BACHELORARBEIT

im Studiengang Bachelor Informatik

Pathfinding-Algorithmen in verschiedenen Spielegenres

Ausgeführt von: Andreas Hofmann
Personenkennzeichen: 1010257010

BegutachterIn: DI Stefan Reinalter

Hollabrunn, 07.01.2013

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht."

Hollabrunn, 07.01.2013	
Ort, Datum	Unterschrift

Kurzfassung

Einem Computer beizubringen den Weg von A nach B zu finden ist - trotz der simpel scheinenden Aufgabe – ein komplexes Problem. Die Lösung kann nur mithilfe exakter Algorithmen gefunden werden. Diese stammen aus der mathematischen Graphentheorie. Einige von ihnen, wie der A* Algorithmus, haben sich im Pathfinding zu einem weit verbreiteten Standard entwickelt.

Die Einsatzmöglichkeiten und die Leistung dieser Algorithmen sollen durch deren Analyse aufgezeigt werden. Diese Arbeit bietet gleichzeitig einen Einstieg in das Thema Pathfinding in Videospielen und einen Ausgangspunkt für tiefergreifende Recherche zu den einzelnen Pathfinding Algorithmen.

Schlagwörter: Pathfinding, Algorithmus, A*, Graphentheorie

Abstract

To teach a computer to find a way from A to B is – despite this simple task – a complex problem. The solution can only found by using exact algorithms, developed in the field of mathematical graph theory. Some of this algorithms, like A*, become a widespread standard in path finding.

The field of application and the performance of these algorithms should be displayed by analyzing them. This paper offers also an introduction in path finding and an origin for further research into path finding algorithms.

Keywords: path finding, algorithm, A*, graph theory

Danksagung

Ich möchte mich bei meiner Verlobten Nadine für ihre Unterstützung und Geduld bedanken.

Inhaltsverzeichnis

1 Einführung

Um von Punkt A nach B zu kommen, ist, selbst für menschliche Individuen nicht immer leicht und erfordert oft zahlreiche Hilfsmittel. Dank moderner Technik wie GPS und Computer lassen sich Routen aber immer einfacher planen. Aber wie finden Kartensoftware, autonome Roboter oder aber Figuren in einem Computerspiel von A nach B?

Wo uns unser Auge auf einem Blick den Pfad erkennen lässt, ist der Computer blind. Er muss den Pfad durch mehr oder weniger komplexe Algorithmen berechnen.

Im Game-Development hat sich der A* als Quasi-Standard etabliert. Es existieren viele Abwandlungen aber der Kern-Algorithmus ist immer derselbe. Aber ist der A* so mächtig wie es auf den ersten Blick scheint? Und ist er tatsächlich so verbreitet; gibt es keine besseren Algorithmen?

Tatsächlich findet der A* Anwendung in Shootern wie „Unreal Tournament“, in Rollenspielen wie „The Witcher“ und auch in Point-and-Click Adventures wie „Monkey Island“, die früher meist mit vorberechneten Pfaden gearbeitet haben.

Strategiespiele wie „WarCraft“ arbeiten z.B. mit der A*-Erweiterung HAA*. Aber was steckt hinter den Algorithmen und wie funktionieren sie?

2 Grundlagen

Bevor näher auf die einzelnen Algorithmen selbst eingegangen werden kann, muss zuerst auf einige Grundlagen näher eingegangen werden. Keiner der in dieser Arbeit genannten Algorithmen kann direkt mit der Geometrie eines Gamelevels arbeiten. Ein Grund ist, dass der Dijkstra Algorithmus, die Grundlage des Pathfinding in Games, aus der mathematischen Graphentheorie stammt, näheres dazu im Kapitel 3 Dijkstra.

Algorithmen wie Dijkstra und A* arbeiten mit einer stark vereinfachten Darstellung des Gamelevels, einem Graphen. Genauer gesagt einem gerichteten, nicht-negativ gewichteten Graphen. Pathfinding Algorithmen arbeiten also unabhängig von geometrischen Darstellungen, daher sind sie auf vollkommen unabhängig davon ob ein Gamelevel mit aufwändiger 3D-Grafik oder schlichten 2D Sprites realisiert wurde. Das Ergebnis des Pathfinding muss dementsprechend wieder in die Level-Darstellung übersetzt werden, näheres dazu später.

2.1 Graphen

Der Graph ist eine abstrakte, mathematische Struktur bestehend aus zwei verschiedenen Elementen: den Knoten, oft als Kreis oder Punkt dargestellt und deren Verbindungen oder auch Kanten, die mit Linien zwischen den Knoten dargestellt werden (vgl. [1]).

Jeder Knoten repräsentiert eine gewisse Region des Gamelevels, z.B. einen Raum, einen Abschnitt in einem Gang, eine Kreuzung in einem Labyrinth oder ein kleines Areal in einem

Außenlevel. Jeder Knoten muss über zumindest eine Verbindung erreicht werden können. Der gesuchte Pfad wird dabei durch die Kanten, ausgehen vom Startpunkt bis zum Zielpunkt, gebildet.

2.1.1 Gewichtete Graphen

Als gewichtete Graphen bezeichnet man Graphen die zusätzlich an den Kanten mit einem Zahlenwert versehen sind. Dieser Wert wird in der mathematischen Graphentheorie als Gewicht bezeichnet. Im Zusammenhang mit Pathfinding bezeichnet man den Wert eher als Kosten. Der Wert repräsentiert vereinfacht gesagt die Distanz zwischen den verbundenen Knoten oder auch die Zeit die es dauert von Knoten A nach Knoten B zu gelangen oder aber eine Kombination dieser Faktoren. Repräsentiert ein Knoten den Beginn eines steilen Felshanges, so ist die Distanz zum nächsten Knoten, der das obere Ende repräsentiert, zwar gering, es wird aber länger dauern diesen zu überwinden. Repräsentieren die Knoten hingegen die beiden Enden eines langen, geraden Weges, so ist die Distanz groß, der Zeitfaktor aber geringer als in unwegsamem Gelände.
Es gibt verschiedene Methoden die Kosten einer Kante zu bestimmen.

Nicht-negativ gewichtet

In der mathematischen Graphentheorie sind negative Gewichte erlaubt. Im Pathfinding in Videospielen macht das aber wenig Sinn. Es gibt keine negative Distanz zwischen zwei Orten oder eine negative Zeit die es dauert eine Distanz zu überwinden.
Auch wenn man einen negativ gewichteten Graphen erzeugen würde, Algorithmen die damit umgehen könnten, sind weitaus komplexer. Dijkstra und A* sollten nicht mit solchen Graphen arbeiten.
„In the majority of cases, however, Dijkstra and A* would go into an infinite loop. This is not an error in the algorithms. Mathematically, there is no such thing as a shortest path across many graphs with negative weights; a solution simply doesn't exist." (siehe [2], S 201 f)

2.1.2 Gerichtete Graphen

Der gewichtete, nicht gerichtete Graph allein reicht noch nicht aus um ein Gamelevel glaubhaft abzubilden. In Abbildung 2.1 macht es keinen Unterschied ob man von Knoten A nach Knoten B geht oder umgekehrt, die Kosten bei einem nicht gerichteten Graphen sind in beide Richtungen dieselben. Das macht auch soweit Sinn wenn die Knoten zwei Räume repräsentieren, der Aufwand von einem Raum in den anderen zu gelangen ist immer derselbe. Für einen komplexeren Ansatz benutzt man gerichtete Graphen, in diesem Ansatz wird davon ausgegangen, dass eine Kante und die dazugehörigen Kosten nur in eine Richtung gelten. Von Knoten A nach Knoten B ist somit nicht dieselbe Verbindung wie von Knoten B nach Knoten A. Für den Hin- und Retourweg sind also gegebenenfalls zwei Verbindungen notwendig.

Im Beispiel mit den Räumen reicht eine Verbindung. Anders sieht es z.B. mit einem Felsen aus. Der Sprung von einem Felsen entspricht Kosten von nahezu Null (von A nach B), das heißt aber nicht dass man auch so einfach wieder nach oben kommt. Von B nach A existiert vielleicht gar keine Verbindung, oder eine zwei Verbindung mit anderen Kosten ist vorhanden.

