

# Hauptseminar

## Was Sie schon immer über Spiele wissen wollten...

Wintersemester 2005/06

Ausarbeitung zum Vortrag

**A\***

von

**Stefan K. Baur**

Dozent: Dominic Schell

Lehrstuhl für Programmiersysteme (Lehrstuhl 2)  
Friedrich-Alexander Universität Erlangen-Nürnberg

28. November 2005

# Inhaltsverzeichnis

<b>1 Einführung</b>	<b>2</b>
<b>2 Grundlagen zur Pfadsuche</b>	<b>3</b>
2.1 Die Agenda	3
2.2 Die Kostenfunktion $g$	4
2.3 Die heuristische Funktion $h$	4
<b>3 A*-Suchverfahren</b>	<b>6</b>
3.1 Schätzfunktion $f$	6
3.2 Eigenschaften	7
3.3 Algorithmus	7
<b>4 Implementierung</b>	<b>8</b>
<b>Literaturverzeichnis</b>	<b>9</b>

## 1 Einführung

Es gibt viele Probleme, die bei der Entwicklung eines Computerspiels gelöst werden müssen. Zudem stellt der Spieler immer höhere Ansprüche an Computerspiele und setzt sogar Vieles als selbstverständlich voraus. Bereits die Pfadsuche, die der Spieler als selbstverständlich ansieht, fällt erst bei Fehlverhalten auf. Diese Ausarbeitung zeigt die Schwierigkeiten einer effizienten Pfadsuche in einem zweidimensionalen, rasterisierten Labyrinth (Grid-Map) auf und erklärt die heutzutage populärste Pfadsuche **A\*** (sprich: A-Stern), welche schon 1968 von Peter Hart, Nils Nilson und Bertram Raphael entwickelt wurde[[Wik](#)].

Der A\*-Algorithmus wird üblicherweise in Spiele wie beispielsweise WarCraft III integriert[[Ast](#)]. Der Suchalgorithmus von WarCraft II weist nämlich folgende Defizite auf:

- **Inkorrekte Lösungen**

Besonders unangenehm wird es für den Spielenden, wenn der Suchalgorithmus eine inkorrekte Antwort liefert. In stark verzweigten Labyrinth kommt der Held – wenn auch nur selten – nicht auf dem gewünschten Zielpunkt, sondern in Sackgassen zum Stehen, trotz einer vorhandenen Lösungsmöglichkeit. Die *Korrektheit* der Pfadsuche, d.h. dass die Antwort des Algorithmus eine Lösung des Suchproblems ist sowie die *Vollständigkeit*, d.h. wenn der Algorithmus für ein lösbares Problem stets eine Lösung findet[[Tb104](#)], sollte der Spieler bei einem 2D-Computerspiel als Mindestvoraussetzung annehmen dürfen.

- **Umwege**

Manchmal bewegt sich ein Held im Labyrinth aus unerklärlichen Gründen nicht auf **kürzestem Wege** von **A** nach **B**. Derartige Umwege des Helden bedeuten für den Spielenden meist erhebliche Einbußen an Zeit und schlimmstenfalls sogar den Verlust

des Spiels. Deshalb ist es wichtig einen Pfadsuchalgorithmus im Spiel zu integrieren, welcher *optimal* ist, also den kürzesten Pfad von A nach B für den Helden des Spiels ermittelt.

- **Die “Schrecksekunde”**

Der Spielende sollte nicht den Eindruck haben, dass das laufende Computerspiel aufgrund ineffizienter Algorithmen ins Stocken gerät. Bei manch älteren Spielen muss der Spielende nämlich warten bis die (ineffiziente) Pfadsuche endlich abgeschlossen ist. Erst wenn der Pfad für den Helden gefunden ist, macht dieser sich nach der sogenannten Schrecksekunde auf den Weg. Um diesen Effekt zu minimieren, muss die Pfadsuche extrem schnell sein.

Der A\*-Algorithmus verfügt über Eigenschaften, mit denen diese Defizite nicht mehr zum Tragen kommen.

## 2 Grundlagen zur Pfadsuche

Da die A\*-Suche ein relativ komplexes Suchverfahren ist, werden im Folgenden wichtige Begriffe anhand einfacher Suchverfahren vorgestellt.

