OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

Seminarausarbeitung des wissenschaftlichen Textes: Perception-Driven Sparse Graphs for Optimal Motion Planning

Timo Stadler
Matr.Nr: 3198649
(timo.stadler@st.oth-regensburg.de)

Wissenschaftliches Seminar

OTH Regensburg
Master Informatik

Wintersemester 2019/2020 (Version vom 23. Januar 2020)

Inhaltsverzeichnis

Zusammenfassung

Dieses Dokument fasst die Erkenntnisse aus dem Paper **Perception-Driven Sparse Graphs for Optimal Motion Planning** zusammen und stellt diese neue Methode zur Kartierung und Ermittlung eines kürzesten Pfades einem bereits etablierten Verfahren gegenüber. Zusätzlich werden die beiden Suchalgorithmen A* und D* Lite erläutert, um einen kurzen Pfad durch den erzeugten Graphen zu ermitteln. Außerdem enthält diese Ausarbeitung grundlegende Erklärungen von Begriffen aus dem Bereich der Robotik, sodass die Anwendung des Algorithmus auf eine reale Problemstellung nachvollziehbar wird.

1 Einleitung

Denkt man an die Robotik schweben einem zunächst Bilder von Hallen, die mit Laufbändern und Industrierobotern gefüllt sind, vor. Das Wort Robotik an sich deckt aber einen viel größeren Bereich ab. Gerade durch die Fortschritte in den Bereichen Computervision, Maschinelles Lernen und die Möglichkeit immer kleinere, effizientere Sensoren und Rechner herzustellen, verschob sich das Anwendungsgebiet von den Industrierobotern immer mehr hin zu Robotern als Dienstleister für Privatpersonen. Beispiele hierfür sind Saugroboter zur Reinigung der Wohnung, autonom fahrende Fahrzeuge, bis hin zu sogenannten Unmanned Aerial Vehicles (UAVs), die als 'Flugtaxis' dienen sollen [1].
Während personenbefördernde Fahrzeuge genügend Fläche besitzen, um entsprechend viele Kameras, Radarsysteme und weitere Sensoren tragen zu können, ist dies bei kleinen Drohnen oder minimal ausgestatteten Robotern nicht möglich. Außerdem können bei solchen Maschinen aufgrund des knappen Kostenrahmens nicht beliebig viele Sensoren verbaut werden. Jüngste Arbeiten untersuchen daher die Kopplung des Kartierungs- und des Planungsprozesses. Bei sehr großen oder komplexen Räumen muss der Roboter viele Gebiete wahrnehmen und in der Planung verarbeiten, obwohl er sie nie betreten wird. Daher entstand die Idee die Wahrnehmung von Hindernissen in die Planung zu integrieren, um so die Zeit bis zum Erreichen des Ziels zu verringern [2].
Die hier vorgestellte Methode der „Wahrnehmungsgetriebenen dünnbesetzten Graphen" liefert einen Algorithmus, der sowohl in einem zweidimensionalem, als auch in einem dreidimensionalen Raum selbstständig eine Karte der Umgebung erstellt und zusätzlich den schnellsten Weg durch diese mit Hindernissen versehene Karte findet. In dieser Arbeit werden zunächst Standardverfahren erläutert, die zur Wegfindung oder der Erzeugung einer Karte eingesetzt werden. Im Anschluss wird ein neuer Ansatz vorgestellt, der es erlaubt mit möglichst wenigen Informationen eine Karte der Hindernisse zu erzeugen und in dieser den schnellsten Weg zum Ziel zu finden. Es folgt die Beschreibung des Testaufbaus, der diesen neuen Algorithmus mit der bereits etablierten Methode der Erzeugung eines Gittergraphen vergleicht. Zuletzt wird auf die Stärken und Schwächen, sowie mögliche Anwendungsbereiche des Algorithmus eingegangen.

2 Stand der Technik

Durch den 'Sparse-Plan' Algorithmus werden zwei Prozesse miteinander gekoppelt, die normalerweise unabhängig voneinander ablaufen. Der Kartierungsprozess ist hierbei für das Erfassen von Hindernissen und Eintragen in eine vom Roboter verwaltete Karte zuständig. Der Prozess der Planung wird durch einen separaten Suchalgorithmus umgesetzt und ermittelt einen kürzesten Pfad durch die mit Hindernissen versehene Karte. Das folgende Kapitel stellt die Funktionsweise des A*-Algorithmus, als Beispiel eines informierten Suchalgorithmus dar. Die Idee von A* wird im nächsten Schritt an einem

Beispiel in die Simulation einer realen Umgebung übertragen. Es folgt eine Abwandlung von A* hin zum 'D* Lite' Algorithmus, der sich auch auf wandelbare Umgebungen anwenden lässt.

