Umsetzung

4.1 Vereinfachungen

Zur Vereinfachung des Problems gehe ich davon aus, dass das Auto sich beim vorwärts und rückwärts Fahren identisch verhält, d.h. wo das Auto vorwärts durchpasst, passt es auch rückwärts durch. Besonders bei engen Situationen wie dem Einparken ist dies aber leider nicht der Fall. In solchen Fällen sollte das Auto aber sowieso eher langsam fahren und man hat daher die Zeit aufwändigere Kollisions- und Kurvenberechnungen durchzuführen. Des Weiteren wird bei der Kollisionserkennung die Größe des Autos auf die Hindernisse aufaddiert, sodass man nur Punkte auf dem Weg testen muss.
Als Testfahrzeug wurde das Modelauto der FU-Berlin "AutoNOMOS Mini v3.1" [5] sowie der Seat Car Simulator [6] verwendet, wobei als Framework zur Kommunikation mit dem Auto ROS [7] eingesetzt wurde. Da dies jedoch keinen großen Einfluss auf die Umsetzung hat, wird darauf hier nicht weiter eingegangen.
Der für diese Arbeit entstandene Quellcode ist in Form eines ROS-Package im Gitlab des Fachbereichs Mathematik und Informatik der Freien Universität Berlin zu finden [2].

4.2 Modularer Aufbau

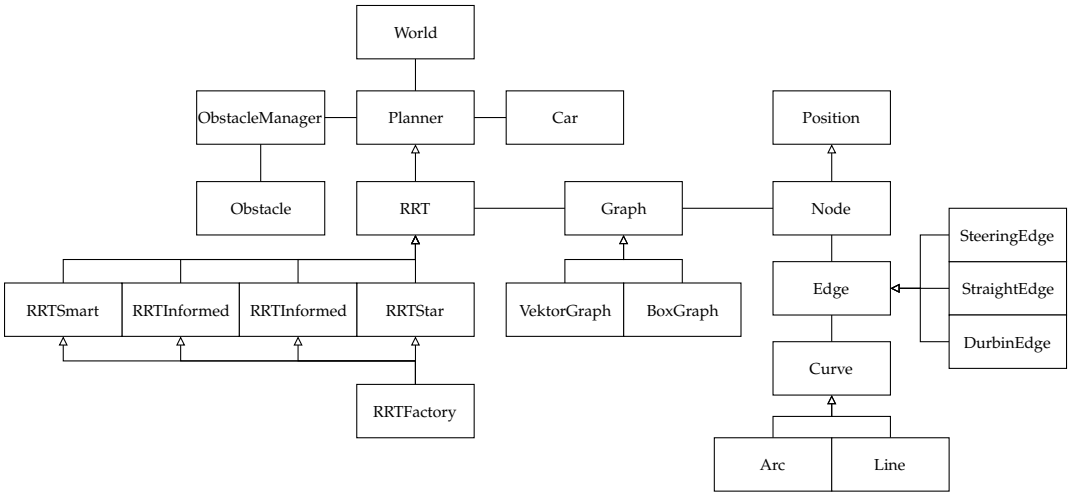

Abbildung 9: Klassenstruktur

Ein Ziel der Arbeit war es Quellcode objektorientiert und möglichst modular aufzubauen, um die verschiedenen Erweiterungen gut implementieren und vergleichen zu können. Der erste Schritt dazu war, alles Externe, also u.a. die Kommunikation mit dem Auto über ROS, von allem Internen, also der Implementierung von RRT, abzugrenzen. Deshalb gibt es die Oberklasse Planner, diese ist zentral in der Klassenstruktur und wird als Interface zum RRT verwendet. An Planner hängt alles Externe, u.a. die Klasse Car, welche die aktuelle Position des Autos bestimmt und World, was Informationen zur aktuellen Umgebung beinhaltet.

Des Weiteren kümmert sich Planner um den ObstacleManager, der für die Kollisionsberechnung mit Hindernissen zuständig ist. Genauere Infos dazu später in Kapitel 4.7.

4.3 RRT

Von der Klasse Planner erbt die Klasse RRT. Hier ist die eigentliche Implementierung des Algorithmus RRT. Dabei wurde der Algorithmus in möglichst viele inhaltliche Teilschritte zerlegt, damit diese dann später einzeln von Erweiterungen angepasst werden können.
Von der Klasse RRT erben dann die Erweiterungen, welche neue Funktionen bereitstellen, sie überschreiben die vorhandenen Funktionen aber noch nicht. Denn das Zusammenspiel der verschiedenen Algorithmen untereinander kann nicht immer allgemein definiert werden.
Stattdessen gibt es die Klasse RRTFactory, sie erzeugt RRT-Objekte mit verschiedener Zusammensetzung aus den Erweiterungen und achtet darauf, dass sie korrekt miteinander zusammenarbeiten.

4.4 Graph

Der RRT benötigt einen Graph bzw. einen Tree. Da wir hier teilweise auch alle Nachbarn speichern hat er den Namen Graph bekommen. Dieser ist in die Klasse Graph ausgelagert, welche als Interface für die zwei Implementierungen VectorGraph und BoxGraph dient. Der VektorGraph verwendet einfach nur die Datenstruktur Vector, eine Art von Array. Der BoxGraph unterteilt die Knoten in Boxen, um einfacher nahe Knoten zu finden, mehr dazu in Kapitel 4.6.
Der Graph besteht aus Knoten (Klasse Node) und Kanten (Klasse Edge).
Die Knoten sind dabei eine Unterklasse von der Klasse Position, welche die Informationen zum Status des Fahrzeug beinhaltet. Die Klasse Node fügt alles hinzu, was für die Benutzung im Graphen benötigt wird.

4.5 Kanten

Die Kanten sind wie in Kapitel 2.3 erwähnt die Statusübergänge. Implementiert wurde das Fahren mit festen Lenkeinstellungen sowie Dubins Kurven. Um die Implementierung der Kanten zu verstecken, besitzen Kanten wiederum Kurven (Klasse Curve). Das sind entweder Geraden (Klasse Line) oder Kreisbögen (Klasse Arc). Damit lassen sich konstante Lenkeinstellungen abbilden, Dubins Kurven werden entsprechend aus drei Kurven-Objekten zusammengesetzt. Auch Reeds-Shepp-Kurven wären so möglich. Dynamische Kurven, bei denen sich die Lenkeinstellung zwischen zwei Knoten kontinuierlich ändert, können jedoch nicht dargestellt werden. Dafür müsste man jedoch nur eine neue Unterklasse von Curve einfügen, welche sie geometrisch beschreibt.

4.6 BoxGraph

Wie bereits in Kapitel 2.4.2 angesprochen wird die Ebene in Boxen aufgeteilt, in denen die einzelnen Knoten gespeichert werden.
Der einfachste Ansatz wäre ein gegebenes Rechteck zu unterteilen und alle Boxen durchzunummerieren. Dies hat jedoch den Nachteil, dass Knoten außerhalb des Rechtecks nicht eingefügt werden können. Man könnte sie zwar in die Randboxen packen, diese wären dann aber stärker gefüllt als andere. Des Weiteren ist es nicht möglich, nachträglich das Gesamtfeld zu ändern.
Ein besserer Ansatz ist also die gesamte Ebene als mögliches Feld zu nehmen und mit einer Map nur die nötigen Boxen zu speichern.
Als Index wird dazu ein Paar aus x-Index und y-Index verwendet $I = (\lfloor \frac{x}{k} \rfloor, \lfloor \frac{y}{k} \rfloor)$, dieser wird dann auf ein Array gemappt, welches die Knoten enthält.