### 2.1 Die Agenda

Im Prinzip arbeitet jedes Suchverfahren mit einer Liste, die sogenannte **Agenda**. Zu Beginn einer Suche enthält die Agenda lediglich den Startknoten. In jedem Schritt wird der Knoten mit der höchsten Priorität aus der Agenda entfernt und geprüft, ob es sich um einen Zielknoten handelt. Handelt es sich um einen Zielknoten, so liegt eine Lösung des Suchproblems vor, falls nicht, so werden die Nachbarknoten (Adjazenzknoten) in die Agenda mitaufgenommen. Wenn die Agenda leer ist, gibt es keine Lösung des Suchproblems.[\[Tb104\]](#)

- Bei der **Tiefensuche** (Depth-First-Search) zum Beispiel verhält sich die Agenda wie ein Keller, der sogenannte **Stack**. Der Knoten mit der höchsten Priorität befindet sich in der Agenda stets an erster Stelle. Die besagten Nachbarknoten werden der Agenda vorne angefügt.
- Und bei der **Breitensuche** (Breadth-First-Search) verhält sich die Agenda wie eine Warteschlange, die sogenannte **Queue**. Der Knoten mit der höchsten Priorität befindet sich in der Agenda stets an erster Stelle und die Nachbarknoten werden der Agenda hinten angefügt.

Jeder einzelne Knoten verfügt über Attribute wie zum Beispiel welcher Knoten sein Vorgänger ist, so kann bei einer erfolgreichen Suche über den Zielknoten der Pfad bis zum Startknoten zurückverfolgt werden. Aber auch weitere Attribute, welche je nach Suchverfahren Kosten- und Schätzwerte speichern, können einem Knoten angehören.

## 2.2 Die Kostenfunktion $g$

Suchverfahren, welche als Lösung stets den kürzesten Pfad zurückgeben, die sogenannten *optimalen Suchverfahren*, bedienen sich einer Kostenfunktion.

“Eine **Kostenfunktion** ist eine Funktion, die jedem Knoten  $k$  im Suchraum ein Gewicht  $g(k) \geq 0$  zuordnet. Eine Kostenfunktion  $g$  heißt **streng monoton**, wenn  $g(k_1) < g(k_2)$  gilt für alle Knoten  $k_1, k_2$ , so dass  $k_1$  ein Vorgänger von  $k_2$  ist.” [TbI04]

Die Kostenfunktion gibt Aufschluss darüber wie hoch der Suchaufwand vom Startknoten bis zum jeweiligen Knoten des Suchbaumes ist.

- Der **Dijkstra-Suchalgorithmus** zum Beispiel verwendet eine streng monotone Kostenfunktion. Die Agenda der Dijkstra-Suche verhält sich wie eine Prioritätswarteschlange, wobei die Priorität der Knoten über die Kostenfunktion ermittelt wird. Je geringer die Kosten eines Knotens sind, desto höher die Priorität, deshalb handelt es sich hier um die sogenannte **Min-Priority-Queue**. Mit der Kostenfunktion  $g$  verfolgt der Dijkstra-Algorithmus ausgehend vom Startknoten den kostengünstigsten Pfad zuerst.

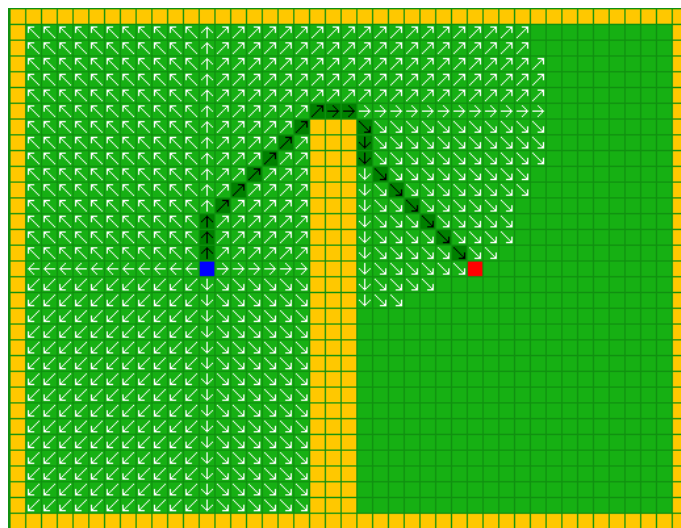


Abbildung 1: Beispiel einer Dijkstra-Suche

Bei den eben angesprochenen Suchverfahren wie Tiefensuche, Breitensuche und Dijkstra-Suche handelt es sich um *uninformierte Suchverfahren*, d.h. diese Verfahren erhalten keinerlei Information über Ort und Entfernung des Ziels.