2.1 Durchsuchen eines bereits existierenden Graphen mit dem A*-Algorithmus

A* ist ein Graph-basierter Suchalgorithmus, der zur Berechnung des kürzesten Pfades zwischen zwei Knoten verwendet wird. A* zählt hierbei zur Klasse der einfachen Suchalgorithmen, da er mit nur wenig Code-Zeilen umgesetzt werden kann und der Algorithmus somit auch einfach zu verstehen ist. A* erweitert den Dijkstra-Algorithmus (vgl. [3]) um die Verwendung einer Heuristik, also einer ungefähren Schätzung des Endergebnisses. Diese ermöglicht es, einen Graphen zielgerichtet zu durchsuchen und somit eine kürzere Laufzeit als beim Dijkstra-Algorithmus zu erreichen. Die verwendete Schätzfunktion kann frei ausgewählt werden. Meistens wird aber der optimistisch schätzende Luftlinienabstand zum Zielknoten als Schätzung verwendet. Die Grundidee von A* besteht darin einen Graphen aufzubauen, dessen Knoten genau einen der drei folgenden Zustände haben können: Der Knoten ist unbekannt, der Knoten befindet sich in der Warteschlange oder der Knoten ist fertig abgearbeitet.
Der Algorithmus beginnt mit dem Einfügen des Startknotens in die Warteschlange. Im nächsten Schritt wird der Knoten mit dem geringsten $f(x)$ zur Warteschlange hinzugefügt, wobei gilt: $f(x) = g(x) + h(x)$. Hierbei bezeichnet für einen Knoten x, $g(x)$ die aufsummierten Kosten, die notwendig sind, um x vom Startknoten aus zu erreichen. $h(x)$ bezeichnet die geschätzten Kosten bis zum Zielknoten vom Knoten x aus. Es wird der erste Knoten aus der Warteschlange entnommen und alle seine Nachbarknoten werden jetzt betrachtet. Wie schon erwähnt, weißt jeder Knoten einen Status auf. Ist der Nachbarsknoten bereits fertig abgearbeitet, passiert mit diesem nichts. Befindet sich der Knoten in der Warteschlange, werden die Kosten beider Wege miteinander Verglichen und nur der 'günstigere Weg' für den Knoten verbleibt in der Warteschlange. Ist der Knoten noch nicht in der Warteschlange, wird der Knoten anhand seines Wertes von $f(x)$ in die Warteschlange eingefügt. Der Algorithmus endet, wenn der Zielknoten aus der Warteschlange entnommen wird oder die Warteschlange komplett leer ist. Sollte der zweite Fall eintreten, so konnte kein Weg vom Start- zum Zielknoten gefunden werden.

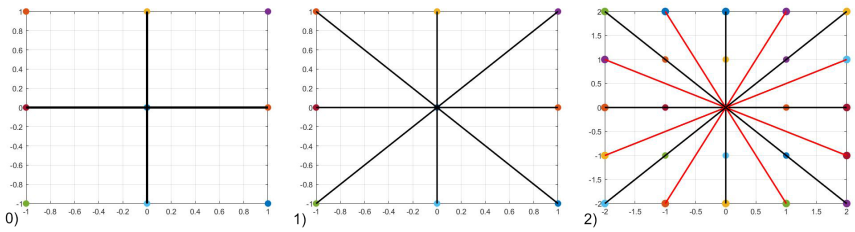


Abbildung 1: Mögliche Verbindungen bei den Werten 0, 1 und 2 für den Konnektivitätsparameter

Der A*-Algorithmus kann auch in der Robotik zur automatischen Wegfindung eingesetzt werden. Hierfür sind nur wenige zusätzliche Bedingungen notwendig. In einem zweidimensionalem (2D) Raum wird eine quadratische Fläche mit fester Kantenlänge als Knoten behandelt. Je nachdem, wie die Kantenlänge gewählt wird, kann eine höhere räumliche Auflösung bei gleichzeitig steigender Rechenlaufzeit für den Algorithmus erreicht werden. Zusätzlich wird die Verwendung eines Konnektivitätsparameters eingeführt. Dieser Parameter gibt an, mit wie vielen Nachbarn im Gitter ein Knoten durch Kanten verbunden wird. Abbildung 1 zeigt die möglichen Verbindungen für die Werte 0, 1 und 2 des Konnektivitätsparameters. Die Kantengewichte sind in dem Anwendungsfall der Robotik für alle Knoten einer Konnektivitätsordnung gleich groß. So kann

z.B. für alle Verbindungen von nicht-diagonalen Knoten ein Gewicht von 1 gelten und für alle Verbindungen zu direkt diagonalen Nachbarknoten ein Kantengewicht von 1,4. Hindernisse und Wände werden als Knoten dargestellt, deren Kanten ein Gewicht von ∞ aufweisen. Ein mit diesen Eigenschaften erstellter Graph wird auch als Karte bezeichnet.

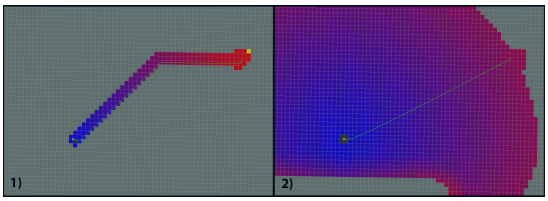


Abbildung 2: Gegenüberstellung des A*(1)- und des Dijkstra(2)-Algorithmus im Robotic Operating System (ROS). Die grauen Felder stehen hierbei für nicht behandelte Knoten und die farbigen Felder für Punkte, die in die Warteschlange gelangten und untersucht wurden.