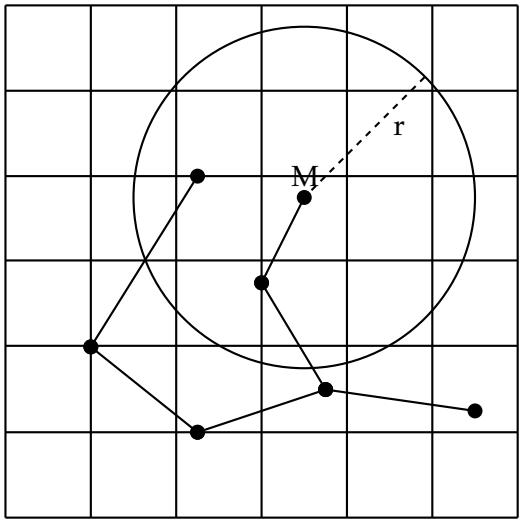

Abbildung 10: Alle Knoten in Radius r finden

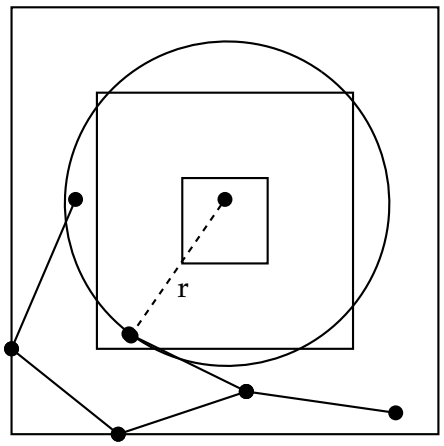

Abbildung 11: Nächsten Knoten finden

Wir haben zwei Anforderungen:

Anforderung 1: Es sollen alle Knoten in einem Radius r um Knoten M gefunden werden.
Die erste Einschränkung, die man machen kann, ist dass man nur Boxen in einem Quadrat um den Knoten M durchsuchen muss. Dazu berechnet man den minimalen und maximalen x sowie y Index aus dem Knoten und dem Radius.
Um nicht unnötigerweise alle Knoten in einer Box zu prüfen, prüft man ob die Box existiert und ob die nächste Ecke näher ist als r. Nur dann wird der Inhalt der Box geprüft, sonst nicht.
Dies sorgt dafür, dass nur die Knoten geprüft werden, die eine Chance haben näher als r zu sein und reduziert die Anzahl deutlich. In Abbildung 10 ist der Graph mit den Boxen und dem Radius r dargestellt.

Anforderung 2: Es soll der nächsten Knoten gefunden werden.
Hierzu wird schrittweise vorgegangen um möglichst früh aufhören zu können, man geht also mit quadratischen Ringen vor, wie in Abbildung 11 dargestellt. Sobald in

einem Ring ein bester Knoten gefunden wurde, sucht man mit dem Algorithmus aus 1. nochmal alle Boxen zusammen, in denen sich Knoten befinden können, die näher als der aktuell nächste Knoten sind. Dann werden alle Knoten in diesen Boxen noch geprüft, es sei denn, die Boxen wurden bereits geprüft. Damit wird sichergestellt, dass kein Knoten übersehen wurde, der eventuell näher ist.
Bei der Umsetzung war wichtig auf den Umgang mit den Daten zu achten. Das Kopieren der Arrays ist aufwändig, man sollte also immer darauf achten nur den Index durchzureichen und mit diesem auf die Daten zuzugreifen. Ein erster Umsetzungsversuch hat über die Listen iteriert und sie dabei kopiert. Dadurch war die Performance schlechter als direkt durch alle Knoten zu iterieren.

4.7 Kollisionsberechnung

Die Kollisionsberechnung sollte möglichst einfach und unabhängig von der Art der Kurven sein. Deshalb basiert sie nur auf Punkten. Jede Kurve berechnet Punkte, die in einem regelmäßigen Abstand auf ihr liegen. Diese werden dann auf Kollision geprüft. Um diesen Vorgang zu Beschleunigen wird eine Kollisions-Map verwendet, die wie der BoxGraph die Ebene in kleine Quadrate teilt, die entweder eine Kollision haben oder nicht.

4.8 Technische Herrausforderungen

Das Programm ist in C++ geschrieben und aus Performancegründen werden viele Zeiger verwendet, dies hat aber auch zu Problemen geführt. Für die Kanten war es möglich Smartpointer zu verwenden, für die Knoten ist dies in bei der vorhandenen Klassenstruktur problematisch. Daher muss man sich bei jeder Erweiterung, die selber Knoten erzeugt, löscht oder zwischenspeichert Gedanken über die Lebensdauer der Knoten machen.
Dies behinderte bei der Entwicklung und sorgte regelmäßig für schwer nachzuvollziehende Segfaults. Durch einen Umbau auf Smartpointer auch für die Knoten, kann dies aber behoben werden.
Das Problem ist dabei der Zusammenhang zwischen Position und Node. Knoten werden aus Performancegründen per Pointer durchgereicht, Position will man einfach per Referenz durchreichen. Einige Funktionen müssen aber Beides akzeptieren, was eher unschön ist. Die Lösung ist hier wahrscheinlich Position nicht mehr als Oberklasse von Node zu verwenden, sondern stattdessen als Membervariable.
Hinzu kommen Performanceprobleme wegen des dauernden ”allocate” und ”free” von kleinen Speicherbereichen, was durch Smartpointer sogar noch verschlimmert wird. Dies machte teilweise 20 bis 30% der Laufzeit aus laut Valgrind [3] (mit Tool Callgrind). Hier könnte man z.B. mit einem ObjectPool, wo Speicherbereiche in größeren Mengen reserviert und dann verteilt werden, die Performance verbessern.

5 Experimente

5.1 Methodik

Die Umsetzung der Algorithmen wurde auf dem "AutoNOMOS Mini v3.1" [5] getestet, dieses besitzt einen Odroid Xu4 mit 2GB RAM, auf dem der Code ausgeführt wurde.
Der Odroid Xu4 [1] hat als CPU eine Exynos 5422, die aus vier schnellen und vier stromsparenden ARM-Kernen besteht. Die CPU ist im Vergleich zu x86-CPUs eher langsam, besonders da RRT nicht einfach parallelisierbar ist. So war ein Intel Core i5-6500 sieben mal so schnell in den folgenden Tests. Für die folgenden Ergebnisse wurden die Laufzeiten von mehreren Durchläufen gemessen und dann gemittelt. Für die Performancetests in Kapitel 5.2 wurden dazu je fünf Messwerte auf einem Intel Core i5-6500 ermittelt, für die Vergleiche der Erweiterungen in Kapitel 5.3 waren es zehn Messwerte auf dem Modelauto.
Bei den Vergleichen der Erweiterungen werden auch die Strecken verglichen, hierbei ist zu beachten, das nicht jeder Durchlauf einen Weg gefunden hat. Das Laufzeitdiagram zeigt also den Mittelwert von zehn Messwerten, während es bei der Strecke eventuell nur einer ist. Dies beeinflusst teilweise auch stark die Laufzeit, da einige Erweiterungen nur bei gefundenen Wegen aktiv werden und so nur in einem oder zwei der Durchläufe die Laufzeit beeinflusst haben. Dies wird aber nochmals bei der Analyse der einzelnen Diagramme in Kapitel 5.3 erwähnt.

5.2 Performance-Optimierungen

Zum Messen der Performanceunterschiede wurden die zwei Implementierungen des Graphen in einem einfachen Testfeld gegenübergestellt.
Es handelte sich um eine leere Fläche von 10 x 10 m, Start und Ziel befanden sich in zwei gegenüberliegenden Ecken, 2 m von den Seitenwänden entfernt. Es wurden keine Kurven verwendet sondern einfach nur Geraden für die Berechnung der Abstände zwischen den Knoten.
Der Test wurde für jede Einstellung fünf mal durchgeführt mit 5.000 Knoten pro Graph und Rewireradius r = 0,3 m.

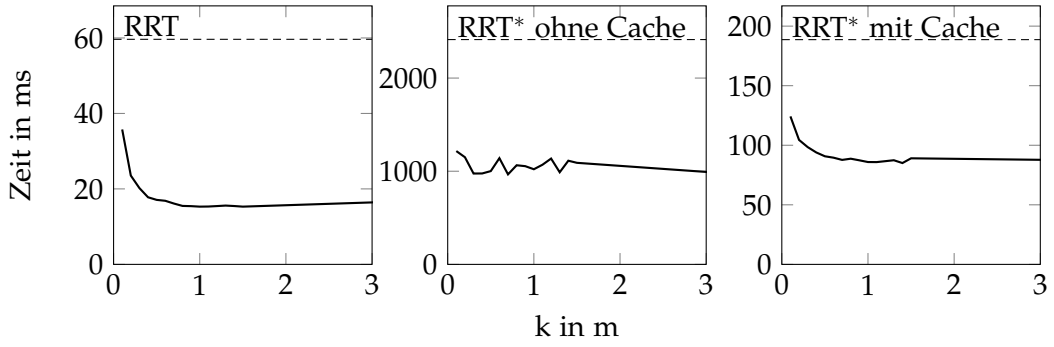

Abbildung 12: Laufzeit abhängig von Boxgröße k

Als Erstes erfolgt die Bestimmung der optimalen Boxgröße k. Zu kleine Boxen haben den Nachteil, dass zu viel Management für sie betrieben werden muss. Bei zu großen

