Seminar Sommersemester 2013:
Automobile Systeme in der Automatisierung
Prof. Dr. Dieter Zöbel, Universität Koblenz-Landau, FB Informatik

Die Planungsmethode Rapidly-exploring Random Tree Konzepte und Varianten

Benedikt Jöbgen

Eingereicht: 25.08.2013 / Fertiggestellt: 05.10.2013

Zusammenfassung *Die vorliegende Arbeit vermittelt einen Einblick in die Bewegungsplanung, im speziellen die Trajektorienplanung von Fahrzeugen, auf Basis des* rapidly exploring random tree*(RRT) Algorithmus, welcher erstmals von Steven M. LaValle [17][1] veröffentlicht wurde. Neben der RRT werden außerdem der RRT* [2] sowie weitere Heuristiken und Erweiterungen vorgestellt. Abschließend werden noch Probleme der Trajektorienplanung, welche in der Praxis auftreten, erläutert und mögliche Lösungsansätze diskutiert. Hierzu wird der Fahrtplannungs-Algorithmus Talos [9] des MIT als ein mögliches Beispielsystem vorgestellt.*

Schlüsselwörter motion planning, realtime motion planning, RRT, trajectory planning, sampling-based planning, autonomous driving

1 Einleitung

Der *rapidly-exploring random tree*(RRT) ist ein probabilistischer Bewegungsplanungsalgorithmus, welcher für viele Problemstellungen eine Lösung erstellt. Diese Ausarbeitung beschäftigt sich jedoch in erster Linie mit der Pfad- und Trajektorienplanung autonomer Fahrzeuge. Der Sampling-basierte Ansatz baut von einem vordefiniertem Startzustand eine zufällige Baumstruktur auf und hat eine Lösung für das gefragte Problem gefunden, sobald dieser Baum einen Zielzustand erreicht. Hierbei wird darauf geachtet, dass keine Kollisionen mit Hindernissen entstehen. Dieser Algorithmus wird sehr oft genutzt, da er sehr schnell einen möglichen Pfad findet und selbst bei komplexen nicht-linearen und nichtholonomen Bewegungsmodellen und in hochdimensionalen Zustandsräumen angewandt werden kann.
Ein großes Makel des einfachen RRT ist seine Tendenz, möglichst schnell einen möglichen Zielpfad zu finden, ohne jedoch dabei auf Optimalität zu achten, wodurch der erzeugte Pfad oft große "Umwege" beinhaltet. Hier verschafft der RRT* [2] Abhilfe, weil er genau an diesem Problem ansetzt. Er verbindet die einzelnen Knoten des Baumes intelligenter und läuft selbst nach dem Finden des ersten möglichen Zielpfades

Universität Koblenz-Landau, Institut für Informatik
bjoebgen@uni-koblenz.de
http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGZoebel

weiter, um diesen zu optimieren.
Da der RRT ein sehr gängiger Algorithmus zur Bewegungsplanung ist, gibt es zahlreiche Erweiterungen und Heuristiken. In dieser Arbeit werden Heuristiken gezeigt, welche das zufällig Sampling zum Erweitern des Baumes leicht abändern [11], um somit dem Algorithmus eine bestimmte Tendenz zu geben. Des Weiteren werden aber auch signifikante Erweiterungen behandelt, wie den bidirektionalen RRT [14][10], welcher zwei Bäume gleichzeitig aufbaut, oder den dynamic-domain-RRT [13], welcher den RRT in Hindernisreichen Umgebungen beschleunigt. Der eigene Beitrag in dieser Arbeit stellt das rekursive ReWire dar, welches den RRT* um eine nützliche Invariante bereichert. Dieses wird als letzte Abänderung des Algorithmus vorgestellt und diskutiert.
Wird der RRT in der Praxis in einem autonomen Fahrzeug eingesetzt, so fallen einige Probleme auf, welche der RRT jedoch alle bewältigt, falls man diesen entsprechend anpasst. Hierzu gehört zum Beispiel das Planen während der Fahrt [2], um somit, trotz dynamischer Umgebung und begrenzter Sensorik, durch eine unbekannte Umwelt fahren zu können. Weitere Probleme und mögliche Lösungsansätze können an dem Planungssubsystem "Talos" [5][9] demonstriert werden, welches dem Massachusetts Institute of Technology (MIT) in der DARPA Urban Challenge 2007 diente.

2 RRT

2.1 Problemdefinition

Das sogenannte **motion planning problem** (bzw. piano movers problem) beschreibt die Suche eines Pfades von einem Startzustand zu einem Zielzustand, ohne dabei mit Hindernissen zu kollidieren. Im vorliegenden Papier wird hierbei hauptsächlich auf die Trajektorienfindung für ein Fahrzeug von einer Startpose hin zu einer möglichen Zielpose eingegangen. Diese Posen sind Punkte im Zustandsraum des Fahrzeuges $X \subseteq \mathbb{R}^n$, neben welchem ebenso auch noch der Arbeitsraum der möglichen Steuerbefehle $U \subseteq \mathbb{R}^m$ sowie die Dynamik des Subjekts $\dot{x}(t) = f(x(t), u(t))$ beschrieben werden muss. Des Weiteren wird eine ausreichende Beschreibung der Umgebung $X_{free} = X \setminus X_{obs}$ verlangt, um Kollisionsvermeidung durchführen zu können.
Ziel des motion planning problems ist es nun, eine Trajektorie von einem vordefinierten Startzustand $x(0) = x_{init}$ in der Zeit $t \in [0, T]$ zu einem beliebigen Zustand aus einem Set von Zielzuständen $x(T) \in X_{goal}$ zu finden. Soll diese Trajektorie genutzt werden, um ein Fahrzeug direkt anzusteuern, so ist es außerdem notwendig, eine Sequenz von Kontrollbefehlen $u : [0, T] \rightarrow U$ zu erlangen, welche das Fahrzeug unter idealen Bedingungen entlang dieser Trajektorie führt. Während der Ausführung dieser Trajektorie darf das Subjekt mit keinen Hindernissen kollidieren, es muss sich zu jeder Zeit in einem freien Zustand befinden: $\forall t : x(t) \in X_{free}$.
Wird nun außerdem auch noch versucht einen optimalen Weg zu finden, so spricht man von dem **optimal motion planning problem**. Hierzu muss die Kostenfunktion $J(x) = \int_0^T g(x(t))dt$ minimiert werden. Hierbei ist die Definition der Kosten nicht weiter eingeschränkt. Sie können sowohl die Streckenlänge, als auch die benötigte Zeit für ihre Ausführung sein. Zusätzlich können aber auch noch weitere Aspekte mit einfließen, beispielsweise die Sicherheit des Pfades.

2.2 RRT - Algorithmus

Der **rapidly-exploring random tree** (RRT) ist ein probabilistischer Ansatz zum Lösen des oben beschriebenen motion planning problems. Er baut ausgehende von dem Startzustand $z_{init} \subseteq \mathbb{R}^n$ einen Baum auf, welcher sich solange durch eine zufallsbestimmte Heuristik im Zustandsraum ausbreitet, bis ein Zielzustand erreicht wurde. Schließlich bricht der Algorithmus ab und es kann von dem gefunden Zielzustand rückwärts entlang des Baumes gegangen werden, um den gesuchten Pfad zu erlangen.

$T \leftarrow InsertNode(\emptyset, z_{init})$;
1 **for** $i = 1$ ***to*** N
2 **do**
3 | $z_{rand} \leftarrow Sample(i)$;
4 | $extend(T, z_{rand})$;
5 **end**
6 **return** T

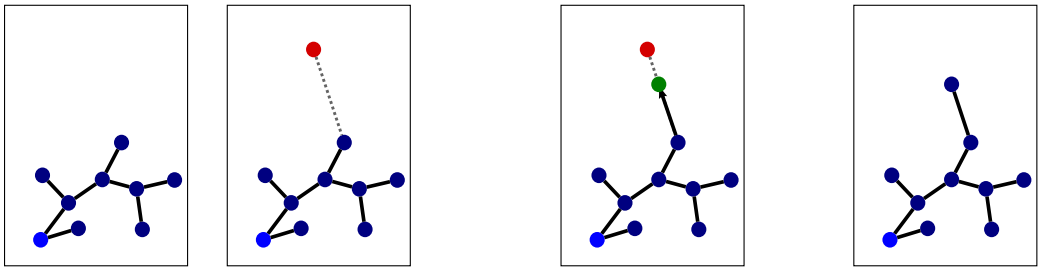
Abb. 1 Gerüst des rapidly-exploring random tree Algorithmus

In Abbildung 1 ist der RRT-Algorithmus beschrieben. Er beginnt mit einem leeren Baum T, welchem die Startkonfiguration z_{init} also Wurzel übergeben wird (Zeile 1). Anschließend werden so lange neue Knoten, welche Konfigurationen des Fahrzeuges entsprechen, an diesen Baum hinzugefügt (Zeilen 3-4), bis einer dieser Knoten ein Zielzustand beschreibt. Sollte allerdings nach N 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 ebenso ab (Zeile 2).

