Algorithmen zur Pfadplanung

Martin Helmreich Sebastian Prokop Chris Schwemmer

Seminar im Grundstudium: Umgebungsexploration und Wegeplanung mit Robotern am Beispiel Katz und Maus

Informatik LS12, Universität Erlangen

21.06.2005

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit stellt die schriftliche Ausarbeitung eines Vortrages über *„Algorithmen zur Pfadplanung“* dar, der im Rahmen des Proseminars „*Umgebungsexploration und Wegeplanung mit Robotern am Beispiel Katz und Maus*“ gehalten wurde. Als Quellen liegen in erster Linie die Arbeiten [lavalle] und [rao] zugrunde.

Um über die Probleme und möglichen Lösungen bei der Navigation von Robotern zu diskutieren, ist es sinnvoll zuerst einige Begriffe einzuführen. Das allgemeine Problem, „einen Roboter zu einem Ziel zu bewegen und dabei Hindernissen auszuweichen, welche vorher nicht zwingend bekannt sind und über Sensoren erfasst werden“ bezeichnet man als das Navigations- oder Suchproblem.

Man unterscheidet hierbei zwischen unbekanntem und bekanntem Terrain. Ist das Gebiet vollständig bekannt, ist das Suchproblem identisch mit der Suche in einem Graphen, also geometrisch lösbar. Ist das Gelände gar nicht oder nur unvollständig bekannt, so basieren alle Berechnungen auf lokalen Teilinformationen und erfolgen inkrementell wenn neue Informationen hinzukommen. Diese Informationen können nur von Sensoren kommen, weshalb man hier von sogenannten On-Line-Algorithmen spricht.

Die Auswahl der Sensoren ermöglicht auch eine Unterscheidung der Algorithmen: Solche, die Informationen von Berührungssensoren auswerten und solche, die Informationen von optischen Sensoren auswerten.

Bevor auf diese Algorithmenklassen und einige exemplarische Algorithmen daraus eingegangen wird, sollen erst noch einige historische Suchalgorithmen für Graphen vorgestellt werden, um zu zeigen dass das Suchproblem keine neue Problemstellung ist.

Abschließend soll in Abschnitt 2.4 auch noch auf die heuristischen Suchalgorithmen eingegangen werden, welche unter Aufgabe des Anspruchs, den kürzesten Weg zu finden, Geschwindigkeitsvorteile bei der Berechnung bringen können.

2 Algorithmen zur Pfadplanung

2.1 Historische, einfache Algorithmen

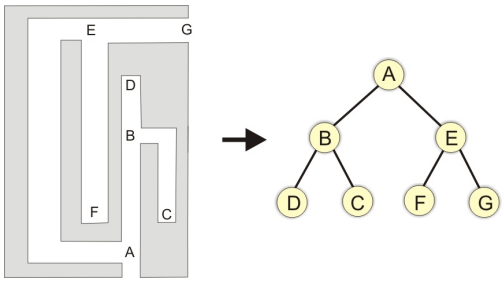
Einfache Pfadplanungs- und Labyrinthalgorithmen gehen bis auf das Jahr 1873 zurück, wie z. B. das Königsberger Brückenproblem u.ä. Man geht davon aus, dass man eine

Maschine bzw. einen Roboter hat, der berühren und/oder sehen kann. Bei Labyrinthen gibt es zwei grundlegende Hauptbereiche:

- einen Weg in das Labyrinth zu finden, um einen bestimmten Gegenstand, z. B. einen Schatz, zu erreichen.
- die Flucht aus dem Labyrinth von einer unbekannten Stelle aus.

Der Zusammenhang zwischen Graphensuche und Suche in Labyrinthen wurde sehr bald festgestellt. Deswegen kann das Labyrinth als Graph gezeichnet werden, indem jeder Korridor als Kante und jede Kreuzung als Knoten dargestellt wird.

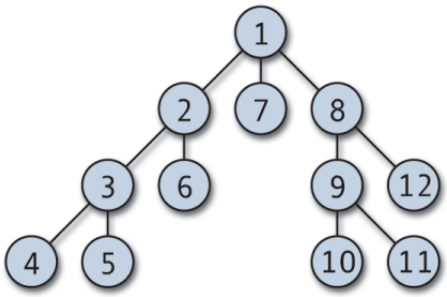
Abbildung 1: Zusammenhang zwischen Labyrinth und Graph



Nun kann man bekannte Suchverfahren für Graphen anwenden. Als einfachste Suchalgorithmen gibt es bei Bäumen (azyklische gerichtete Graphen) die Tiefen- und Breitensuche.