Boxen müssen aber zu viele Abstände berechnet werden.
Im Abbildung 12 sieht man die Laufzeit für verschiedene k's für RRT, RRT* ohne Cache und RRT* mit Cache. Es wurde für k = 10 cm bis 1,5 m in 10 cm Schritten gemessen und dann nochmals für 3 m, um den Trend für größere k zu prüfen. Als Vergleich ist die Laufzeit mit VectorGraph als gestrichelte Linie dargestellt.
Bei RRT und RRT* mit Cache kann man eindeutig den Mehraufwand durch das Management der Boxen sehen, bei k = 0,1 m und 0,2 m ist die Laufzeit deutlich größer. Bei RRT* ohne Cache ist der Ausschlag bei 0,1 m kaum vom Rest des Graphens zu unterscheiden. Bei 0,5 bis 1,5 m liegt bei RRT und RRT* mit Cache das Minimum, danach steigt die Laufzeit wieder an, wenn auch langsam. Aus RRT* ohne Cache lässt sich nichts genaueres ablesen, da die Messwerte zu sehr varieren.
Es lässt sich also schließen, dass für diese Faktoren zwischen 0,5 m und 1,5 m ein guter Wert für k ist, deshalb wurden für alle weiteren Messungen k = 1 m verwendet. Bei geänderten Faktoren wie Knotendichte und Rewireradius wäre natürlich ein anderes k optimal.

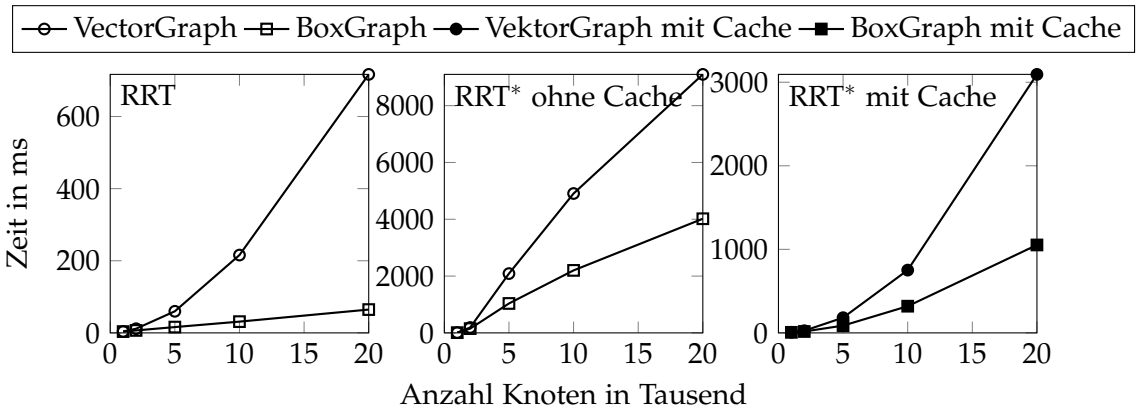

Abbildung 13: Performancevergleich Box- versus Vektorgraph

In Abbildung 13 wird der BoxGraph im Vergleich zum VektorGraph für RRT, RRT* ohne Cache und RRT* mit Cache bei verschiedenen Knotenzahlen dargestellt.
Bei RRT skaliert der BoxGraph fast linear, während der Vektorgraph wie erwartet eine Parabel bildet. Bei RRT* ohne Cache sind die Ergebnisse überraschend, hier würde man auch eine Parabel erwarten. Bei RRT* mit Cache erhält man dann wieder eine Parabel. Bei allen drei Algorithmen ist der BoxGraph deutlich schneller als VektorGraph, wobei sich der Abstand bei mehr Knoten sogar noch vergrößert.
Dass bei RRT und RRT* mit Cache der Abstand größer wird liegt daran, dass das Finden des nächsten Nachbarn deutlich schneller wird. Bei mehr Knoten ist der nächste Nachbar näher und die Wahrscheinlichkeit das man nur die aktuelle Box prüfen muss steigt, die Laufzeit sinkt also.
Bei RRT* ohne Cache müssen aber häufig alle Nachbarn gefunden werden. Hier ist die Laufzeit abhängig von der Knotendichte und daher relativ zur Anzahl der Knoten. Die Laufzeit steigt also relativ, wie als würden wir alle Knoten prüfen.
Für den relevanten Bereich von unter 10.000 Knoten bei RRT* mit Cache kann so die Laufzeit um einen Faktor von zwei bis drei verbessert werden.

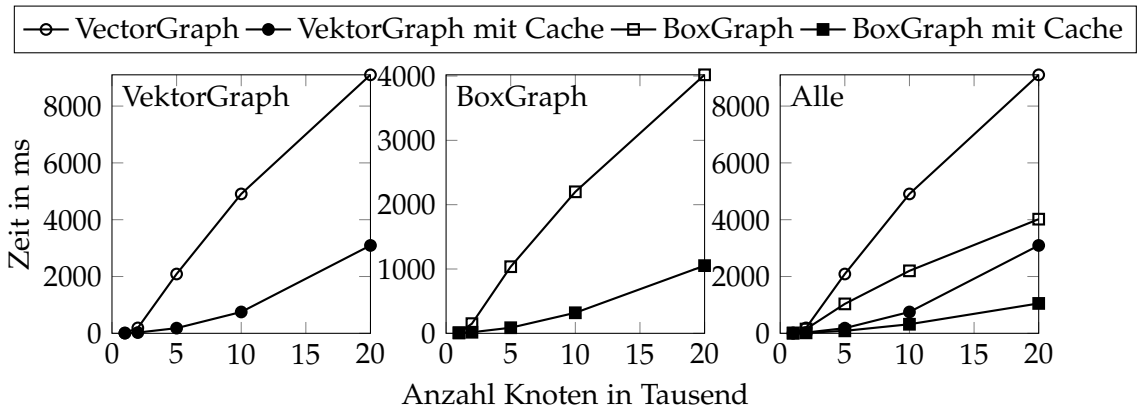

Abbildung 14: Performancevergleich mit/ohne Cache

In Abbildung 14 sieht man nochmal die Daten aus Abbildung 13, diesmal aber nur für RRT* und aufgeteilt nach Art der Graph-Implementierung, damit man sehen kann, wie sich das Zwischenspeichern der Nachbarknoten auswirkt.
Bei 1.000 und 2.000 Knoten ist der Unterschied noch minimal, da die Knotendichte noch nicht so hoch genug ist. Damit kommt es eher selten zu einem rekursiven Rewiring, sodass der Cache kaum Vorteile bietet. Bei 5.000 und mehr Knoten steigt der Abstand aber stark an, hier kommt es wegen der höheren Knotendichte häufiger zu rekursivem Rewiring, sodass der Cache das häufige Neuberechnen spart.
Für den Bereich von 5.000 bis 10.000 Knoten, ist der RRT* mit Cache teilweise bis zu 10 mal so schnell wie RRT* ohne Cache.
Wenn man jetzt noch ohne Optimierung, also VektorGraph ohne Cache mit BoxGraph mit Cache vergleicht, so ist der BoxGraph mit Cache teilweise 20mal schneller, die Optimierung hat sich also gelohnt.
Abhängig von der Komplexität der Kurven spielt dies aber später keine so große Rolle mehr, da die Kurven abhängig von der Anzahl der Knoten in der Umgebung berechnet werden und dies wird nicht durch die Implementierung des Graphens beeinflusst.

	5.000 Knoten	10.000 Knoten	20.000 Knoten
ohne Cache	13 MB	15 MB	18 MB
mit Cache	31 MB	90 MB	320 MB

Abbildung 15: Speicherverbraucht mit/ohne Caching

Des Weiteren hat das Zwischenspeichern aber auch einen Nachteil, so wird deutlich mehr Arbeitsspeicher verbraucht. Wie in Abbildung 15 zu sehen, ist der Speicherverbrauch ohne Cache relativ gering, steigt mit Cache aber deutlich an. Bei komplexeren Kurven, einem großen Rewireradius r und hoher Knotendichte kann er sogar auf über ein Gigabyte steigen. Man sollte also bei aktiviertem Cache immer einen Blick auf den Arbeitsspeicherverbrauch haben.

5.3 RRT-Erweiterungen

5.3.1 Testszenarien

Im Folgenden sind ein paar Testszenarien zu finden, die zur Bewertung der Erweiterungen verwendet werden.