Zur Simulation von Robotern wird das ROS verwendet. Hierbei handelt es sich um ein Framework zur Entwicklung von Robotik-Anwendungen. Abbildung 2 stellt die Planung des Fahrtweges eines Roboters im ROS durch den A*-Algorithmus und den Dijkstra-Algorithmus dar. Um Missverständnisse auszuschließen, muss angemerkt werden, dass die Implementierung der Karte im ROS alle Koordinaten um 0,5 Gitterzellen in X und Y-Richtung verschiebt. Außerdem stimmt der Anfang des berechneten Pfades nicht mit dem eigentlichem Startpunkt überein. Abbildung 2 zeigt, dass bei A* viel weniger Punkte betrachtet werden, als bei der Verwendung des Dijkstra-Algorithmus und sich daraus eine kürzere Rechenzeit ergibt [4]. In diesem Fall ist der ermittelte Pfad bei A* länger, da nur ein Konnektivitätsparameter von $n = 1$ verwendet wurde. Ein höherer Konnektivitätsparameter könnte einen noch kürzeren Pfad erzeugen.

2.2 Anpassungen des A*-Algorithmus zu D* Lite

Der A*-Algorithmus ist allerdings unflexibel und kann nicht eingesetzt werden, wenn im Graph Veränderungen, wie etwa neue Hindernisse, auftreten oder der Graph expandiert wird. Der erstellte Pfad müsste hierbei komplett verworfen werden und der Algorithmus müsste auf dem neu entstandenen Graphen angewandt werden. Da das Ziel des vorgestellten Papers in der Ermittlung eines kürzesten Weges mit relativ wenig Rechenaufwand, auf nicht a priori determinierten Karten liegt, kommt A* als Suchalgorithmus also nicht in Frage. Stattdessen wird von den Erstellern der vorgestellten Arbeit vorgeschlagen, den D*- oder D* Lite-Algorithmus auf die simultan generierte Karte anzuwenden. Bei D* handelt es sich lediglich um eine Modifikation von A*, um durch sich dynamisch verändernde Karten navigieren zu können [5]. An dieser Stelle wird statt dem D*-Algorithmus D* Lite beschrieben, da dessen Idee einfacher verständlich ist und daher auch in weniger Zeilen Code umgesetzt werden kann. D* Lite ist eine Erweiterung des 'Lifelong Planning A*'-Algorithmus, berechnet als Ergebnis aber den gleichen Pfad wie D* [6]. Umgekehrt zu A* wird bei D* Lite der Weg vom Zielknoten aus zum Startknoten ermittelt, wodurch die exakten Kosten zum Zielknoten jedem Knoten des Graphen zur Verfügung stehen. Sobald der Graph analog zu A* vollständig expandiert ist, kann der Roboter den Graph durchschreiten, sodass die Kosten der restlichen Strecke reduziert werden, bis der Zielknoten erreicht ist. Sollte während des Durchlaufens des Algorithmus eine Kollision mit einem neuen, zuvor nicht bekannten, Hindernis auftreten, können zur Neuberechnung die beiden heuristischen Parameter $g(x)$, das Gewicht aller Kanten von diesem Knoten bis zum Zielknoten und $rhs(x)$, eine 1-Schritt-Vorschau des Gewichts aller Kanten von diesem Knoten bis zum Zielknoten, welche für

jeden Knoten gespeichert wurden, verwendet werden. Hierbei ist zu beachten, dass der $rhs(x)$-Wert aus den $g(x)$ Werten aller Vorgängerknoten aufsummiert wird. Zusätzlich wird der Begriff der Konsistenz eingeführt. Falls für einen Knoten x gilt: $g(x) \neq rhs(x)$ ist dieser inkonsistent und wird zur Bearbeitung in die bereits aus A* bekannte Warteschlange gesetzt. Die inkonsistenten Knoten werden in der Queue nach aufsteigendem $rhs(x)$-Wert sortiert. Zunächst wird der rhs Wert des Hindernisknotens auf ∞ gesetzt. Zusätzlich werden die $rhs(x)$ Werte aller durch eingehende und ausgehenden Kanten mit dem neuen Hindernisknoten verbundenen Knoten aktualisiert und auf inkonsistenz überprüft. Hierbei handelt es sich um einen rekursiven Prozess, der alle Knoten betrifft, die durch das hinzufügen eines Nachbarknotens zur Queue inkonsistent wurden. Sobald alle Inkonsistenzen aufgelöst sind kann wieder der kürzeste Pfad berechnet werden. Mit diesen neuen Definitionen kann nun eine neue Wegfindung stattfinden, falls ein neues Hindernis auftritt. Abbildung 3 zeigt die wichtigsten Schritte bei der Anwendung des D* Lite Algorithmus an einem Beispiel.