Tiefensuche (depth-first-search)

Abbildung 2: Beispiel für Tiefensuche

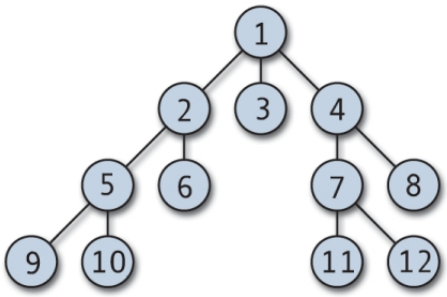


Bei der Tiefensuche wird der Baum zuerst in die Tiefe abgearbeitet, d.h. man startet bei der Wurzel und folgt so lange den Kanten in die Tiefe, bis es eine Sackgasse gibt. Dann geht man zu der letzten Verzweigung zurück, die einen noch unbesuchten Ast enthält, nimmt den nächsten unbesuchten Ast und arbeitet sich wieder in die Tiefe. Dieser Vorgang wird wiederholt, bis der ganze Baum abgearbeitet ist. Die Tiefensuche

ist nicht vollständig, da bei einem unendlichen Baum die Suche sich in dem unendlichen Ast verliert und eventuell keine Lösung gefunden werden kann. Die Tiefensuche ist auch nicht optimal, da oftmals Lösungen gefunden werden, die tiefer im Baum liegen. Um die optimale Lösung zu finden, muss der Baum immer komplett abgearbeitet werden und dann der kürzeste Weg als Ergebnis verwendet werden.

Breitensuche (breadth-first-search)

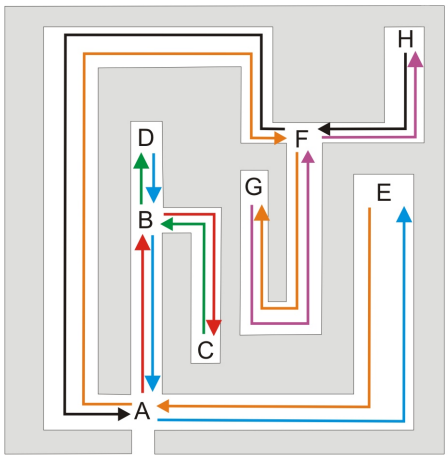
Abbildung 3: Beispiel für Breitensuche



Hierbei wird der Baum Ebene für Ebene abgearbeitet, und es muss immer wieder zu den vorhergehenden Knoten zurückgegangen werden. Der Vorteil bei der Breitensuche ist, dass sobald eine Lösung (z. B. ein Weg aus dem Labyrinth) gefunden wurde, die Suche beendet werden kann, da diese Lösung zugleich auch ein Optimum ist. Der Nachteil der Breitensuche ist, dass das Verfahren sehr speicheraufwändig ist, da sämtliche Knoten und Kanten zwischengespeichert werden müssen.

Tarry & Trémaux Algorithmus

Abbildung 4: Beispiel für den Tarry & Trémaux Algorithmus



Für Roboter, die den ganzen Weg abfahren müssen, ist die Tiefensuche geeigneter, da im Gegensatz zur Breitensuche nicht so viele Wege doppelt gefahren werden müssen. Wenn man einen Weg zum Ziel gefunden hat, kann man aufhören. Der gefundene Weg muss dann nicht unbedingt das Optimum sein, hat aber auch zum Ziel geführt. Da ein Labyrinth endlich ist gibt es auch keine Probleme mit der Vollständigkeit der Tiefensuche.

Der Tarry & Trémaux Algorithmus ist ein Beispiel für die klassische Tiefensuche. Die Richtung, in der das zu suchende Objekt liegt, ist unbekannt und der Graph kann auch Zyklen enthalten. Für die Ausführung wird ein zyklisch gerichteter Weg durch jede Kante konstruiert, wobei jede Kante maximal einmal pro Richtung besucht wird.

Algorithm 1 Tarry & Trémaux

- Der Algorithmus startet willkürlich an einem Knoten und folgt einem möglichen Pfad (Gang) und markiert dabei jede Kante in welcher Richtung sie bereits betreten worden ist.
- Kommt man an einen Knoten (Kreuzung), wählt man einen beliebigen Pfad aus, der noch nicht betreten worden ist. Sind alle Kanten schon betreten, dann eine auswählen, die bis jetzt nur einmal in die Gegenrichtung betreten worden ist.
- Trifft man auf eine Sackgasse oder auf einen schon besuchten Gang, dann muss man zurück zur letzten Kreuzung.
- Man darf keinen Pfad betreten, der schon in beide Richtungen besucht wurde.
- Der Algorithmus ist beendet, wenn man wieder am Startpunkt angekommen ist.