## 2.3 Die heuristische Funktion $h$

*Heuristische bzw. informierte Suchverfahren* greifen auf eine Heuristik zurück, um zielgerichtet zu suchen [Wik], dazu erhalten sie die genaue Position des Ziels. Die Knoten werden

daraufhin geprüft, welcher der Wahrscheinlichkeit nach dem Ziel am nächsten ist, dies wird anhand einer heuristischen Funktion  $h$  ermittelt.

“Eine **heuristische Funktion**  $h$  ordnet jedem Knoten  $k$  im Suchraum eine nicht negative Zahl  $h(k)$  zu, die eine Schätzung für die Entfernung des Knotens  $k$  zum nächsten Zielknoten angibt.” [Tb104]

- Die **Bestensuche** (Best-First-Search), häufig auch Greedy-Search genannt, behilft sich einer heuristischen Funktion. Genau wie bei der Dijkstra-Suche, verhält sich die Agenda der Bestensuche wie eine **Min-Priority-Queue**, nur dass anstelle der Kostenfunktion  $g$  die heuristische Funktion  $h$  verwendet wird. Je kleiner der heuristische Wert eines Knotens ist, desto höher die Priorität des Knotens in der Agenda.

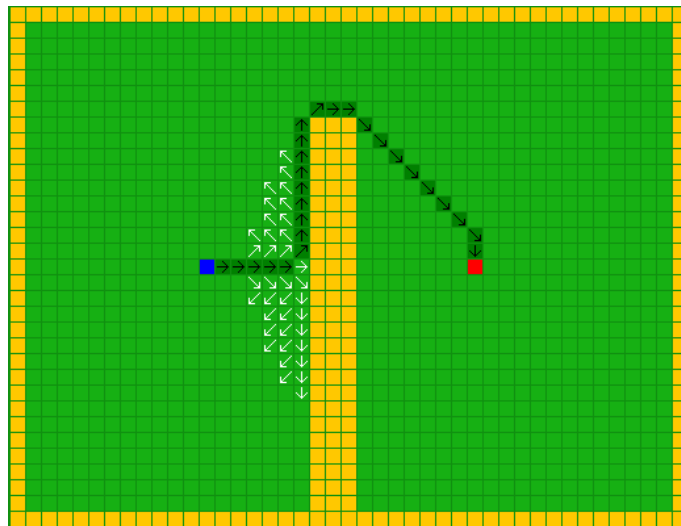


Abbildung 2: Beispiel einer Bestensuche (nicht optimal)

Damit aber die heuristische Suche optimal ist, muss die Schätzung optimistisch bzw. zulässig sein. Die Heuristik darf nie schlechter als der tatsächlich mögliche Pfad sein. “Eine Schätzfunktion  $h$  heißt **zulässig**, wenn sie die Kosten, um von einem Knoten  $k$  zum Ziel zu kommen, niemals überschätzt, das heißt wenn für jeden Knoten  $k$  und für jeden Zielknoten  $z$ , der von  $k$  aus erreichbar ist, die folgende Ungleichung gilt:  $h(k) \leq g(z) - g(k)$ ” [Tb104]. Deshalb seien im Folgenden einige zulässige Heuristiken für Grid-Maps angeführt, siehe [Pat03]:

- **Manhattan-Abstand**

Gibt es zu einem Knoten maximal vier Adjazenzknoten, also für jede Himmelsrichtung maximal einen Knoten, dann kann der *Manhattan-Abstand* als heuristische Funktion gewählt werden:

$$h(n) = D * (|n.x - goal.x| + |n.y - goal.y|),$$

wobei  $D$  der minimale Abstand zu einem Nachbarknoten,  $n$  den aktuellen Knoten und  $goal$  das Ziel bezeichnet.