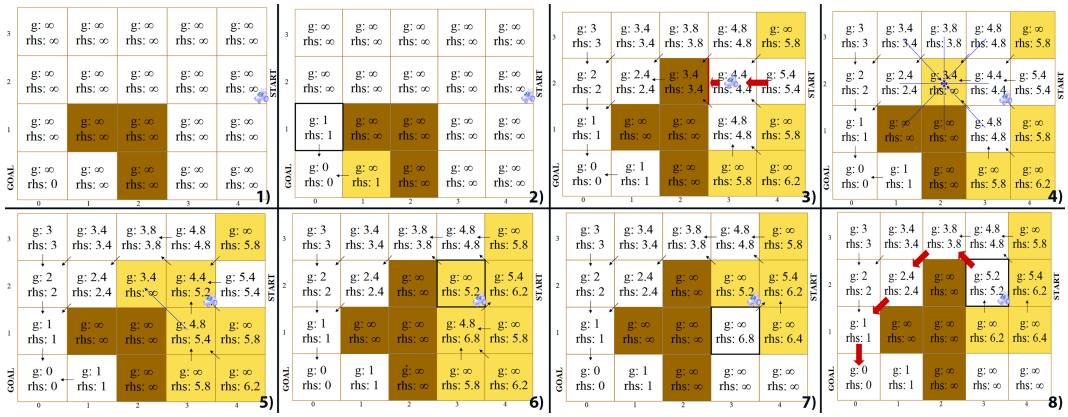


Abbildung 3: Wichtige Schritte bei der Durchführung des D* Lite Algoritmus: Braune Felder stellen Hindernisse dar, Gelbe Felder sind gerade in der Queue eingetragen.
(1) Ausgangssituation, (2) $g(x)$ und $rhs(x)$ Werte schrittweise updaten, (3) Fahren des kürzesten Weges, (4) Feststellung eines neuen Hindernisses, (5) Updaten der Nachbarknoten, (6) Nachbarsknoten von (3,2) werden aktualisiert, (7) Inkonsistenter Knoten (3,1) wird in die Queue eingefügt, (8) Fahren des neuen kürzesten Weges [7].

2.3 Dynamische Robotik Systeme

Dynamische Robotik Systeme dienen in einer Simulation zur Modellierung von verschiedenen real existierenden Robotern. Die Art des Roboters nimmt hierbei erheblichen Einfluss auf die Bewegungsregelung. Der 'Sparse-Graph' Algorithmus wurde mit holonomen 2D und dreidimensionalen (3D) Robotern, sowie einem 'Dubins Car' getestet. Bei einem holonomen Roboter handelt es sich um ein System, das ohne Rotationsbewegung jeden Punkt im 2D oder 3D Raum erreichen kann [8]. Die Freiheitsgrade des Roboters sind also komplett unabhängig voneinander und eine Bewegung kann als Vektor umgesetzt werden. Beim 'Dubins Car' handelt es sich dagegen um einen nicht-holonomen Roboter, der als vereinfachtes Modell für Kraftfahrzeuge eingesetzt wird. Das 'Dubins Car' kann sich, aufgrund des verbauten Differentialantriebs nur in drei Richtungen bewegen: Vorwärts, Linkskurve und Rechtskurve. Der minimale Wenderadius des Fahrzeugs wird als Parameter festgelegt und eine Drehung auf der Stelle ist in diesem Modell nicht möglich. Um den Rückwärtsgang eines Fahrzeuges in der Simulation umzusetzen wird oft noch die vierte Bewegungsrichtung 'Rückwärts' eingeführt [9]. Mit diesem Modell kann der kürzeste Pfad mit relativ einfacher Geometrie berechnet werden, während in

anderen dynamischen Systemen komplexe Matrixoperationen notwendig sind [10].

3 Problemdefinition

Es sei $x(t) \in \mathcal{X}$ der Knoten an dem sich der Roboter befindet und $u(t) \in U$ sei die, durch den Algorithmus ermittelte, Eingabe zum Zeitpunkt t. Hieraus ergibt sich die Dynamik des Roboters als folgende Differentialgleichung mit ausschließlicher Ableitung nach der Zeit: $\dot{x}(t) = f(x(t), u(t))$
Eine *dynamisch realisierbare Bahn* von einem Startpunkt $x_a \in \mathcal{X}$ zu einem Endpunkt $x_b \in \mathcal{X}$ ist eine Abbildung von $\tau : [0, T) \to \mathcal{X}$, sodass $\tau(0) = x_a, \tau(T) = x_b$ gilt. Die Kostenfunktion, welche jeder Kante einen Kostenwert zuweist, sei definiert durch $\mathcal{C}(\mathcal{S}) = min_{\tau \in \mathcal{S}} \mathcal{C}(\tau)$. Außerdem wird eine Karte mit Hindernissen durch $\mathcal{M} \subset \mathcal{X}$definiert [11]. Bei Hindernissen handelt es sich, wie bereits beschrieben, um Knoten die von dem Roboter nicht erreicht werden können. Folglich wird jeder Kante, die durch einen solchen Knoten läuft ein unendlich hoher Kostenwert zugeordnet. Durch den 'Sparse Graph' Algorithmus soll das Problem der optimalen Bewegungsplanung gelöst werden. Es soll also unter Berücksichtigung der Dynamik des Roboters, eine Bahn mit minimalen Kosten vom Startknoten x_s zum Zielknoten x_g gefunden werden. Eine Bahn ist definiert als die Verkettung mehrerer Kanten. Weißt das Ergebnis endliche Kosten auf, existiert ein möglicher Pfad und dieser ist kollisionsfrei. Hat der Pfad unendlich hohe Kosten gibt es keinen möglichen Pfad.
Der hier entwickelte Algorithmus soll eine Karte und gleichzeitig dazu den Plangraphen erstellen, um das definierte Problem der optimalen Bewegungsplanung zu lösen. Alle Hindernisse in der höchstmöglichen Auflösung zu erkennen, würde zu einer umfangreichen Berechnungszeit allein für die Phase der Kartierung führen. Daher liegt das Ziel des Algorithmus darin, den optimalen Pfad zu berechnen während nur Hindernisse zur Karte hinzugefügt werden, wenn es notwendig ist. Dies wird umgesetzt indem eine Sequenz von Karten generiert wird und die Bewegungsplanung in allen nachfolgenden Karten betrachtet wird. M_k sei die Sammlung aller Hindernisse in der $k$ten Planungsphase. Das Ziel des Algorithmus lässt sich in 3 Unterziele aufteilen:

1) Erstellung eines effizienten Plangraphens.

2) Sicherstellung der Vollständigkeit und Optimalität der erhaltenen Lösung auf der Grundlage aller verfügbaren Sensor-Daten.

3) Minimierung der Menge an Sensordaten, die verarbeitet werden müssen, um 2.) erreichen zu können.

Zu diesem Zweck wurde ein gekoppelter Kartierungs- und Planungsalgorithmus erstellt. Dieser erzeugt stufenweise die Karte, sowie den Plangraphen, indem er Hindernisse zur Karte und Kantengewichte zum Plangraphen nach Bedarf hinzufügt.

4 Vorstellung eines Algorithmus zur Erzeugung von wahrnehmungsgetriebenen dünnbesetzten Graphen

Dieses Kapitel beschreibt den neu entwickelten Konstruktions-Algorithmus für die eng gekoppelte Erzeugung der Karte und des dazugehörigen Plangraphens. Die Kollisionserkennung erfolgt hierbei nach dem 'Lazy'-Prinzip. Das bedeutet sie wird nur ausgeführt, wenn ein Zusammenstoß mit dem Hindernis unmittelbar bevorsteht. Es werden nur Objekte in die Karte eingetragen, die den geplanten Weg schneiden und der Plangraph wird lokal aktualisiert. Im Weiteren wird die Kantenverkettung dargestellt, durch die eine optimale Lösung aus den optimalen Bahnen im freien Raum erzeugt werden kann. Im An-

schluss folgt die Beschreibung des Algorithmus zur Konstruktion einer Karte, gekoppelt mit der Erzeugung des Plangraphen.

4.1 Kantenverkettung

Bevor der Algorithmus beschrieben wird müssen noch einige Eigenschaften für die Verbindung von Kanten erläutert werden. Geht man von zwei Bahnen τ_1 und τ_2 aus, so ist die Verkettung dieser beiden definiert als:

$$(\tau_1 + \tau_2)(t) = \begin{cases} \tau_1(t) & \text{für alle } t \in [0, T_1) \\ \tau_2(t - T_1) & \text{für alle } t \in [T_1, T_1 + T_2). \end{cases}$$

Zwei Bahnen können somit nur verbunden werden, wenn gilt: $\tau_1(T_1) = \tau_2(0)$, also der Endpunkt einer Bahn dem Anfangspunkt der zu verbindenden Bahn entspricht. Zudem können zwei Mengen von Bahnen unter Beachtung dieser Regel miteinander konkateniert werden. Hierbei muss der Endpunkt der ersten Verkettung dem Startpunkt der zweiten Verkettung entsprechen oder umgekehrt.

4.2 Lösungskandidaten

Über das Problem des kürzesten Weges werden zwei Annahmen getroffen:

Annahme 1: Die Kostenfunktion $C(\cdot)$ erfüllt die Dreiecksungleichung für alle $t \in \mathcal{S}$

Annahme 2: Es existiert ein Planer, der eine Menge an lokalen Minima-Bahnen vom Punkt x_1 zum Punkt x_2 durch den freien Raum generieren kann.

Diese beiden Annahmen sind bei der Verwendung der vorgestellten dynamischen Systeme holonome Roboter und 'Dubins Car' erfüllt. Aufgrund dieser beiden Annahmen lassen sich Aussagen über die Struktur einer optimalen Lösung des Problems der Bewegungsplanung treffen. Die getroffene Aussage lautet folgendermaßen:
„Angenommen Annahme 1 und 2 sind wahr, so lässt sich ein optimaler Pfad von Startpunkt x_s zum Zielpunkt x_g durch eine endliche Anzahl an Kurven und die Verbindungen dieser an den Hindernissen finden."
Basierend auf dieser Aussage kann ein Graph $G_{complete}$ erzeugt werden, der garantiert jede optimale Lösung enthält. Um den Rechenaufwand zu verringern wird $G_{complete}$ durch ein festes Intervall δ zu $\tilde{G}_{complete}$ diskretisiert. Die Diskretisierung bewirkt, dass es nur endlich viele Kantenpunkte an einem Hindernis geben kann [11]. Für diesen Graphen soll nun durch den vorgestellten Algorithmus eine Lösung des Problems ermittelt werden.