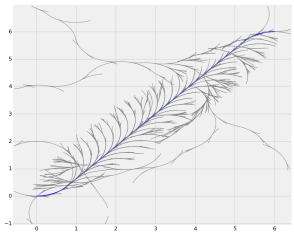

Abbildung 16: leeres Feld

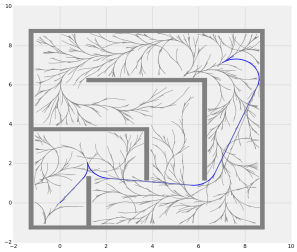

Abbildung 17: Labyrinth

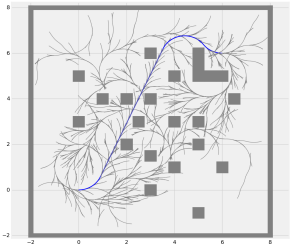

Abbildung 18: Hindernisse

leeres Feld (Abb. 16) Der einfachste Test ist ein leeres Feld. Dies testet wie gut der Algorithmus sich in der Fläche ausbreitet, wie schnell ein Pfad gefunden werden kann und wie gut dieser optimiert wird.

Labyrinth (Abb. 17) Hier wurden um die Ebene eine Mauer gezogen und auch ein paar Mauern in den Weg gestellt. Dadurch steigt die Weglänge, wodurch es länger dauert bis der Graph sich weit genug ausgebreitet hat, um einen Weg zu finden.

Feld mit Hindernissen (Abb. 18) Ein einfaches Feld mit zufällig verteilten Hindernissen. Durch die Hindernisse gibt es viele verschiedene optimierte Wege, die gefunden werden können. Optimierer können hier also ein lokales Optimum finden und stehenbleiben.

Einparken in Parklücken (Abb. 19 und 20) Dies war ein Versuch, das Verhalten bei Engstellen zu untersuchen. So wurde einmal das parallele und einmal das senkrechte Parken modelliert. Jedoch schafte es nur die Kombination aus RRT*Smart mit der besseren Vorgängerauswahl einen Weg zu finden und auch dies nicht bei jedem Durchlauf.

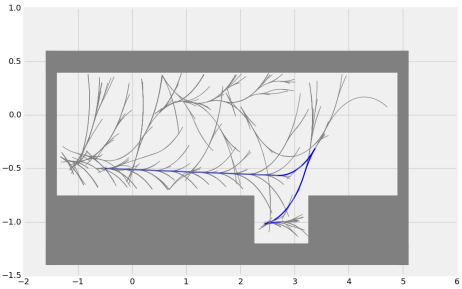

Abbildung 19: Einparken parallel

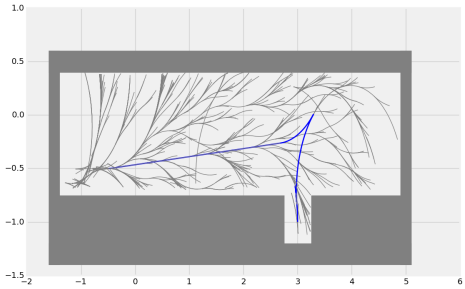

Abbildung 20: Einparken senkrecht

5.3.2 RRT* Hybrid - Verknüpfung mit dem Ziel

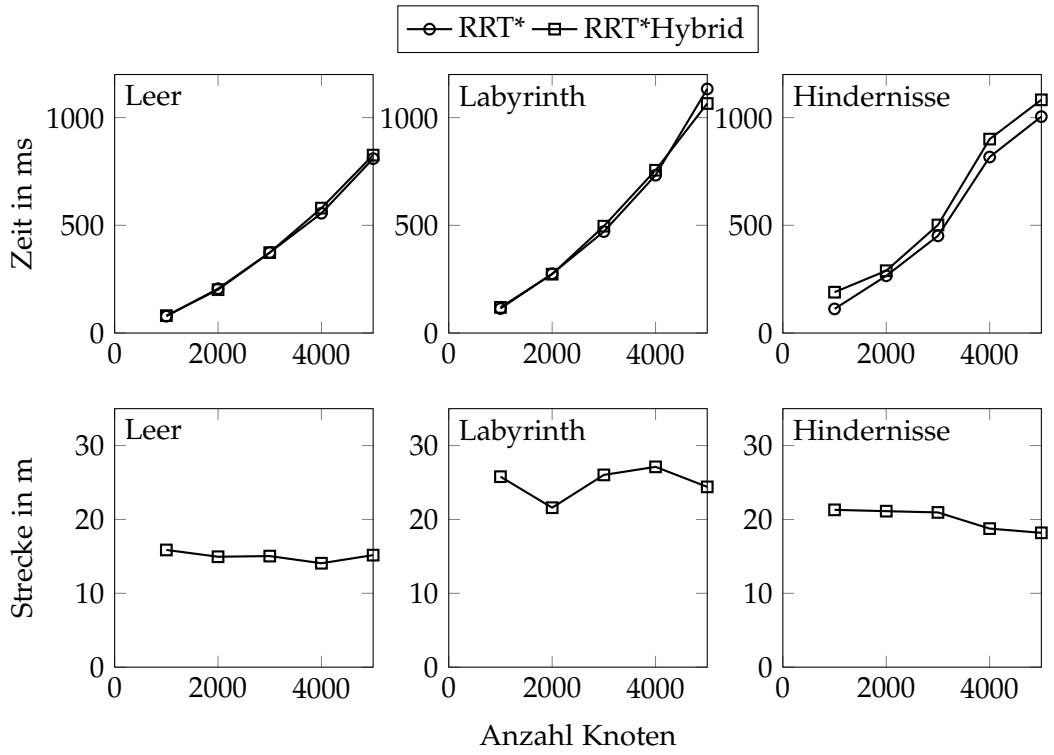

Abbildung 21: Vergleich RRT* versus RRT* Hybrid

In Abbildung 21 sieht man den Vergleich mit der Erweiterung RRT* Hybrid. In der 1. Spalte sieht man den Vergleich in der leeren Ebene. RRT* Hybrid ist gleich schnell, schafft es dafür aber einen Weg zu finden, der mit steigenden Knotenanzahl auch etwas kürzer wird. RRT* schafft es hingegen nie einen Weg zu finden.
In der 2. Spalte sieht man den Vergleich im Labyrinth. Auch hier ist RRT*Hybrid gleichschnell wie RRT* und schafft es Wege zu finden, während RRT* keine findet.
In der 3. Spalte bei den Hindernissen ist RRT* Hybrid jedoch langsamer als RRT*, wenn auch nur minimal. Dafür schafft auch hier RRT* Hybrid einen Weg zu finden, der mit steigender Knotenanzahl langsam kürzer wird.
Insgesamt ist die Verknüpfung vom Ziel mit dem Graphen per komplexen Kurven (in diesem Fall Dubins Kurve) eine gute Erweiterung. Sie kostet keinen oder wenig Zeitaufwand, liefert aber Ergebnisse, die ohne sie nicht zustande gekommen wäre, wie in Kapitel 3.6 erklärt.
Deshalb ist sie in allen weiteren Tests von Erweiterungen aktiviert, da RRT*Smart und Informed RRT* einen gefunden Weg brauchen und sonst nicht aktiviert werden. Und ohne gefundenen Weg kann man die Graphen nur nach Laufzeit vergleichen, was ohne gefunden Weg wenig Aussagekraft hat.