$extend(T, z_{rand})$
1 $z_{nearest} \leftarrow Nearest(T, z_{rand})$;
2 $(x_{new}, u_{new}, z_{new}) \leftarrow Steer(z_{nearest}, z_{rand}, \Delta t)$;
3 **if** $ObstacleFree(x_{new})$ **then**
4 | $T \leftarrow InsertNode(z_{nearest}, z_{new}, T)$;
5 **end**

Abb. 2 extend-Funktion

Zum Erstellen eines neuen Knotens, wird zunächst ein zufälliger Knoten z_{rand} gewählt (Abb. 1 Zeile 3, sowie Abb. 3b roter Punkt). Hierzu wird gleichverteilt aus dem kompletten Konfigurationsraum gezogen. Nun wächst der bereits bestehende Baum in Richtung dieser Zufallskonfiguration (Abb. 1 Zeile 4, sowie Abb. 2). Zunächst wird aus dem bereits bestehenden Graphen der nächstliegende Nachbarknoten $z_{nearest}$ gesucht (Abb. 2 Zeile 1, sowie Abb. 3b 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 zum Beispiel die euklidische Distanz oder besser die Dusbinspfadlänge genommen werden.
Ist $z_{nearest}$ nun bekannt, so wird von diesem aus eine gewisse Strecke(Δt) in Richtung z_{rand} gesteuert (Abb. 2 Zeile 2), wobei Δt lediglich als obere Schranke zu verstehen



a)
bereits bestehender Baum

b)
$z_{rand} \leftarrow Sample(i)$;
$z_{nearest} \leftarrow Nearest(T, z_{rand})$;

c)
$(x_{new}, u_{new}, z_{new}) \leftarrow$
$Steer(z_{nearest}, z_{rand}, \Delta t)$;

d)
if$(ObstacleFree(x_{new}))$
$InsertNode(z_{nearest}, z_{new}, T)$;

Abb. 3 Hinzufügen eines neuen Knotens in den Graphen

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 muss, neben dem nun neu erreichten Konfigurationspunkt z_{new} (Abb. 3c grüner Punkt), sowohl die abzufahrende Trajektorie x_{new} von $z_{nearest}$ bis z_{new} als auch die Kontrollbefehle u_{new}, welche für die Ausführung dieser Strecke benötigt werden, zurückgeben. Falls diese neue Strecke x_{new} hindernisfrei ist (Abb. 2 Zeile 3) kann z_{new} als neuer Punkt in den Baum eingetragen werden, mit einer Kannte von $z_{nearest}$ zu z_{new} (Abb. 2 Zeile 4, sowie Abb. 3d). Genaueres zur Kollisionsvermeidung findet sich in [12].

2.3 rapidly exploring

Das folgende Kapitel soll klären, warum sich dieser Algorithmus "rapidly-exploring", also "sich schnell ausbreitend", nennt. Um dies zu zeigen, ist es sinnvoll das Voronoi-Diagramm eines rrt-Graphen zu betrachten. Abbildung 4 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 definieren hingegen 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 nächstliegendste Knoten erweitert, so bedeutet dies, dass die Wahrscheinlichkeit, dass ein spezieller Konten erweitert wird proportional zu der Größe seiner Voronoi-Region ist.

Betrachtet man nun Abbildung 4 so erkennt man, dass die Voronoi-Regionen der außen liegenden Knoten, auch **frontier**-Knoten genannt, wesentlich größer sind 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 Knofiguationsraumes auszubreiten. Dies ist eine der wichtigsten Eigenschaften, welche auch in allen Erweiterungen und Heuristiken auf jeden Fall beibehalten werden muss.

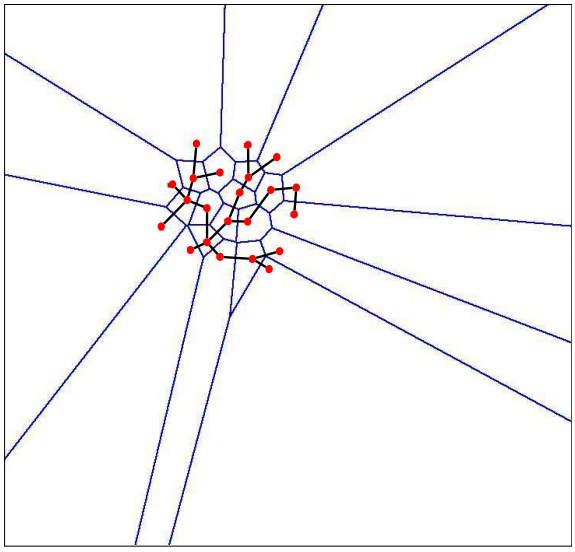


Abb. 4 Voronoi-Gebiete eines rrt-Graphen

2.4 Errungenschaften pt.1

Zusammenfassend kann nun gesagt werden, dass der rapidly-exploring random tree ein **single query Pfadplaner** ist, was bedeutet, dass der Algorithmus lediglich für ein spezielles motion planning problem ausgeführt wird und er nicht, wie zum Beispiel die probabilistische road map (PRM), einmal ausgeführt wird und dann mehrere Abfragen beantworten kann. Dies wirkt zwar auf dem ersten Blick wie ein Kritikpunkt, allerdings kann das bei dem Aufbau des Baumes bestehende Wissen über Start- und Zielpunkte sinnvoll genutzt werden, wodurch der Algorithmus performanter ist.
Ein sehr großer Vorteil des RRT liegt darin, dass er keine Einschränkungen in der Steuer-Funktion stellt und somit auch auf **nichtholonomen** und **nicht-linearen Modellen** angewendet werden kann.
Will man einen Verfahren bewerten, 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 korrekte Ergebnis. Es ist noch interessant zu betrachten, inwiefern dies eingehalten werden kann, wenn die Umgebung nicht statisch sondern dynamisch ist. Wie der RRT auch in solchen Umgebungen sinnvoll eingesetzt werden kann, wird in späteren Kapitel noch näher erläutert. Die Vollständigkeit im Sinne der Fahrtplanung bedeutet, dass der Algorithmus einen Weg finden muss, falls es einen in der Realität gibt.Hier kann es jedoch tatsächlich vorkommen, dass der RRT sehr lange braucht um durch gewisse Engstellen zu kommen, was auch bedeuten kann, dass vor Ablauf der Maximalzeit kein Pfad gefunden wurde, obwohl es einen gäbe. Allerdings kann behauptet werden, dass die Wahrscheinlichkeit einen Weg zu finden, falls einer existiert, mit der Anzahl der Knoten und somit mit der Laufzeit steigt. Dies nennt man **probabilistische Vollständigkeit**. Ein sehr großer Makel hat der normale RRT in Hinsicht auf

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**. Diese Eigenschaft wird allerdings schon im folgenden Kapitel angestrebt.

3 RRT*

3.1 Motivation

Der größte Mangel des einfachen RRTs ist, dass er keine Optimalität anstrebt. Wird eine neuer Knoten hinzugefügt, so wird lediglich darauf geachtet, dass das neu entstandene Stück Pfad möglichst kurz ist, allerdings wird nicht auf die Gesamtlänge vom Knotenpunkt bis zu der neuen Konfiguration geachtet. 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* [4] an. Im Grunde erweitert sie den normalen RRT um zwei Funktionalitäten. Das **ChooseParent** sorgt dafür, dass eine neu entstehende Kante nicht lediglich die kürzeste, sondern global gesehen die optimalste ist. Das **ReWire** schaut nach dem Setzen eines neuen Knotens, ob durch diesen andere Knoten schneller erreicht werden können und strukturiert Kanten gegebenenfalls um.

3.2 Algorithmus

Der veränderte Algortihmus ist in Abbildung 5 zu sehen, welcher sich bis zur Zeile 6 nicht von dem normalen RRT (Abb. 1) unterscheidet. Erst nachdem ein neuer Knoten (z_{new}) mit gültigem Pfad (x_{new}) gefunden wurde, gibt es Unterschiede zum ursprünglichen Algorithmus. Bevor die $ChooseParent$ und $ReWirer$ Funktionen einge-

$\underline{T = (V, E) \leftarrow RRT^*(z_{init})}$

```
 1 T ← InsertNode(∅, z_init);
 2 for i = 1 to N do
 3 |   z_rand ← Sample(i);
 4 |   z_nearest ← Nearest(T, z_rand);
 5 |   (x_new, u_new, z_new) ← Steer(z_nearest, z_rand, Δt);
 6 |   if ObstacleFree(x_new) then
 7 |   |   Z_near ← Near(T, z_new, |V|);
 8 |   |   z_min ← ChooseParent(Z_near, z_nearest, z_new, x_new);
 9 |   |   T ← InsertNode(z_min, z_new, T);
10 |   |   T ← ReWire(T, Z_near, z_min, z_new);
11 |   end
12 end
13 return T
```

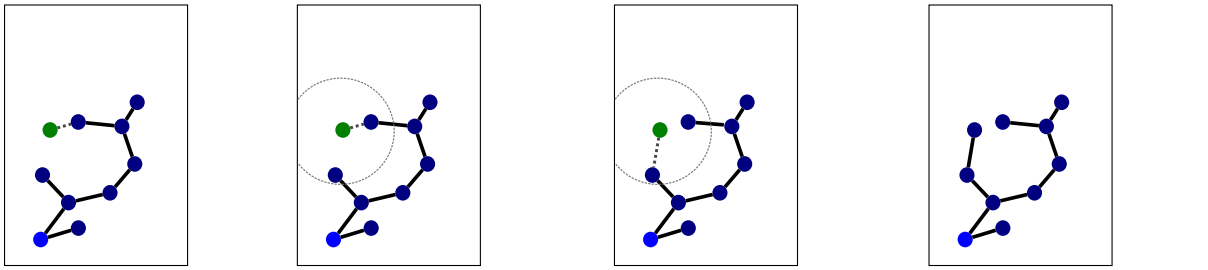
Abb. 5 RRT* - Algorithmus

setzt werden können, wird die von diesen Funktionen benötigte Menge von Knoten (Z_{near}) gesucht, deren Elemente sich innerhalb einer vordefiniert-großen ($|V|$) Kugel

um z_{new} befinden (Zeile 7). Hierbei sollte $|V|$ so gewählt werden, dass alle Knoten in dieser Kugel liegen, von welchen aus z_{new} in einem Schritt(Δt) zu erreichen ist (siehe Abbildung 6 b)).

