FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Pfadsuche in einer Triangulation Reduction im Mammoth Massive Multiplayer Online Research Framework

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Patrick Klaffenböck
Matrikelnummer 0125335

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl

Wien, 20.03.2014

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Patrick Klaffenböck
Rotenmühlgasse 14/1, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Kurzfassung

Die Suche nach optimalen Pfaden ist ein Bereich der künstlichen Intelligenz, die in vielen Computerspielen eine Rolle spielt. Dabei sollten die berechneten Pfade einerseits natürlich intelligent wirken, die Berechnung selbst sollte aber möglichst wenig Zeit in Anspruch nehmen, weil die zu bewegenden Objekte im Idealfall sofort beginnen sollen sich auf ihre Zielposition zu zu bewegen. Die Spielwelt wird für das *Pathfinding* in eine Graphenstruktur überführt, in der dann mit dem A*-Algorithmus der kürzeste Pfad zwischen Start- und Zielpunkt gesucht wird. Um mit großen Suchräumen, besser umgehen zu können, wurden einige Methoden entwickelt die Spielwelt oder deren Graphendarstellung zu abstrahieren. Im Rahmen dieser Arbeit wurde eine dieser Abstraktionsmethoden, *Triangulation Reduction*, im *MMO Research Framework* Mammoth[1] implementiert und mit zwei konkreten Suchalgorithmen getestet.

Abstract

Pathfinding is an area in artificial intelligence, that plays an important role in many computer games. On the one hand, the calculated paths should look natural and intelligent. On the other hand it is very importat, that the paths are computed very quickly, since the objects should start to move more or less immediately. The game world is usually transformed into a search graph, on which the A* algorithm is used to determine the shortest path between the start and the goal points. To better deal with large search spaces, quite a number of abstractions on the game world or the corresponding graph have been designed. As part of this thesis, one of these abstraction methods, Triangulation Reduction, has been implemented in the MMO Research Framework Mammoth[1], and has been tested with two concrete search algorithms.

---

[1] `http://mammoth.cs.mcgill.ca/`

Inhaltsverzeichnis

Abbildungsverzeichnis

Pseudocodeverzeichnis

KAPITEL 1

Einleitung

Die Suche der kürzesten Verbindung zwischen zwei Punkten, ist in sehr vielen Computerspielen von großer Bedeutung. Auch wenn für die Spieler selbst keine Pfadsuche nötig ist, weil sie entweder keine Objekte kontrollieren, oder diese vollständig selbst steuern, müssen dennoch sogenannte Nicht-Spieler-Charaktere ihren Weg durch die Spielwelt finden. Meistens stehen für die Berechnung eines Pfades nur sehr wenige Rechenzyklen zur Verfügung, weil natürlich möglichst keine Verzögerung bemerkbar sein soll, wenn der Befehl gegeben wird, einen Spielercharakter zu bewegen. Der Pfadsuche-Algorithmus muss auch deswegen sehr effizient sein, weil für gewöhnlich sehr viele entsprechende Anfragen kommen, vor allem in einem *Massively Multiplayer Online Game* (MMOG).

Der Suchalgorithmus selbst arbeitet normalerweise auf einem gewichteten Graphen, oder in einem gleichmäßigen Raster (das sehr intuitiv als eine Graphen-Struktur, bei der alle Kanten das gleiche Gewicht haben, interpretiert werden kann). Kapitel 2 bietet eine Übersicht über die Methoden eine Spielwelt in eine Graphen-Darstellung zu überführen, sowie eine Auswahl an Abstraktionsmöglichkeiten um den Suchraum möglichst klein zu halten.

Im Rahmen dieser Arbeit wurde ein Pfadsuche-Algorithmus der mit einer *Triangulation Reduction*, siehe [3], arbeitet im *MMO-Research-Framework* Mammoth implementiert. In diesem Framework wurden bereits mehrere Pfadsuche-Algorithmen realisiert, unter anderem auch *Triangulation A** (TA*), eine A*-Variante, die für die Anwendung in einer Triangulation entwickelt und ebenfalls in [3] vorgestellt wurde. Es gab aber noch keine Implementation der *Triangulation Reduction*. Als konkreter Suchalgorithmus wurde neben *Triangulation Reduction A** (TRA*) auch eine Variante von *Fringe-Search*, siehe [1], implementiert.

Fringe-Search schneidet im direkten Vergleich mit A* in einer gleichmäßigen Raster-Umgebung sehr gut ab, und hier sollte getestet werden, ob sich auch *Fringe-*

Search für eine Adaption zu einer Suche in einer *Triangulation Reduction* eignet.

1.1 Aufbau der Arbeit

Der Rest dieser Arbeit gliedert sich in die folgenden Kapitel.

Kapitel 2 Dieses Kapitel beschreibt die Basis-Variante des A*-Algorithmus, der den de facto Standard für die Suche des kürzesten Pfades in einem gewichteten Graphen darstellt, sowie mehrere Möglichkeiten der Repräsentation einer Spielwelt, die sich für die Suche mit A* eignen.

Kapitel 3 Hier folgt eine Beschreibung, der für den praktischen Teil der Arbeit relevanten Technologien und Algorithmen: Das *MMO-Research-Framework* Mammoth, die *Triangulation Reduction* und die A*-Variante *Fringe-Search*.

Kapitel 4 In diesem Kapitel werden die Beiträge dieser Arbeit zur Pfadsuche in einer *Triangulation Reduction* im Allgemeinen und deren Implementierung in Mammoth im Speziellen näher beschrieben und ausgewählte Algorithmen als Pseudocode vorgestellt.

Kapitel 5 Als nächstes folgt eine Beschreibung und Auswertung der durchgeführten Tests mit den implementierten Suchalgorithmen *Triangulation Reduction A** und *Fringe-Search*.

Kapitel 6 Abschließend werden die wichtigsten Punkte dieser Arbeit zusammengefasst und es folgt ein Ausblick auf mögliche Themen für weitere Arbeiten.

KAPITEL 2

Grundlagen

2.1 Einleitung

Aus der Sicht eines Level-Designers besteht eine Umgebung aus ihren Abmessungen und einer Ansammlung an geometrischen Formen, die als Hindernisse gelten, und mit denen sich ein bewegliches Objekt zu keiner Zeit überschneiden darf. Pfadsuche-Algorithmen benötigen allerdings eine abstraktere Darstellung, die sich als Graph interpretieren, oder einfach in einen Graphen überführen lässt. Bei der Art, wie eine Umgebung zerlegt und im Computer dargestellt wird, gibt es mehrere Punkte zu beachten, vgl. [12].

Wenn sich ein Objekt von einer Position zu einer anderen bewegen soll, muss der *Pathfinder* die beiden Positionen in der Spielwelt effizient in Knoten eines Graphen „übersetzen“ können. Diesen Vorgang nennt man *Quantisierung*. Der umgekehrte Vorgang, *Lokalisierung*, wird benötigt, wenn ein vom *Pathfinder* gefundener Pfad in eine Reihe von Positionen in der Spielwelt konvertiert werden muss.

Der *Pathfinder* sollte natürlich nur gültige Pfade liefern. Ein Knoten im Graphen entspricht einem Bereich in der Spielwelt. Wenn also eine Kante zwischen Knoten A und B existiert, dann muss *jeder* Punkt des Bereichs B von *jedem* Punkt des Bereichs A aus ohne Kollision erreichbar sein. Diese Einschränkung kann bei vielen Zerlegungen abgeschwächt werden (wenn Pfade zum Beispiel sowieso zwecks *Pathsmoothing* nachbearbeitet werden).

2.2 Der A*-Algorithmus

Für die tatsächliche Suche in der Graphendarstellung einer Umgebung wird für gewöhnlich eine Variante des in [7] vorgestellten A*-Algorithmus verwendet. Eine gute Übersicht über A* inklusive verschiedener Implementierungsdetails und Varianten findet sich in [13]. Ähnlich wie bei Dijkstra's Algorithmus, siehe [5], werden, ausgehend vom Startknoten, jedem Knoten des Graphen Labels zugewiesen, die einen Verweis auf den Vorgänger des Knotens und dessen Abstand vom Startknoten beinhalten. In jeder Runde wird derjenige Knoten, mit der geringsten Distanz aus der Liste der zu verarbeitenden Knoten entfernt und die Labels für alle seine Nachbarknoten werden aktualisiert.

Im Gegensatz zum Algorithmus von Dijkstra, speichert A* nicht nur die Distanz vom aktuellen Knoten zum Startknoten, den sogenannten g-Wert, sondern auch eine Abschätzung der Distanz vom aktuellen Knoten zum Ziel, den h-Wert. Für jeden Knoten n wird $f(n) = g(n) + h(n)$ berechnet und in jeder Iteration wird derjenige Knoten mit dem niedrigsten f-Wert als nächstes expandiert.

Es gibt zwei interessante Eigenschaften, die die Heuristik h haben kann: *Zulässigkeit* und *Konsistenz*. Eine *zulässige* Heuristik darf den Abstand zum Ziel nicht überschätzen. Es muss also für alle Knoten n gelten, dass der tatsächliche kürzeste Abstand von n zu einem Zielknoten größer oder gleich $h(n)$ ist. Wenn eine zulässige Heuristik verwendet wird, liefert A* immer einen optimalen Pfad. Wenn die Heuristik aber manchmal überschätzt, kann es sein, dass der Zielknoten expandiert und der Algorithmus dadurch beendet wird, obwohl noch ein Knoten in der *Open*-Liste ist, der einen kürzeren Pfad zum Ziel ermöglichen würde.