4.3 Erzeugung eines dünnbesetzten Graphen

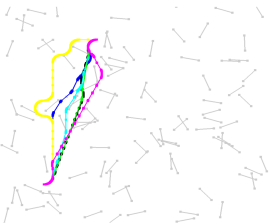
Obwohl $\tilde{G}_{complete}$ endlich ist, ist der Graph immer noch unverhältnismäßig groß, um effektiv Durchsucht zu werden. Stattdessen erzeugt der vorgestellte Algorithmus einen steuerbaren Teilgraphen G_{sparse}, der einen optimalen Bewegungsplan enthält, falls ein solcher existiert. Dieser spärlich besetzte Teilgraph wird generiert, indem nur Objekte zur Karte hinzugefügt werden, falls dies nötig ist. Das Ziel des Algorithmus ist also, nach dem 'Divide-and Conquer'-Prinzip, das Problem des minimalsten Pfades zwischen Ziel- und Anfangsknoten in mehrere Unterprobleme ${}_a\mathcal{P}_b$ aufzuteilen, um den kürzesten Pfad zwischen den Punkten x_a und x_b zu finden [11]. Aus diesen Unterproblemen formt der Algorithmus wieder eine gesamte Lösung, indem die Start und Endpunkte von Lösungen der Unterprobleme, wie im Kapitel Kantenverkettung beschrieben, sequenziell miteinander verbunden werden. Hierbei pflegt jedes Unterproblem eine Karte mit allen Hindernissen, die nach dem 'Lazy'-Prinzip erfasst wurden und eine Liste von sog. 'Parents'. Bei den 'Parents' handelt es sich um andere Unterprobleme, die das berechnete

```
Result: τ* = arg min_{τ∈S(x_s,x_g)} C(τ)
1  Algorithm SparseShortestPath(x_s, x_g)
       /* Add the start and goal                                  */
2      addNodes(G_sparse, {x_s, x_g})
3      addProblem(x_s, x_g)
4      M_added = ∅
5      while true do
           /* Solve with any optimal graph search
              algorithm                                           */
6          τ_0 + ··· + τ_n = Solve(G_sparse)
           /* Check the solution                                  */
7          for τ_i ∈ τ do
8              for M_j ∈ M do
9                  if blocked(τ_i, M_j) then
10                     C(τ_i) = ∞
11                     if M_j ∉ M_added then
12                         addNodes(G_sparse, B̃(M_j))
13                     x_a = τ_i(0), x_b = τ_i(T_i)
14                     addObstacle(aP_b, M_j)
15                     Go to line 6
           /* Solution is unblocked, then it is
              optimal                                             */
16         return τ_0 + ··· + τ_n

17 Procedure addObstacle(aP_b, M_j)
18     if M_j ∉ aP_b.map then
19         aP_b.map += M_j
20         for x_k ∈ B̃(M_j) do
21             addProblem(x_a, x_k)
22             aP_k.parents += aP_b
23             aP_b.children += aP_k
24             addProblem(x_k, x_b)
25             kP_b.parents += aP_b
26             aP_b.children += kP_b
               /* Recursively add to parents of the
                  path                                            */
27             for cP_d ∈ aP_b ∪ aP_b.parents do
28                 for M_l ∈ M_j ∪ aP_k.map ∪ kP_b.map do
29                     addObstacle(aP_b, M_l)

30 Procedure addProblem(x_a, x_b)
31     if aP_b does not exist then
32         aP_b.parents = ∅
33         aP_b.children = ∅
34         aP_b.map = ∅
35         addEdges(G_sparse, S_∅(x_a, x_b))|
```

Abbildung 4: Pseudocode des vorgestellten Algorithmus



Color	Planner	Grid Discr.	Angular Discr.	Conn.	Cost	Time (ms)
Green	Sparse	–	$\pi/8$	–	**20.71**	**16.9**
Black	Grid	0.25	$\pi/8$	4	20.86	751.3
Cyan	Grid	0.5	$\pi/8$	4	20.88	486.7
Blue	Grid	1.0	$\pi/4$	2	21.43	49.1
Purple	Grid	0.5	$\pi/8$	2	21.86	110.9
Yellow	Grid	1.0	$\pi/2$	0	25.56	25.6