3.2.1 ChooseParent

In Abbildung 6 erkennt man die Einordnung der ChooseParent-Funktion und ihre Aufgabe. Nachdem mittels der steer-Funktion ein neuer Knoten z_{new} ermittelt wurde, schaut diese in dessen unmittelbarer Umgebung, ob es einen optimaleren Vaterknoten als $z_{nearest}$ gibt, durch den die Kosten von der Startkonfiguration bis zu z_{new} minimiert würden.



a) $(x_{new}, z_{new}) \leftarrow Steer(z_{nearest}, z_{rand}, \Delta t)$;

b) $Z_{near} \leftarrow Near(T, z_{new}, |V|)$;

c) $z_{min} \leftarrow ChooseParent(Z_{near}, z_{nearest}, z_{new}, x_{new})$;

d) $InsertNode(z_{min}, z_{new}, T)$;

Abb. 6 Hinzufügen eines neuen Knotens in den Graphen mittels ChooseParent-Funktion

$ChooseParent(Z_{near}, z_{nearest}, z_{new}, x_{new})$

```
1  z_min ← z_nearest;
2  c_min ← Cost(z_nearest) + c(x_new);
3  for z_near ∈ Z_near do
4  |   (x', T') ← Steer(z_near, z_new);
5  |   if (ObstacleFree(x')) ∧ (x'(T') = z_new) then
6  |   |   c' = Cost(z_near) + c(x');
7  |   |   if (c' < c_min) then
8  |   |   |   z_min ← z_near;
9  |   |   |   c_min ← c';
10 |   |   end
11 |   end
12 end
13 return z_min
```

Abb. 7 ChooseParent - Funktion

Wie der ChooseParent-Algorithmus im Detail funktioniert sieht man in Abbildung 7. Nachdem die Z_{near}-Kugel ermittelt wurde, geht die Funktion über jeden Knoten aus dieser Menge (Zeile 3) und überprüft, ob dieser ein potentieller Vaterknoten für z_{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 z_{new} wirklich erreicht wird (Zeile 5). Nun muss noch unter allen potentiellen Vaterkonfigurationen dieser bestimmt werden, über welche die Kosten von der Startkonfiguration bis hin zu dem neuen Zustand z_{new} minimal sind (Zeilen 6-7). Es wird als Vaterknoten also der Knoten gesucht, welcher folgende Gleichung erfüllt:

$$z_{min} = \underset{z_{near} \in Z_{near}}{\arg\min} \; (Cost(z_{near}) + c(x_{z_{near}, z_{new}}),$$

wobei $Cost(z)$ die Kosten vom Startzustand bis zu einem bestimmten Knoten beschreibt, $c(x)$ die Kosten eines bestimmten Pfades beschreibt und $x_{z_{near}, z_{new}}$ der neue Pfad von z_{near} bis z_{new} ist.

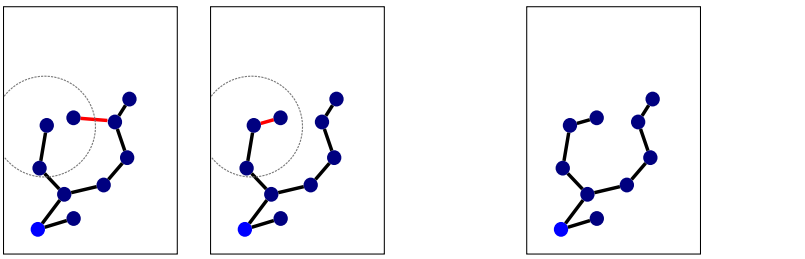
3.2.2 ReWire

Nachdem mit der CooseParent-Funktion bereits der bestmögliche Vaterknoten für z_{new} ermittelt und in den Graphen eingetragen wurde, schaut die ReWire-Funktion nach, ob sich die Kosten von einer der Knotenpunkte aus der Z_{near}-Kugel dadurch senken lassen, dass dieser nun die z_{new}-Konfiguration als Vaterknoten nimmt. Eine solche

$ReWire(T, Z_{near}, z_{min}, z_{new})$

```
1 for z_near ∈ Z_near \ {z_min} do
2 |   (x', T') ← Steer(z_new, z_near);
3 |   if (ObstacleFree(x')) ∧ (x'(T') = z_near)
4 |      ∧(Cost(z_new) + c(x') < Cost(z_near)) then
5 |     T ← ReConnect(z_new, z_near, T);
6 |   end
7 end
8 return T
```

Abb. 8 ReWire - Funktion



e) $foreach$ $(z_{near} \in Z_{near})$

f) $if(Cost(z_{near}) > Cost(z_{new} + c(x_{z_{new}, z_{near}}))$

g) $ReConnect(z_{new}, z_{near}, T);$

Abb. 9 Optimierung des Baumes mittels ReWire-Funktion

Beispielsituation ist in Abbildung 9 als Fortführung von Abbildung 6 abgebildet. Den genauen Algorithmus hierzu sieht man in Abbildung 8. Hierin wird wieder über jeden Knoten aus Z_{near} außer z_{min} iteriert und überprüft, ob diese Knoten von z_{new} aus erreichbar sind (Zeilen 1-4). Falls dies der Fall ist, wird geschaut, ob die Kosten sinken würden und gegebenenfalls wird z_{new} als neuer Vaterknoten deklariert (Zeilen 5-6).

3.3 Errungenschaften pt.2

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. Welche Probleme die Nichtholonomie jedoch trotzdem mit sich bringt, wird in einem späteren Kapitel beschrieben. Die signifikante Veränderung des RRT* ist ein großer Schritt hin zur Optimalität. Sobald ein gültiger Pfad gefunden wurde, wird dieser Pfad stetig weiter verbessert, solange der Algorithmus weiter läuft. Dadurch, dass Pfadstücke nur verändert werden, wenn dadurch auch die Kosten sinken, kann es nicht passieren, dass sich die Gesamtkosten wieder verschlechtern. Sie nähern sich also immer weiter dem absoluten Optimum an. Daher spricht man in diesem Fall davon, dass der RRT* **asymptotisch optimal** [4] [8] ist.

4 RRT Erweiterungen

Das folgende Kapitel gibt einen Überblick über eine Reihe kleinerer Erweiterungen und Heuristiken des RRT. Hier eine Übersicht über die beschriebenen RRT-Varianten:

4.1 Sample-Heuristiken
  4.1.1 goal bias
  4.1.2 local bias
  4.1.3 node rejection
4.2 bidirektionaler RRT
  4.2.1 RRT-ExtExt
  4.2.2 RRT-Connect
4.3 Dynamic-Domain RRT
4.4 recursive ReWire

4.1 Sample Heuristiken

4.1.1 goal bias

Der goal bias [11] 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. Ziel dieser Heuristik ist es, dem Baumwachstum eine Tendenz in Richtung der Zielzustände zu geben, um einen Zielpfad schneller zu finden. Dies funktioniert besonders gut, falls es bereits eine Stelle gibt, an der sich zwischen den Baumblättern und den Zielzuständen keine Hindernisse mehr befinden. Erreicht wird diese Orientierung dadurch, dass die *sample*-Funktion zu einem gewissen Anteil ein Zielzustand z_{goal} anstatt eines zufälligen Zustandes zurück gibt, wie in Abbildung 10 zu sehen ist.

4.1.2 local bias

Der local bias [11] ist ebenso wie der goal bias eine Heuristik, welche das sampling abändert, um dem Baumwachstum eine gewisse Richtung zu geben. Allerdings ist

gb_sample(i)

```
1 p ← rand(0.0, 1.0);
2 if ((p > α) ∧ (¬Pathfound))
3 then
4 |  return z_goal;
5 else
6 |  return sample(i);
7 end
```

Abb. 10 goal-bias sampling

der local bias erst sinnvoll, 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 komplett ausgeschlossen werden, da hierdurch eventuell auch noch bessere Wege gefunden werden können. 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, zu erkennen im oberen Algorithmus der Abbildung 11. Implementiert wird diese Heuristik, indem der *local_bias_path*() eine Zufallskonfigura-

lb_sample(i)

```
1 p ← rand(0.0, 1.0);
2 if ((p > β) ∧ (Pathfound))
3 then
4 |  return local_bias_sample(path);
5 else
6 |  return sample(i);
7 end
```

local_bias_sample(path)

```
1 z ← random_node(path);
2 z_1 ← path(z − 1);
3 z_2 ← path(z + 1);
4 z_tmp ← (z_1 + z_2)/2 − z;
5 z_rand ← z + z_tmp/|z_tmp| * rand(r_min, r_max);
  return z_rand;
```

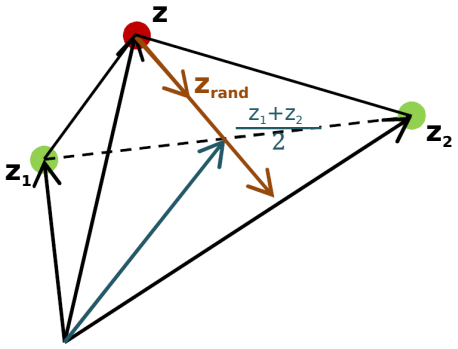


Abb. 11 local-bias sampling

tion zurück gibt, welche in Pfadnähe liegt, mit der Absicht diesen dadurch zu optimieren. Um ein sample in Pfadnähe zu bekommen, wird zunächst zufällig ein Knoten z aus dem bereits gefundenen Pfad ausgewählt, sowie sein Vorgänger z_1 und Nachfolger z_2 bestimmt. Da sich die Zustande im $\mathbb{R}^n$ befinden, können die Knoten als Vektoren in diesem Raum vorgestellt werden und mittels einfacher Vektorarithmetik einen neuen zufälligen Knoten nahe z, zwischen z_1 und z_2 ermittelt werden.Dies wurde zum besseren Verständnis in der Skizze in Abbildung 11 visualisiert.