Während es zwar von Vorteil ist, wenn A* immer eine optimale Lösung liefert, bedeuten niedrigere h-Werte im Allgemeinen auch eine höhere Ausführungszeit, weil sich A* dann mehr auf Knoten in der Nähe des Startpunktes konzentriert, als auf die vielversprechenden Knoten, die näher am Ziel sind. Wenn die Anforderungen also nicht-optimale Lösungen nicht verbieten, und die entstehenden Pfade intelligent und natürlich *wirken*, kann es sinnvoll sein, auf die Einschränkung einer zulässigen Heuristik zu verzichten, siehe [12].

Die zweite interessante Eigenschaft einer Heuristik ist ihre *Konsistenz* oder *Monotonie*. Eine Heuristik ist konsistent, wenn sie die Dreiecksungleichung erfüllt. Für jeden Knoten n und jeden Nachfolger m von n, muss gelten, dass

$$h(n) \leq d(n, m) + h(m)$$

wobei $d(n, m)$ das Gewicht der Kante (n, m) beschreibt.

Jede konsistente Heuristik ist auch zulässig. Wenn A* eine konsistente Heuristik verwendet, wird jeder Knoten erst dann expandiert, wenn ein optimaler Pfad zu diesem Knoten gefunden wurde. In diesem Fall ist es nie nötig einen Knoten von der *Closed*-Liste wieder zu öffnen.

Algorithmus 1 zeigt den prinzipiellen Ablauf einer A*-Suche.[1] Dabei handelt es sich bei O um eine Prioritätswarteschlange, die die gespeicherten Knoten aufstei-

---

[1] Abgewandelt von [13].

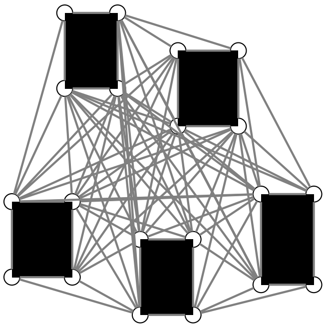


Abbildung 2.1: Worst-Case für einen Corner-Graph

gend sortiert nach ihrem f-Wert zurückliefert, und bei $\mathrm{N}(v)$ um eine Funktion, die alle Nachbarn des Knotens v liefert, während $d(v, u)$ die Kosten der Kante (v, u) beschreibt.

2.3 Reine Graphendarstellung

Man kann für die Pfadsuche auch einen Graphen verwenden der nicht auf der Zerlegung der Spielwelt beruht, indem man an bestimmten Koordinaten der Welt sogenannte *Waypoints* platziert, die gleich den Knoten des Graphen entsprechen. Diese *Waypoints* können vom Level-Designer platziert, oder automatisch verteilt werden. Dafür eignen sich zum Beispiel die Eckpunkte der Hindernisse, weil der optimale Pfad immer nur an genau diesen Punkten Biegungen hat, siehe [12].

Zwei Knoten in diesem Graphen werden verbunden, wenn die Strecke zwischen den entsprechenden Koordinaten keine Hindernisse kreuzt, oder anders ausgedrückt, wenn eine direkte Sichtlinie besteht, daher auch der Name *Points of Visibility*. Im schlimmsten Fall hat so ein Graph allerdings quadratisch viele Kanten und wird damit schnell unhandlich, siehe Abb. 2.1.

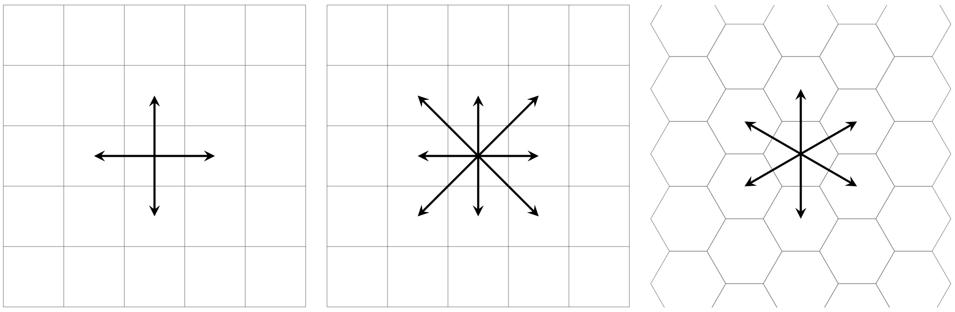
Ein weiterer Nachteil einer reinen Graphendarstellung ist, dass die Start- und Endpunkte einer Anfrage in den meisten Fällen nicht mit einem bestehenden Knoten zusammenfallen, also müssen temporär an diesen Stellen neue Wegpunkte angelegt werden. Dann müssen (zumindest einige) Knoten gefunden werden, zu denen eine direkte Sichtverbindung besteht. Außerdem liefern *Waypoint*-Graphen keine Informationen über die tatsächliche Geometrie der Spielwelt. Wenn also die direkte Verbindung zwischen zwei Wegpunkten temporär durch ein anderes Objekt blockiert ist, kann der *Pathfinder* keine Angaben darüber machen, ob es möglich ist, dieses Hindernis mit einem Schritt zur Seite zu umgehen oder nicht. Dafür müssen dann teure Kollisionsabfragen durchgeführt werden.

Egal ob die Wegpunkte automatisch an den Ecken der Hindernisse oder manuell

Algorithmus 1 astar(V, E, s, g)

Eingabe: Den Graphen $G = (V, E)$, sowie Startknoten s und Zielknoten g
Rückgabe: Den kürzesten Weg in G von s nach g

```
$g(v) \leftarrow \infty \; \forall v \in V$
$p(v) \leftarrow$ NULL $\forall v \in V$        ▷ Verweise auf die Vorgängerknoten
$g(s) \leftarrow 0$
$f(s) \leftarrow h(s)$
$O = \langle s \rangle$                          ▷ Die Open-Liste
$C = \emptyset$                                  ▷ Die Closed-Liste

while $\neg$empty($O$) do
    $v \leftarrow$Knoten in $O$ mit dem kleinsten $f$-Wert

    if $v = g$ then
        Rekonstruiere den Pfad mittels den Verweisen in $p$
        Abbruch: Fertig
    end if

    Entferne $v$ aus $O$
    Speichere $v$ in $C$

    for all $u \in N(v)$ do
        $g \leftarrow g(v) + d(v, u)$

        if $g(u) > g$ then
            if $u \in O$ then
                Entferne $u$ aus $O$
            else if $u \in C$ then
                Entferne $u$ aus $C$
            end if
        end if

        if $u \notin O \land u \notin C$ then
            $g(u) \leftarrow g$
            $f(u) \leftarrow g + h(u)$
            $p(u) \leftarrow v$
            Speichere $u$ in $O$
        end if
    end for
end while

Abbruch: Es existiert kein Pfad von $s$ nach $g$
```



(a) Quadtiles (b) Octiles (c) Hextiles

Abbildung 2.2: Verschiedene Tiles

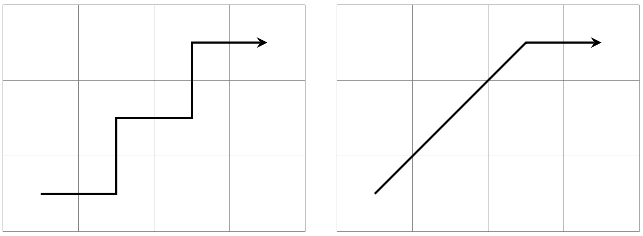
vom Leveldesigner gesetzt werden, wirken die resultierenden Pfade oft unnatürlich: Bei einem *Corner-Graph* bewegt sich ein Objekt zuerst sehr zielstrebig direkt auf ein Hindernis zu, und geht dann immer an den Wänden entlang, zumindest bei längeren Pfaden, die keine direkte Verbindung von Start- und Endpunkt sind, siehe [18]. Bei manuell gesetzten Wegpunkten entsteht, vor allem in offenem Terrain, der Eindruck die Objekte bewegten sich wie „auf Schienen“ und eine Annäherung an den optimalen Pfad ist sehr selten. Dem kann man nur entgegenwirken, indem mehr Wegpunkte gesetzt werden, was den Speicherverbrauch für den Graphen und die Komplexität für den *Pathfinder* erhöht, siehe [18].

2.4 Zerlegungen

2.4.1 Raster

Eine Möglichkeit die Spielwelt zu zerlegen besteht darin, ein gleichmäßiges Raster darüber zu legen und sie dadurch in lauter gleiche *Tiles* zu unterteilen. Am häufigsten sieht man Unterteilungen in Quadrate oder gleichmäßige Sechsecke, siehe Abbildung 2.2.

Jedes dieser *Tiles* ist entweder passierbar oder unpassierbar, wobei passierbare *Tiles* Knoten im Pfadsuche-Graphen entsprechen und zwei Knoten im Graphen durch eine Kante verbunden sind, wenn die entsprechenden *Tiles* benachbart sind. Ein Schritt von einem Knoten zu einem anderen entspricht in der Spielwelt, einer Bewegung vom Mittelpunkt des entsprechendem Start-*Tiles* zum Mittelpunkt des End-*Tiles*.