Abbildung 5: Von einem Dubins Car gefahrene Wege für sechs Plangraphen

Ergebnis zur Lösung ihres eigenen Problems verwenden können, also solche Probleme deren Start- oder Endpunkt mit x_a oder x_b übereinstimmt. Die Kartierung, umgesetzt durch Kollissionserkennungen, führt hierbei zur Generierung neuer Unterprobleme und zum Hinzufügen von Hindernissen zu diesen neu erstellten Sub-Karten. Während jeder Iteration berechnet jedes Unterproblem einen minimalen Kostenwert für das Teilergebnis, welcher in jedem Fall niedriger oder gleich den Kosten des Subproblems auf der kompletten Karte ist. Mit jeder Iteration werden die Sub-Karten, basierend auf der besten Lösung der letzten Iteration, miteinander verbunden, bis eine echte optimale Lösung gefunden wurde. Der komplette Algorithmus ist als Pseudocode in Abbildung 4 dargestellt. An dieser Stelle sei erwähnt, dass es sich bei dem vorgestellten Algorithmus lediglich um einen Graph-Konstruktionsalgorithmus handelt. Die gleichzeitige Erstellung eines Plangraphens ist in Abbildung 4 durch die Funktion $Solve(G_{sparse})$ dargestellt. Diese Funktion kann durch einen beliebigen Wegfindungsalgorithmus für Graphen, wie z.B. D* [6] oder D* Lite, ersetzt werden und gibt einen kürzesten Weg zum Ziel als Output zurück.

5 Computergestützte Versuche

Um den vorgestellten Algorithmus zu Validieren und seine Effizienz zu testen wird dieser einem etablierten Gitter-basierten Verfahren zur Erzeugung eines Graphen gegenübergestellt. Beide erzeugten Graphen werden mit D* Lite als Suchmethode und mit 'lazy-checking' als Kollisionserkennung durchsucht. Getestet wurden die Algorithmen mit holonomen Robotern im 2D- und 3D Raum, sowie mit 'Dubins Car'.

5.1 Versuchsaufbau

Umgesetzt wurde der im Paper vorgestellte Vergleich als computergestützte Simulation. Der generierte Testraum beträgt eine Größe von $30x30x30$ Einheiten und der Startpunkt wird an die Stelle $x_s = (5,5,5)$ gesetzt. Der Zielpunkt wird in eine zufällige Richtung 20 Einheiten vom Startpunkt in den positiven Quadranten gesetzt [11]. Für Tests im 2D-Raum entfällt die dritte Dimension. In diesen generierten Raum werden zusätzlich Hindernisse platziert. Im 2D-Raum handelt es sich hierbei um Linien mit einer festen Länge, deren Start und Endpunkte zufällig verteilt sind, und im 3D-Raum um zufällig verteilte Würfel mit einer festen Seitenlänge. Der gitterbasierte Algorithmus, mit dem der vorgestellte Algorithmus verglichen wird, setzt im erzeugten Raum Knotenpunkte

mit einem festen räumlichen Abstand zueinander. Die Kanten zwischen den Punkten dieses erzeugten Gitters werden anhand des bereits im Kapitel 2.1 beschriebenen Konnektivitätsparameters gesetzt.

5.2 Versuchsdurchführung

Die Überprüfung auf Kollisionen wird im 2D-Raum als Erkennung von Schnittpunkten zweier Linien umgesetzt und im 3D-Raum mithilfe von 'Raycasting' in der Octomap-Library[12]. Jeder Kollisionserkenner agiert als simulierter Sensor, der entlang der gefahrenen Bahn, freien oder durch Hindernisse blockierten Raum kartiert. Abbildung 5 zeigt die von sechs unterschiedlichen Plangraphen erzeugten Wege, die von einem 'Dubins Car' auf einer einzelnen KArte gefahren werden.

5.3 Vergleich der Versuchsergebnisse

Für jedes der drei dynamischen Systeme wurden 200 zufällig generierte Karten mit Hindernissen erzeugt, auf denen der neu vorgestellte Algorithmus, sowie der gitterbasierte Algorithmus mit verschiedenen Parametern für Konnektivität und räumliche Auflösung, ausgeführt wurden [11]. Die Ergebnisse der drei Systeme wurden für die verschiedenen Konstruktionsalgorithmen auf die Länge des erzeugten Pfades im Verhältnis zur zeitlichen Dauer für die Generierung des Pfades verglichen. Diese Ergebnisse legen dar, dass der 'Sparse Algorithmus' Lösungen erzeugt, die schneller berechnet werden können und gleichzeitig niedrigere Kosten aufweisen.
Für das Experiment mit dem 'Dubins Car' wurde zusätzlich die Tabelle 1 erzeugt, welche die Mittelwerte aller Einzelversuche für Pfad-Kosten, Planungszeit (in ms), Anzahl generierter Knoten, Anzahl generierter Kanten und den durch Sensoren erfassten Raum festhält. Diese Tabelle zeigt, dass der neue Sparse-Planning Algorithmus mit einer Winkelauflösung von $pi/16$ die geringsten Pfadkosten, bei entsprechend niedriger Planungszeit aufweist und hierbei nur wenige Knoten und Kanten berechnet. Der gitterbasierte Algorithmus weist für alle Parameter höhere Pfadkosten auf, ab einer Gitter-Diskretisierung von 0,5 ist seine Planungszeit aber deutlich geringer, als die des Sparse-Planners. Bei der Wahl des einzusetzenden Algorithmus ist also zu beachten, dass stets ein Kompromiss zwischen Pfadkosten und Planungszeit geschlossen werden muss.