4.1.3 node rejection

Eine dritte Erweiterung des Sampling-Vorganges ist die node rejection [11], welche ebenfalls lediglich 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. Abbildung 12 zeigt, dass keine Zufallskonfigurationen akzeptiert werden, deren minimalen Kosten von der Startkonfiguration z_{init} bis zu dem Sample z_{rand} und von z_{rand} zu einer Zielkonfiguration z_{goal} bereits größer sind als die Kosten c_{best} des aktuell besten Zielpfades. Die minimalen Kosten müssen auf jeden Fall eine untere Schranke sein (hier zum Beispiel die euklidische Distanz) und es können nicht die aktuell angenommen Kosten des Knotens genommen werden, da sich diese je nach RRT-Algorithmus noch verringern könnten. Eine weitere Alternative dieser Heuristik besteht darin, in einer gewissen Frequenz über

nr_sample(i)

```
1 repeat
2 |  z_rand ← sample(i);
3 until (|z_rand − z_init| + |z_goal − z_rand| <= c_best);
4 return z_rand;
```

Abb. 12 node-rejection sampling

alle bereits bestehenden Knoten zu iterieren und zu testen, ob diese noch potentiell den aktuellen Pfad verbessern könnten und sie ansonsten aus dem Baum zu entfernen.

4.2 bidirektionaler RRT

Wird der RRT mit einem bidirektionalem Ansatz erweitert, so wird nicht nur ein Baum vom Startzustand aus aufgebaut, sondern parallel wird noch ein zweiter Baum von einem Zielzustand aus aufgespannt. Sobald sich die beiden Bäume treffen, kann, durch die Verknüpfung dieser, ein Pfad vom Startzustand bis hin zu dem Zielzustand ermittelt werden. Das Ziel dieses Ansatzes ist es Rechenzeit einzusparen, dadurch dass zwei kleinere Bäume, welche sich in der Mitte treffen, insgesamt weniger Knotenpunkte benötigen als ein Großer.

4.2.1 RRT-ExtExt

Eine Möglichkeit des bidirektionalen RRTs ist der RRT-ExtExt[10]. Für diesen kann die $extend(T, z_{rand})$-Funktion (Abb. 2) weitestgehend unverändert bleiben. Es wird lediglich noch ein Rückgabewert erwartet, welcher angibt, ob der Zielknoten z_{rand} tatsächlich erreicht wurde (*Reached*), ob der Baum lediglich um einen Knoten in Richtung z_{rand} erweitert wurde (*Advanced*) oder ob es durch eine Hinderniskollision zu keiner Erweiterung gekommen ist ($Trapped$). Mithilfe dieser Rückmeldung kann der RRT-Algorithmus wie in Abbildung 13 erweitert werden. Statt eines einzigen Baumes wird hier nun ein Baum T_a vom Start ausgehend und ein Baum T_b von einem Zielzustand ausgehend initialisiert (Zeilen 1-2). Wird nun T_a erfolgreich um z_{new} erweitert (Zeile 6), so wird versucht, T_b in Richtung dieses neuen Zustandes zu expandieren

$RRT - ExtExt(z_{init}, z_{goal})$

```
 1 T_a ← InsertNode(∅, z_init);
 2 T_b ← InsertNode(∅, z_goal);
 3 for i = 1 to N
 4 do
 5 |   z_rand ← Sample(i);
 6 |   if (extend(T_a, z_rand)! = Trapped)
 7 |   then
 8 |   |   if (extend(T_b, z_new) = Reached)
 9 |   |   then
10 |   |   |   return path(T_a, T_b);
11 |   |   end
12 |   |   swap(T_a, T_b);
13 |   end
14 end
15 return T
```

Abb. 13 RRT-ExtExt

(Zeile 8). Falls mit dieser Erweiterung bereits der neue Knoten z_{new} erreicht wurde, sind damit die beiden Bäume verbunden und es kann direkt der Pfad von Start- zu Zielknoten ermittelt werden. Falls dies allerdings noch nicht der Fall ist, so wird dieser Vorgang wiederholt, mit vertauschen Rollen der Bäume (Zeile 12). Ein Problem, welches bei dem Verbinden zweier Bäume mit nichtholonomen Fahrzeugen auftritt, wird in Kapitel 5.1.1 näher erläutert.

4.2.2 RRT-Connect

Eine Erweiterung des RRT-ExtExt ist der RRT-Connect [14], bei der es lediglich eine Veränderung in dem Versuch den zweiten Baum T_b in Richtung des neuen Zustandes z_{new} zu Erweitern (Abb. 13, Zeile 8) gibt. In dieser wird die *extend*-Funktion durch die *connect*-Funktion ersetzt, welche in Abbildung 14 zu sehen ist. Hier wird nicht nur *ein* Erweiterungsschritt durchgeführt, sondern es wird so lange *extend*ed wie dieses ein *Advanced* zurück gibt. Dies bedeutet, dass die Verbindung der beiden Bäume lediglich dann gestoppt wird, wenn sich ein Hindernis im Weg befindet oder wenn die Zusammenkunft erfolgreich war. Durch diese Veränderung wird erreicht, dass die Bäume direkt zusammenwachsen, sobald sich kein Hindernis mehr zwischen diesen befindet. Somit kann schneller ein gültiger Pfad gefunden werden.

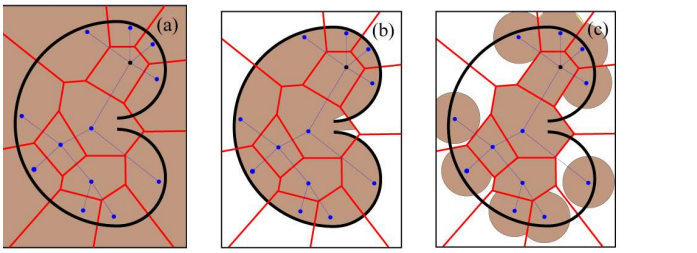
$connect(T, z)$

```
1 repeat
2 |   S ← extend(T, z);
3 until (S! = Advanced);
4 return S;
```

Abb. 14 Erweiterung des RRT-Connect

4.3 Dynamic-Domain RRT

Bei dem Dynamic Domain RRT [13] wird das Sampling in den Bereichen reduziert, in deren Richtung der Baum nicht weiter wachsen kann, da er sonst gegen Hindernisse laufen würde. LaValle [13] demonstrierte dieses Problem am Beispiel der sogenannten bugtrap (Abbildung 15 a)). Zu sehen ist hier das Hindernis (schwarz) sowie der bereits aufgespannte RRT-Baum (blau). Am eingezeichneten Voronoi Diagramm (rot) erkennt man, an welchem Knoten der Baum wachsen würde abhängig von der Stelle im Konfigurationsraum, an der der neue Zufallsknoten liegt(vgl. Kapitel 2.3). Hierdurch lässt sich leicht erkennen, dass es nur einen sehr kleinen Bereich gibt, an den das nächste sample fallen müsste, damit der Baum aus der "Falle" heraus wachsen könnte. Würde das Sample an irgend einer 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 Falle heraus wächst. Somit ergibt sich das Problem, dass es einen *enormen Leistungsverlust des RRT-Algorithmus gibt, wenn viele* ***frontier nodes*** *(≙ Randknoten) gleichzeitig auch* ***boundary nodes*** *(≙ Hindernisnahe Knoten) sind.* Die ideale Lösung wären hierfür die *visible regions*, zu sehen in Abbildung 15 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 Hang gegen Hindernisse zu laufen entfernt würde und trotzdem der sehr wichtige Bias des Wachsens in unbekannte Gebiete weiter unterstützt würde. Daher versucht der dynmaic domain RRT diesen Idealfall anzunähern, mit sehr geringem Rechenaufwand und ohne die Vollständigkeit des RRT zu verlieren. Hierbei kann der Drang gegen die Hindernisse nicht komplett entfernt, aber trotzdem merklich verringert werden und der Bias in die unbekannten Gebiete bleibt trotzdem bestehen. Erreicht wird dies, in-



a) "bugtrap"
b) ideales sampling
c) dynamic domain sampling

Abb. 15 sampling Bereiche

dem jedem Knoten ein Radius, welcher im Normalfall unbegrenzt ist (Abb. 16 Zeile 8), mitgegeben wird. 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 (Abb. 16 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 vorgenommen werden muss, die, dass ein neues Sample z_{rand} nur dann akzeptiert wird, wenn es innerhalb des Radius seines