(a) Pfadlänge 5 (b) Pfadlänge 3,8

Abbildung 2.3: Pfadlängen bei Quad- und Octiles

Quadrate

Eine Unterteilung in Quadrate ist sehr naheliegend, weil man damit sehr gut arbeiten kann: Quantisierung und Lokalisierung sind sehr einfach durchzuführen und man kann ein gewohntes Koordinatensystem für die *Tiles* verwenden. Als Nachbarn gelten entweder nur angrenzende *Tiles*, die sich eine Kante teilen (*Quadtiles*), oder alle acht angrenzenden *Tiles* (*Octiles*).

Quadtiles haben den Vorteil, dass ein Schritt zu einem Nachbarknoten (der ja ein passierbares *Tile* repräsentiert) immer gültig ist, solange das zu bewegende Objekt als Ganzes innerhalb eines *Tiles* Platz hat. Außerdem muss A* bei jedem Knoten (außer dem Startknoten) maximal drei Nachfolger in die *Open*-Liste aufnehmen.

Durch die Einschränkung, sich nur parallel zu den Achsen zu bewegen, entstehen allerdings unnatürliche Zick-Zack-Muster und dadurch längere Pfade, als wenn man auch diagonale Bewegungen erlaubt, siehe Abbildung 2.3.

Während *Octiles* kürzere und glattere Pfade ermöglichen, verursachen sie aber auch zusätzlichen Aufwand: Zum Beispiel ist nicht jeder benachbarte Knoten in einem gültigen Schritt erreichbar, siehe Abbildung 2.4. Der Schritt in Richtung B ist definitiv ungültig, während der Schritt zum Punkt A eventuell noch als gültig angesehen werden kann. Da die zu bewegenden Objekte für gewöhnlich einen Radius größer Null haben, würde aber auch hier ohne Nachbearbeitung eine Kollision entstehen.

Der Verzweigungsgrad scheint auf den ersten Blick bei einem *Octile*-Raster viel höher zu sein, als bei *Quadtiles*. Wie man allerdings an dem Beispiel in Abbildung 2.5(b) sieht, fallen zumindest manche Schritte weg, weil sie einen Umweg darstellen würden: Die hellgrauen Felder hätte man auch in einem einzigen Schritt erreichen können; vgl. [19]. Im Beispiel wird angenommen, dass das Objekt gerade den durch den dicken Pfeil gekennzeichneten Schritt durchgeführt hat. Die drei mit H markierten Felder sind definitiv unnötig, weil sie entweder zum Ausgangsfeld zurückführen, oder mit zwei Schritten auf ein Feld führen, das eigentlich in einem

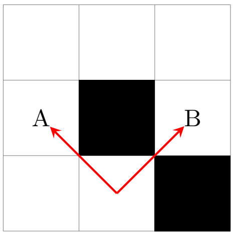
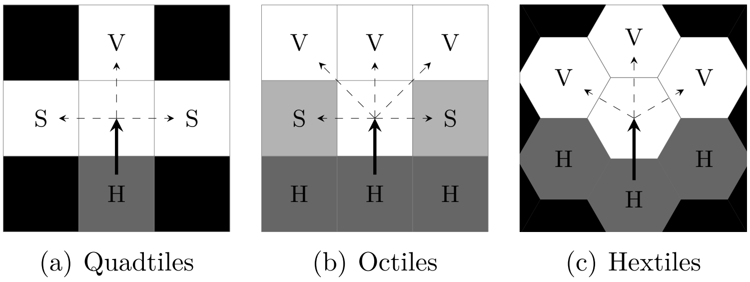


Abbildung 2.4: Ungültige Schritte in einem Octile-Raster



(a) Quadtiles (b) Octiles (c) Hextiles

Abbildung 2.5: Verzweigungsgrade

hätte erreicht werden können. Die beiden mit S gekennzeichneten Felder müssen nur betrachtet werden, wenn ein einziger diagonaler Schritt vom Ausgangspunkt aus nicht möglich war, zum Beispiel wenn das links untere Feld unpassierbar ist.

Hextiles

Hextiles haben einerseits die nette Eigenschaft von *Quadtiles*, dass ein passierbares Nachbar-*Tile* auch sicher in einem gültigen Schritt erreichbar ist, und bieten andererseits aber auch diagonale Schritte an, was die Pfade verkürzt, wenn auch nicht so stark wie *Octiles*, vgl. [19]. Obwohl jedes *Hextile* sechs Nachbarn hat, ist der Verzweigungsgrad in einer A*-Suche, wie man in Abbildung 2.5(c) sieht, derselbe wie bei *Quadtiles*. Ein weiterer Vorteil von *Hex*- und *Quadtiles*, ist die Tatsache, dass alle Schritte die gleichen Kosten haben, während bei *Octiles* die diagonalen Schritte teurer sein sollten als Schritte, die parallel zu einer der Achsen verlaufen.

Für die Pfadsuche an sich haben *Hextiles* also hauptsächlich angenehme Eigenschaften, allerdings ist das Entwickeln von Algorithmen für *Hextiles* komplexer, weil Quantisierung und Lokalisierung aufwändiger sind. Außerdem kann man nicht mehr mit einem gewohnten Koordinatensystem arbeiten: Will man vom *Hextile* mit den Koordinaten (x, y) zum „rechts unteren“ Nachbarn, dann hat dieser Nachbar nur dann die Koordinaten $(x+1, y+1)$ wenn x gerade ist. Bei ungeradem x entspricht „rechts unten“ den Koordinaten $(x + 1, y)$.

Alternativen

Es gibt Alternativen zu dem was oben über *Tiles* gesagt wurde, die zumindest kurz erwähnt werden sollten. Zum Beispiel ist es natürlich auch möglich, die Umgebung in gleichseitige Dreiecke zu unterteilen. In [19] wird das *Tex-Grid* vorgestellt, ein quadratisches Raster bei dem jede zweite Spalte um die halbe *Tile*-Höhe nach unten verschoben ist, wodurch bestimmte Vorteile und Nachteile eines Hex-Rasters mit denen eines quadratischen Rasters verbunden werden können.

Außerdem sind durchaus andere Abbildungen vom Raster zu einem Graphen möglich, als auf Seite 7 beschrieben. Zum Beispiel indem ein Knoten des Graphen nicht ein gesamtes *Tile* bzw. dessen Mittelpunkt repräsentiert, sondern eine Kante zwischen zwei benachbarten passierbaren *Tiles*, oder ein von vier passierbaren *Tiles* umgebener Eckpunkt, siehe [13].

Raster-Auflösung

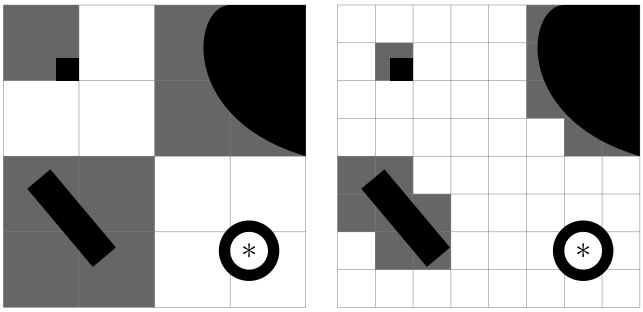
Unabhängig von der Form der *Tiles*, sind nicht am Raster ausgerichtete bzw. sehr kleine Hindernisse immer problematisch, weil dann manche *Tiles* als unpassierbar markiert sind, obwohl sich nur ein kleiner Teil mit einem Hindernis überlappt und sich das zu bewegende Objekt durchaus zwischen den Hindernissen durch bewegen könnte, siehe Abbildung 2.6(a).

Dass alle Hindernisse am Raster ausgerichtet sind, ist allerdings oft eine inakzeptable Einschränkung. Eine andere Möglichkeit ist die Auflösung des Rasters zu erhöhen, was aber die Anzahl an Knoten stark erhöht und damit die Suche aufwändiger macht. Wird das Raster zu eng passt das Objekt, das man bewegen möchte, nicht mehr in ein einziges *Tile*. Dadurch wird der Aufwand für den *Pathfinder* noch einmal größer, weil für einen einzigen Schritt nicht nur das Ziel-*Tile* sondern auch die angrenzenden *Tiles* passierbar sein müssen.

Neben nicht ausgerichteten Hindernissen sind auch große weite Flächen deswegen ein Problem, weil sie zur Darstellung sehr viele *Tiles* benötigen, die eigentlich alle frei passierbar wären. Das erhöht den Speicherverbrauch und die Länge des Pfades, und damit natürlich auch die Laufzeit des *Pathfinders*.

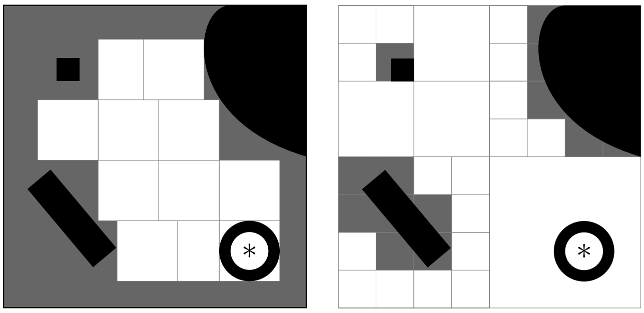
2.4.2 Dynamische „Raster“

Es folgen nun einige Methoden, die zwar ebenfalls eine Art Raster erstellen, bei dem allerdings die einzelnen *Tiles* nicht alle gleichmäßig groß bzw. gleichmäßig verteilt sind. Dadurch können Probleme durch ungünstige Verhältnisse zwischen Objekt- und *Tile*-Größe bzw. unnötig viele *Tiles* in einer großen passierbaren Region vermindert werden.