5.3.3 RRT* Smart

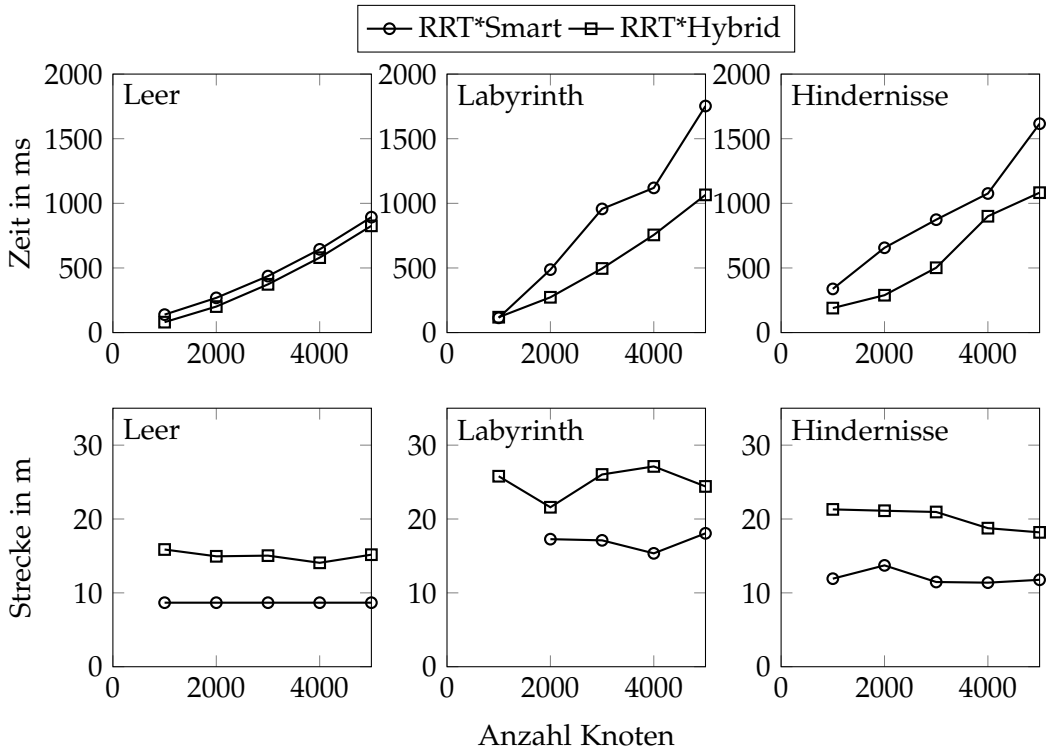

Abbildung 22: Vergleich RRT*-Smart versus RRT*Hybrid

In Abbildung 22 ist der Vergleich von RRT* Smart mit RRT* Hybrid gezeigt. In der 1. Spalte der Abbildung sieht man den Vergleich bei einer leeren Ebene. Die Laufzeit von RRT*Smart ist dabei nur geringfügig länger als die ohne RRT*Smart. Dies liegt daran, dass der Weg einfach zu optimieren ist. Dementsprechend ist die Strecke auch bereits bei kleiner Knotenanzahl fertig optimiert, was man an der niedrigen Weglänge sieht. Und die Implementierung merkt sich, wenn sie fertig mit Optimieren ist und fängt erst wieder an, wenn der Weg sich geändert hat.
In der 2. Spalte beim Labyrinth ist die Laufzeit erst gleich, wird dann aber deutlich größer als die Laufzeit der Durchläufe ohne die Erweiterung. Gleichzeitig ist die Laufzeit deutlich geringer als ohne die Erweiterung. Bei 1.000 Knoten findet RRT*Smart aber keinen Weg. Dies liegt daran, dass die hohe Weglänge es schwer macht, mit wenigen Knoten sicher einen Weg zu finden. Wenn aber kein Weg gefunden wurde, wurde auch nicht optimiert und daher kam RRT* Smart nicht zum Einsatz. Dies ist auch die Ursache, warum bei 1.000 Knoten der Algorithmus gleich schnell war, egal ob RRT*Smart aktiviert oder deaktiviert war. Bei 2.000 Knoten wurde der Weg wahrscheinlich erst gegen Ende gefunden, deshalb musste RRT* Smart nur wenig optimieren und die Laufzeit ist daher nur leicht größer.
In der 3. Spalte bei den Hindernissen ist die Laufzeit wieder höher, diesmal um einen eher konstanten Betrag. Dies liegt daran, das hier leicht ein Weg gefunden wurde, der dann aber auch deutlich optimiert wurde. So kann hier bereits mit 3.000 Knoten ein guter Weg gefunden werden, sodass mehr Knoten kaum Verbesserung bringen.

RRT* Smart ist damit ebenfalls eine gut geeignete Erweiterung. Die Laufzeit ist zwar größer, die Wege sind dafür aber auch deutlich kürzer. Wenn man das Ergebnis aber nach einer festen Zeit vergleichen würde, was praxisnäher wäre, so liefert RRT* Smart die deutlich besseren Ergebnisse.

5.3.4 Informed RRT*

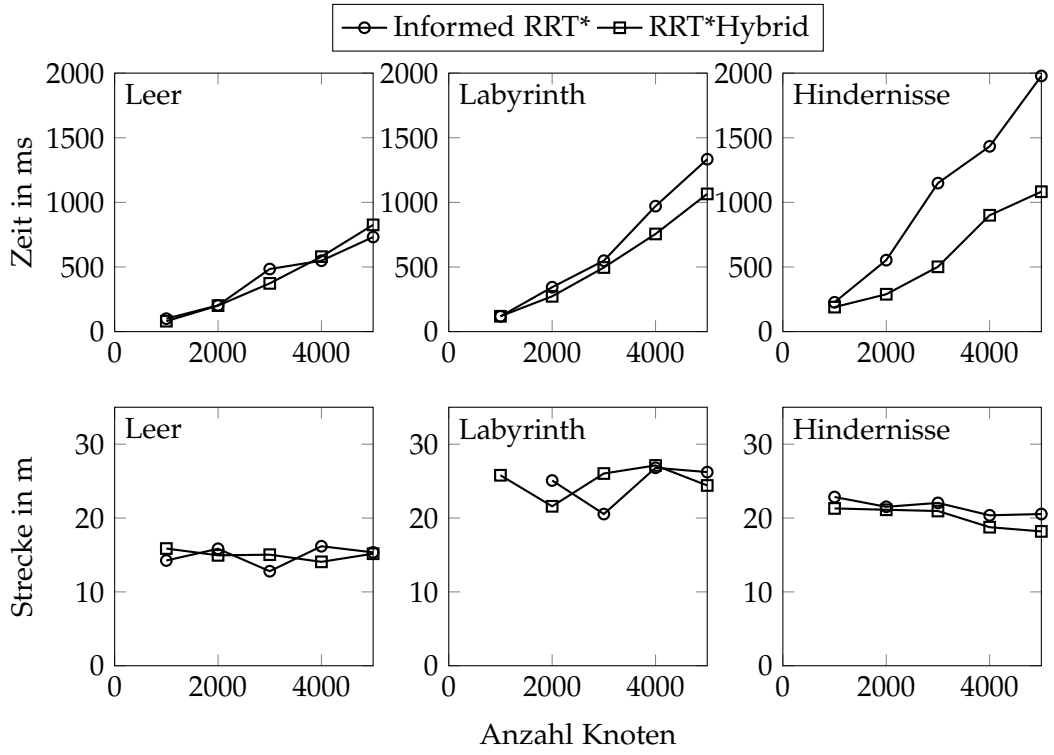

Abbildung 23: Vergleich Informed RRT* versus RRT*Hybrid

In Abbildung 23 sieht man den Vergleich mit und ohne Informed RRT*.
Informed RRT* ist im leeren Feld (1. Spalte) gleich schnell wie RRT* Hybrid, liefert aber auch vergleichbare Ergebnisse.
Im Labyrinth ist RRT* Informed etwas langsamer, besonders bei mehr Knoten und liefert wieder vergleichbare Weglängen. Die schlechtere Laufzeit bei 4.000 und 5.000 Knoten liegt daran, dass der Weg erst mit vielen Knoten zuverlässig gefunden wird, sodass auch erst dann Informed RRT* viel Arbeit hat.
Bei den Hindernissen ist Informed RRT* deutlich langsamer und liefert dann auch noch schlechtere Ergebnisse.
Die Ursache ist dabei, dass Informed RRT* von der gefundenen Weglänge abhängig ist.
Beim leeren Feld ist die Ellipse ähnlich groß wie das Rechteck, in dem sonst die Punkte gesampled werden. Deshalb unterscheidet sich das Ergebnis kaum. Im Labyrinth und bei den Hindernissen ist der Weg jedoch länger, sodass auch die Ellipse größer wird. Sowohl das Labyrinth als auch die Hindernisse haben aber im Gegensatz zum leeren Feld eine Außenmauer. Dadurch dass viel außerhalb dieser Mauer gesampled

wird, werden viele Punkte am Rande des Feldes und neben dieser Mauer erstellt. Und wenn versucht wird einen Knoten hinter die Mauer zu platzieren, so wird dieser verworfen. Es wurde also ein Knoten unnötigt berechnet. Wenn dies häufig genug passiert, was bei den Hindernissen der Fall ist, da bereits früh ein Weg gefunden wurde, dann steigt die Laufzeit natürlich an. Für Fälle wie unsere Testszenarien, wo der zu samplende Raum gut begrenzt ist, ist Informed RRT* also nicht hilfreich. Wenn jedoch der zu samplende Raum unbekannt ist, kann dies helfen den Raum schnell einzugrenzen. Man sollte aber eine Obergrenze für den Radius einbauen. Denn wenn zu lange Wege gefunden werden, so sorgt dies dafür, dass plötzlich in einem deutlich größeren Raum gesampled wird.