$T = (V, E) \leftarrow DynamicDomain\_RRT(z_{init})$

```
1  T ← InsertNode(∅, z_init); for i = 1 to N do
2  |  repeat
3  |  |  z_rand ← Sample(i);
4  |  |  z_nearest ← Nearest(T, z_rand);
5  |  until dist(z_nearest, z_rand) ≤ z_nearest.radius;
6  |  (x_new, z_new) ← Steer(z_nearest, z_rand, Δt);
7  |  if ObstacleFree(x_new) then
8  |  |  z_new.radius ← ∞;
9  |  |  T ← InsertNode(z_nearest, z_new, T);
10 |  else
11 |  |  z_nearest.radius ← R;
12 |  end
13 end
14 return T
```

Abb. 16 dynamic domain RRT

nächsten Nachbarn liegt (Abb. 16 Zeile 5). Dies bewirkt, dass die Zufallskonfigurationen zwar immer noch im kompletten Konfigurationsraum gezogen werden, allerdings werden bei den meisten Punkten, bei denen die Erweiterung fehl schlagen würde, von vornherein auf die sehr kostenintensive Steuerfunktion(*steer*) und die Kollisionserkennung (*ObstacleFree*) verzichtet. Abbildung 17 verdeutlicht noch einmal, inwiefern sich das Sampling im Vergleich zu den normalen RRT und den visible regions verändert. Die graubraun eingefärbten Gebiete sind die, in denen neue Zufallskonfigurationen z_{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 nodes auf, welche gleichzeitig auch frontier nodes sind.

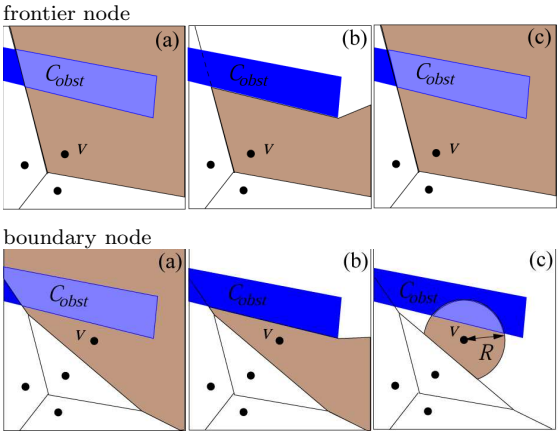


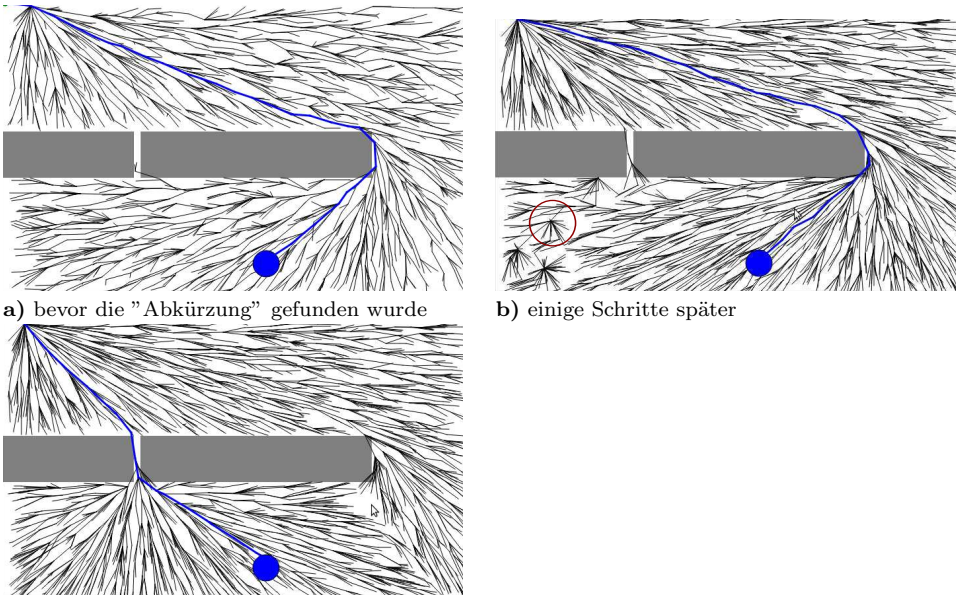
Abb. 17 **a)**normales sampling **b)** visible regions sampling **c)** dynamic domain sampling

Experimente mit dem bidirektionalem Dynmaic Domain RRT-Connect haben gezeigt, dass die beschriebene Erweiterung in Sonderfällen wie etwa in Abbildung 15 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 [13]. Da ein solcher Sonderfall in der Realität jedoch kaum auftritt, wurden auch Experimente mit einem motion planning problem 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 [13]. Es sollte jedoch beachtet werden, dass die Messergebnisse nicht von unabhängigen Dritten stammen, sondern von LaValle selbst.

4.4 recursive ReWire

Das folgende Kapitel erläutert ein Mangel/Defizit/Makel des RRT*, welcher mir bei den Recherchen aufgefallen ist, und zeigt meinen ersten Ansatz zu seiner Behebung. Der RRT* versucht den RRT so abzuändern, dass das Ergebnis der Optimalität 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 aber 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 hat. Offensichtlich wird die an dem Beispiel in Abbildung 18 a). Hierbei ist die Startkonfiguration 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 ge-



a) bevor die "Abkürzung" gefunden wurde

b) einige Schritte später

c) mit recursiv ReWire

Abb. 18 Motivation und Ergebnis des rekursiven ReWire

plant. Sobald nun aber doch der kürzere Weg durch das Hindernis hindurch gefunden wurde, sieht der Baum im normalen RRT so aus wie in Abbildung 18 b). Wird ein neuer Knoten gesetzt, welcher Anschluss an den kürzeren neuen Pfad hat, wird dieser mit dem kurzen Pfad verbunden (durch das *chooseParent*) und alle Knoten im engeren Umfeld nutzen nun diesen neuen Knoten durch das ReWire als Vaterknoten, da dieser wesentlich geringere Kosten hat. Hierbei ist nun gut zu erkennen, dass dieser Optimierungsschritt lediglich in dem kleinen Z_{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 *reconnect*en ihre Kosten senken könnten.
Eine einfache Lösung dieses Gedankens findet man in Abbildung 19. Hier wird das ReWire rekursiv an jeden Knoten weitergegeben, welcher seine Kosten senken konnte. Dies ist jedoch nur ein erster Gedanke und wäre in der Laufzeit noch zu verbessern, zum Beispiel indem man statt Tiefensuche die Breitensuche für die ReWire-Weitergabe nutzt. Als Ergebnis erhält man einen Baum, wie in Abbildung 18 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.

$ReWire(T, Z_{near}, z_{min}, z_{new})$

```
1 for z_near ∈ Z_near \ {z_min} do
2 |   (x', T') ← Steer(z_new, z_near);
3 |   if (ObstacleFree(x')) ∧ (x'(T') = z_near) ∧ (Cost(z_new) + c(x') < Cost(z_near))
  |   then
4 |   |   T ← ReConnect(z_new, z_near, T);
5 |   |   Z'_near ← Near(T, z_near, |V|);
6 |   |   ReWire(T, Z'_near, z_new, z_near)
7 |   end
8 end
9 return T
```

Abb. 19 rekursive ReWire-Funktion

In der Literatur lässt sich ein solcher Ansatz jedoch nicht finden. Dies könnte daran liegen, dass ohne diese Erweiterung das Hinzufügen eines Knotens eine sehr konstante Zeit in Anspruch nimmt, welches durch die Rekursion nicht mehr gegeben ist. 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 läuft. Jedoch ergibt sich durch das recursive ReWire auch eine neue sehr gute Invariante: *Es ist zu jeder Zeit gegeben, dass zu jedem Knoten der über die bereits bestehenden Knoten optimalste Pfad führt.*

5 Trajektorienplanung in der Praxis

Die vorherigen Kapitel behandelten den Aufbau des RRT-Algorithmus, ohne zu beachten, inwiefern dieser in der Praxis anwendbar ist. Das folgende Kapitel

beschäftigen sich nun mit dem Einsatz des RRT als Echtzeit-Trajektorienplanung für autonome Fahrzeuge. Es werden Probleme betrachtet, welche durch die Anwendung im realen Straßenverkehr auftreten und mögliche Lösungsansätze besprochen. Neben der Trajektorienplanung wird der RRT auch oft für die Steuerung von Roboterarmen oder Flugzeugen genutzt, welche in diesem Paper aber keine Rolle spielen werden.

5.1 Steuerfunktion

Das wohl größte Problem des RRT ist die Behandlung von komplexeren Fahrzeugmodellen. Der RRT an sich hat zwar keine Beschränkung in der Höhe der Dimensionen des Zustandraumes, jedoch wird die Aufstellung der *steer*-Funktion irgendwann nicht mehr möglich. Benötigt wird für die meisten RRT-Varianten eine Funktion $f(z, z') = u$, welche einen Start- und einen Zielzustand entgegen nimmt und die gesuchte Steuergrößen zurück gibt. Bei einfachen Fahrzeugmodellen wie zum Beispiel den Dubins Fahrzeuge (Abb. 20 a) nach [16]) ist dies noch einfach zu realisieren. Soll das Fahrzeug jedoch realitätsgetreu modelliert werden, so muss auch die instabile Dynamik sowie ein nicht zu vernachlässigbarer Drift mit bedacht werden. Ein Beispiel hierfür ist das Rallye-Auto-Modell von Karaman und Frazzoli (Abb. 20 b) nach [3]).