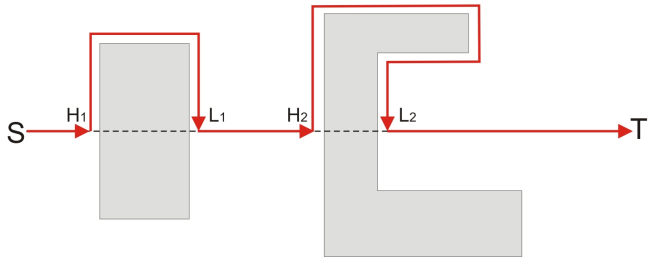
Mit dieser Methode wird das Labyrinth komplett durchlaufen (jeder Pfad zweimal) und man kann einen Gegenstand oder den Ausgang finden. Will man ein spezielles Ziel erreichen, kann man bei Eintreffen des Ereignisses den Algorithmus schon vorher beenden.

2.2 Navigation mit Tastsensoren (IR)

Für die nächsten Algorithmen ist die Richtung, in der das Ziel liegt, bekannt und der Roboter hat Berührungssensoren (IR). Es wird immer eine Linie vom Start (S) zum Ziel (T) konstruiert, auf der sich der Roboter nach Möglichkeit bewegt. Wird ein Objekt getroffen, besitzt der Roboter die Fähigkeit, dieses zu umfahren. Objekte haben einfache geschlossene Kurven als Begrenzungen. Die Hindernisse werden solange umfahren, bis ST wieder getroffen wird. Dann wird versucht auf ST weiter in Richtung T zu fahren. Hierfür haben die Algorithmen Bug2 und Alg1 kleine Unterschiede.

Bug2 (Lumelsky)

Abbildung 5: Beispiel für Bug2



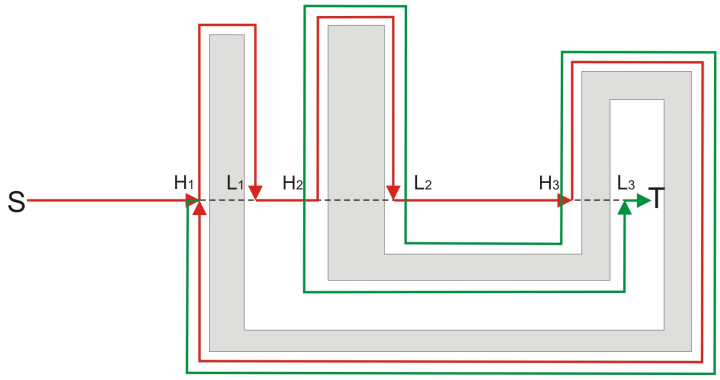
Algorithm 2 Bug2

Start mit $i = 1$

1. Auf der Geraden ST in Richtung T fahren bis:
   a) das Ziel erreicht wird $\rightarrow$ Algorithmus beenden
   b) ein Hindernis getroffen wird. Hitpoint H_i setzen und weiter mit Schritt 2
2. Um das Objekt herumfahren bis:
   a) das Ziel erreicht wird $\rightarrow$ Algorithmus beenden
   b) die Gerade ST in einem Punkt Q getroffen wird & $\overline{QT} < \overline{H_iT}$ & QT kreuzt das aktuelle Objekt nicht (es kann auf QT in Richtung T verlassen werden). $\rightarrow$ Q als Leavepoint L_i setzen, i um 1 erhöhen und weiter mit Schritt 1
   c) man zu H_i (dem zuletzt gesetzten Hitpoint) zurückkehrt. Es wurde eine geschlossene Kurve gefahren. Algorithmus abbrechen, da es keinen Weg zu T gibt

Alg1 (Sankaranarayanan)

Abbildung 6: Beispiel für Alg1



Alg1 versucht möglichst wenig an den Objektgrenzen und möglichst viel auf ST zu fahren. Der Vorteil zu Bug2 liegt darin, dass Alg1 ein mehrmaliges Besuchen eines Objektes verhindert und dadurch Weg einspart.

Algorithm 3 Alg1

Start mit $i = 1$

1. Auf der Geraden ST fahren bis:

   a) das Ziel erreicht wird $\rightarrow$ Algorithmus beenden

   b) ein Hindernis getroffen wird. Hitpoint H_i setzen und weiter mit Schritt 2

2. Der Roboter folgt der Objektgrenze in einer bestimmten Richtung (zu Beginn links) bis:

   a) as Ziel erreicht wird $\rightarrow$ Algorithmus beenden

   b) die Gerade ST in einem Punkt Q getroffen wird und dieser Treffpunkt Q die kürzeste Entfernung zu T hat, die der Roboter bis jetzt erreicht hat und dieser Punkt Q auf der Geraden ST in Richtung T verlassen werden kann. $\rightarrow$ Q als Leavepoint L_i setzen, i um 1 erhöhen und weiter mit Schritt 1

   c) ein vorher definierter Hitpoint oder Leavepoint Q_k ($k < i$) wird wieder getroffen. Dann muss der Roboter zurück zu H_i fahren und die andere Seite des Objektes von H_i besuchen