(a) Zu niedrige Auflösung (b) Zu hohe Auflösung

Abbildung 2.6: Probleme mit der Rasterauflösung



(a) Ghost Player (b) Quadtree-Zerlegung

Abbildung 2.7: Dynamische „Raster“

Dafür verzichtet man aber auf eine der größten Stärken von gleichmäßigen Rastern: Zugriff auf ein bestimmtes *Tile* (Quantisierung) in konstanter Zeit. Darauf verzichtet man aber nicht nur bei dynamischen Rastern, sondern bei allen Arten der Umgebungsrepräsentation, die nicht auf einem gleichmäßigen Raster beruhen.

Ghost Player

Die ideale Auflösung des Rasters hängt mit den Abmessungen des Objektes zusammen, das bewegt werden soll. Daraus ergibt sich allerdings das Problem, dass für verschieden große Objekte auch verschiedene Raster erstellt werden sollten, was bei einer großen Spielwelt und vielen verschiedenen Größenklassen von Objekten nicht praktikabel ist.

In früheren Versionen von Mammoth wurde das Konzept eines *Ghost Players* verwendet um Probleme mit der Auflösung des Rasters zu vermeiden, siehe [11].

Dabei wird gar kein explizites Raster erzeugt sondern ein unsichtbares Objekt mit denselben Eigenschaften, wie das zu bewegende Objekt, wird in diskreten

Schritten durch die Spielwelt bewegt, natürlich nur, wenn ein Schritt keine Kollision verursacht. Bei dem Beispiel in Abbildung 2.7(a) wurde willkürlich eine Mindestschrittweite gewählt, die dem Objektradius entspricht.

Dadurch können kurzfristige Änderungen der Spielumgebung, wie zum Beispiel verschobene Hindernisse, sehr einfach behandelt werden, und die „Auflösung“ des Rasters kann leicht an die Größe des Objektes und die Umgebung angepasst werden, indem zum Beispiel in einem Gebiet mit wenig bzw. vielen Hindernissen, die Schrittweite erhöht bzw. verringert wird.

Weiters ist nur zusätzlicher Speicher für diejenigen Bereich der Welt notwendig, in denen eine Pfadsuche durchgeführt wird. Dafür können diese dynamischen Raster nicht von mehreren Objekten benutzt werden, sondern jede Anfrage an den *Pathfinder* erstellt ein eigenes Raster.

Bei jedem Schritt des Ghost-Players ist eine Kollisionsabfrage nötig um zu Bestimmen, ob dieser Schritt möglich ist, bzw. wie lange der Schritt in die gewählte Richtung sein kann. Diese Kollisionsabfragen sind in der Regel sehr teuer dafür, dass sie so oft ausgeführt werden müssen, wobei Kollisions-Caching hier entgegenwirken kann, siehe [11].

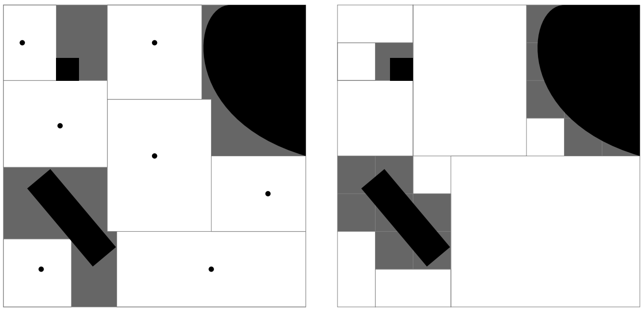
Quadtrees

Eine Möglichkeit die Anzahl an Knoten zu verringern ohne dabei zu viele Informationen über die Spielwelt zu verwerfen, wodurch dann eventuell gültige Pfade nicht gefunden werden können, besteht darin, die Spielwelt durch einen *Quadtree* aufzuteilen, siehe [6]. Ein Quadtree ist eine Baumstruktur, bei der jeder innere Knoten vier Kinder hat. Ein Knoten entspricht dabei einem Ausschnitt aus der Spielwelt. Befinden sich in diesem Ausschnitt keinerlei Hindernisse, wird der Knoten zu einem Blattknoten gemacht. Wenn die repräsentierte Region Hindernisse enthält, wird sie in vier gleichgroße Teile zerlegt, die dann jeweils den vier Kindern des aktuellen Knotens entsprechen. Diese Teile der Umgebung, und die dazugehörigen Knoten, werden dann rekursiv weiter zerlegt, siehe Abbildung 2.7(b).

Space-Filling Volumes

Space-Filling Volumes unterteilen die Spielwelt in Rechtecke, wobei jedes Rechteck eine frei passierbare Fläche und einen Knoten im Pfadsuche-Graphen darstellt. Es gibt zwei Möglichkeiten, wie man die Unterteilung in *Space-Filling Volumes* erstellen kann, siehe [18].

1. Man verteilt automatisch oder manuell kleine Quadrate in der Spielwelt und lässt diese in beide Richtungen „wachsen“, bis sie entweder mit einem Hindernis oder einem anderen Rechteck zusammenstoßen, siehe Abbildung 2.8(a).



(a) „Wachsen“ (b) Zusammenlegung

Abbildung 2.8: Zerlegung in Space-Filling Volumes

2. Man beginnt mit einem fein auflösenden Raster und legt Rechtecke zusammen bis das nicht mehr möglich ist, siehe Abbildung 2.8(b).

2.4.3 Polygone

Schließlich kann man jede Regelmäßigkeit bei der Zerlegung fallen lassen und die Spielwelt in beliebige konvexe Polygone unterteilen. Die Einschränkung, dass die Polygone konvex sein müssen, stellt sicher, dass jeder Punkt im Polygon von jedem anderen Punkt im selben Polygon über eine gerade Strecke erreichbar ist. Außerdem befindet sich dann der geometrische Schwerpunkt sicher innerhalb des Polygons.

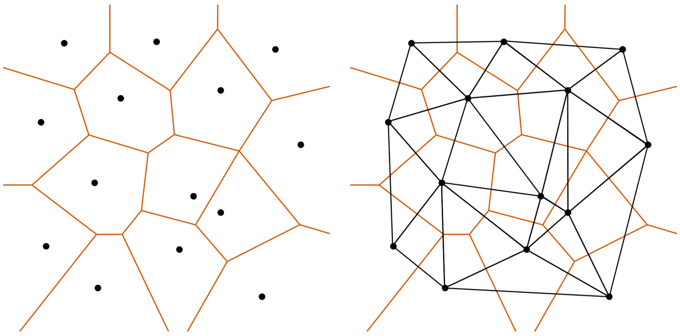
Jedes Polygon entspricht einem Knoten im Graphen und zwei Knoten sind verbunden, wenn ihre zugehörigen Polygone sich eine Kante teilen. Bei der Quantisierung werden alle Punkte innerhalb eines Polygons auf eben dieses abgebildet, bei der Lokalisierung wird ein beliebiger innerer Punkt (zum Beispiel der geometrische Schwerpunkt verwendet.

Anders als bei regelmäßigen Rastern, ist die Anzahl der benötigten Polygone nur von der Anzahl und Form der Hindernisse abhängig und nicht von der Größe der dargestellten Umgebung. Außerdem stellen Hindernisse, die nicht parallel zu den Achsen verlaufen, kein Problem mehr dar.

Von den verschiedenen Möglichkeiten, wie man nun genau die Spielwelt in Polygone unterteilt, werden nur zwei näher betrachtet: Die Dirichlet-Zerlegung und die (Delaunay) Triangulation.

Dirichlet-Zerlegung

Bei der Dirichlet-Zerlegung (auch Voronoi-Diagramm) werden mehrere charakteristische Punkte gewählt und der Raum wird so in Regionen unterteilt, dass alle



(a) Dirichlet-Zerlegung (b) Delaunay-Triangulation

Abbildung 2.9: Dirichlet-Zerlegung und Delaunay-Triangulation

Punkte innerhalb einer Region näher an dessen charakteristischem Punkt liegen, als an allen anderen. Wenn man wirklich nur mit charakteristischen *Punkten* arbeitet, sind die Grenzen dieser Regionen alle gerade Strecken und man erhält eine Unterteilung in konvexe Polygone. Erlaubt man hingegen auch charakteristische Liniensegmente oder Flächen, sind manche der Regionengrenzen Kurven.

Die Quantisierung erfolgt wie oben beschrieben, und die Lokalisierung liefert immer den entsprechenden charakteristischen Punkt.

Wenn man die oben erwähnten *Waypoints*, siehe Abschnitt 2.3 Seite 5, als charakteristische Punkte wählt, kann man einige der oben genannten Nachteile von *Waypoints* umgehen, zum Beispiel müssen keine temporären Knoten mehr eingeführt werden.