5.3.5 Informed RRT* mit RRT* Smart

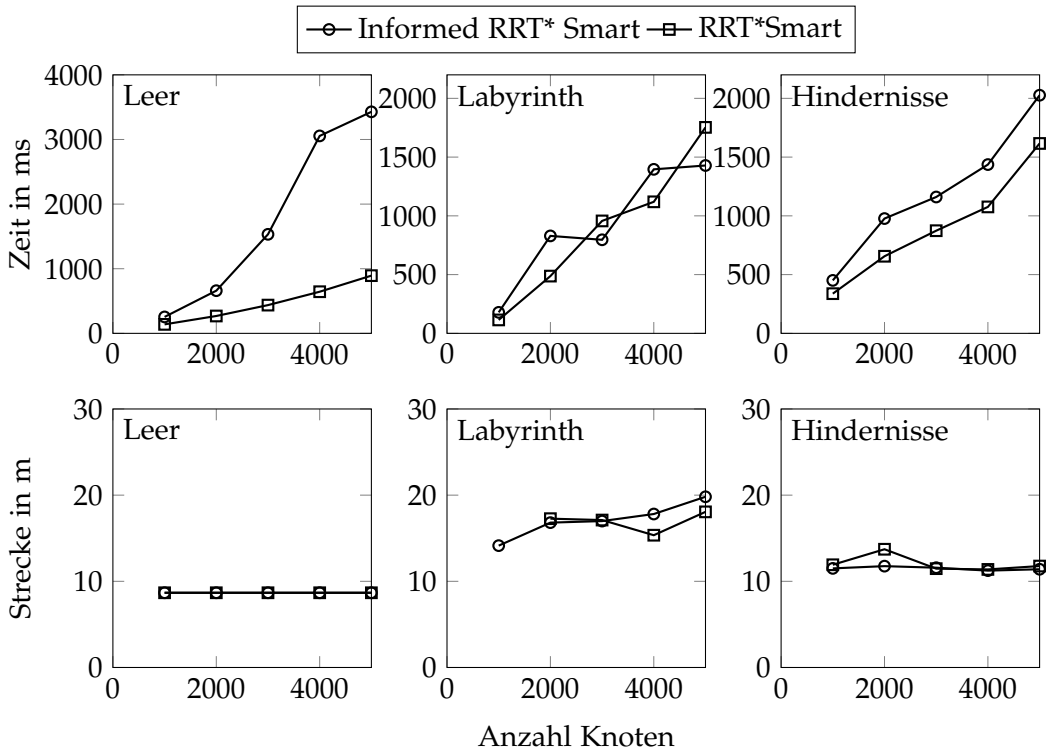

Abbildung 24: Vergleich Informed RRT*Smart versus RRT*Smart

Da Informed RRT* stark von der gefunden Weglänge abhängt, ist ein Vergleich zusammen mit RRT* Smart interessant. Als Vergleich wird dabei RRT* Smart genommen, denn Informed RRT* hat keine besseren Ergebnisse geliefert, sondern nur mehr Zeit verbraucht.
In Abbildung 24 sieht man also den Vergleich von Informed RRT* Smart zu RRT*Smart. Es fällt auf, dass bei der leeren Ebene die Kombination aus RRT*Smart und Informed RRT* die Laufzeit deutlich nach oben treibt. Dies liegt daran, dass der durch RRT*Smart optimierte Weg so kurz ist, dass die Fläche der Ellipse sehr klein ist. Neue Punkte werden also nur noch auf dem gefundenen Weg und direkt daneben gesampled. Dadurch steigt die Knotendichte dort stark an und alle Operationen brauchen

deutlich länger. Auch der Speicherverbrauch steigt an. Während auf leerer Fläche mit 5.000 Knoten RRT* Smart noch 60 MB Arbeitsspeicher brauchte, waren es bei Informed RRT* mit RRT*Smart sogar 180 MB. Im Labyrinth liefert die Kombination durchmischte Ergebnisse, mal besser und mal schlechter als RRT*Smart. Und bei den Hindernissen stieg die Laufzeit, aber die Weglänge blieb vergleichbar.
Zur Bewertung von Informed RRT* aus Kapitel 5.3.4 muss also hingefügt werden, dass man bei der Kombination von RRT*Smart und Informed RRT* auch eine Untergrenze braucht. Wenn der Weg zu gut ist, darf Informed RRT* nicht samplen, da es sonst die Laufzeit deutlich verschlechtert. Aber wenn man dies erreicht hat, kann man auch gleich aufhören, denn besser wird der Weg dann nicht mehr.

5.3.6 RRT* BestAncestor - Bessere Vorgängerauswahl

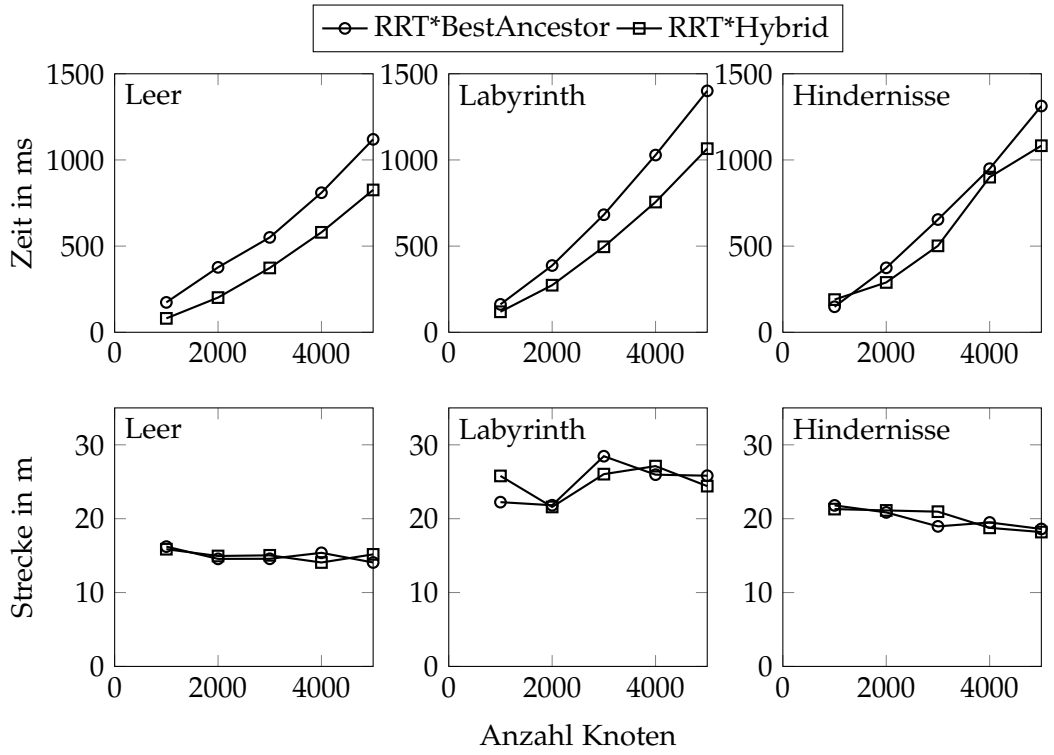

Abbildung 25: Vergleich RRT*BestAncestor versus RRT*Hybrid

Die Abbildung 25 lässt sich einfach zusammenfassen: Die Laufzeit mit dieser Erweiterung wird größer, die Weglänge bleibt aber gleich.
RRT* BestAncestor bietet in diesen Testszenarien keine Vorteile, aber im Testszenario Einparken war es nötig, um überhaupt Ergebnisse zu bekommen. Der Ansatz hat also Potential, kann dieses aber hier nicht ausnutzen und muss daher erst noch überarbeitet werden.