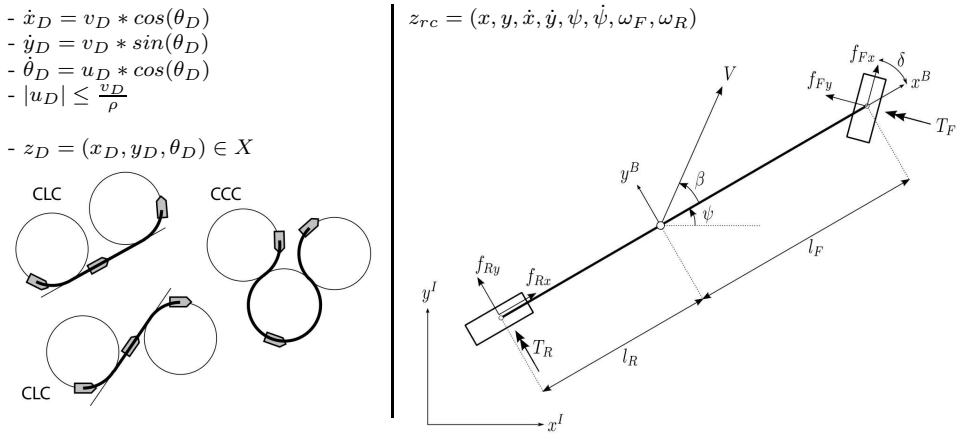
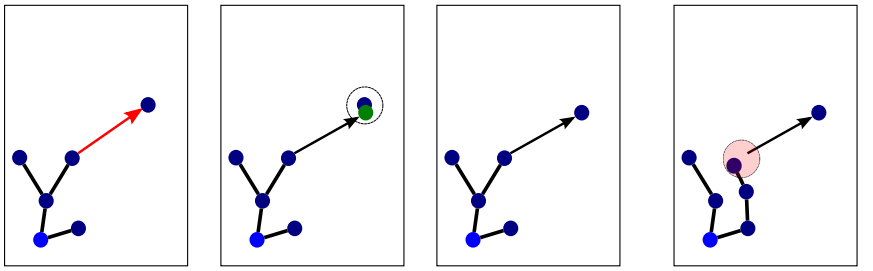


Abb. 20 a) Dubins vehicle Modell [16] **b)** high-speed off-road rally car [3]

5.1.1 Toleranzkugel

Bei solchen Fahrzeugmodellen ist es nicht mehr so einfach eine passende *steer*-Funktion nach $f(z, z') = u$ zu finden (Abb. 21 a) u = Pfeil). Einfach ist jedoch das Erstellen einer Funktion $f(z, u) = z'$ (Abb. 21 b) z' = grüner Punkt). Diese Funktion berechnet die Zielkonfiguration abhängig von der Startkonfiguration und den Steuerbefehlen, welches in einigen Fällen bereits genügt. Es muss lediglich eine Kugel um die gewünschte Zielkonfiguration erlaubt werden, in der die angefahrene Zielkonfiguration liegen darf (Abb. 21 b) schwarzer Kreis), um als neuer Zielpunkt akzeptiert zu werden (Abb. 21 c)). Diese Vorgehen bereitet jedoch Probleme, falls die Zielkonfiguration



a) $f(z, z') = u$ **b)** $f(z, u) = z'$ **c)** $InsertNode(z, z')$ **d)** ReWire-Problem

Abb. 21 steering-Problem

fest definiert ist. Dies ist zum Beispiel der Fall beim $ReWire$ing im RRT*. Wird ein Knoten $ReConnect$ed, so lässt sich dabei die Position dieses Knotens nicht einfach ändern, da hierdurch die Wege von diesem Knoten zu seinen Kinderknoten ungültig werden (Abb. 21 d)). Aufgrund dessen ist es nun notwendig, alle Pfade der Kinderknoten rekursiv neu zu berechnen, was zu enormen Rechenaufwand führen kann. Noch gravierender ist dieses Problem bei einem bidirektionalem RRT, da hierbei die beiden Bäume miteinander verbunden werden müssen und dies nur möglich ist, wenn die Steuergrößen u zwischen zwei bereits bestehenden Knoten ermitteln werden können.

5.1.2 Dimensionsreduzierung

Um dieses Problem zu umgehen, kann die Trajektorienplanung in zwei Schichten aufgeteilt werden [5], um dadurch den RRT zu vereinfachen. Ziel dieses Vorgehens ist es, die Dimensionen des Konfigurationsraumes zu reduzieren und die komplizierte fahrzeugpezifische Steuerung an einen separaten Regelalgorithmus zu übergeben. Ein Modell des MIT sieht so aus, dass der RRT-Planer lediglich die Zielkonfiguration von einer höher liegenden Schicht erhält, zum Beispiel einen GPS-Navigationssystem, sowie die aktuellen Lagen der Hindernisse von der Sensorik des Fahrzeuges (Abb. 22, "Motion Planner"). Die Aufgabe des RRT ist es nun, lediglich einen zweidimensionalen (x,y)-Pfad zu planen, welcher den Hindernissen ausweicht, sich aber nicht um die genau Fahrzeugdynamik kümmert. Hierbei kann nun der Pfad zwischen zwei Konfigurationspunkten mit den einfach zu berechnenden Dubinspfaden (Abb. 20 a)) beschrieben werden oder wie in Abbildung 22 mittels gerader Verbindungen. Dieser Pfad wird nun an einen Regelalgorithmus weitergegeben (Abb. 22, "Controller"), welcher dafür sorgt, dass das Fahrzeug den vorgegebenen Pfad entlang fährt.

Da der Controller mit höherer Taktrate laufen kann als der Planer, ist es für diesen auch einfach, auf Ungenauigkeiten und Rauschen in der Sensorik zu reagieren. So muss der RRT gar nicht erst erfahren, wenn das Fahrzeug nicht ganz genau auf dem Pfad liegt, da der Regler dies wieder ausgleichen wird. Lediglich wenn ein sehr großer Unterschied zwischen dem geplanten Pfad und der tatsächlichen Pose liegt, wird dies an den Planer weitergegeben, damit dieser einen neuen Pfad ermittelt. In Abbildung 22 stellt der gelbe Pfad den vom RRT geplanten Pfad dar und die grüne Linie ist die vom Regler geplante Trajektorie. Zu beachten ist, dass der Regler hier auch aus zwei Teilen besteht, einmal die Vorhersage, welche die Trajektorie vorausberechnet und auf Gültigkeit prüft und ein zweiter, welcher diese Trajektorie dann ausführt.

Durch das Separieren der Pfadplanung im Großen (RRT) und der Trajektorienplanung

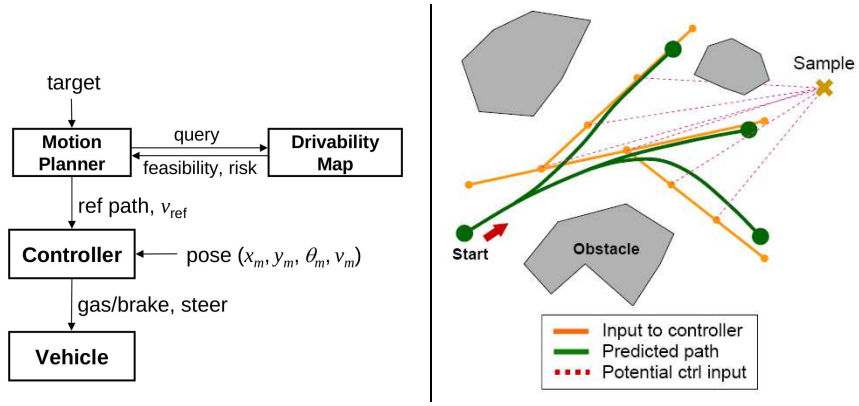


Abb. 22 Aufteilung der Trajektorienplanung in mehrere Schichten [5]

im Kleinen (Regler) wird erreicht, dass sich der RRT schneller ausbreiten kann und dass es einfacher ist, weitere Änderungen zur Optimierung hinzuzufügen. Des Weiteren kann ein Regler schneller und präziser auf unvorhersehbare Änderungen im Pfad reagieren, da nicht der komplette Suchbaum neu aufgebaut werden muss. Es ist weiterhin sinnvoll, die Geschwindigkeitsplanung dem Regler zu überlassen, anstatt diese mit zu samplen, da jede weitere Dimension im RRT eine merkliche Aufwandserhöhung darstellt.