2.2 Repräsentation

Mit der oben beschriebenen visuellen Darstellung können die Pathfinding Algorithmen aber noch nicht arbeiten. Der Graph muss durch eine geeignete Datenstruktur repräsentiert werden. [2] beschreibt folgendes Pseudocode Interface:

```
class Graph:
    # Returns an array of connections (of class
    # Connection) outgoing from the given node
    def getConnections(fromNode)

class Connection:
    # Returns the non-negative cost of the
    # connection
    def getCost()

    # Returns the node that this connection came
    # from
    def getFromNode()

    # Returns the node that this connection leads to
    def getToNode()
```

Die Graph Klasse liefert einfach eine Auflistung aller Verbindungen zu einem bestimmten Knoten. Von jeder Verbindung können dann Kosten, sowie der Knoten von dem sie ausgeht und der Knoten zu dem sie hinführt, abgefragt werden. Dieses Interface dient nur als Richtlinie und ist keines Falls die einzige mögliche Repräsentation des Graphen. Je nach Implementierung können Änderungen und Optimierungen erforderlich sein. Auch die Wahl der Datentypen und –strukturen hängt von der Implementierung ab.

3 Dijkstra

Der Dijkstra Algorithmus ist eine wichtige Grundlage im Pathfinding. Obwohl er nicht dafür entwickelt wurde, kann er trotzdem dafür verwendet werden, außerdem bildet er die Basis für

den A*. Der Algorithmus wurde für das mathematische Problem „kürzester Pfad“ entwickelt. Die Lösung des Dijkstra sind die kürzesten Pfade zu jedem Knoten innerhalb eines Graphen, ausgehen von einem Startknoten.

Die Lösung die im Pathfinding gesucht wird, der kürzeste Weg von Start zum Ziel, ist also eine Teillösung des Dijkstra.

Die Verwendung des Dijkstra im Pathfinding bringt aber einig Probleme. um einen ist es Verschwendung die gefundenen, aber nicht benötigten Pfade zu verwerfen, zum anderen ist der Dijkstra auch nach Anpassung und Optimierung immer noch äußerst ineffizient. Daher ist er in der Produktion nicht zu finden (vgl. [2], S 204 f).

3.1 Problemstellung

Gegeben sind ein gerichteter, nicht-negativ gewichteter Graph sowie ein Start- und ein Zielknoten. Gesucht ist der kürzeste Pfad zwischen den beiden Knoten. Der kürzeste Pfad ist jener, dessen Gesamtkosten kleiner sind als die Gesamtkosten aller anderen Pfade. Möglicherweise gibt es mehr als einen Pfad mit den minimalen Gesamtkosten, als Ergebnis wird aber nur einer davon erwartet.
Das Ergebnis ist eine Auflistung der betroffenen Verbindungen nicht der Knoten, da, wie in Kapitel 2.1.2 beschrieben, zwischen zwei Knoten mehrere Verbindungen existieren können.
„… it may be possible to either fall off a walkway or climb down a ladder […]. We therefore need to know which connections to use; a list of nodes will not suffice.“ (siehe [2], S 205)

3.2 Funktionsweise

Grundlegend kann das Funktionsprinzip des Dijkstra folgendermaßen beschrieben werden: Ausgehend vom Startpunkt durchläuft der Algorithmus immer weiter entferntere Knoten und merkt sich dabei von wo er gekommen ist. Am Ziel angekommen kann er die Referenzen bis zum Startpunkt zurückverfolgen. Während der Algorithmus die Knoten durchläuft, stellt er sicher, dass die Referenz immer auf den kürzesten Pfad zeigt.

Der Algorithmus arbeitet in Iterationen. In jeder dieser Iteration wird ein Knoten verarbeitet. Ausgehend vom aktuellen Knoten werden alle abgehenden Verbindungen untersucht. Die zugehörigen Endknoten werden erfasst und die bisherigen Kosten werden vermerkt.

Die bisherigen Kosten eines Endknotens ergeben sich aus den bisherigen Kosten des aktuellen Knotens und den Kosten der Verbindung. Die Knoten die der Dijkstra bearbeitet werden in zwei Listen verwaltet: Offen und Geschlossen. In der Offen-Liste werden alle Knoten gespeichert die erfasst, deren Kosten aber in einer eigenen Iteration noch nicht erfasst wurden. In der Geschlossen-Liste werden alle Knoten abgespeichert die schon verarbeitet wurden.

Zu Beginn der Suche enthält die Offen-Liste nur den Startknoten mit den Kosten 0 und die Geschlossen-Liste ist leer. Als aktueller Knoten wird immer jener aus der Offen-Liste gewählt, der die niedrigsten bisherigen Kosten hat. Nachdem er wie oben beschrieben verarbeitet wurde, wird er aus der Offen-Liste gelöscht und in die Geschlossen-Liste eingetragen.

Wenn die Verbindungen des aktuellen Knoten geprüft werden, wird ein noch nicht besuchter Knoten erwartet. Was aber wenn der Knoten offen oder geschlossen ist?

Solche Knoten haben schon einen Wert für bisherige Kosten, eine normale Iteration würde diesen Wert überschreiben. Die neu errechneten bisherigen Kosten müssen mit den bereits eingetragenen verglichen werden. Ist der neue Wert höher (das wird fast immer der Fall sein) passiert gar nichts, der Wert wird nicht aktualisiert und der Knoten bleibt in der Liste in der er sich gegenwärtig befindet. Ist der neue Wert aber niedriger, werden die bisherigen Kosten aktualisiert, außerdem wird der Knoten gegebenenfalls in die Offen-Liste verschoben.

Der Dijkstra Algorithmus wird niemals einen kürzeren Pfad zu einem geschlossenen Knoten finden. Auf die Berechnung der bisherigen Kosten kann in diesem Fall also verzichtet werden.

Per Definition terminiert der Dijkstra erst dann, wenn die Offen-Liste leer ist. Im Pathfinding ist aber nur der Pfad zum Zielknoten relevant. Die Dijkstra Implementierung kann also terminieren sobald der Zielknoten der Knoten mit den geringsten Kosten in der Offen-Liste ist. Würde der Algorithmus terminieren sobald er den Zielknoten erreicht, könnte ein noch kürzerer Pfad übersehen werden.

In der Praxis brechen Implementierungen aber dennoch ab sobald der Zielknoten erreicht wurde, da der erste gefundene Pfad meist der kürzeste ist und selbst wenn ein noch kürzerer existieren sollte, ist dieser nur minimal kürzer.

Ausgehend vom Zielknoten wird dann die Verbindung, über die er erreicht wurde, zum vorherigen Knoten zurückverfolgt. Die Verbindung wird gespeichert und der Vorgang wird wiederholt bis der Startknoten erreicht wurde. Die Liste der gespeicherten Verbindungen entspricht dem gesuchten Pfad, allerdings in der verkehrten Reihenfolge. Die Liste muss nur noch umgekehrt werden.

3.2.1 Beispiel

Das folgende Schritt für Schritt Beispiel soll die genaue Funktionsweise des Dijkstra Algorithmus veranschaulichen. Gegeben sei ein gerichteter, nicht-negativ gewichteter Graph (Abb. 3.1) mit dem Startknoten A und dem Zielknoten F.

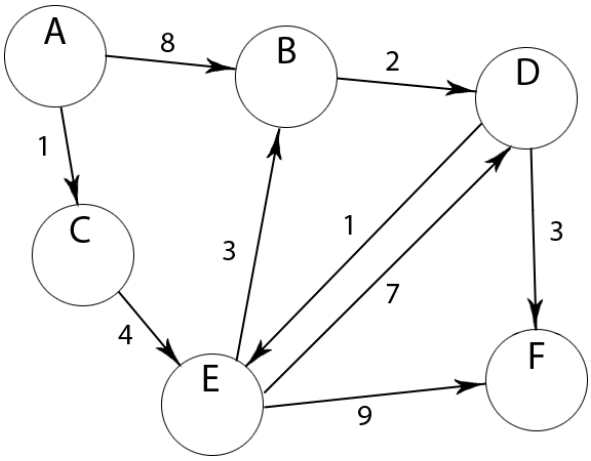

Abbildung 3.1: gegebener Graph mit Kantengewichtung

Im ersten Schritt wird der Startknoten in die Offen-Liste eingetragen und zum aktuellen Knoten gemacht (Tab. 3.1). Die bisherigen Kosten für den Startknoten sind immer 0, dieser Wert wird eingetragen. Danach werden die von A ausgehenden Verbindungen untersucht. Die Knoten B und C werden gefunden und in die Offen-Liste eingetragen.