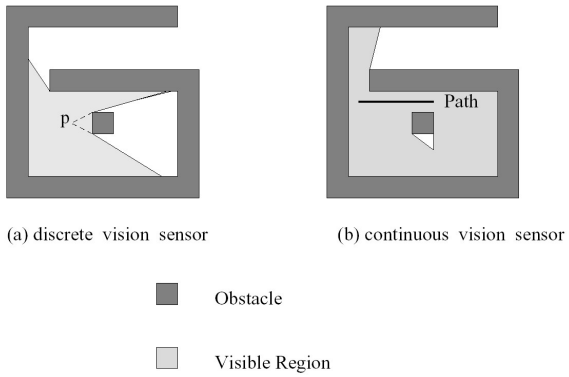
   d) Der zuletzt vorher definierter Hitpoint wird wieder getroffen: $Q = H_i \rightarrow$ Algorithmus abbrechen, da keine Lösung existiert

---

2.3 Navigation mit optischen Sensoren (Kamera)

Wie so oft in der Technik, diente auch hier die Natur als Vorlage; insbesondere ein Lebewesen, das wir in seinen Verhaltensweisen gut studieren können und auch nachfragen können, was es und warum es das so getan hat: Der Mensch. Nur lässt sich leider die Natur nicht 1:1 auf Roboter umsetzen. Schon bei der Sensor-Art gibt es zwei verschiedene Moeglichkeiten:

Abbildung 7: Kontinuierliche und diskrete Sensoren



Einerseits **kontinutierliche** Sensoren, die ununterbrochen Werte liefern und damit eine gleitende Aktualisierung der Karte ermöglichen. Andererseits **diskrete** Sensoren, die nur zu bestimmten Zeiten Werte liefern, wodurch nur ein „Schnappschuss der Wirklichkeit“ erstellt werden kann. Daraus kann man leicht erkennen, dass Algorithmen, die für diskrete Sensoren geschrieben wurden, auch auf koninutierlichen funktionieren, aber nicht umgekehrt. Man könnte meinen, dass eigentlich doch diskrete Sensoren ausreichen müssten (die Robertinos in unserem Seminar sind ja auch nur mit diskreten Sensoren bestückt) - für einfache, mit Polygonen beschreibbare Gebiete mag das gelten, sobald die Umgebungen jedoch natürlicher werden und nicht mehr so leicht darstellbar sind, stoßen diskrete Sensonen schnell an ihre Grenzen.

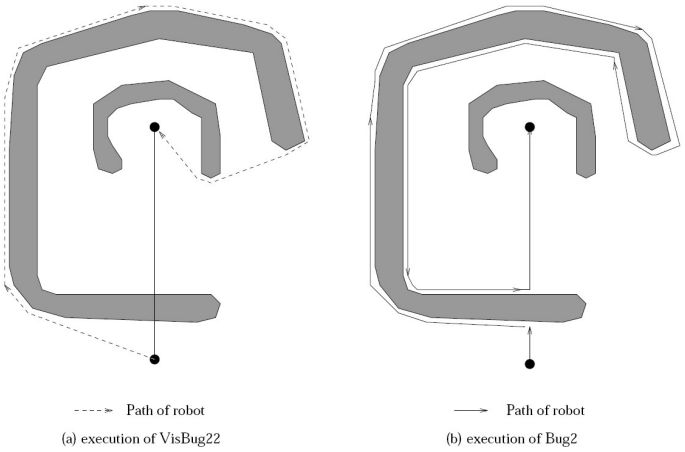
VisBug22 vs. Bug2

VisBug22 basiert auf Bug2. Allerdings kann VisBug22, da er die optischen Sensoren nutzen kann, den Weg, den Bug2 nehmen würde, auf dem sichtbaren Gebiet vorherberechnen. (Sinnvollerweise muss dazu der Sicht-Radius größer als die Hindernisse sein.) Dann bestimmt VisBug22 den dem Ziel am nächsten gelegenen Punkt auf dem Pfad vom Bug2 und der Robotor fährt dort hin. Wenn ein kürzerer Weg zum Ziel als der von Bug2 ersichtlich ist, kann der Bug2-Pfad auch verlassen werden. Wenn der berechnete Punkt erreicht ist, starte den Algorithmus erneut.