6 Erkenntnisse

6.1 Vergleich von Kurven

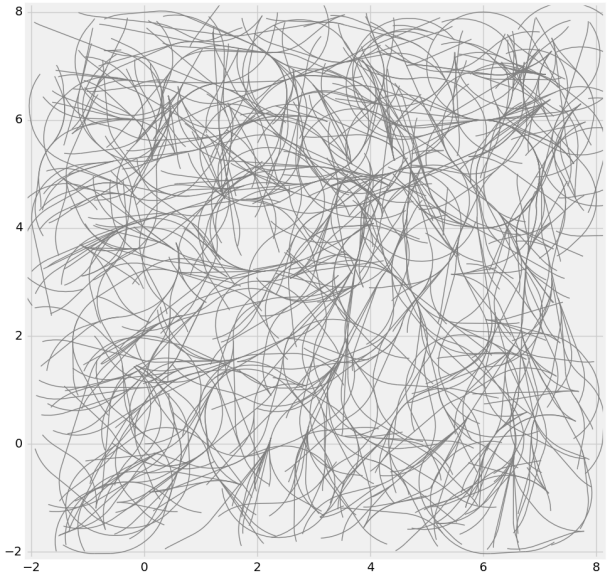

Abbildung 26: RRT* Dubins Kurven

Abbildung 26 zeigt einen Graphen, der mit Dubins Kurven erstellt wurde. Es fällt auf, dass der Graph viele kreuzende Kurven und viel Kreise hat. Wenn man das Programm mit Dubins Kurven laufen lässt, kann man zusätzlich folgendes beobachten:
1. der Algorithmus läuft sehr lange, bei 3.000 Knoten braucht er über 100mal so lange wie RRT*Hybrid.
2. der Arbeitsspeicherverbrauch steigt auf über 1 GB.
3. der gefundene Weg ist meist doppelt so lang wie ein von RRT*Hybrid gefundener Weg.
Dies liegt daran, dass Dubins Kurven immer einen Weg finden, solange es keine Hindernisse gibt. Die Wege sind dann zwar umständlicher, aber die gesampleten Positionen werden nicht verschoben.
Wenn man darauf RRT* anwenden will und nicht möchte, dass das Auto mehrere Meter fährt, bloß um 30 cm voran zu kommen und in eine ganz andere Richtung zu schauen, so muss man den Rewireradius vergrößern. Das reine Vergrößern des Rewireradius reicht aber nicht aus, um die gefundene Strecke zu verkürzen. Denn sobald viele Hindernisse vorhanden sind, haben Dubins Kurven ein Problem Wege zu finden. Denn die Dubins Kurven sind länger und haben daher eine höhere Wahrscheinlichkeit mit Hindernissen zu kollidieren. Hinzu kommt der steigenden Aufwand für die Kollisionsberechnung, da die Kurven länger sind und es wegen dem größeren Rewireradius auch mehr Kurven gibt. Und wenn man diese vielen Kurven cacht, dann steigt wie beobachtet die Arbeitsspeicherauslastung.
Dass die Laufzeit über 100 mal so lang ist, resultiert zum Teil auch aus der nicht op-

timierten Implementierung der Dubins Kurve. Aber selbst mit Optimierung wird es immer noch mehr als 10 mal so lange brauchen.
Deshalb habe ich mich in dieser Arbeit auf die Vor- und Nachteile von einfachen Kurven fokussiert.
Einfache Kurven schieben sich die Knoten so hin, dass sie besser erreicht werden können. Damit findet man einen besseren Weg, da die Kurven nicht so umständlich sind. Die einfachen Kurven haben zwar das Problem, den Anschluss an das Ziel zu schaffen, aber dafür kann man auch die komplexen Kurven einsetzen, wie man an RRT*Hybrid in Kapitel 5.3.2 sehen kann. Diese komplexe Kurve muss man dann nur in einfache Kurven zerlegen, was bei Dubins Kurven einfach ist. Auch dass die erzeugten Wege nicht optimal sind, kann man mit RRT* Smart beheben, der ebenfalls komplexe Kurven benutzt.

6.2 RRT* BestAncestor - Wahl des Vorgängerknotens

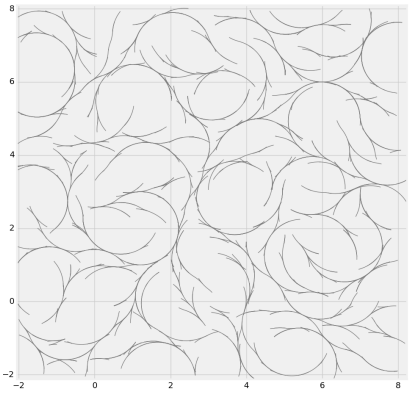

Abbildung 27: RRT* mit festen Lenkeinstellungen

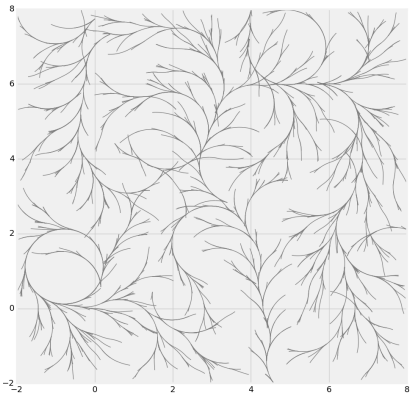

Abbildung 28: RRT* mit festen Lenkeinstellungen und BestAncenstor

Ein Problem haben wir aber noch mit den einfachen Kurven. Durch das erneute Verschieben der Knoten nach dem zufälligen Samplen verlieren wir die Zufälligkeit. Die Wahl des Vorgängers ist also wichtig, damit der Graph sich gut ausbreitet.
In Abbildung 27 sieht man den Graphen, wenn einfach der nächste Knoten gewählt wird. In Abbildung 28 sieht man einen Graphen, wo bei der Wahl des Vorgängers auch auf die Ausrichtung des Vorgängerknoten geachtet wurde.
Dies ermöglicht u.a., dass nahe Knoten deutlich unterschiedliche Winkel haben. Wenn man die Ausrichtung des Vorgängerknotens nicht beachtet, so nimmt man immer den nächsten Knoten und alle Knoten, die nah zu einander gelegen sind, haben dadurch immer sehr ähnliche Ausrichtungen. Die bessere Auswahl sorgt also für ein besseres Ergebnis an Engstellen, wie man in der Abbildung 30 im Vergleich zu Abbildung 29 sieht.

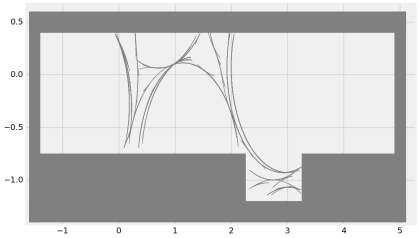

Abbildung 29: Einparken mit RRT* mit festen Lenkeinstellungen

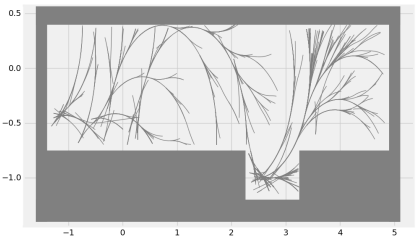

Abbildung 30: Einparken mit RRT* mit festen Lenkeinstellungen und BestAncenstor

Die Wahl des Vorgängerknotens anhand seiner Ausrichtung reicht aber noch immer nicht aus, um eine gleichmäßige Verteilung der Knoten zu erhalten. Die Knoten sind zwar bereits besser in der Ebene verteilt und auch die Ausrichtung der Knoten ist etwas besser verteilt, aber es ist noch nicht zufällige genug.
Es fehlt noch die Beachtung der Ausrichtung des gesampleten Knoten. Wenn man diese auch noch beachten würde, könnte man wahrscheinlich auch bessere Ergebnisse erzielen.

6.3 RRT* Informed - Besseres Sampling

RRT* Informed ist sicherlich ein guter Ansatz, benötigt aber noch ein paar Anpassungen.
Den Vorteil von Informed RRT*, das Einschränken eines sehr großen Raumes auf einen kleineren, konnte es in unseren Testszenarien nicht ausnutzen, da diese bereits relativ klein sind. In einem größeren Testfeld hätte man ein anderes Ergebnis erhalten.
Eine Anpassung wäre das bessere Samplen in der Ellipse, dieses ist nicht direkt machbar, sodass man z.B. erst in einem Kreis Punkte erstellt und diesen Kreis dann in die gewünschte Richtung streckt. Wobei das dazu führt, dass die Punkte nicht mehr komplett gleich verteilt sind. Eventuell kann man hier ein mathematisches Modell finden womit man einen annähernd gleich verteiltes Ergebnis direkter berechnen kann.
Des Weiteren sollte man, wie bereits in Kapitel 5.3.4 und 5.3.5 erwähnt, eine Unter- und Obergrenze für den Radius festlegen.