- **Diagonal-Abstand**

Falls sich die Einheiten aber auch diagonal auf der Grid-Map bewegen können, d.h. wenn es zu einem Knoten maximal acht Adjazenzknoten gibt, dann würde nämlich der Manhattan-Abstand nicht zulässige Schätzwerte liefern. Abhilfe schafft eine weitere heuristische Funktion, der *Diagonal-Abstand*:

$$h(n) = D * \max(|n.x - goal.x|, |n.y - goal.y|).$$

- **Euklidischer Abstand**

Und falls die Einheiten in jede Richtung ziehen können, so ist der *Euklidische Abstand* angebracht:

$$h(n) = D * \sqrt{(n.x - goal.x)^2 + (n.y - goal.y)^2}.$$

Mit dem Euklidischen Abstand wird die Luftlinienentfernung vom aktuell betrachteten Knoten bis zum Zielknoten bestimmt.

### 3 A\*-Suchverfahren

Die A\*-Suche kombiniert die Kostenfunktion  $g$  der Dijkstra-Suche und die heuristische Funktion  $h$  der Bestensuche[Ast].

Im Prinzip sind sich die Dijkstra-Suche und die Bestensuche sehr ähnlich. Beide Suchverfahren verfügen über eine Agenda, die sich wie eine Min-Priority-Queue verhält. Sie unterscheiden sich lediglich in der Art der Prioritätsfunktion ( $g$  vs.  $h$ ).

- **Nachteil der Dijkstra-Suche**

Wird die Dijkstra-Suche in einer Grid-Map, in der benachbarte Knoten stets das gleiche Kantengewicht haben, ausgeführt, verhält sich diese Suche wie eine Breitensuche.

- **Nachteil der Bestensuche**

Die Bestensuche macht sich gefräßig (greedy) in Richtung Ziel auf, falls es auf dem Wege zum Ziel Hindernisse geben sollte, so ist diese Suche in der Regel *nicht optimal*.

Mit der Vereinigung jedoch heben sich die Nachteile dieser Suchverfahren gegeneinander auf.

#### 3.1 Schätzfunktion $f$

Die A\*-Suche benutzt ebenfalls eine Agenda, die sich wie eine Min-Priority-Queue verhält, die neue Prioritäts- bzw. Schätzfunktion  $f$  setzt sich wie folgt zusammen:

$$f(k) = g(k) + h(k),$$

wobei die  $g$  eine streng monotone Kostenfunktion und  $h$  eine zulässige heuristische Funktion ist [Tb104].

Bezüglich der Schätzfunktion  $f$  ergeben sich für den Spieleentwickler hervorragende Eigenschaften.

### 3.2 Eigenschaften

Die Eigenschaften des A\*-Suche sind gleichzeitig auch seine Vorteile. Der A\*-Suche ist:

- **korrekt**, d.h. das Suchergebnis, das die A\*-Suche liefert, ist eine Lösung des Suchproblems [Tb104].
- **vollständig**, d.h. wenn es eine Lösung des Suchproblems existiert, so wird diese auch gefunden.
- **optimal**, d.h. dass die Lösung des Suchproblems der kürzeste Pfad zum Zielknoten ist.
- **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." [Asa]

Sein Hauptnachteil ist, ähnlich wie bei der Breitensuche, seine ungünstige Speicherplatzkomplexität mit  $O(\text{Branchingfaktor}^{\text{Suchtiefe}})$  [Tb104].

### 3.3 Algorithmus

Gegeben sind der STARTKNOTEN, der ZIELKNOTEN sowie die SCHÄTZFUNKTION.

```
Erzeuge die Agenda als Min-Priority-Queue (OPEN)
Erzeuge eine Liste aller abgearbeiteten Knoten (CLOSED)
Füge den STARTKNOTEN in OPEN ein.
```

```
Tue solange die OPEN nicht leer ist
```

```
(
  Entnimm den KNOTEN gemäß der SCHÄTZFUNKTION
  Ermittle alle ADJAZENZKNOTEN von KNOTEN
  Für alle ADJAZENZKNOTEN je ADJ
  (
    Setze ADJ.Vorgänger auf KNOTEN
    Berechne SCHÄTZNEU mit Hilfe der SCHÄTZFUNKTION(ADJ)
    Falls sich ADJ in OPEN befindet und  $SCHÄTZALT \leq SCHÄTZNEU$ ,
      dann verwerfe ADJ.
    Falls sich ADJ in CLOSED befindet und  $SCHÄTZALT \leq SCHÄTZNEU$ ,
      dann verwerfa ADJ
    Andernfalls lösche alle gleichen Knoten aus OPEN und CLOSED.
```