Aktueller Knoten	Offen-Liste	Geschlossen-Liste
A	A(0)	

Tabelle 3.1: Listen vor der ersten Iteration

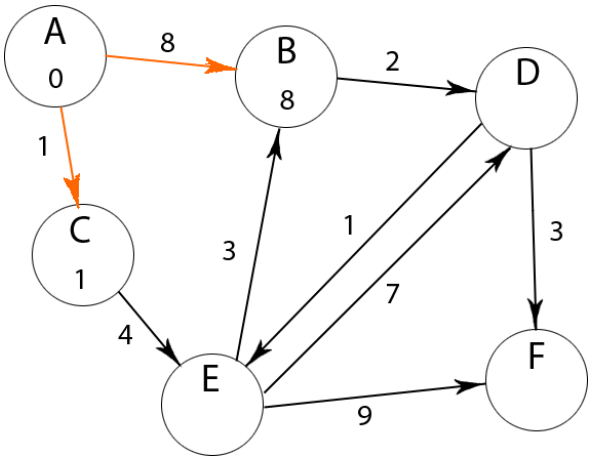

Abbildung 3.2: Knoten B und C gefunden

Die Kosten werden berechnet (Abb. 3.2):

Kosten B = bisherige Kosten A + Kosten Verbindung A → B

Kosten B = 0 + 8 = 8

Kosten C = bisherige Kosten A + Kosten Verbindung A → C

Kosten C = 0 + 1 = 1

Der Knoten A ist fertig verarbeitet und wird in die Geschlossen-Liste verschoben. Im nächsten Schritt wird die Offen-Liste nach dem Knoten mit den niedrigsten Kosten durchsucht (Knoten C) und dieser zum aktuellen Knoten gemacht (Tab. 3.2).

Aktueller Knoten	Offen-Liste	Geschlossen-Liste
C	B(8), C(1)	A

Tabelle 3.2: C ist jetzt der aktuelle Knoten

Von Konten C ausgehend wird der Knoten E gefunden, in die Offen-Liste eingetragen und seine Kosten berechnet (Abb. 3.3):

Kosten E = bisherige Kosten C + Kosten C → E = 1 + 4 = 5

Knoten C wird in die Geschlossen-Liste verschoben.

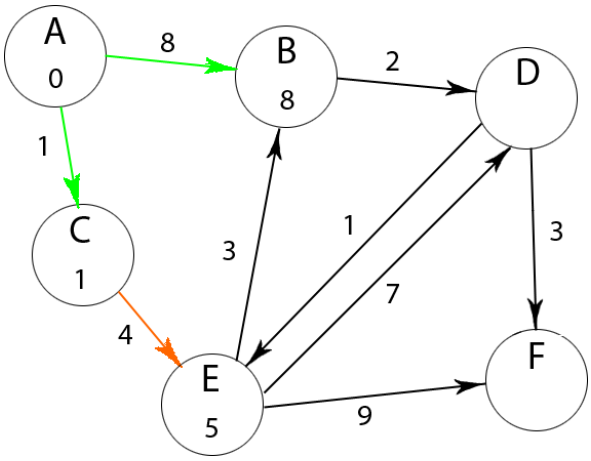

Abbildung 3.3: Eintrag für Knoten E

Jetzt wird erneut nach dem Knoten mit den niedrigsten Kosten in der Offen-Liste gesucht (Tab. 3.3).

Aktueller Knoten	Offen-Liste	Geschlossen-Liste
E	B(8), E(5)	A, C

Tabelle 3.3: Die Listen während der 3. Iteration

Ausgehend von E werden die Knoten B. D und F gefunden. B befindet sich schon in der Offen-Liste, die Berechnung der Kosten entscheidet ob B aktualisiert werden muss:

Kosten B = bisherige Kosten E + Kosten E → B = 5 + 3 = 8

Kosten D = bisherige Kosten E + Kosten E → D = 5 + 7 = 12

Kosten F = bisherige Kosten E + Kosten E → F = 5 + 9 = 14

Der neue Pfad zu B hat dieselben Kosten wie der alte Pfad, B muss also nicht aktualisiert werden (Abb. 3.2).

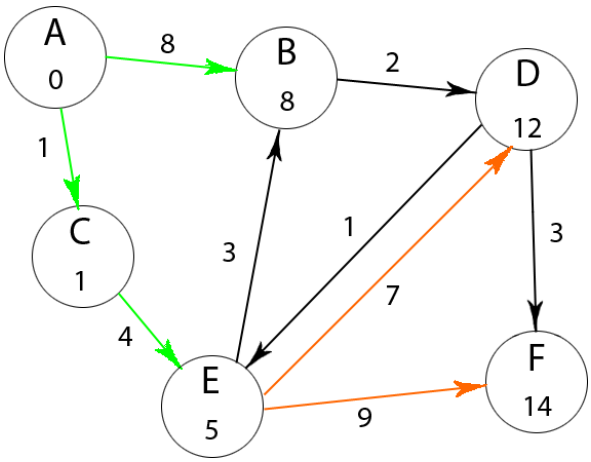

Abbildung 3.4: keine Aktualisierung von B

Jetzt ist B die Knoten mit den geringsten Kosten. Der Algorithmus läuft also einen Pfad nicht bis zum Ende durch sondern verarbeitet den bisher kürzesten Pfad. Auch ist der Algorithmus noch nicht zu Ende, obwohl der Zielknoten erreicht wurde, es kann ein kürzerer Pfad existieren (Tab. 3.4).

Aktueller Knoten	Offen-Liste	Geschlossen-Liste
B	B(8), D(12), F(14)	A, C, E

Tabelle 3.4: B als aktueller Knoten

In der Iteration von Knoten B wird der Knoten D aktualisiert da die Kosten über B geringer sind als über E (Abb. 3.5):

Kosten D = bisherige Kosten B + Kosten B → D = 8 + 2 = 10

Auch die Referenz von D auf E wird in B geändert.

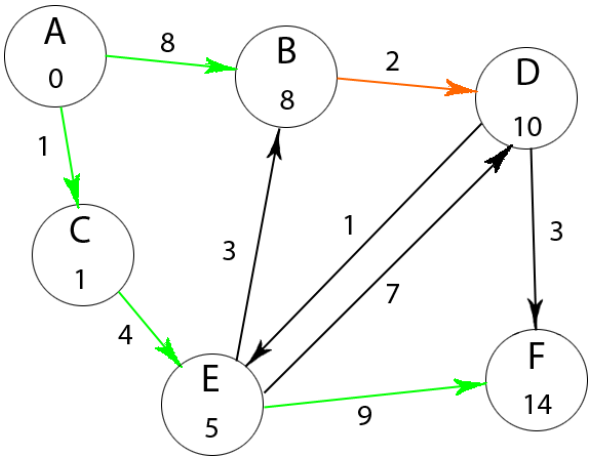

Abbildung 3.5: Aktualisierung von D

Aktueller Knoten	Offen-Liste	Geschlossen-Liste
D	D(10), F(14)	A, C, E, B

Tabelle 3.5: Der aktualisierte Knote D als aktueller Knoten

Von D aus existiert zwar eine Verbindung auf E, dieser Knoten muss aber nicht aktualisiert werden, da die Kosten höher sind als die bisherig berechneten:

Kosten E = bisherige Kosten D + Kosten D → E = 10 + 1 = 11

Anders sieht es mit dem Zielknoten F aus, die neuen Kosten sind niedriger als die bisherigen, es führt also ein noch kürzerer Pfad zum Ziel (Abb. 3.6):

Kosten F = bisherige Kosten D + Kosten D → F = 10 + 3 = 13

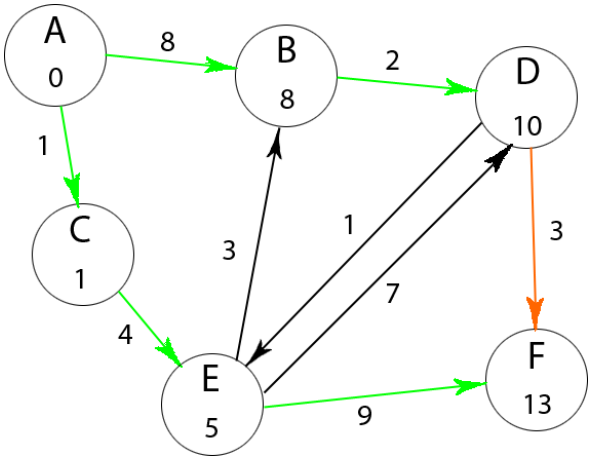

Abbildung 3.6: ein neuer, kürzerer Pfad zu F wurde gefunden

Die Zielknoten ist der einzige der noch auf der Offen-Liste steht (Tab. 3.6), von ihm gehen auch keine Verbindungen mehr aus. Gäbe es noch welche würde der eigentliche Dijkstra noch weiterlaufen, spätestens hier brechen die üblichen Implementierungen ab.