Eine alternative Anwendungsmöglichkeit für Voronoi-Diagramme besteht darin, genau die Hindernisse als charakteristische Punkte zu wählen. Die Punkte, die genau auf den Grenzen des Voronoi-Diagramms liegen, sind dann genau jene, die am weitesten von allen Hindernissen entfernt liegen.

Delaunay-Triangulation

Bei einer Delaunay-Triangulation wird eine Punktemenge so zu Dreiecken verbunden, dass der kleinste Winkel aller Dreiecke möglichst groß ist. Sehr dünne Dreiecke verkomplizieren die Pfadsuche, siehe [4].

Eine solche Triangulation verhält sich genau dual zu einer Dirichlet-Zerlegung: Die charakteristischen Punkte der Zerlegung sind die Eckpunkte der Dreiecke und deren Seiten verbinden benachbarte Regionen, so dass sie orthogonal zur Regionsgrenze verlaufen.

Abbildung 2.9(a) zeigt eine Dirichlet-Zerlegung für eine gegebene Punktemenge. In Abbildung 2.9(b) ist über der Dirichlet-Zerlegung (braun) die Delaunay-

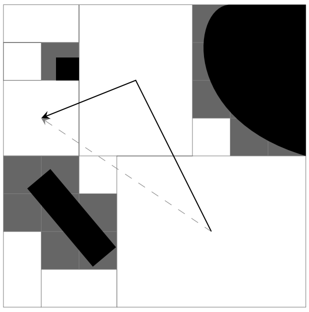


Abbildung 2.10: Suboptimaler Pfad

Triangulation (schwarz) eingezeichnet.[2]

Will man eine Umgebung triangulieren, verwendet man dazu normalerweise eine beschränkte Delaunay Triangulation, in der die Hindernisse als eine Menge von Liniensegmenten dargestellt werden. Die Endpunkte dieser Liniensegmente bilden die zu triangulierende Punktemenge, wobei die Segmente selbst sozusagen vorgegebene Dreieckskanten sind, die zu einer kompletten Triangulation erweitert werden müssen. Natürlich sind nur diejenigen Kanten eines Dreiecks passierbar, die nicht einem Hindernis entsprechen. Im Beispiel in Abbildung 2.12 sind nur die strichlierten Kanten offen, und alle anderen entsprechen Hindernissen und sind beschränkt.

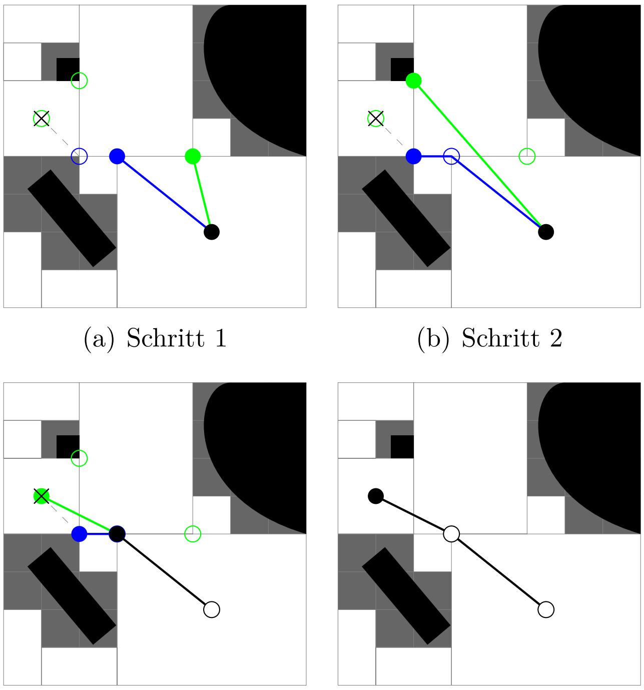
2.4.4 Funnel-Algorithmus

Wenn die Spielwelt in Bereiche zerlegt wird, die deutlich größer sind, als das sich bewegende Objekt, so wie es üblicherweise bei den genannten Verfahren in den letzten beiden Abschnitten der Fall ist, kann man nicht mehr exakt sagen, welche Strecke beim Übergang von einem Bereich zum nächsten zurückgelegt wird. Zum Beispiel könnte ein sehr großer Bereich nur kurz an einer Ecke passiert werden. Pfade die in so einem Fall immer von einem Mittelpunkt zum Nächsten gehen, sehen sehr unnatürlich aus, und sind sehr wahrscheinlich weit vom Optimum entfernt, siehe Abbildung 2.10.

Wenn man die drei Bereiche, durch die der Pfad in Abbildung 2.10 führt, als Kanal betrachtet in dem sich ein Weg vom Start- zum Endpunkt befindet, kann man mit dem sogenannten *Funnel*-Algorithmus den kürzesten Weg zwischen den beiden Punkten innerhalb des Kanals berechnen. Diesen Kanal erhält man durch eine Suche mit A*. Da allerdings A* die Abstände zwischen den Bereichen nur abschätzen kann, kann es vorkommen, dass ein eigentlich „längerer“ Kanal trotzdem

---

[2] Quelle: `http://de.wikipedia.org/wiki/Delaunay-Triangulation` (Oktober 2013)


(a) Schritt 1 (b) Schritt 2

(c) Schritt 3a (d) Schritt 3b

Abbildung 2.11: Funnel-Algorithmus

einen kürzeren Weg enthält: Um den strichliert eingezeichneten optimalen Weg zu finden, muss in einem Kanal gesucht werden, der vier statt drei Bereiche umfasst.

Der *Funnel*-Algorithmus betrachtet nacheinander die sogenannten *Portale* – die Stellen an denen man sich von einem Bereich in den Anderen bewegt. Ausgehend vom aktuellen Scheitelpunkt (*Apex*) wird dabei der Trichter (*Funnel*) berechnet, dessen linke bzw. rechte Seite jeweils die kürzesten Wege vom aktuellen Scheitelpunkt zu den linken bzw. rechten Endpunkten der nachfolgenden Portale darstellen. Der erste Scheitelpunkt entspricht dabei dem Startpunkt der Suche. Befindet sich die Strecke vom Scheitelpunkt zum rechten Endpunkt des nächsten Portals zumindest teilweise links außerhalb des Trichters, wird der Scheitelpunkt zum Ergebnispfad hinzugefügt, und wandert dann zum nächsten Eckpunkt der linken Trichterseite, analog wenn sich der linke Endpunkt des nächsten Portals rechts außerhalb des Trichters befindet.

Abbildung 2.11 zeigt einen beispielhaften Ablauf des *Funnel*-Algorithmus. Es wird ein virtuelles Portal eingeführt, bei dem ein Endpunkt zum letzten echten Portal gehört und der andere Endpunkt mit dem Zielpunkt zusammenfällt.

Zuerst entspricht der Scheitelpunkt genau dem Startpunkt. Der Trichter verbindet den Startpunkt mit den Endpunkten des ersten Portals (die rechte Seite

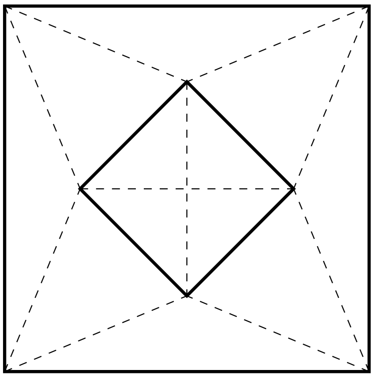


Abbildung 2.12: Beispiel einer beschränkten Delaunay Triangulation

des Trichters ist grün, die linke blau), siehe Abbildung 2.11(a).

Der rechte Endpunkt des nächsten Portals befindet sich genau indem vom Trichter definierten „Sichtbereich“ des Scheitelpunktes, weshalb man hier den Trichter „verengen“ kann. Der linke Endpunkt des nächsten Portals befindet sich „links“ vom Trichter, also wird die linke Seite des Trichters um diesen Punkt erweitert, siehe Abbildung 2.11(b).

Im dritten Schritt muss für die linke Trichterseite nichts aktualisiert werden, weil der linke Endpunkt des nächsten Portals identisch ist mit dem des vorherigen Portals. Die direkte Verbindung zwischen Scheitelpunkt und rechtem Endpunkt des nächsten Portals befindet sich teilweise links außerhalb des Trichters, also wird der Scheitelpunkt zum ersten Punkt im Ergebnispfad und wandert dann auf der linken Seite des Trichters um eins nach vor. Danach kann der Trichter so verengt werden, dass der einzige Punkt auf der rechten Seite gleich dem Zielpunkt entspricht, siehe Abbildung 2.11(c).

Es gibt jetzt keine weiteren Portale mehr, und da wir das virtuelle Portal so eingefügt haben, dass der Zielpunkt auf der rechten Seite lag (und der Trichter ja genau die kürzeste Verbindung vom Scheitelpunkt zum Endpunkt des Portals darstellt), brauchen wir nur die rechte Seite des Trichters zum Ergebnispfad hinzufügen um den kürzesten Pfad zwischen Start- und Endpunkt *innerhalb* des gegebenen Kanals zu erhalten, siehe Abbildung 2.11(d).

2.5 Abstraktionen

Durch Abstraktion versucht man die Anzahl an Knoten des Pfadsuchegraphen zu verringern um den Suchraum für A* zu verkleinern. Dafür nimmt man in Kauf, dass dabei Informationen verworfen werden, die man eigentlich für das Auffinden eines optimalen Pfades brauchen würde, solange die erhaltenen Pfade nahe genung am Optimum liegen. Eine Suche in solch einem abstrahierten Graphen liefert schnell einen groben Pfad, der nicht auf die einzelnen Schritte eingeht. Zum Beispiel könnte so ein Pfad nur die Räume aufzählen die man durchqueren muss, und dabei ignorieren *wie* man durch die einzelnen Räume kommt.