Versteckspiel - Visibility based pursuit evation

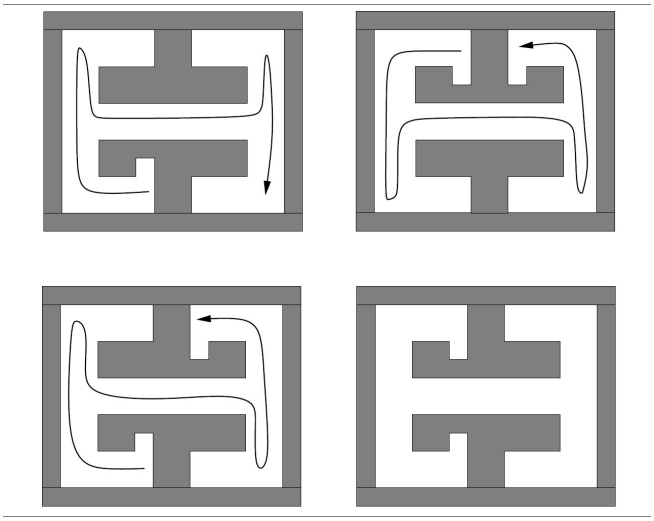
Die grundlegende Problemstellung: In einem unbekannten Gebiet befindet sich ein (oder auch kein) Verfolgter und ein Verfolger, der den Verfolgten finden muss oder sicherstellt, dass es gar keinen Verfolgten gibt. Beispiele zur Anwendung im „täglichen Leben“ wären zum Beispiel Feuerwehrleute, die ein brennendes Haus nach Personen

Abbildung 8: VisBug22 vs. Bug2



durchsuchen, Polizisten, die eine Geiselnahme in einem unbekannten Haus beenden wollen oder die Armee, die feindliches Terrain aufklärt. Wann kann sichergestellt werden, dass eine Lösung gefunden wird? Wenn sich ein Problem nicht mit einem Verfolger lösen lässt, kann man es mit mehreren Verfolgern lösen?

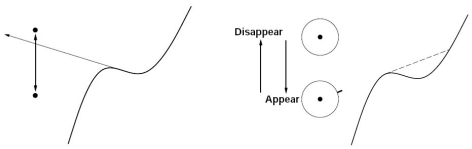
Abbildung 9: Beispiele für Versteckspiel



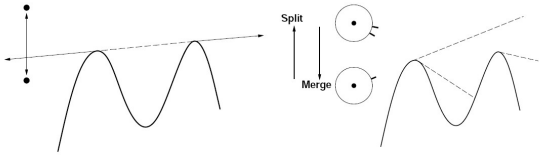
Es lässt sich leicht erkennen, dass man das Gebiet rechts unten in Abbildung 9 nicht alleine aufklären kann. Mit zwei Verfolgern könnte man das Problem hingegen lösen.

Cell decomposition

Auf was muss man achten, um das gesamte Gebiet abzudecken? Dazu lässt sich folgendes feststellen: Es gibt zwei verschiedene „Schwellen“, bei deren „Überfahren“ verschiedene Effekte auftreten:

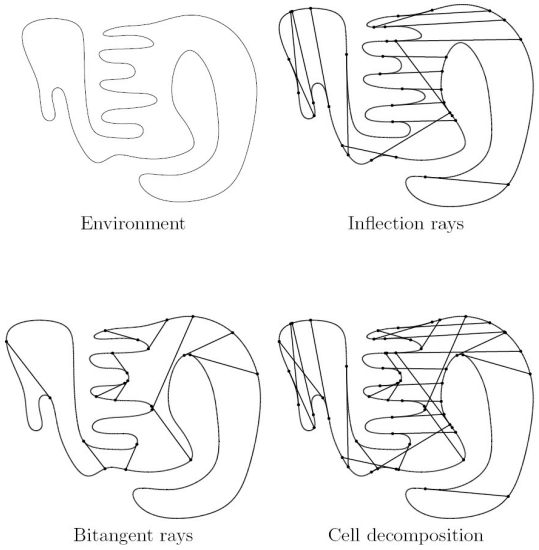


„**Inflection rays**“: Beim „Überfahren“ taucht entweder eine Schatten-Fläche auf, oder eine verschwindet.



„**Bitangent rays**“: Beim „Überfahren“ wird eine Schatten-Fläche geteilt oder zwei Flächen verschmelzen zu einer.

Abbildung 10: Cell decomposition



Wenn man in ein Gebiet beide Schwellen einzeichnet, erhält man einen Graphen, der alle kritischen Bereiche markiert. Man sieht: bewegt man sich ausschließlich in einer der so entstandenen Zellen, ändert sich der „Wissensstand“ über den Inhalt anderer Zellen nicht. Daher ist es nicht wichtig, welchen Weg der Roboter innerhalb einer Zelle nimmt.