Aktueller Knoten	Offen-Liste	Geschlossen-Liste
F	F(13)	A, C, E, B, D

Tabelle 3.6: der Zielknoten als letzter Knoten in der Offen Liste

Das Beispiel zeigt auch dass der Dijkstra zu jedem Konten den kürzesten Pfad liefert (Abb. 3.7, grüne Pfeile).

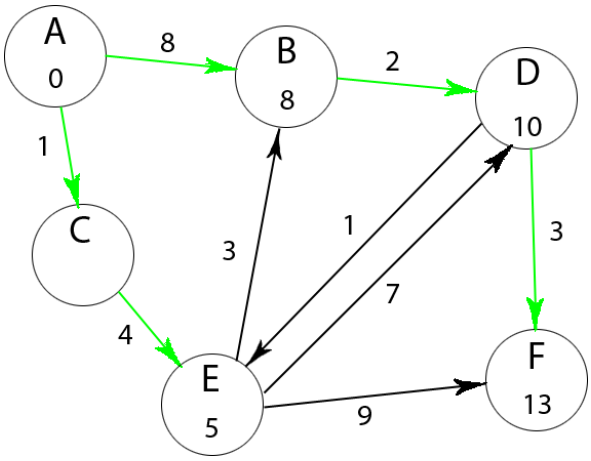

Abbildung 3.7: das Ergebnis des Dijkstra

3.3 Datenstrukturen

Drei wichtige Datenstrukturen werden für den Algorithmus verwendet, eine einfache Liste die den gefundenen Pfad repräsentiert, sie ist nicht sehr performance-kritisch, da sie erst am Ende des Algorithmus gebraucht wird. Für diese Liste reicht eine einfache Datenstruktur wie eine verkettete Liste (z.B. std::list in C++) oder ein erweiterbares Array (std::vector in C++).
Bei den anderen beiden Datenstrukturen handelt es sich um die beiden Listen für die Offen- und Geschlossen-Liste. Diese beiden Listen sind kritische Strukturen und haben einen großen Einfluss auf die Performance des Dijkstra. Die kritischen Operationen sind:

- einen Eintrag in die Liste einfügen
- einen Eintrag entfernen
- das kleinste Element finden
- ein bestimmtes Element finden

Standard Datenstrukturen werden hier nicht ausreichen, da sie nicht genügend Möglichkeiten zur Optimierung bieten.

Eine Pseudo-Code Beispielimplementierung ist im Anhang A zu finden.

3.4 Performance

Der Algorithmus verarbeitet jeden Knoten der näher liegt als der Zielknoten. Diese Anzahl sei n. Die innere Schleife verarbeitet jede ausgehende Verbindung für jeden dieser Knoten. Die durchschnittliche Anzahl an ausgehenden Verbindungen sein m. Daraus ergibt sich ein Arbeitsaufwand von O(nm).

Terminiert der Algorithmus befinden sich n Knoten in der Geschlossen Liste und weniger als nm Knoten in der Offen-Liste (in der Regel weniger als n). Der Speicherverbrauch ist daher im schlimmsten Fall O(nm).

Wie oben erwähnt sind die Listenoperationen die aufwändigsten. Das Hinzufügen und das Finden wird nm mal ausgeführt, das Entfernen und die Suche nach dem kleinsten Element erfolgt n mal (vgl. [2], S 214).

3.5 Probleme

Der Dijkstra Algorithmus wurde entworfen die kürzesten Pfad zu jedem Knoten zu finden. Genau das macht ihn ineffizient für Punkt-zu-Punkt Pfade. Selbst ein früheres Terminieren des Algorithmus verhindert nicht, dass beinahe alle Knoten verarbeitet werden. Das Verarbeiten kostet Zeit und Speicher. Diese Nachteile wiegen im Game-Development schwer, daher wird der Dijkstra kaum in der Praxis eingesetzt. Aus ihm geht aber der A* hervor.

4 A*

Der A* ist das Synonym für Pathfinding im Game-Development. Er ist der am weitesten verbreitete Algorithmus, da er einfach zu implementieren und effizient ist. Weiters bietet er viel Platz für Optimierungen und ist einfach zu erweitern, um auch komplexere Fälle zu behandeln.

„Every pathfinding system we've come across in the last 10 years has used some variation of A* as its key algorithm, …“ siehe [2], S 215)

Bekanntestes Beispiel dafür ist die Path Engine, sie wird unter anderem von CDProjekt in der „The Witcher“ Reihe und von NCSoft für „Guild Wars 2“ benutzt.

4.1 Unterschiede zum Dijkstra

Der A* funktioniert ähnlich wie der Dijkstra, aber anstatt den Knoten mit den geringsten bisherigen Kosten zu wählen, wählt der A* den Knoten der am ehesten zum Ziel führt. Dieser Wert wird Heuristik genannt. Die Heuristik ist nur ein Schätzwert und gibt nicht die genaue Distanz zum Zielknoten an. Diese muss ja erst errechnet werden. Ist die Heuristik gut gewählt ist der A* äußerst effizient, ist sie schlecht gewählt dann kann der Aufwand noch höher als beim Dijkstra sein.

Der A* steht und fällt also mit der Heuristik, näheres dazu in Kapitel 4.2 Heuristik.

Der Knoten muss einen weiteren Wert speichern, die geschätzten Gesamtkosten. Diese errechnen sich aus den bisherigen Kosten plus den geschätzten, noch anfallenden Kosten, dem heuristischen Wert. Das unten stehende Beispiel soll dies verdeutlichen.

Ein weiterer, sehr wichtiger Unterschied zum Dijkstra ist, dass der A* sehr wohl einen besseren Pfad zu einem bereits verarbeiteten Knoten finden kann. Wo der Dijkstra Knoten in der Geschlossen-Liste nicht aktualisiert, macht der A* das sehr wohl. Das alleine reicht aber noch nicht. Wird nur der Knoten aktualisiert stimmen die Pfade die auf ihn beruhen nicht mehr. Das heißt auch die von ihm ausgehenden Verbindungen müssen neu verarbeitet werden. Um das zu erreichen wird der aktualisierte Knoten wieder in die Offen-Liste verschoben.

4.1.1 Beispiel

Das Beispiel soll die Funktionsweise des A* verdeutlichen. Die Knoten haben Werte für die Heuristik (h), die bisherigen Kosten (g) sowie den geschätzten Gesamtkosten (f). Die Verbindungen weisen weiterhin ihre Kosten (c) auf.

Gegeben ist ein Graph, die Verbindungen der Knoten gehen in beide Richtungen, sind wegen der Übersichtlichkeit aber als eine Verbindung dargestellt (Abb. 4.1).

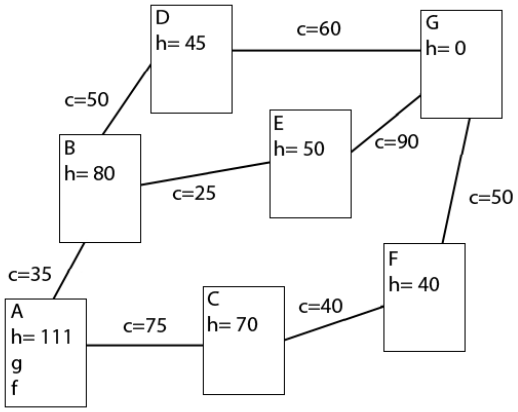

Abbildung 4.1: der gegebene Graph

Der Startknoten wird zuerst verarbeitet, die bisherigen Kosten betragen 0, daher ergibt sich für die geschätzten Gesamtkosten der Wert der Heuristik. Anhand der ausgehenden Verbindungen werden die Knoten B und C gefunden. Deren geschätzte Gesamtkosten berechnen sich folgendermaßen (Abb. 4.2):

$f_B = g_A + c_{AB} + h_B = 0 + 35 + 80 = 115$

$f_C = g_A + c_{AC} + h_C = 0 + 75 + 70 = 145$

Die geschätzten Gesamtkosten werden eingetragen, auch die bisherigen Kosten werden ermittelt, die errechnen sich aus den bisherigen Kosten des Vorgängerknotens und den Kosten der Verbindung.