Die Länge dieser abstrakten Pfade kann man zum Beispiel verwenden um die Heuristik für einen A*-Lauf im nicht abstrahierten Graphen zu unterstützen. Würde die Heuristik für A* immer exakt den kürzesten Abstand vom aktuellen Knoten zum Zielknoten liefern, würde A* fast ausschließlich Knoten betrachten, die sich tatsächlich auf dem kürzesten Pfad befinden. Wenn man also die Genauigkeit der Heuristik erhöht, kann das die Effizienz von A* stark steigern.

Häufiger werden diese abstrakten Pfade aber dazu verwendet um zu Bestimmen, welche Knoten in einem folgendem A*-Lauf besucht werden können, der auf der nächst-niedrigeren Abstraktionsebene gestartet wird. Hierarchische Pfadsuche versucht also meistens eine Anwendung von A* in einem großen Graphen, durch mehrere Anwendungen zu ersetzen, die in deutlich kleineren Graphen ausgeführt werden, vgl. [2, 17].

Die Konkretisierung des abstrakten Pfades muss dabei nicht am Stück erfolgen. So eine „*lazy evaluation*“ hat zum einen den Vorteil, dass die Planung des Pfades und die tatsächliche Bewegung verzahnt werden können, was die Verzögerung bis zur ersten Bewegung eines Objekts stark verringern kann, und dass bei langen Pfaden die Wahrscheinlichkeit sinkt, dass ein berechnetes Pfadstück verworfen werden muss, weil der Pfad bis zum Eintreffen des Objekts nicht mehr gültig ist, vgl. [2, 17].

Der Nachteil von Abstraktionen ist, dass man dadurch im Allgemeinen die Optimalität der gefundenen Pfade aufgibt, weil es sein kann, dass eine direkte Verbindung zweier Punkte, in der anfangs verwendeten Abstraktionsstufe nicht mehr abgebildet ist. Der grobe Pfad der dann in der Abstraktion gefunden wird, kann dann verhindern, dass diese Abkürzung auf einer niedrigeren Ebene verwendet wird.

2.5.1 Auf Graphen-Ebene

Die Autoren die in [17] *Partial Refinement A** (PRA*) vorstellen, gehen zwar laut den Illustrationen von einem *Octile*-Raster aus, die eigentliche Abstraktion beruht aber rein auf der Struktur des Pfadsuche-Graphen. Der eigentliche Graph, stellt die Abstraktionsebene Null dar. Um die nächste Ebene zu bilden, werden alle Knoten der aktuellen Ebene mit Grad Zwei oder höher zu möglichst großen Cliquen zusammengefasst. Knoten mit Grad Eins werden zur selben Clique zugeordnet, wie ihr Nachbar. Diese „Cliquen“ entsprechen den Knoten der nächsthöheren Ebene. Zwei Knoten auf der Ebene n sind genau dann durch eine Kante verbunden, wenn auf der Ebene $n-1$ eine Kante zwischen zwei Knoten der entsprechenden „Cliquen“

existiert. Dieser Vorgang wird wiederholt bis der Graph nur noch Knoten und keine Kanten mehr enthält, wenn der ursprüngliche Graph zusammenhängend war, existiert auf der höchsten Abstraktionsebene nur noch ein einziger Knoten.

Soll nun ein Pfad von s nach g, beides Knoten der Ebene Null, gesucht werden, wird zuerst ein geeigneter Abstraktionslevel bestimmt. In [17] wird dafür $x/2$ vorgeschlagen, wobei x die niedrigste Ebene ist, auf der s und g das erste mal in den selben Knoten abstrahiert werden. Nachdem auf dieser Ebene ein Pfad gefunden wurde, werden die ersten k Schritte in der nächst niedrigeren Ebene verfeinert, wobei bei diesem A*-Aufruf nur Knoten aus denjenigen „Cliquen“ betrachtet werden, die auf die Knoten des abstrakteren Pfades abgebildet werden.

2.5.2 Raster

In [2] wird mit *Hierarchical Pathfinding A** (HPA*) eine Möglichkeit vorgestellt, eine Hierarchie für eine Rasterzerlegung zu erzeugen, und in [8] finden sich einige Verbesserungen dafür. HPA* fasst die Zellen des Rasters in gleichgroße *Cluster* zusammen. Dann werden die Bereiche bestimmt, an denen man von einem Cluster zum Anderen wechseln kann, die sogenannten *entrances*, und abhängig von der Größe eines solchen Eingangs, werden ein oder zwei Übergänge (*transitions*) gewählt. Solche Übergänge sind Paare von Rasterzellen, die sich beide in angrenzenden Clustern befinden. Diese Übergänge bilden die Knoten des abstrahierten Graphen.

Die Kanten des abstrakten Graphen bestehen zum Einen aus den sogenannten *Inter*-Kanten, die ein zusammengehöriges Paar von Übergängen verbinden, die entsprechenden Zellen befinden sich direkt nebeneinander, die Gewichte dieser Kanten stehen also sofort zur Verfügung. Außerdem gibt es noch die *Intra*-Kanten, die die Übergangsknoten innerhalb eines Clusters verbinden. Deren Gewichte werden mit A* oder dem Algorithmus von Dijkstra ermittelt.

Der erste Schritt bei der Suche in diesem Abstrakten Graphen besteht darin den Start- und den Zielknoten in den Abstrakten Graphen einzufügen indem er mit allen Übergangsknoten des Clusters verbunden wird in dem sich der Knoten befindet. Dann wird im abstrakten Graphen eine A*-Suche durchgeführt, die bestimmt durch welche Cluster man sich bewegen muss. Dieser abstrakte Pfad wird verfeinert, indem man nacheinander mit A* einen Weg durch den aktuellen Cluster sucht, sofern der Weg zwischen den Übergangsknoten nicht gespeichert wurde. Da die Abstraktion die möglichen Verbindungspunkte zwischen zwei Clustern absichtlich deutlich beschränkt, muss der verfeinerte Pfad nochmals geglättet werden um unnatürliche Zick-Zack-Bewegungen zu vermeiden, zum Beispiel indem man wiederholt Pfadabschnitte x, y, z sucht, bei denen zwischen x und z eine direkte Verbindung möglich ist und dann den Knoten y entfernt.

Es ist auch möglich, weitere Abstraktionsebenen zu erzeugen, indem man die Cluster der aktuellen Ebene wiederum zusammenfasst, alle Übergangsknoten zwischen den zusammengefassten Clustern entfernt, und die Intra-Kanten der übrig gebliebenen Knoten, diejenigen, die sich am „Rand des übergeordneten“ Clusters befinden, neu berechnet.

2.5.3 Triangulation Reduction

Die in [4] vorgestellte *Triangulation Reduction* weißt jedem Knoten des Basisgraphen, die genau den Dreiecken der Delaunay Triangulation entsprechen, eines der Konzepte *Teil einer Sackgasse* (Level 1), *Teil eines Korridors* (Level 2) oder *Entscheidungspunkt* (Level 3) zu. Die Level 0 Inseln – Knoten mit Grad 0, also Dreiecke mit drei beschränkten Seiten – sind ein trivialer Spezialfall der hier weggelassen wird. Die Idee ist, dass sich A* mit Sackgassen gar nicht beschäftigen muss, denn diese Knoten kommen im abstrakten Graphen nicht mehr vor, und dass er Korridore in einem einzigen Schritt durchqueren kann, so dass wirklich nur bei den Level 3 Dreiecken Entscheidungen vom Algorithmus getroffen werden müssen.

Zu Sackgassen gehören die Knoten aus Untergraphen, die Bäume darstellen. Auf jedem Dreieck, dass zu einer Sackgasse gehört, kann man sich maximal in eine der drei Richtungen weiterbewegen, ohne dass man früher oder später vor einer Wand steht, zumindest, wenn man keine Schritte wieder rückgängig machen darf. Wenn es in einer zusammenhängenden Umgebung keine freistehenden Hindernisse gibt, ist der ganze Graph, der durch die Triangulation definiert wird, ein Baum. Ist das nicht der Fall, „mündet“ so eine Sackgasse in einen Korridor.

Korridore sind Ketten von Knoten, die entweder einen Kreis bilden, oder die nicht notwendigerweise verschiedene Entscheidungspunkte miteinander verbinden. Im Gegensatz zu Sackgassen, kann man ein Dreieck eines Korridors in zwei Richtungen verlassen, ohne dass man früher oder später vor einer Wand steht. Es gibt aber keine Abzweigungen, die eine Suche berücksichtigen müsste. Wenn man einen Korridor von einem Entscheidungspunkt aus betritt, gibt es in jedem Dreieck das man passiert nur eine Richtung, die nicht zurück führt. Die einmündenden Sackgassen kommen im abstrakten Graphen nicht mehr vor und müssen von A* nicht betrachtet werden.

Entscheidungspunkte sind schließlich Dreiecke mit drei passierbaren Seiten, deren Nachbarn Korridordreiecke oder andere Entscheidungspunkte sind. Egal von welcher Seite man so ein Dreieck betreten hat, man muss sich zwischen zwei Seiten entscheiden über die man es wieder verlassen kann.