)  
)  
Füge den ADJ in OPEN ein.

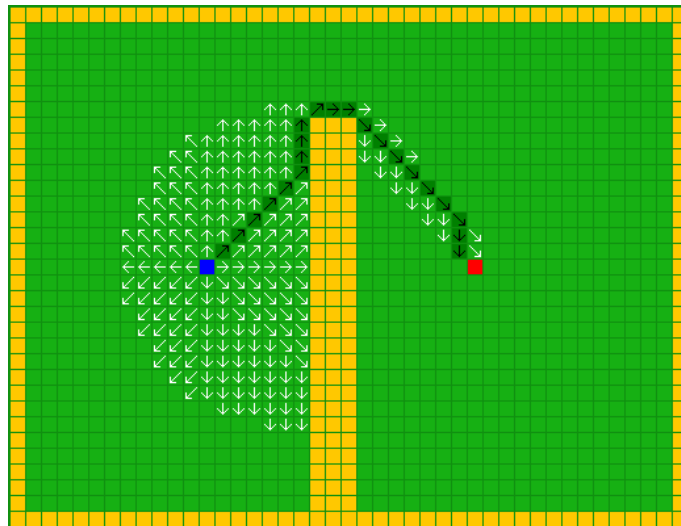


Abbildung 3: Beispiel einer A\*-Suche

## 4 Implementierung

Die Implementierung besteht praktisch nur aus einer einzigen Klasse `AStar`. Der A\*-Suchalgorithmus ist eine Black-Box, der man die Map sowie Start- und Zielknoten übergibt. Die Methode `get_path()` gibt einen `vector` von `Position` zurück. Ist der Rückgabvektor leer, so existiert keine Lösung.

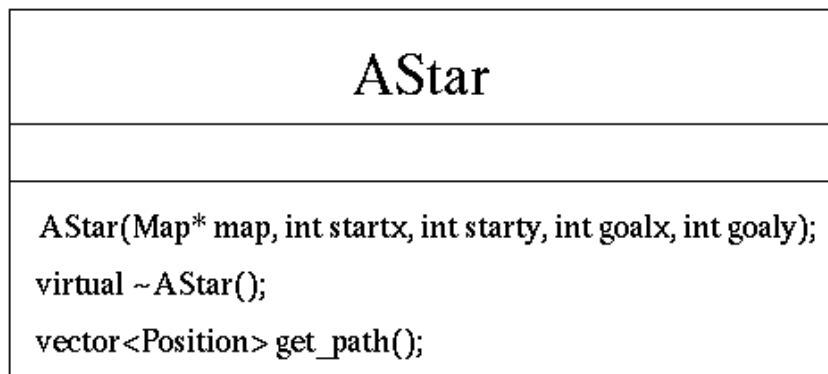


Abbildung 4: Klassendiagramm zur A\*-Suche

## Literatur

- [Asa] A\*-Algorithmus.  
<http://a-stern-algorithmus.lexikona.de/art/A-Stern-Algorithmus.html>.
- [Ast] Pathfinding using A\*.  
<http://rajiv.macnn.com/tut/search.html>.
- [Pat03] Amit J. Patel. Amit's Thoughts on Path-Finding and A-Star, 2003.  
<http://theory.stanford.edu/~amitp/GameProgramming/>.
- [Tb104] *Taschenbuch der Informatik*. Fachbuchverlag im Carl Hanser Verlag, 2004.  
ISBN: 3-446-22584-6.
- [Wik] A\*-Algorithmus bei Wikipedia.  
<http://de.wikipedia.org/wiki/A-Stern-Algorithmus>.