Algorithmenskizze

Aufgabe: Du bist der Verfolger. Stelle sicher, dass sich in dem vorgegebenen Gebiet kein Verfolgter befindet.

Lösungsidee: Alle „kritischen Schwellen“, die durch die Cell Decomposition gefunden wurden, müssen analysiert werden, d.h. der entstandene Graph muss „abgearbeitet“ werden.

- Teilgebiete, die von der momentanen Position nicht eingesehen werden können, heißen „Schatten-Elemente“
- Wenn ein Element als „sauber“ gekennzeichnet ist, bedeudet das, dass sich dort kein Verfolgter aufhält
- Wenn ein Element als „kontaminiert“ gekennzeichnet ist, kann sich dort evtl. ein Verfolgter aufhalten

---

Algorithm 4 Visibility based pursuit evation

- Alle Teilgebiete werden als kontaminiert gekennzeichnet
- Wenn ein „inflection ray“ getroffen wird und ein neues Schatten-Element erscheint, wird es als „sauber“ gekennzeichnet (denn dann war es gerade eben noch sichtbar)
- Wenn ein „bitangent ray“ getroffen wird müssen zwei Fälle unterschieden werden:
  - Wenn ein Schatten-Gebiet in zwei Teile geteilt wurde, werden die beiden entstandenen Teile genauso gekennzeichnet, wie das ursprüngliche Element gekennzeichnet war („sauber“ → „sauber“ + „sauber“ bzw. „kontaminiert“ → „kontaminiert“ + „kontaminiert“)
  - Wenn zwei Schatten-Elemente zu einem neuen verschmelzen, wird das neue nur genau dann als „sauber“ gekennzeichent, wenn beide ursprünglichen „sauber“ waren, sonst wird es „kontaminiert“. (Dadurch kann es durchaus zu einer Wiederkontaminierung kommen und vorhergegangene Arbeit zunichte gemacht werden)
- Durchquere das Gebiet solange, bis alle Teilgebiete als „sauber“ markiert wurden.

Bei diesem Algorithmus kann es vorkommen, dass der Roboter ein Gebiet mehrfach und mit verschiedenem Wissensstand über den Inhalt anderer Gebiete besucht.

2.4 Heuristische Suchalgorithmen

Heuristiken

> Als **Heuristik** (abgeleitet von [alt]griechisch ευρισκω, *heurísko*, zu deutsch *ich finde*) bezeichnet man eine Strategie, die das Streben nach Erkenntnis und das Finden von Wegen zum Ziel planvoll gestaltet.[1]

[1] Aus: `http://de.wikipedia.org/wiki/Heuristik`

Für unsere Zwecke ist nur die Bedeutung von „Heuristik“ im Bezug auf das Suchproblem relevant. Hier stellt die Heuristik eine Bewertungsfunktion dar, welche die Entfernung von möglichen Wegpunkten bei der Wegeplanung zum Ziel schätzt. Heuristische Suchalgorithmen wählen also aus einer Liste von möglichen Wegpunkten für den nächsten Schritt solche aus, die von der Heuristik als besser bewertet wurden. Da eine Heuristik die Entfernung nur schätzt, im allgemeinen Fall also keine exakte Aussage treffen kann, garantieren heuristische Suchalgorithmen allgemein auch keine optimalen Lösungen. Daher sind sie für Anwendungsfälle, in denen der kürzeste Weg garantiert gefunden werden muss, nicht ohne weiteres geeignet. Dagegen bieten sie für Anwendungen, in denen nur ein möglichst kurzer Weg - aber nicht unbedingt der Kürzeste - gefunden werden muss, deutlich kürzere Rechenzeiten.

Der A*-Algorithmus - Einführung

Der A*-Algorithmus in seiner allgemeinen Form ist ein Suchalgorithmus für den kürzesten Pfad in kantengewichteten Graphen (mit nicht-negativen Gewichten). Als heuristischer Suchalgorithmus ist die Grundlage von A* die heuristische Funktion $H(n)$. Diese schätzt die minimale Entfernung vom Knoten n zum Zielknoten, liefert also eine untere Schranke für den noch zurückzulegenden Weg. Die Wahl der Heuristik bestimmt nahezu vollständig die Eigenschaften von A*. Eine Auswahl an gängigen Heuristiken folgt später. Zu beachten ist jedoch eine sehr wichtige Bedingung: $H(n)$ darf die Entfernung zum Zielknoten nie zu hoch einschätzen. Sonst werden potentiell richtige Knoten auf dem Weg zum Ziel nicht untersucht und der Algorithmus findet evtl. den kürzesten Weg nicht. Weiter gibt es eine Funktion $G(n)$, welche die Kosten für den Weg vom Startknoten zum Knoten n liefert. Dabei bestehen die Kosten aus der Entfernung und evtl. unterschiedlichen Kosten für unterschiedliches Gelände. So kann man bergaufgehen „teurer“ bewerten und damit das Umgehen von Anhöhen bevorzugen oder Straßen „billiger“ bewerten und damit die Benutzung von Straßen bevorzugen. Damit folgt als Gesamtkostenfunktion $F(n)$ für einen Knoten n: $F(n) = G(n) + H(n)$. Zuletzt benötigt man noch zwei Listen A und B, in denen man die noch zu untersuchenden Knoten (A) bzw. die bereits abgearbeiteten Knoten (B) speichert. Nun kann man folgendes Ablaufschema für A* angeben:

Algorithm 5 A*

1. Füge den Startknoten in A ein
2. Wiederhole:
   a) Wähle den Knoten *n* mit den niedrigsten Kosten $F(n)$ aus A aus und verschiebe ihn in B
   b) Für jeden an *n* direkt angrenzenden Knoten *m*:
      i. Wenn *m* nicht betretbar (Hindernis, Wasser, etc.) oder bereits in B ist, ignoriere ihn
      ii. Füge *m* in A ein, wenn er noch nicht enthalten ist
      iii. Trage die Kosten $F(m)$ und $G(m)$ ein und vermerke als Vorgänger *n* bzw. aktualisiere sie wenn *m* schon enthalten war und ein Weg über *n* mit kleinerem $G(m)$ gefunden wurde
3. Wenn der Zielknoten in A eingefügt wurde, ist ein Weg gefunden worden. Wenn A leer geworden ist, ohne den Zielknoten zu finden, existiert kein Weg

Ein Beispiellauf des A*-Algorithmus ist in den Folien zu diesem Vortrag oder bei [lester] zu finden.

Wenn es in Schritt 2a mehrere Knoten mit den gleichen, minimalen Kosten gibt, erfolgt die Auswahl nach einer vorher festzulegenden Strategie. Die Wahl dieser Strategie beeinflusst, welcher Weg am Ende gefunden wird, wenn mehrere kürzeste Wege existieren. Sie kann unter Umständen auch die Laufzeit des Algorithmus beeinflussen. Auf diese Strategien soll hier aber nicht näher eingegangen werden. Im einfachsten Fall kann man z.B. den zuletzt eingefügten Knoten nehmen.

Eigenschaften von A*

- Der A*-Algorithmus ist **vollständig**. Das bedeutet dass wenn ein Pfad zum Zielknoten existiert, dieser garantiert gefunden wird. Dies sollte offensichtlich sein, da der Algorithmus alle betretbaren Knoten absucht, bis er einen Weg vom Startknoten zum Zielknoten findet. Er terminiert, wenn alle betretbaren Knoten besucht wurden, unabhängig ob ein Pfad gefunden wurde oder nicht.

- A* ist **optimal** unter der Bedingung dass die Heuristik die Entfernung zum Zielknoten nie überschätzt. Damit kann man zeigen, dass A* bei korrekter Wahl der Heuristik garantiert den kürzesten Weg findet - obwohl er ein heuristischer Suchalgorithmus ist.

- A* ist sogar **optimal effizient**, „d.h. jeder andere optimale und vollständige Algorithmus, der dieselbe Heuristik verwendet, muss mindestens so viele Knoten betrachten wie A*, um eine Lösung zu finden“[a*-wiki].

Diese Eigenschaften lassen A* zu einem der wichtigsten Algorithmen der Pfadplanung werden, denn er ist - bei korrekter Wahl der Heuristik - damit der effizienteste Algorithmus bzgl. des Berechnungsaufwandes, der garantiert der kürzesten Pfad findet.

Heuristiken für A*

Eine gute Heuristik muss eine möglichst hohe untere Schranke für die Entfernung zum Zielknoten liefern. Eine perfekte Heuristik würde immer die exakte Entfernung liefern, allerdings ist dies in der Praxis nur selten möglich. Bei einer ebenen Fläche ohne Hindernisse wäre z.B. die Luftlinie eine perfekte Heuristik. Allerdings spielt es für die meisten Anwendungen auch keine Rolle wenn die Heuristik nicht perfekt ist, solange sie nicht überschätzt. Eine überschätzende Heuristik macht A* nicht mehr optimal, was je nach Anwendung kritisch oder auch völlig unproblematisch sein kann.