Tabelle 1: Auswertung der Experimente für 'Dubins Car' [11]

Planner	Grid Discretization	Angular Discretization	Connectivity	Path Cost	Plan Time (ms)	Nodes	Edges	Area Sensed
Sparse	–	$\pi/16$	–	22.315	1645	282	4838	436
Sparse	–	$\pi/8$	–	22.328	140	140	1369	418
Sparse	–	$\pi/4$	–	22.357	20	70	383	382
Grid	0.25	$\pi/16$	4	22.400	35871	49318	1101100	528
Grid	0.25	$\pi/8$	4	22.424	2541	25276	278740	513
Grid	0.25	$\pi/4$	4	22.635	258	13704	77655	522
Grid	0.50	$\pi/16$	2	22.710	715	13069	113290	544
Grid	0.50	$\pi/8$	2	22.783	78	6677	27560	546
Grid	0.50	$\pi/4$	2	23.090	29	3954	11733	449
Grid	1.00	$\pi/16$	1	24.214	55	4266	19424	417
Grid	1.00	$\pi/8$	1	25.280	15	2287	5897	403
Grid	1.00	$\pi/4$	1	25.837	7	887	1780	282

6 Zusammenfassung und Fazit

In dem zusammengefassten Paper wurde ein neuer Planalgorithmus vorgestellt, der die beiden Probleme der Kartierung und der Planung enger miteinander verbindet. Dieser Algorithmus erlaubt es also auch Robotern, die nur mit wenigen Sensoren ausgestattet sind und eine geringe Rechenleistung aufweisen, einen kürzesten Weg durch einen

Raum mit Hindernissen zu finden. Zunächst erscheint der Algorithmus durch die notwendigen mathematischen Formulierungen als komplex. Eine klarere Erklärung durch eine bildliche Darstellung der einzelnen Schritte trüge dem besseren Verständnis bei. Ist die Grundidee des Algorithmus, Kanten an Kollisionspunkten von Objekten miteinander zu verbinden, verstanden, ist der Algorithmus aber geradezu trivial. Der Einsatz dieses Algorithmus in personenbefördernden Fahrzeugen ist als kritisch anzusehen. Hier wünscht man sich ein System, das vorausdenkt und Kollisionen möglichst vermeidet. Das 'Lazy'-Checking würde dazu führen, dass das Fahrzeug erst kurz vor einer Wand zum Stehen kommt. Für günstige robuste Roboter, die als Rettungsroboter [13] eingesetzt werden ist dieser Algorithmus aufgrund der geringen Rechenleistung, die hierfür benötigt wird optimal.

Abkürzungsverzeichnis

ROS Robotic Operating System

2D zweidimensional

3D dreidimensional

UAVs Unmanned Aerial Vehicles

Literaturverzeichnis

[1] T. Hutchings, S. Jeffryes, and S. Farmer, "Architecting uav sense & avoid systems," in *2007 Institution of Engineering and Technology Conference on Autonomous Systems*, pp. 1 – 8, 12 2007.

[2] W. Pryor, Y.-C. Lin, and D. Berenson, "Integrated affordance detection and humanoid locomotion planning," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, IEEE, nov 2016.

[3] T. H. Cormen *et al.*, *Algorithmen - Eine Einführung*. Oldenbourg Wissenschaftsverlag; Auflage: überarbeitete und aktualisierte Auflage (18. August 2010), 2010.

[4] M. F. David V. Lu, "ROS-Wiki: Global Planner." http://wiki.ros.org/global_planner, July 2019.

[5] A. Stentz, "The focussed d* algorithm for real-time replanning," in *In Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1652–1659, 1995.

[6] S. K. M. Likhachev, "D*Lite," *Eighteenth national conference on Artificial intelligence*, 2002.

[7] H. Choset, *Principles of Robot Motion - Theory, Algorithms and Implementation (OIP)*. MIT Press, 2005.

[8] M. Oubbati, "Einführung in die Robotik." https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.130/Mitarbeiter/oubbati/RobotikWS1113/OubbatiSkript.pdf, Oct. 2009.

[9] C. Siedentop, R. Heinze, D. Kasper, G. Breuel, and C. Stachniss, "Path-Planning for Autonomous Parking with Dubins Curves," 01 2015.

[10] J. A. Reeds and L. A. Shepp, "Optimal paths for a car that goes both forwards and backwards.," *Pacific Journal of Mathematics Vol. 145*, 1990.

[11] T. Sayre-McCord and S. Karaman, "Perception-driven sparse graphs for optimal motion planning," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018.

[12] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: an efficient probabilistic 3d mapping framework based on octrees," *Autonomous Robots*, vol. 34, pp. 189–206, feb 2013.

[13] A. Davids, "Urban search and rescue robots: from tragedy to technology," *IEEE Intelligent Systems*, vol. 17, pp. 81–83, mar 2002.

Tabellenverzeichnis

Abbildungsverzeichnis