6.4 RRT* Smart - Optimierung von gefundenen Wegen

RRT* Smart hat die größten Verbesserungen gebracht. Es gleicht den Nachteil der einfachen Kurven aus, indem es die Schlangenlinien entfernt und sorgt so für einen besseren Weg.
Die Implementierung mit Dubins Kurve ist aber nicht perfekt, mit Verwendung von besseren Kurvenalgorithmen kann man hier vermutlich noch bessere Ergebnisse erreichen. So kann man im Weg eine Kurve mit einem großen Radius haben, die zwischen Hindernissen hindurchführt. Beim Optimieren mit Dubins Kurven würde man hier enge Kurven abwechselnd mit geraden Strecken erhalten.
Eine wichtige Frage ist, wie viele Kurven man probiert auf einmal zu optimieren. In

Kapitel 3.3 wurde schon gezeigt, dass es mindestens 3 Kurven sein müssen. Wenn man aber auch größere Abkürzungen finden will, so muss man längere Strecken optimieren. Dies erhöht aber den Aufwand, da man so mehr komplexe Kurven berechnen und auf Kollision prüfen muss.
Ein weiteres Problem ist, dass die Verknüpfung des Ziels mit dem Graphen bei RRT* Hybrid meist sehr unschöne Verbindungen baut. Auch um diese zu optimieren, muss man längere Strecken optimieren.
Hinzu kommt, dass man teilweise den falschen Weg zuerst findet.
Angenommen es gibt 2 Wege um ein Hindernis herumzufahren. Linksrum ist länger, wird aber als erstes gefunden und optimiert. Rechtsrum ist kürzer, wird aber erst als zweites gefunden. Zu diesem Zeitpunkt ist der linke Weg bereits so optimiert, dass er kürzer ist als der nicht optimierte rechte Weg. Wir würden zwar irgendwann den rechten Weg als kürzer entdecken, aber das kann dauern.
Hier wäre es hilfreich, wenn man auch alternative Wege erkennt und diese optimiert. Damit würde man größere Abkürzungen entdecken, ohne dass man die Anzahl der Kurven, die man optimiert, zu erhöhen.

6.5 Offene Punkte

Wie bereits in Kapitel 5.3.5 erwähnt, kann eine zu hohe Knotendichte ein Problem sein. Sie sorgt für deutlich höhere Ausführungszeit und viele der Knoten liefern keinen Mehrwert. Deshalb ist es nötig, die zu hohe Knotendichte zu verhindern. Beim Einfügen neuer Knoten darauf zu achten ist schwer, da man schlecht den Mehrwert der Knoten einschätzen kann, bevor er in den Graph integriert wurde. Besser ist es, regelmäßig Knoten zu entfernen, die keinen Mehrwert liefern. Dazu existiert auch bereits ein Ansatz mit RRT*Fixed Nodes, wie in Kapitel 3.4 beschrieben.
Ein weiteres Problem ist, dass zu häufig rückwärts gefahren wird. Dies liegt daran, dass der Graph, wenn er erstmal eine Orientierung in einem Bereich hat, diese nicht mehr so schnell ändert, wie bereits in Kapitel 6.2 beschrieben. Hier fehlt noch die Gewichtung des Rückwärtsfahrens, zusammen mit der in Kapitel 6.2 erwähnten Anpassung. Auch durch die Beachtung der Ausrichtung der gesampleten Position, könnte man hier eventuell ein besseres Ergebnis erzielen. Alternativ könnte man probieren, Strecken zu erkennen, wo lange rückwärts gefahren wird und an dieser Stelle Knoten samplen, die die Strecke in entgegengesetzter Richtung abfahren. Dies funktioniert natürlich nur, wenn am Anfang und Ende der Strecke genug Platz zum Wenden ist.

7 Fazit

Mit dieser Arbeit konnte gezeigt werden, dass ein RRT* basierter Planer auch auf langsamen Modellautos in akzeptabler Zeit einen guten Weg planen kann. So hat RRT* Smart in Abbildung 22 mit 3.000 Knoten eine Laufzeit von unter 0,5 Sekunden erreicht und gleichzeitig einen guten Weg gefunden. Da der Graph im Ziel verankert ist, kann sogar bei Abfahren des Weges ein regelmäßiges Aktualisieren des Graphen erfolgen, wodurch auch ein Abkommen vom geplanten Weg erkannt, korrigiert und optimiert werden kann.

Das gute Ergebnis konnte dabei einerseits durch die Performanceoptimierungen an der Datenstruktur und das Cachen der Kanten erreicht werden, andererseits auch durch die Kombination aus einfachen und komplexen Kurven.
Einfache Kurven, wie Kurven mit festen Lenkeinstellungen, sind schneller zu berechnen, liefern dafür aber keine optimalen Wege. In Kombination mit komplexen Kurven, in diesem Fall Dubins Kurven, konnten diese Wege aber schnell optimiert werden, sodass ein Kompromiss zwischen Qualität des Weges und Berechnungsdauer gefunden werden konnte.

Es gibt aber immer noch Potential für Verbesserungen, so kann eine andere komplexe Kurve verwendet werden oder man kann den Raum um die aktuelle Lenkeinstellung erweitern. Damit könnte man besser abfahrbare Wege finden, auch wenn die Berechnung länger dauern würde. Des Weiteren fehlt auch noch eine Behandlung von neu auftauchenden Hindernissen. Weitere Probleme und Ansätze sind in Kapitel 6 ausführlicher beschrieben.

Die Entscheidung zu einer modularen Struktur war für diese Arbeit wichtig, sie hat das Implementieren der einzelnen Erweiterungen deutlich vereinfacht und den Vergleich zwischen den Algorithmen erst möglich gemacht. Darüber hinaus ist Sicherheit eine wichtige Eigenschaft, die autonome Autos besitzen müssen, denn sonst wird sie keiner nutzen. Das betrifft nicht nur die physikalische Sicherheit, wie z.B. ein guter Aufprallschutz, sondern auch die Stabilität und Korrektheit der Steuerungssoftware. Eine gut modularisierte Software lässt sich deutlich besser auf Fehler untersuchen und testen als ein Monolith.

Literaturverzeichnis

[1] Odroid Xu4. https://wiki.odroid.com/odroid-xu4/odroid-xu4, Nov 2018.

[2] Quellcode. https://git.imp.fu-berlin.de/linushelfman/rrt, Nov 2018.

[3] Valgrind. http://www.valgrind.org/, Nov 2018.

[4] O. Adiyatov and H. A. Varol. Rapidly-exploring random tree based memory efficient motion planning. In *2013 IEEE International Conference on Mechatronics and Automation*, pages 354–359, Aug 2013.

[5] Arbeitsgruppe Robotics FU Berlin. Hardware (AutoNOMOS Model v3.1). https://github.com/AutoModelCar/AutoModelCarWiki/wiki/Hardware-(AutoNOMOS-Model-v3.1), Nov 2018.

[6] Institut de Robòtica i Informàtica Industrial. Seat Car Simulator. https://gitlab.iri.upc.edu/seat_adc/seat_car_simulator, Nov 2018.

[7] Open Source Robotics Foundation. Robot Operating Systems. http://www.ros.org/, Nov 2018.

[8] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. Informed RRT*: Optimal Incremental Path Planning Focused through an Admissible Ellipsoidal Heuristic. *CoRR*, abs/1404.2334, 2014.

[9] Andrew Giese. A Comprehensive, Step-by-Step Tutorial to Computing Dubin's Paths, 2012.

[10] Lukas Gödicke. RRT based Path Planning in Static and Dynamic Environment for Model Cars, März 2018.

[11] Fahad Islam, Jahanzeb Nasir, U. Malik, Yasar Ayaz, and Osman Hasan. RRT*-Smart: Rapid convergence implementation of RRT* towards optimal solution. *2012 IEEE International Conference on Mechatronics and Automation*, pages 1651–1656, 2012.

[12] Sertac Karaman and Emilio Frazzoli. Sampling-based Algorithms for Optimal Motion Planning. *CoRR*, abs/1105.1186, 2011.

[13] James Kuffner and Steven M. LaValle. RRT-Connect: An Efficient Approach to Single-Query Path Planning. volume 2, pages 995–1001, 01 2000.

[14] Steven M. LaValle. Rapidly-Exploring Random Trees: A New Tool for Path Planning. 1998.

[15] Michael Otte and Emilio Frazzoli. RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning. *The International Journal of Robotics Research*, 35(7):797–822, 2016.

[16] Bernd Sahre. Untersuchung der Effizienz von RRT* bei autonomen Autos, Mai 2018.