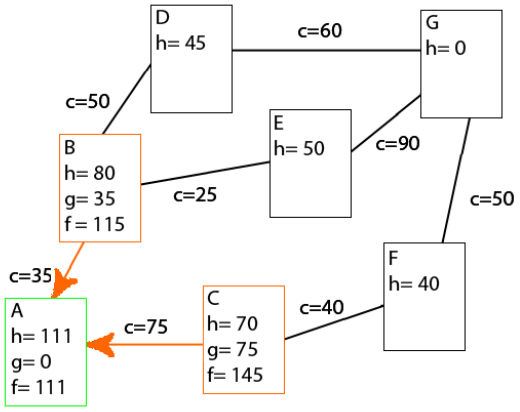

Abbildung 4.2: Verarbeitung des Startknotens

Der Startknoten ist somit abgearbeitet und kommt auf die Geschlossen Liste.
Der A* berücksichtigt bei der Wahl des nächsten Knotens nicht den Knoten mit den geringsten bisherigen Kosten, sondern denjenigen mit den geringsten noch zu erwartenden Kosten. Als nächster Knoten wird B gewählt und die Knoten D und E gefunden und in die Offen-Liste eingetragen (Abb. 4.3):

$f_D = g_B + c_{BD} + h_D = 35 + 50 + 45 = 130$

$f_E = g_B + c_{BE} + h_E = 35 + 25 + 50 = 110$

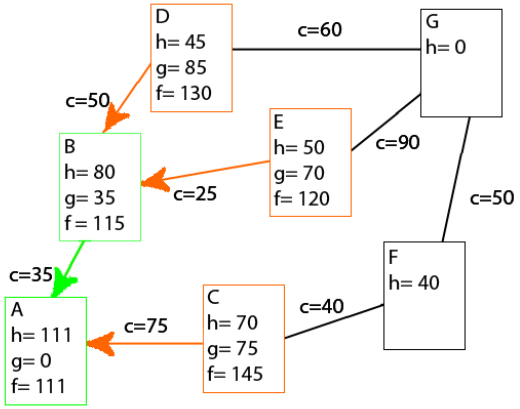

Abbildung 4.3: weitere Knoten werden gefunden

Der Zielknoten G wird gefunden und mit den folgenden Werten in die Offen-Liste eingetragen (Abb. 4.4):

$f_G = g_E + c_{EG} + h_G = 70 + 90 + 0 = 160$

Da ein kürzerer Pfad existieren könnte terminiert der A* hier noch nicht, gleich dem Dijkstra.

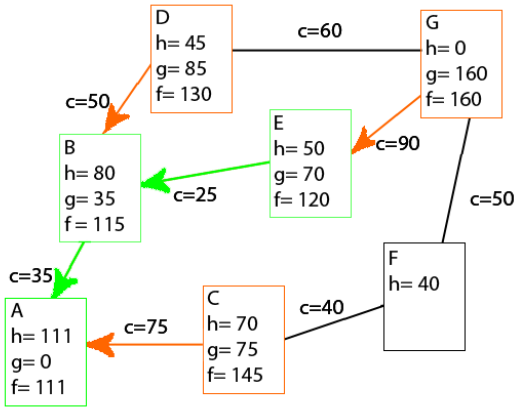

Abbildung 4.4: erstes Verarbeiten des Zielknotens

In der nächsten Iteration wird D mit den geringsten zu erwartenden Kosten gewählt. Von ihm aus existiert ebenfalls eine Verbindung zum Knoten G. Die Kosten für G müssen also erneut berechnet werden.

$$f_G = g_D + c_{DG} + h_G = 85 + 60 + 0 = 145$$

Der neue geschätzte Gesamtwert ist niedriger als der zuvor errechnete. Der Knoten G wird also aktualisiert und die Referenz wird von E auf D geändert (Abb. 4.5).
Danach wird der Knoten C gewählt und verarbeitet (Abb. 4.6).

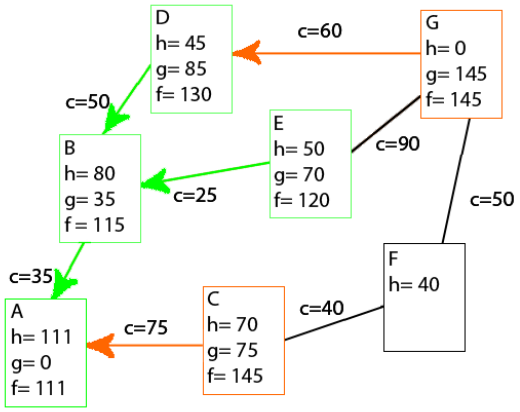

Abbildung 4.5: Knoten G wird aktualisiert

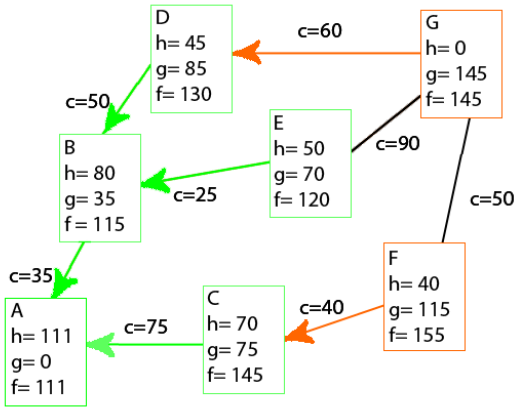

Abbildung 4.6: A* sucht nach einem möglichen kürzeren Pfad

Der Zielknoten ist jetzt der Knoten mit den geringsten Kosten in der Offen-Liste, der A* Terminiert. Ausgehend vom Zielpunkt kann jetzt der Pfad rekonstruiert werden (Abb. 4.7).

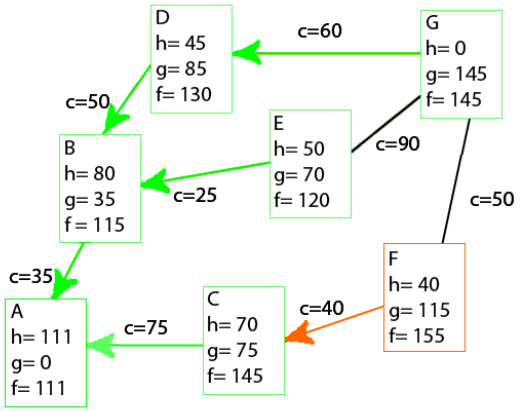

Abbildung 4.7: G ist der Knoten mit den geringsten Kosten

4.2 Heuristik

Wie das obige Beispiel zeigt, hat die Heuristik keinerlei Einfluss auf die eigentlichen Kosten des Pfades. Der heuristische Wert fließt lediglich in die geschätzten Gesamtkosten ein, die lediglich für die Auswahl des nächsten Knotens berücksichtigt werden.

Die Heuristik ist ein kritischer Punkt im Algorithmus, da es sich um einen Schätzwert handelt, muss dahinter ein Algorithmus stecken der einen enormen Aufwand bedeuten kann. Wird dies nicht bedacht, helfen alle Optimierungsversuche des A* nichts.

Die optimale Heuristik wäre die kürzeste Distanz zwischen zwei Knoten, der Aufwand des A* würde sich auf O(p), wobei p für die Anzahl der Schritte steht, belaufen. Aber die kürzeste Distanz zwischen zwei Knoten ist genau das was der A* liefern soll.

Unterschätzt die Heuristik die aktuelle Pfadlänge läuft der A* länger. Knoten die näher am Start liegen werden eher bevorzugt. Das führt dazu dass der A* zwar den kürzesten Pfad findet, der Ablauf aber genau dem des Dijkstra entspricht.