5.2 anytime motion planner

Eine weitere Herausforderung des Einsatzes des RRTs in der Trajektoreienplanung ist die Handhabung einer dynamischen Umgebung. In einer statischen, vollständig bekannten Umgebung ist es möglich, vor Fahrbeginn mittels RRT eine Trajektorie zu bestimmen und dieses anschließend auszuführen. Möchte man den Algorithmus jedoch für eine reale Fahrt in dynamischer Umgebung nutzen, so muss während der Fahrt weiter geplant werden, um immer auf den aktuellen Umgebungssensordaten arbeiten zu können. Ebenso ist es dadurch möglich, mit begrenzter Sensorik in unbekannter Umgebung zu fahren, da sich hier das Umgebungsbild erst während der Fahrt weiter aufbaut.
Realisieren lässt sich dies mit dem *online RRT* [2], indem zuerst ein initiale Pfad geplant wird, bevor das Fahrzeug startet. Sobald es diesen hat kann es beginnen diesen Abzuführen, jedoch muss der RRT-Algorithmus weiterlaufen, um den Pfad weiter zu optimieren und an die Umgebung anzupassen. Hierzu wird in einer gewissen Frequenz das erste Stück des aktuell besten Pfades an die Fahrzeugsteuerung weitergegeben, welche dieses Teilstück ausführt. Der Endpunkt dieses Teilstückes dient als neue Startkonfiguration des RRT. Nun wird jedoch nicht komplett neu geplant, sondern der bestehende Baum erweitert. Nutzt man den RRT* oder einen ähnlichen Algorithmus, so ist dies auch in statischen und vollständig bekannten Umgebungen nützlich, da hierbei die Ausführungszeit zur weiteren Optimierung des Pfades genutzt werden kann und das Fahrzeug daher früher starten kann und trotzdem eine möglichst optimalen Trajektorie fährt.
Betrachtet man nun die Pfadplanung, so erinnert diese an einen Regelalgorithmus

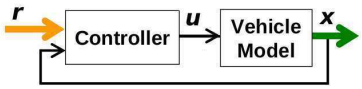


Abb. 23 closed loop RRT

(Abb. 23), da er einen Sollwert (Zielkonfiguration) als Eingabe erhält und den Istwert (aktuelle Konfiguration) abhängig von den Messwerten (Umgebungssensorik) auf dieses Ziel einstellt. In diesem Fall spricht man auch von einem *closed loop RRT* [5][6].

5.3 Talos

5.3.1 Einordnung von Talos

Abschließend wird noch als Beispielsystem auf den Pfadplanungsalgorithmus **Talos** [9] [7] [15] eingegangen, um einen Einblick zu geben, wie man den RRT im Straßenverkehr nutzen kann und um zu demonstrieren, wieviel tatsächlich noch hinzugefügt werden muss, bevor ein solches System straßentauglich ist.

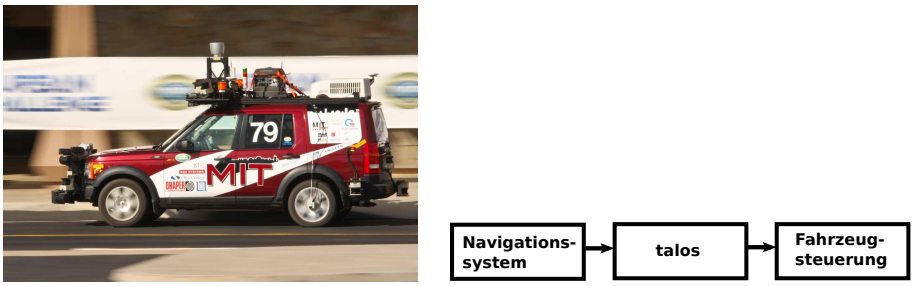


Abb. 24 Talos Subsystem

Talos ist ein Subsystem zwischen dem höher liegenden Navigationssystem und dem tieferliegendem Controller (Abb.24), welches das Massachusetts Institute of Technology (MIT) in der DARPA Urban Challenge 2007 erfolgreich nutzte. In der DARPA Urban Challenge ging es darum, mit 70 anderen Verkehrsteilnehmern in einer städtischen Umgebung autonom zu fahren, wobei sich an alle Verkehrsregeln gehalten werden musste und verschiedene Manöver, wie zum Beispiel Überholmanöver oder Einparken, durchgeführt wurden. Ziel des MIT war es, hierfür einen universellen Pfadplanungsalgorithmus zu entwickeln, welcher in allen möglichen Situationen und Manövern einsetzbar sein sollte.
In Abbildung 25 ist der vollständige Algorithmus zu sehen. Man erkennt, dass hier der online-RRT (vgl. kapitel 5.2) eingesetzt wurde, der lediglich im Zweidimensionalem arbeitet und seinen besten Pfad an einen Regelalgorithmus weitergibt, welcher die Trajektorie im Detail plant und ausführt (vgl Kapitel 5.1.2). Im Folgenden wird auf einige spezifische Besonderheiten eingegangen.

```
 1 repeat
 2 |  Receive current vehicle states and environment
 3 |  repeat
 4 |  |  Take a sample for input to controller
 5 |  |  Select a node in tree using heuristics
 6 |  |  Propagate from selected node to the sample until the vehicle stop
 7 |  |  Add branch nodes on the path
 8 |  |  if propagated path is feasible with the drivability map then
 9 |  |  |  Add sample and branch nodes to tree
10 |  |  else
11 |  |  |  if all the branch nodes are feasible then
12 |  |  |  |  Add branch nodes to the tree and mark them as unsafe
13 |  |  |  end
14 |  |  end
15 |  |  for each newly added node v do
16 |  |  |  Propagate to the target
17 |  |  |  if propagated path is feasible with drivability map then
18 |  |  |  |  Add path to tree
19 |  |  |  |  Set costs of propagated path as upper bound of cost-to-go at v
20 |  |  |  end
21 |  |  end
22 |  until the time limit is reached;
23 |  Choose best safe trajectory in tree and check feasibility with latest drivability map
24 |  if best trajectory is infeasible then
25 |  |  Remove infeasible part from tree
26 |  |  goto line 23
27 |  end
28 |  Send the best trajectory to controller
29 until Vehicle reaches the target;
```

Abb. 25 Talos-Algorithmus

5.3.2 bias sampling (Zeile 4)

Wie oben bereits beschrieben, war es ein Ziel einen universellen Planer zu entwickeln, der sowohl Autobahnfahrten als auch zum Beispiel Einparken meistert. Um dies zu unterstützen, wurde eine parametrisierbarer Sampler eingesetzt. Für die Wahrscheinlichkeit, dass ein Sample an einer gewissen Stelle gezogen wird, wurde eine Gaußglocke

steuerbare
Zufallskonfiguration:
$z_{rand} = (x_{rand}, y_{rand}) \in X$

mit
- $x_{rand} \leftarrow x_0 + r * cos(\theta)$;
- $y_{rand} \leftarrow y_0 + r * sin(\theta)$;
- r, θ sind Gaussverteilungen um r_0, θ_0

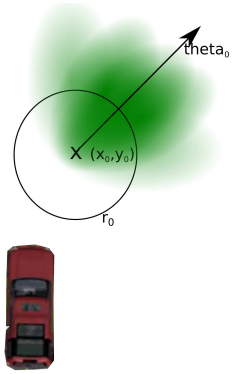


Abb. 26 parametrisierbares Sampling

in den 2D-Konfigurationsraum gesetzt. Diese lässt sich in Position, Ausrichtung und Größe einstellen. Im Detail ist dies in Abbildung 26 zu sehen. Diese Einstellungen werden von der Schicht über Talos vorgenommen und es kann so gezielt eine schnelle Pfadfindung erfolgen.

5.3.3 Distanz-Heuristiken (Zeile 5)

Um nach dem Sampling den nächsten Nachbar zu finden, nutzt Talos zwei verschieden Heuristiken. Die "Explorations-Heuristik" sorgt dafür, dass der Algorithmus sich eher schnell ausbreitet und so komplett neue Wege findet. Sie ist besonders sinnvoll, solange noch kein Zielpfad gefunden wurde. Hierbei wird als nächster Nachbar $z_{nearest}$ einer Zufallskonfiguration z_{rand} genommen, welcher die kürzeste Distanz (Dubinspfadlänge) zu diesem hat. Wurde jedoch bereits ein ausführbarer Pfad bis zur Zielkonfiguration gefunden, so ist eher ein Bias zur Pfadoptimierung erwünscht. Daher wird mit der "Optimierungs-Heuristik" als Vaterknoten nicht einfach der nächstliegende genommen, sondern der, der in der Summe der Dubinspfadlänge und eigener Kosten am geringsten ist.

$$Nearest_{exploration}(T, z_{rand}) = \underset{z_{near} \in Z}{\arg\min}(DubinDistanz(z_{near}, z_{rand}))$$
$$Nearest_{optimization}(T, z_{rand}) = \underset{z_{near} \in Z}{\arg\min}(DubinDistanz(z_{near}, z_{rand}) + Cost(z_{near}))$$

Um sich aber zu jeder Zeit alle Möglichkeiten offen zu halten, werden stets beide Heuristiken parallel genutzt. Bevor eine Zielpfad gefunden wurde, kommt die Explorations-Heuristik zu 70% zum Einsatz und die Optimierung-Heuristik lediglich zu 30%. Sobald der erste Zielpfad gefunden wurde, wechseln diese Wahrscheinlichkeiten, da nun nicht mehr die Suche sondern die Optimierung des Zielpfades im Vordergrund steht.

5.3.4 Notstopp (Zeile 6)

Um Sicherheit zu garantieren, führt Talos nur dann eine Ast des Baumes aus, wenn an mindestens einem seiner Blätter ein sicherer Zustand ist. Ein sicher Zustand ist hierbei ein solcher, in dem das Fahrzeug eine unbestimmt lange Zeit verbleiben kann, ohne dass eine Straßenregel verletzt wird oder es mit einem statischen oder dynamischen Gegenstand kollidiert, falls dieser seinen aktuellen Pfad beibehält. Beachtet man dies, so ist ausschließlich ein Pfad mit vollständigem Bremsweg(Nothalteweg) ein sicherer Zustand. Dies bedeutet, dass ein Pfad nur dann ausgeführt wird, wenn zu jeder Zeit ein potentieller Haltezustand erreicht werden kann.