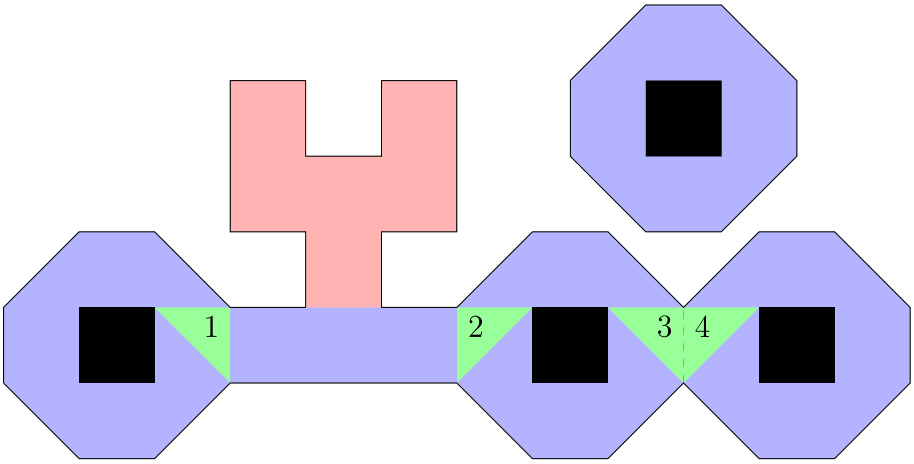


Abbildung 2.13: Triangulation Reduction Konzepte

Der Abstrakte Graph, der durch die *Triangulation Reduction* (TR) entsteht, enthält Knoten, die den Level 3 Dreiecken entsprechen und Kanten, die Ketten von Level 2 Dreiecken darstellen. Zwei interessante Eigenschaften dieser Abstraktion sind, dass einerseits alle wichtigen topologischen Informationen über die Umgebung erhalten bleiben, so dass man immer noch den optimalen Pfad finden kann. Dafür erschwert die Triangulation selbst, das Auffinden des optimalen Pfades: Da man nicht genau weiß, welche Strecke beim Durchqueren eines Dreiecks zurückgelegt wurde, ist der erste Pfad den man zum Zieldreieck findet im Allgemeinen nicht der Kürzeste, man muss also A* weiterlaufen lassen, nachdem man den Zielknoten bereits aus der *Open*-Liste entfernt hat. Außerdem ist die Anzahl der Knoten im Abstrakten Graphen nur linear in der Anzahl der freistehenden Hindernisse: $2n - 2$ Knoten für n Hindernisse.Größe oder Form der Hindernisse haben auf die Anzahl der Knoten keinen Einfluss. Ein kleines quadratisches Hindernis verursacht genauso viele Knoten, wie eine große Koch-Schneeflocke.

Abbildung 2.13 zeigt ein Beispiel für eine solche Einteilung in Sackgassen, Korridore und Entscheidungspunkte (ohne die Triangulation): Rote Bereiche gehören zu Sackgassen und die blauen Bereiche stellen Korridore dar, die die grünen Entscheidungspunkte verbinden. In diesem Beispiel sieht man, dass dieser Graph im Allgemeinen nicht schlicht ist: Es gibt sowohl Korridore, die Entscheidungspunkte mit sich selbst verbinden, als auch zwei Entscheidungspunkte, die durch zwei verschiedene Korridore miteinander verbunden sind.

KAPITEL 3

Mammoth, Triangulation Reduction & Fringe-Search

3.1 Mammoth

Mammoth[1] ist ein *Massively Multiplayer Online Game Research Framework* der Fakultät für *Computer Science* an der McGill Universität in Kanada[2]. Es soll Studenten eine Möglichkeit bieten, an den verschiedenen Bereichen, die ein MMOG benötigt, zu arbeiten, zum Beispiel Netzwerkkommunikation, Objekt-Replikation, Interessens-Management, Fehlertoleranz und künstliche Intelligenz. Der modulare Aufbau des Mammoth-Frameworks ermöglicht es einfach verschiedene Algorithmen für eine bestimmte Anforderung zu testen und zu vergleichen.

Das Spiel kann sowohl klassisch als *Client-Server*-Anwendung, als *Peer-to-Peer*-Anwendung oder als *Standalone-Client* ausgeführt werden. Sowohl der Client als auch die Server-Anwendung sind in Java implementiert, der 3D-Client verwendet die jMonkey-Engine für die Darstellung (die Abbildungen 3.1 und 3.2 zeigen Screenshots des Spiels).

Neben den normalen Client- und Serveranwendungen, gibt es auch noch einen Kommandozeilen-Client und einen Karten-Editor.

3.1.1 Pfadsuche im Mammoth-Framework

Um einen neuen *Pathfinder*-Algorithmus in Mammoth zu integrieren, gibt es zwei Einstiegspunkte. In der Klasse `Mammoth.XML.Reader.WorldReader` muss man in

---

[1] `http://mammoth.cs.mcgill.ca/`

[2] `http://www.cs.mcgill.ca/`

Abbildung 3.1: Mammoth Screenshot

```
FrameBuffers (M) = 1
FrameBuffers (F) = 1
FrameBuffers (S) = 2
Textures (M) = 137
Textures (F) = 134
Textures (S) = 250
Shaders (M) = 12
Shaders (F) = 9
Shaders (S) = 38
Objects = 930
Un[illegible]
Triangles = 442851
Vertices = 1131677
```

Debug

☐ Wireframe
☐ Bounding boxes
☐ Interest Radius
☐ Reach Radius
Camera type: Isometric
☐ Snow effect
☐ Tile overlay
☐ Rain effect
☐ Real Time
Weather Intensity
Time
☐ Bloom effect
Bloom Intensity
☐ Hide world

Chat

Welcome to Mammoth!

Use your mouse to move around world.
Type below to send chat to those around you.
Hello there!

Abbildung 3.2: Mammoth Screenshot

der Methode `load()` dafür sorgen, dass der neue *Pathfinder* alle nötigen Informationen über die gerade geladene Karte erhält. Außerdem muss der neue *Pathfinder* in der Methode `setupPathFinding()` in der Klasse `Mammoth.Client.Client` registriert werden. Dann kann der neue Algorithmus in der globalen Konfigurationsdatei als Pfadsuche-Algorithmus ausgewählt werden.

Wenn eine Pfadsuchen-Anfrage gestellt wird, erzeugt Mammoth ein leeres `Path`-Objekt und übergibt es zusammen mit dem Start- und dem Endpunkt an die `findPath()`-Methode der entsprechenden Implementation. Dem *Pathfinder* kann über die Methode `Path.cancel()`, dass die Pfadsuche abgebrochen werden soll. Ob der Pfadsuche-Algorithmus fertig ist oder nicht kann über den Aufruf von `Path.isDoneComputing()` festgestellt werden.

3.2 Triangulation Reduction

Einen Überblick über TR findet sich im Abschnitt 2.5.3. Hier sollen nun die Erstellung und die Suche in einer solchen Abstraktion näher beschrieben werden.

3.2.1 Aufbau der Abstraktion

Der Abstraktionsprozess nimmt die Dreiecke der beschränkten Delaunay-Triangulation und generiert den sogenannten *Most Abstract Graph*, in dem die Knoten den Level 3 Dreiecken entsprechend. Die Kanten entsprechen Ketten von Level 2 Dreiecken, und Dreiecke die zu Sackgassen gehören kommen in der Abstraktion gar nicht mehr vor.

Zu Beginn sind alle Dreiecke unmarkiert. Während der Abstraktion werden dann die folgenden Schritte ausgeführt:

1. Die Level 1 Dreiecke zu Sackgassen zusammenfassen
2. Die Level 3 Dreiecke identifizieren und Knoten im Graphen zuweisen
3. Die Level 2 Dreiecke identifizieren und den Kanten des Graphen zuweisen

Wobei es in der tatsächlichen Implementierung effizienter ist, die Schritte 2 und 3 zu kombinieren.

Level 1 Dreiecke

Im ersten Durchlauf werden alle Dreiecke, die nur eine offene Kante haben als Level 1 Dreiecke markiert und ihre einzigen erreichbaren Nachbar-Dreiecke in einer Liste zur weiteren Verarbeitung gespeichert.

Solange diese Liste nicht leer ist, wird das nächste Dreieck daraus entnommen. Wenn dieses Dreieck nun höchstens eine offene Kante hat, die nicht zu einen bereits markierten Level 1 Dreieck führt, wird es ebenfalls als Sackgassen-Dreieck markiert und das nicht markierte Dreieck auf der anderen Seite der Kante (falls es existiert) in die Liste aufgenommen.

Wenn die Liste leer ist, sind alle Sackgassen Dreiecke als solche markiert. Diejenigen Dreiecke, die zwar in die liste aufgenommen, aber dann nicht markiert wurden, werden später als Korridor-Dreiecke erkannt werden.

Sollte es in der triangulierten Karte keine frei stehenden Hindernisse geben, sind nach diesem Schritt alle Dreiecke der Triangulation als Level 1 Dreiecke markiert.