Ein Überschätzen führt dazu, dass der Algorithmus nicht den optimalen Pfad findet. Er bevorzugt den Pfad mit weniger Knoten, selbst wenn die Verbindung zwischen den Knoten sehr hohe Kosten haben. Somit kann ein Pfad mit viel geringeren Kosten übersehen werden. In der Praxis hat sich die Euklidische Distanz bewiesen. Sie gibt nichts anderes an als die direkte Distanz in Luftlinie, unabhängig von irgendwelchen Hindernissen. Die euklidische Distanz ist immer unterschätzt oder genau, das eignet sich vor allem für weitläufige Außenareale in denen praktisch jeder Punkt erreichbar ist. In Innenarealen mit vielen Hindernissen und Sackgassen kann es zu einer höheren Laufzeit kommen und beinahe alle Knoten werden verarbeitet.

4.3 Datenstrukturen

Die Datenstrukturen des Graphen sind identisch mit denen die der Dijkstra benutzt. Eine kleine Änderung ist nur bei der Methode `smallestElement` notwendig. Bisher lieferte sie den Knoten mit den geringsten bisherigen Kosten. Für den A* wird der Knoten mit den geringsten geschätzten Gesamtkosten gesucht.

Wie schon beim Dijkstra sind die Offen- und Geschlossen-Liste die Datenstrukturen die das größte Optimierungspotential. Hier kann man die verschiedensten Ansätze verfolgen, hier sind einige Beispiele angeführt.

4.3.1 Priority Queue

Die Priority Queue ist der einfachste Ansatz. Es handelt sich um eine verkettete Liste oder ein Array. Diese Datenstrukturen sind sortiert, d.h. nach dem kleinsten Element muss nicht gesucht werden, es befindet sich an erster Stelle. Die geringe Performance beim Suchen wird durch ein teures Einfügen bezahlt. Gerade bei einer verketteten Liste muss beim Einfügen so lange gesucht werden, bis ein größeres Element gefunden wurde. Für das Array kann zwar die binäre Suche verwendet werden, allerdings erfordert das Einfügen das Verschieben der bisherigen Elemente.

4.3.2 Priority Heap

Der Priority Heap ist nichts anderes als ein Array das eine Baumstruktur abbildet. Jedes Element im Baum kann bis zu zwei Kindelemente haben, beide müssen größer als ihr Elternelement sein. Das Array wird ebenen-weise, von links nach rechts befüllt (Abb. 4.8).

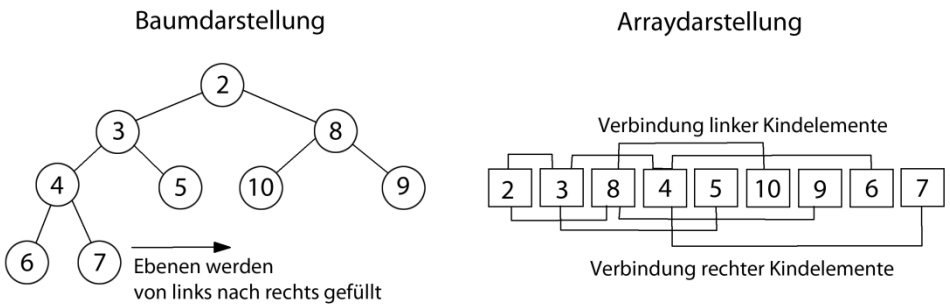

Abbildung 4.8: Priority Heap ([1], S. 225)

Dies erlaubt die Abbildung eines Baumes auf ein einfaches Array. Die Kindelemente können einfach durch *2i* und *2i+1* gefunden werden. Das kleinste Element zu entfernen oder ein neues hinzuzufügen hat hierbei den Aufwand O(log n).

4.3.3 Bucketed Priority Queues

Die bucketed Priority Queue ist eine teilweise Sortierte Datenstruktur. In einer einfachen Liste sind sogenannte Buckets sortiert. Die Buckets repräsentieren einen gewissen Wertebereich und sind selbst wiederum kleine Listen deren Elemente aber unsortiert sind (Abb. 4.9). Um das kleinste Element zu finden wird einfach im ersten nicht leeren Bucket nach dem kleinsten Element gesucht. Hinzugefügt wird ein Element einfach am Ende des entsprechenden Buckets.

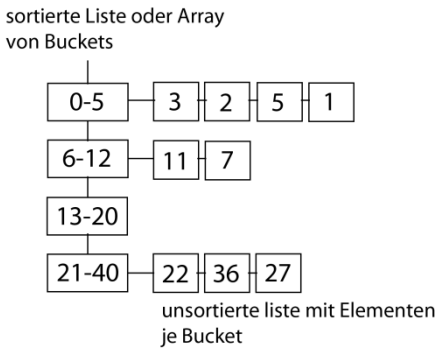

Abbildung 4.9: bucketed Priority Queue ([1], S. 226)

Wie immer gibt es keine optimale Lösung. Die Wahl der Datenstruktur hängt von vielen Faktoren wie Speichergröße und –geschwindigkeit der Zielplattform, Performanceanforderung des Pathfinding ab.

4.4 Performance

Anders als der Dijkstra hängt die Performance nicht von der Gesamtanzahl der Knoten n, sondern der Anzahl Knoten l mit geringeren geschätzten Pfadkosten als dem Zielknoten ab. Die innere Schleife hat eine ähnliche Komplexität wie die des Dijkstra, der Aufwand des A* ergibt daher O (lm).

Die gewählten Datenstrukturen haben einen erheblichen Einfluss auf die Performance, allerdings darf die Berechnung der Heuristik nicht vergessen werden.

4.5 Probleme

Probleme ergeben sich vor allem durch eine schlecht gewählte Heuristik, diese führt zu einer zu hohen Füllrate des Algorithmus. Diese kann ingame oder aber auch durch Statistikausgaben untersucht und die Heuristik optimiert werden. Eine optimale, einheitliche Lösung kann nicht geboten werden, da die Füllrate auch stark vom Leveldesign, und der Umsetzung in einen Graphen, abhängt.
Weitere Probleme des A* werden in den Kapiteln 5 und 6 aufgeführt.

5 Dynamisches Pathfinding

Obwohl Dijkstra und A* den optimalen Pfad finden, haben sie doch einen großen Nachteil: Sie funktionieren nur mit einem statischen Graphen. Wenn sich das Level und somit der Graph ändert, ist die Arbeit der Algorithmen hinfällig und sie müssen von vorne beginnen. Dynamische Ansätze sind vor allem in Echtzeit-Umgebungen wie Shootern, RTS und RPGs unerlässlich. Wird für eine Einheit ein Pfad über eine Brücke geplant und diese Brücke wird währenddessen gesprengt ist der Pfad unbrauchbar.

Der erste Ansatz wäre, den A* jeden Frame erneut aufzurufen. Der initiale Aufruf liefert einen Pfad zurück. Die Einheit bewegt sich nun zum ersten Punkt in der Pfadliste und der A* wird erneut aufgerufen und die Einheit bewegt sich wieder zum nächsten Punkt, solange bis das Ziel erreicht wird. Das Problem hierbei ist die Effizienz. Das Resultat, das Frame für Frameberechnet wird, ist immer derselbe, um das schon zurückgelegte Stück verkürzte Pfad. Erst wenn ein Hindernis auftaucht oder ein Wegstück unpassierbar wird, ändert sich der Pfad. Eine Lösung für dieses Problem lieferte Anthony Stentz mit dem D* (dynamic A*).

5.1 D*

Der D* beruht auf dem A*. Er wurde für Roboter entwickelt, die sich durch unbekanntes Terrain bewegen sollen. Er liefert Anfangs den Pfad und die Einheit beginnt sich entlang des Pfades. Taucht ein Hindernis auf wird die Karte, respektive der Graph aktualisiert und ein neuer Pfad, ausgehend von den aktuellen Koordinaten zum Ziel ausgerechnet.