Die im Vortrag verwendete Heuristik ist die sogenannte „**Manhattan-Methode**“, eine sehr einfache und häufig verwendete Methode. Sie überschätzt allerdings wenn Bewegungen in mehr als 4 Richtungen möglich sind (also auch diagonale Bewegungen zwischen angrenzenden Knoten gestattet sind)! Da die Ergebnisse für die meisten Anwendungen nicht sehr viel schlechter sind als der kürzest mögliche Pfad, wurde sie wegen ihrer Einfachheit dennoch gewählt.

$$H(n) = C \cdot (|n_x - z_x| + |n_y - z_y|)$$

(wobei C die minimalen Kosten für eine Bewegung von einem Knoten zum nächsten und z der Zielknoten sind).

Um diagonale Bewegungen zu gestatten und trotzdem garantiert den kürzesten Weg zu finden, kann man den „**Diagonalen-Abstand**“ verwenden:

$$\begin{aligned} H_{diagonal}(n) &= \min(|n_x - z_x|, |n_y - z_y|) \\ H_{gerade}(n) &= |n_x - z_x| + |n_y - z_y| \\ H(n) &= C_{diagonal} \cdot H_{diagonal}(n) + C_{gerade} \cdot (H_{gerade}(n) - 2 \cdot H_{diagonal}(n)) \end{aligned}$$

(wobei $C_{diagonal}$ die Kosten für eine diagonale Bewegung und C_{gerade} die Kosten für eine gerade Bewegung sind). Der Trick ist, dass ein diagonaler Schritt 2 geraden Schritten aus der Manhattan-Methode entspricht. Damit kann man die Gesamtkosten leicht abschätzen.

Wenn Bewegungen in mehr als 8 Richtungen gestattet werden, kann man die Luftlinie als Heuristik verwenden. Allerdings erhöht dies die Laufzeit von A*, da $G(n)$ nur 8 Bewegungsrichtungen berücksichtigt und damit mehr Wege ausprobiert werden.

Zusammenfassend noch einige interessante Eigenschaften von A* bei bestimmten Heuristiken:

- Wenn $H(n) = 0$ ist, verhält sich A* exakt wie ein Dijkstra-Algorithmus, da immer der vom Startpunkt aus kürzeste Weg zuerst untersucht wird
- Je kleiner $H(n)$ ist, umso mehr Knoten müssen untersucht werden, umso langsamer ist also A*
- Wenn $H(n)$ perfekt ist, folgt A* immer nur dem kürzesten Weg und ist damit sehr schnell
- Wenn $H(n)$ manchmal überschätzt, dann ist er nicht mehr optimal, aber meistens schneller als mit besseren Heuristiken (der Diagonalen-Abstand ist wesentlich aufwändiger zu berechnen als der Manhattan-Abstand)
- Wenn $H(n)$ extrem große Werte liefert, wird $G(n)$ nicht mehr berücksichtigt und A* verhält sich exakt wie Best-First-Search

3 Zusammenfassung

Es hat sich gezeigt, dass sich eine Vielzahl an sehr unterschiedlichen Lösungsansätzen für das Suchproblem finden lässt. Dennoch ist keiner dieser Algorithmen gleichermaßen für jede Problemstellung geeignet - jeder hat seine individuellen Schwächen. Deshalb ist zur Auswahl eines Navigationsalgorithmus zuerst eine genaue Analyse der Situation und Umgebungsparameter nötig. Daraus lässt sich eine Auswahl an geeigneten Sensoren bestimmen, welche wiederum die Auswahl der zur Verfügung stehenden Algorithmen ermöglicht. Unter Umständen kann es auch sinnvoll sein, mehrere verschiedene Algorithmen zusammen zu verwenden, z. B. für die Planung von großen Wegen mit grober Auflösung und kleinen Wegen mit feiner Auflösung.

Die heutigen Algorithmen sind in ihrer Effektivität weniger durch die Rechenzeit zur Planung sondern durch die Qualität der Sensoren und die Rechenzeit zur Auswertung der Sensordaten beschränkt. Fortschritte bei der Sensortechnik lassen auf bessere Ergebnisse bei der Pfadplanung hoffen. Da die Anzahl an autonomen Robotern auch im Alltag langsam zunimmt (Staubsaugerroboter „Roomba“, militärische Erkundungsdrohnen), ist sicher auch ein marktwirtschaftlicher Anreiz für weitergehende Entwicklungen vorhanden.

Literatur

[a*-wiki] http://de.wikipedia.org/wiki/A*-Algorithmus

[lavalle] Lavalle, Planning algorithms, Kapitel 12.3-12.5.

[lester] Lester, A* Pathfinding for Beginners, http://www.gamedev.net/reference/articles/article2003.asp

[rao] Rao et. al, Robot navigation in unknown terrains: introductory survey of non-heuristic algorithms, 1993.