5.3.5 drivability map (Zeile 8)

Die Repräsentation der Umwelt für den Planer erfolgt über die "drivability map" (Abbildung 27). Dies ist eine Karte, in der die bereits analysierten Daten der Umgebungswahrnehmung gespeichert werden. Hierin vermerkt sind Regionen, welche nicht befahren werden dürfen(rot), sowie Regionen, welche durchfahren werden können, in denen aber nicht gehalten werden darf (blau). Die Regionen in Grautönen sind die Gebiete, welche befahren werden dürfen, wobei der Grauwert die Kosten der Strecke definieren, je heller desto höher. Hierdurch lässt sich steuern, dass gewisse Gebiete eher

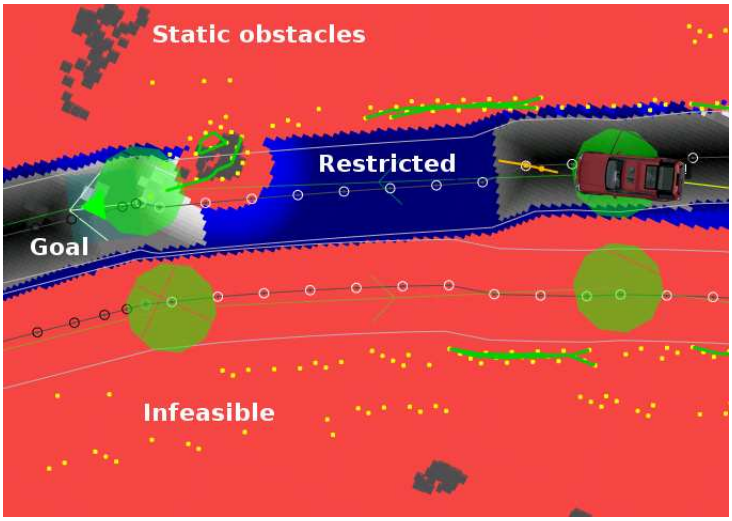


Abb. 27 drivability map

gemieden werden, um zum Beispiel nicht zu nah an Hindernissen vorbei zu fahren. Diese Karte ist die einzige Möglichkeit des RRT, die Pfade auf Kollisionsfreiheit zu testen und die Kosten der Pfade zu ermitteln.

5.3.6 lazy check (Zeile 24)

Da der Baum während der Fahrt weiter aufgebaut wird, kann es passieren, dass Teile des Baumes durch eine Veränderung der Umgebung ungültig werden. Um den Baum immer korrekt zu halten, müsste bei jeder Aktualisierung der drivability map jeder Knoten samt Pfad auf seine Gültigkeit geprüft werden. Dies würde jedoch sehr viel Leistung beanspruchen. Daher wird ein Pfad erstmals nur beim Erstellen mit der drivability map abgeglichen. Um nun aber auch kompatibel zu einer dynamischen Umgebung zu sein, wird das Teilstück, welches als bestes ermittelt wurde, direkt vor seiner Weitergabe an den Controller noch einmal auf seine Gültigkeit überprüft, um nur aktuell-gültige Pfade auszuführen. Hierdurch wird sehr viel Zeit gespart, aber das Risiko eingegangen, dass die Kosten der als-beste-gewählten Trajektorie gestiegen sind und eventuell näher an Hindernissen vorbei gefahren als gewünscht. Dieses Risiko geht man allerdings ein, da die Vorteile überwiegen.

6 Errungenschaften pt. 3

Rückblickend kann gesagt werden, dass der rapidly exploring random tree ein sehr mächtiger und allgemeiner Planungsalgorithmus ist. Der schnelle single-query Planer ist nachweisbar probabilistisch vollständig und korrekt. Durch die Offenheit der Steuerfunktion (*steer*) und der Kollisionsvermeidung ($ObstacleFree$) werden kaum Einschränkungen an das Bewegungsmodell gestellt, wodurch es für fast alle Bewegungsplanungs-Probleme anwendbar ist. Hauptsächlich findet man den Algorithmus in der Fahrtplanung von Fahrzeugen oder in der Ansteuerung von Roboterarmen. Zusätzlich lässt er sich beispielsweise auch in der Flugplanung oder der Entwickliung von Medikamenten einsetzen. Da die Grundversion des RRT, die Steven M. LaValle 1998 entwickelte, sehr allgemein gehalten ist, gibt es zahlreiche Erweiterungen und Heuristiken. Von diesen ist vor allem der RRT* eine signifikante Abänderung, da sie eine asymptotische Optimalität hinzufügt. Andere Erweiterungen versuchen meist den Zielpfad schneller zu finden, um den Rechenaufwand zu verringern, was vor allem in der Echtzeitplanung wichtig ist. Dadurch, dass es eine Vielzahl von Erweiterungen gibt und diese oftmals nur an speziellen Sachverhalten anwendbar sind, ist es schwer einen schnellen Überblick zu gewinnen.
Speziell für die Trajektorienplanung von autonomen Fahrzeugen ist der RRT eine gute Wahl, da er kaum Beschränkungen an das Fahrzeugmodell stellt und sich auch einfach auf dynamische Umgebungen einstellen lässt. Da der Algorithmus jedoch sehr stark streut und der größte Teil des Baumes in ein komplett falsche Richtung wächst, müsste intensiver versucht werden, das Wissen über die Umgebung noch stärker mit einzubinden, um eine noch ausgeprägtere Orientierung in Zielrichtung zu erlangen. Man könnte zum Beispiel versuchen bereits erfolgreiche Methoden mit zu integrieren. Eine Möglichkeit wäre das Aufstellen des Voronoi-Diagramms der bekannten Hindernisse, um diese einfacher umlaufen zu können, oder das Einbringen von Konzepten aus dem A*, um das Wissen über die Zielkonfiguration stärker mit einzubinden.
Abschließend ist zu sagen, dass der RRT eine sehr gute Grundlage für die Bewegungsplanung darstellt. Es ist jedoch wichtig, die richtigen Erweiterungen zu finden, um den Planer gezielt an spezielle Gegebenheiten anzupassen.

Literatur

1. Steven M. LaValle (2006): Planning algorithms. Cambridge University Press.
2. Karaman, S.; Walter, M.R.; Perez, A.; Frazzoli, E.; Teller, S. (2011): Anytime Motion Planning using the RRT*. Proceedings of the IEEE International Conference on Robotics and Automation (ICRA).
3. Jeon, Jeong hwan; Karaman, Sertac; Frazzoli, Emilio (2011): Anytime Computation of Time-Optimal Off-Road Vehicle Maneuvers using the RRT*. CDC-ECE.
4. Karaman, Sertac; Frazzoli, Emilio (2011): Sampling-based Algorithms for Optimal Motion Planning. I. J. Robotic Res.
5. Yoshiaki Kuwata; Justin Teo; Sertac Karaman; Gaston Fiore; Emilio Frazzoli; and Jonathan P. How (2007): Motion Planning in Complex Environments using Closed-loop Prediction.
6. Brandon D. Luders; Sertac Karaman; Emilio Frazzoli; Jonathan P. How (2010): Bounds on Tracking Error using Closed-Loop Rapidly-Exploring Random Trees.
7. Leonard, J.; Barrett, D.; Jones, T.; Matthew, A.; Galejs, R. (2008): A perception-driven autonomous urban vehicle. J. Field Robotics.
8. Karaman, Sertac; Frazzoli, Emilio (2010): Incremental Sampling-based Algorithms for Optimal Motion Planning. Robotics: Science and Systems.
9. Yoshiaki Kuwata; Justin Teo; Gaston Fiore; Emilio Frazzoli; and Jonathan P. How (2008): Motion Planning for Urban Driving using RRT. IROS.
10. Xiaoshan Pan (2003): Wheelchair Robotic Motion Planning. Civil and Environmental Engineering, Stanford University.
11. Baris Akgun; Mike Stilman (2011): Sampling Heuristics for Optimal Motion Planning in High Dimensions. IEEE/RSJ International Conference on Intelligent Robots and Systems.
12. Steven M. LaValle (2012): Motion Planning: The Essentials. Department of Computer Science, University of Illinois.
13. Anna Yershova; Leonard Jaillet; Thierry Simeon; Steven M. LaValle (2005): Dynamic-Domain RRTs: Efficient Exploration by Controlling the Sampling Domain. Department of Computer Science - University of Illinois, LAAS-CNRS.
14. James J. Kuffner Jr.; Steven M. Lavalle (2000): RRT-Connect: An Efficient Approach to Single-Query Path Planning. Proc. IEEE International Conference on Robotics and Automation
15. Yoshiaki Kuwata; Justin Teo; Sertac Karaman; Gaston Fiore; Emilio Frazzoli; and Jonathan P. How (2009): Real-time Motion Planning with Applications to Autonomous Urban Driving. IEEE transactions on control systems technology.
16. Karaman, Sertac; Frazzoli, Emilio (2010): Optimal Kinodynamic Motion Planning using Incremental Sampling-based Methods. Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge.
17. Steven M. LaValle (1998): Rapidly-Exploring Random Trees: A New Tool for Path Planning. Computer Science Dept Iowa State University. TR 98-11.