5.1.1 Funktionsprinzip

Gleich wie beim A* kann ein Knoten unentdeckt, offen oder geschlossen sein. Zusätzlich kann ein Knoten den Zustand „Erhöht“ haben, das bedeutet seine Kosten sind höher als das letzte Mal als er in der Offen Liste war; der zweite Zustand ist „Gesenkt“, der anzeigt, dass die Kosten jetzt niedriger sind. Im Gegensatz zum A* beginnt der D* vom Zielpunkt aus den Startpunkt zu suchen. Das verarbeiten der Knoten wird im Zusammenhang mit D* expandieren genannt. Es wird wieder ein Knoten von der Offen Liste gewählt und seine Kosten berechnet. Die Änderungen werden an die Nachbarknoten übermittelt und diese in die Offen Liste eingetragen. Jeder Expandierte Knoten hat eine Referenz auf den nächsten Knoten in Richtung Ziel. Wurde der Startknoten expandiert terminiert der D*.

Hindernisse

Taucht ein Hindernis auf werden alle betroffenen Knoten erneut auf die Offen Liste gesetzt und mit dem Zustand „Erhöht“ markiert. Bevor die Kosten dieser Knoten erneut berechnet werden, prüft der D* die Nachbarknoten ob sie die Kosten senken können. Wenn das nicht der Fall ist, wird der „Erhöht“ Zustand an alle betroffenen Knoten weitergeleitet, also jenen Knoten die eine Referenz auf den erhöhten Knoten haben. Diese Knoten werden ebenfalls erhöht und leiten dies weiter, eine sogenannte „raise wave“ wird ausgelöst. Diese „Welle“ breitet sich so lange aus, bis sie Auf Knoten trifft, die die Kosten senken können. Der Zustand wird auf „Gesenkt“ gesetzt und zurückgeleitet, eine „lower wave“ wird ausgelöst.

Diese Wellen des Erhöhens und Senkens sind das Herzstück des D*. Erhöhen und senken betrifft nur bereits gefundene Knoten und verhindert das unnötige expandieren weiterer Knoten. Kann aber innerhalb des bereits expandierten Korridors kein alternativer Pfad gefunden werden, müssen neue Knoten expandiert werden.

6 Weitere Ansätze

6.1 A* Erweiterungen

Im Laufe der Jahre wurde der A* ständig weiterentwickelt und neue Abwandlungen geschaffen. So bietet der HPA* einen hierarchischen Ansatz, bei dem das Level in einzelne Grids unterteilt wird. In diesen Grids wird dann mittels A* ein Pfad hindurch gesucht. Die Ein- und Austrittspunkte werden dann zu einem neuen Graphen zusammengesetzt und durch diesen wird wieder mittels A* ein Pfad gesucht.

Eine weitere Abwandlung des A* ist der HAA*, der hierarchically annotated A* wurde Entwickelt um das Bewegen unterschiedlich großer Einheiten in RTS Games zu ermöglichen. Eine Einheit ist eine bestimmte Anzahl Felder groß, eine Einheit die 2x2 Felder misst, kann nicht durch einen Korridor der nur ein Feld breit ist. Um dieses Problem beim Pathfinding mit einzubeziehen werden den einzelnen Knoten sogenannte „clearance values“ zugewiesen. Diese Werte geben an, wie groß eine Einheit mindestens sein muss um diesen

Knoten passieren zu können. Dieser Wert wird beim berechnen des Pfades berücksichtigt und stellt sicher, dass der Pfad auch für die entsprechende Einheit gültig ist (vgl. [3]).

6.2 Navigation Meshes (NavMeshes)

Eine der wichtigsten Optimierungen im Pathfinding haben nichts mit den Algorithmen selbst zu tun. Sie betreffen die Repräsentation des Gamelevel. Bisher war immer nur von Knoten in einem Graphen die Rede. Diese Knoten repräsentieren sogenannte Wegpunkte im Gamelevel. Diese Wegpunkte müssen erst verteilt werden, dies geschieht entweder per Hand bei der Erstellung des Levels oder automatisch zur Laufzeit. Wegpunkte per Hand zu setzen ist ein enormer Arbeitsaufwand, das automatische Setzen geschieht mithilfe verschiedener mathematischer Algorithmen. Jede der Möglichkeiten bietet ihre eigenen vor und Nachteile aber ganz allgemein betrachtet haben Wegpunkte einige Schwachstellen die zwar keinen Einfluss auf die Pathfinding-Algorithmen selbst haben, aber die Qualität der fertigen AI stark beeinträchtigen können.
Folgende Probleme ergeben sich durch die Verwendung von Wegpunkten:

- Anzahl: Für eine ausreichende Repräsentation eines Gamelevels ist eine enorme Anzahl an Wegpunkten notwendig. Das sorgt zum einen für einen hohen Aufwand bei der Erstellung der Wegpunkte aber auch beim Suchalgorithmus selbst.
- Grobe Pfade: Die Pfade die durch die Wegpunkte markiert werden sind nicht sehr geschmeidig. Richtungsänderungen erfolgen abrupt und das Folgen des Pfades wirkt unnatürlich. Natürlich Kann der Pfad durch mathematische Verfahren feiner gemacht werden, aber dadurch entstehen neue Probleme. Was ist wenn die verfeinerte Kurve ein Hindernis schneidet oder die sanfte Richtungsänderung nicht mehr auf der schmalen Brücke liegt?
- keine Dynamische Pfadkorrekturen: Ein plötzlich auftauchendes Hindernis kann dazu führen dass ein Pfad nicht passierbar ist obwohl der Weg nur teilweise von einem Objekt verdeckt wird. Sind nicht genügend Informationen vorhanden (nicht genug Wegpunkte und Verbindungen) kann das dazu führen dass das Ziel gar nicht mehr erreichbar scheint, oder ein riesen Umweg berechnet werden muss.

Natürlich können die beiden zuletzt genannten Probleme durch das hinzufügen weiterer Wegpunkte oder zusätzlicher Verbindungen kompensiert werden, das verstärkt aber Problem Nummer 1.
Die Lösung bieten Navigation Meshes, kurz NavMeshes. NavMeshes sind nichts anderes als konvexe Polygone die Bereiche definieren in denen sich Charaktere bewegen können. Anstatt das Level durch Punkte und deren Verbindungen zu beschreiben, verwendet man die Flächen zwischen den Punkten. Das bietet eine Vielzahl von Vorteilen:

- Geringere Anzahl: Anstatt eine ebene Fläche (z.B. einen Platz) durch mehrere Wegpunkte zu definieren, reicht eine Fläche aus. Das senkt die Anzahl der Knoten im Graphen enorm und führt zu einem viel schnelleren Pathfinding.

- Pfadoptimierung: Die NavMeshes beschreiben begehbare Flächen, der gesuchte Pfad stellt viel öfter eine direkte Line vom Start zum Ziel dar. Außerdem können Pfade einfacher bearbeitet werden, da einfach überprüft werden kann ob das Endresultat immer noch auf einem gültigen Weg liegt.
- Dynamische Pfadkorrekturen: Wird der aktuelle Pfad von einem Hindernis verdeckt, kann man sehr einfach herausfinden ob der begehbare Bereich komplett oder nur teilweise verdeckt ist. Somit sind Korrekturen möglich.

NavMeshes müssen zwar per Hand erzeugt werden, allerdings geschieht das bei der Modellierung der Levels und die gegebene Levelgeometrie vereinfacht die Arbeit. NavMeshes müssen auch nicht besonders detailliert sein wie z.B. Collision Meshes. Es muss nicht jede einzelne Stufe einer Treppe durch ein Polygon beschrieben werden, es reicht wenn der Treppenabschnitt durch ein einziges Polygon definiert wird.

7 Resümee

Der A* ist tatsächlich sehr mächtig und wird in allen Genres eingesetzt. Er ist ein weit verbreiteter Standard, der sowohl in zweidimensionalen als auch in dreidimensionalen Spielen eingesetzt wird. Allerdings kommen durch 3D-Levels ganz neue Anforderungen in den einzelnen Genres auf. Diese haben es notwendig gemacht den A* zu verändern und an die jeweilige Situation anzupassen, z.B. bei unterschiedlich großen Einheiten in einem RTS Game. Der zugrunde liegende Algorithmus ist aber stets derselbe geblieben. Auch neue, dynamische Algorithmen wie der D* haben es nicht geschafft den A* aus dem Pathfinding zu vertreiben. Der D* benötigt eben viel mehr Speicher, der gerade auf den Konsolen sehr knapp ist, die Geschwindigkeitseinbußen nimmt man dank immer schnellerer Prozessoren und der Parallelität von vier Kernen gerne in Kauf.

Die jüngsten Optimierungen im Pathfinding fanden auch nicht auf der Algorithmen-Ebene statt, sondern bei der Level-Repräsentation. Und selbst NavMeshes werden noch nicht überall eingesetzt.

Ein weiterer Grund für die Verbreitung des A* ist, dass viele Developer auch beim Pathfinding auf existierende Middleware wie z.B. die Path Engine, setzen.

Literaturverzeichnis

[1] Franz Embacher, Von Graphen, Genen und dem WWW, [online]. Verfügbar unter: http://homepage.univie.ac.at/franz.embacher/Lehre/aussermathAnw/Graphen.html [Zugang am 09.12.2012].

[2] Ian Millington, Jon Funge, Artificial intelligence for games, 2nd edition, Morgan Kaufmann, Burlington MA 2009.

[3] Daniel Harabor, Clearance-based Pathfinding and Hierarchical Annotated A* Search, 5. May 2009, [online]. Verfügbar unter: http://aigamedev.com/open/tutorial/clearance-based-pathfinding/ [Zugang am 20.12.2012].

[4] Paul Tozour, Fixing Pathfinding Once and For All, 26. Juli 2008, [online]. Verfügbar unter: http://www.ai-blog.net/archives/000152.html [Zugang am 30.12.2012].

Abbildungsverzeichnis

Abkürzungsverzeichnis

AI	Artificial Intelligence (künstliche Intelligenz)
HAA*	Hierarchical Annotated A*
RPG	Role Playing Game
RTS	Real Time Strategy
WWW	World Wide Web

Anhang A: Dijkstra Algorithmus als Pseudo-Code

Wie oben erwähnt gibt es nicht die eine Implementierung. Das Pseudo-Code Beispiel aus [2] nimmt als Übergabeparameter den Graphen, repräsentiert durch das Interface aus Kapitel 2.2 den Startknoten und den Zielknoten.

```
def pathfindDijkstra(graph, start, end):

    # This structure is used to keep track of the
    # information we need for each node
    struct NodeRecord:
        node
        connection
        costSoFar

    # Initialize the record fort he start node
    startRecord = new NodeRecord()
    startRecord.node = start
    startRecord.connection = None
    startRecord.costSoFar = 0

    #Initialize the open and closed lists
    open = PathfindingList()
    open += startRecord
    closed = PathfindingList()

    # Iterate through processing each node
    while length(open) > 0:

        # Find the smallest element in the open list
        current = open.smallestElement()

        # If it is the goal node, then terminate
        if current.node == goal: break

        #Otherwise get its outgoing connections
        connections = graph.getConnections(current)

        # Loop through each connection in turn
        for connection in connections:
```

```
        # Get the cost estimate for the end node
        endNode = connection.getToNode()
        endNodeCost = current.costSoFar +
                       connection.getCost()

        # Skip if the node is closed
        if closed.contains(endNode): continue

        # .. or if it is open and we've found a worse
        # route
        else if open.contains(endNode):

            # Here we find the record in the open list
            # corresponding tho the endNode.
            endNodeRecord = open.find(endNode)

            if endNodeRecord.cost <= endNodeCOst:
                continue

        # Otherwise we know we've got an unvisited
        # node, so make a record for it
        else:
            endNodeRecord = new NodeRecord()
            endNodeRecord.node = endNode

        # We're here if we need to update the node
        # Update the cost and connection
        endNodeRecord.cost = endNodeCost
        endNodeRecord.connection = connection

        # And add it tot he open list
        if not open.contains(endNode)
            open += endNodeRecord

    # We've finished looking at the connections for
    # the current node, so add it tot he closed list
    # and remove it from the open list
    open -= current
    closed += current
```

```
# We’re here if we’ve either found the goal, or
# if we’ve no more nodes to search, find which.
if current.node != goal:

    # We’ve run out of nodes without finding the
    # goal, so there’s no solution
    return None

else:

    # Compile the list of connections in the path
    path = []

    # Work back along the path, accumulating
    # connections
    while current.node != start:
        path += current.connection
        currant = current.connection.getFromNode()

    # Reverse the path, and return it
    return reverse(path)
```

Anhang B: A* Algorithmus als Pseudo-Code

Die Implementierung des A* unterscheidet sich kaum zu der des Dijkstra. Das Berechnen der Heuristik ist nicht Teil des eigentlichen Algorithmus, sie wird in einem anderen Algorithmus berechnet. Im Pseudo-Code aus [2] wird das Abfragen der Heuristik durch `heuristic.estimat()` dargestellt.

```
def pathfindAStar(graph, start, end, heuristic)

    # This structure is used to keep track oft he
    # information we need for each node
    struct NodeRecord:
        node
        connection
        costSoFar
        estimatedTotalCost

    # Initialize the record for the start node
    startRecord = new NodeRecord()
    startRecord.node = start
    startRecord.connection = None
    startRecord.costSoFar = 0
    startRecord.estimatedTotalCost =
        heuristic.estimate(start)

    # Initialize the open and closed lists
    open = PathfindingList()
    open += startRecord
    closed = PathfindingList()

    # Iterate through processing each node
    while length(open) > 0:

        # Find the smallest element in the open list
        # (using the estimatedTotalCost)
        current = open.smallestElement()

        # If it is the goal node, then terminate
        if current.node == goal: break
```

```
# Otherwise get its outgoing connections
connections = graph.getConnections(current)

# Loop through each connection in turn
for connection in connections:

    # Get the cost estimate for the end node
    endNode = connection.getToNode()
    endNodeCost = current.costSoFar +
                  connection.getCost()

    # If the node is closed we may have to
    # skip, or remove it from the closed list.
    if closed.contains(endNode):

        # Here we find the record in the closed list
        # corresponding tot he endNode.
        endNodeRecord = closed.find(endNode)

        # If we didn't find a shorter route, skip
        if endNodeRecord.costSoFar <= endNodeCost:
            continue;

        # Otherwise remove it from the closed list
        closed -= endNodeRecord

        # We can use the node's old cost values
        # to calculate its heuristic without calling
        # the possibly expensive heuristic function
        endNodeHeuristic = endNodeRecord.estimatedTotalCost
                           -  endNodeRecord.costSoFar
    # Skip if the node is open and we've not
    # found a better route
    else if open.conatins(endNode):

        # Here we find the record in the open list
        # corresponding to the endNode.
        endNodeRecord = open.find(endNode)

        # If our route is no better, then skip
        if endNodeRecord.costSoFar <= endNodeCost:
```

```
                continue;

            # We can use the node's old cost values
            # to calculate its heuristic  without calling
            # the possibly expensive heuristic function
            endNodeHeuristic = endNodeRecord.cost –
                                endNodeRecord.costSoFar

        # Otherwise we know we've got an unvisited
        # node, so make a record for it
        else:
            endNodeRecord = new NodeRecord()
            endNodeRecord.node = endNode

            # We'll need to calculate the heuristic value
            # using the function, since we don't have an
            # existing record to use
            endNodeHeuristic = heuristic.estimate(endNode)

        # We're here if we need to update the node
        # Update the cost, estimate and connection
        endNodeRecord.cost = endNodeCost
        endNodeRecord.connection = connection
        endNodeRecord.estimatedTotalCOst =
           endNodeCost + endNodeHeuristic

        # And add it to the open list
        if not open.contains(endNode):
            open += endNodeRecord

    # We've finished looking at the connection for
    # the current node, so add it to the closed list
    # and remove it from the open list
    open -= current
    closed += current

# We're here if we've either found the goal, or
# if we've no more nodes to search, find which.
if current.node != goal:

    # We've run out of nodes without finding the
```

```
        # goal, so there's no solution
        return None

    else:

        # Compile the list of connections in the path
        path = []

        # Work back along the path, accumulating
        # connections
        while current.node != start:
            path += current.connection
            current = current.connection.getFromNode()

        # Reverse the path, and return it
        return reverse(path)
```

Anhang C: erwähnte Spiele und Produkte

Guild Wars 2	© 2005 – 2012 NCsoft
Monkey Island	© 1990 LucasArts
Path Engine	© 2002 – 2011 PathEngine
The Witcher	© 2007 – 2012 CD Project RED sp. z o.o.
Unreal Tournament	© 1999 GT Interactive
WarCraft	© 1994 Blizzard Entertainment