Level 3 Dreiecke

Im nächsten Schritt werden alle noch nicht markierten Dreiecke betrachtet. Hat ein solches Dreieck drei offene Kanten, die zu ebenfalls unmarkierten oder als Level 3 markierte Dreiecken führen, wird es ebenfalls als ein Entscheidungspunkt markiert.

Level 2 Dreiecke

Im letzten Schritt werden alle Level 3 Dreiecke noch einmal betrachtet. Für jeden Entscheidungspunkt werden die drei Kanten überprüft. Führt eine solche Kante zu einem unmarkierten Dreieck, beginnt hier ein Korridor zu einem anderen Entscheidungspunkt. Das Dreieck wird als Level 2 Dreieck markiert und der Algorithmus geht über die eine Kante weiter, die nicht zurückführt und die nicht zu einem als Level 1 markierten Dreieck führt.

Wird dabei ein Entscheidungspunkt erreicht, wurde der zweite Endpunkt des Korridors gefunden und die passierten Dreiecke werden zu einer einzigen Struktur zusammengefasst. Wird ein unmarkiertes Dreieck erreicht, wird dieses ebenfalls als Level 2 Dreieck markiert und der Algorithmus geht weiter über die nächste Kante, die nicht zurück und nicht auf ein Level 1 Dreieck führt.

3.2.2 Suche in einer Triangulation Reduction

Bei der Suche in einer TR wird nur ein einziges Mal eine A* Suche ausgeführt. Dafür werden aber vorher eine ganze Reihe an Vorbereitungsschritten benötigt, was den Algorithmus sehr kompliziert wirken lässt im Vergleich mit den Verfahren, HPA* und PRA*, die die gleiche Vorgehensweise immer wieder auf den verschiedenen Abstraktionsebenen einsetzen, vgl. [2] und [17].

Die erste Phase des TRA* betrachtet die unterschiedlichen Möglichkeiten, in welchen Komponenten sich Start- und Endknoten befinden können, also zum Beispiel ob sie sich in Sackgassen, Korridoren oder auf Entscheidungspunkten befinden, und endet entweder damit, dass bereits ohne eine A*-Suche ein Pfad gefunden wurde, oder das Ergebnis der Phase sind jeweils ein bis zwei Start- und Endknoten

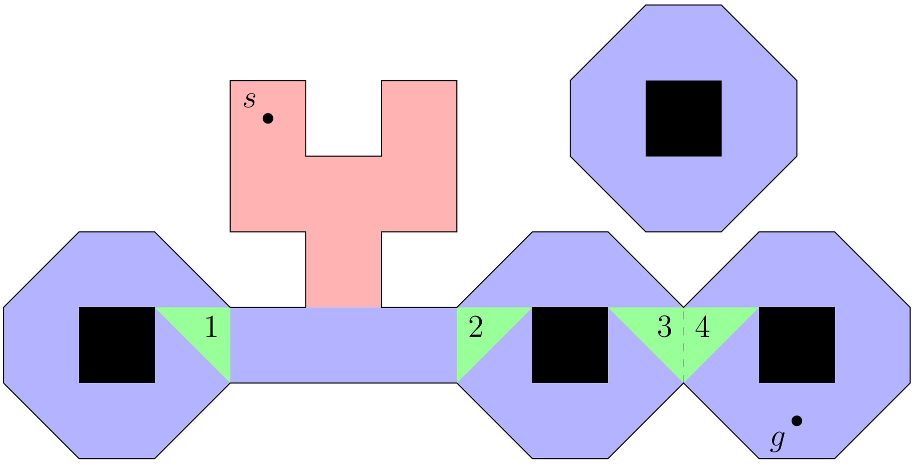


Abbildung 3.3: Suche in einer Triangulation Reduction

im abstrakten Graphen. In der etwaigen zweiten Phase wird mit A* der optimale Weg im abstrakten Graphen gesucht und mit den Informationen aus der Ersten kombiniert um den optimalen Pfad zu erhalten.

Als Beispiel suchen wir einen Weg von s nach g in Abbildung 3.3. Der Startknoten befindet sich in einer Sackgasse. Eine Sackgasse ist ein Teil der Triangulation, die einen Baum bilden, dessen Wurzel ein Level 2 Dreieck des Korridors ist, in den die Sackgasse mündet. Zu jedem Dreieck einer Sackgasse wird ein Verweis auf die Wurzel des entsprechenden Baums gespeichert, sodass die Sackgasse sozusagen in einem Schritt verlassen werden kann. Analog wird zu jedem Dreieck eines Korridors jeweils ein Verweis auf die beiden nicht notwendigerweise verschiedenen Endpunkte gespeichert.

Die Suche wandert also vom Startpunkt sofort in den angrenzenden Korridor. Hierfür existiert nur ein einziger Weg, der durch Elternverweise sehr einfach rekonstruiert werden kann. Der Korridor in dem man dadurch landet ist nicht derselbe, wie der Korridor in dem sich g befindet. Also werden zunächst die Wege zu den beiden Endpunkten des Korridors (die Knoten 1 und 2) konstruiert, und diese beiden Knoten werden zu den Startknoten für die A*-Suche. Analog werden die Wege zum Endpunkt des Korridors in dem sich g befindet ermittelt (zwei verschiedene Wege zu Knoten 4) und A* gestartet um den kürzesten Weg vom Knoten 1 oder 2 zum Knoten 4 zu finden.

Hat A* einen Weg gefunden, hat man eine Folge von Dreiecken, die man auf dem Weg von s nach g passiert, einen sogenannten *Kanal*, in dem man mittels des *Funnel*-Algorithmus den kürzesten Weg zwischen s und g bestimmen kann. Da man vom Knoten 4 aus, sowohl im Uhrzeigersinn, als auch im Gegenuhrzeigersinn zu g gelangen kann, müssen beide entsprechenden Kanäle untersucht werden sobald man einen Weg zum Knoten 4 gefunden hat.

Außerdem darf A* nicht abbrechen sobald der Knoten 4 das erste mal von der *Open*-Liste genommen wurde, weil es zwei verschiedene Wege zwischen den Knoten 2 und 3 gibt. Und auch wenn der Weg im Uhrzeigersinn der kürzeste ist um einfach nur die Dreiecke 2 und 3 zu verbinden, kann es sein, dass der Kanal, den man erhält wenn man im Gegenuhrzeigersinn vom Knoten 2 zum Knoten 3 geht, einen besseren Weg zwischen s und g enthält.

3.3 Fringe-Search

Das in [1] vorgestellte *Fringe Search* versucht einen Kompromiss zwischen der Perfomanz von A* und dem niedrigeren Speicherverbrauch von *Iterative Deepening A** (IDA*) zu finden. Dies wird erreicht, indem einerseits die g-Werte von bereits behandelten Zuständen gecached werden und indem die Zustände, die die „Grenze“ zwischen besuchten und expandierten Zuständen und nur besuchten Zuständen bilden, gespeichert werden, damit in der nächsten Iteration nicht die ganze Arbeit wiederholt werden muss.

In jeder Iteration von IDA* werden die Nachfolger eines Zustandes nur erzeugt, wenn der f-Wert des Zustands nicht über dem aktuellen f_{Limit} liegt. Wenn die *Open*-Liste leer ist und das Ziel noch nicht erreicht wurde, wird f_{Limit} entsprechend angehoben und die nächste Iteration beginnt mit der Suche von Vorne, vgl. [9]. Bei *Fringe-Search* ist die *Open*-Liste (oder auch *Fringe*) unterteilt in eine Liste von Zuständen, die in dieser Iteration noch betrachtet werden. Diese *Now*-Liste enthält die Zustände deren f-Wert das aktuelle f_{Limit} nicht übersteigen. Liegt der f-Wert eines betrachteten Zustandes über dem Limit, wird er aus der *Now*-Liste entfernt und in die *Later*-Liste eingefügt. Ist eine Iteration beendet, wird die *Later*-Liste zur *Now*-Liste.

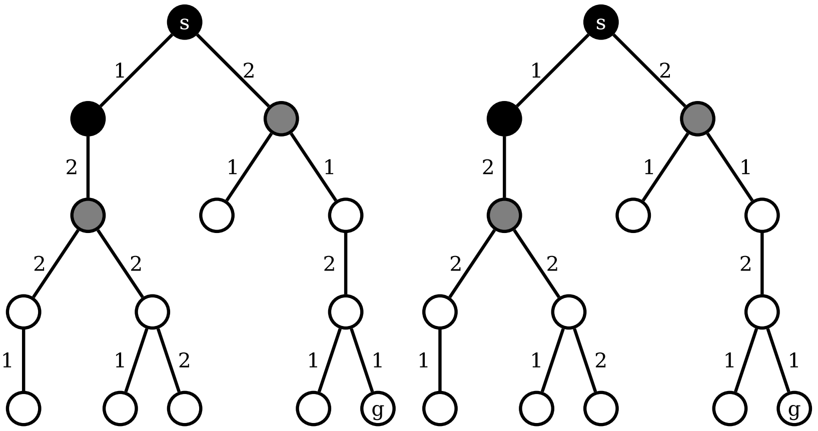
Konkret werden in [1] drei Faktoren identifiziert, die sich bei IDA*, verglichen mit A*, negativ auf die Effizienz auswirken:

1. mehrfach besuchte Zustände,
2. mehrfach durchgeführte Arbeit während der Suche, und
3. die Reihenfolge in der expandierte Zustände bearbeitet werden.

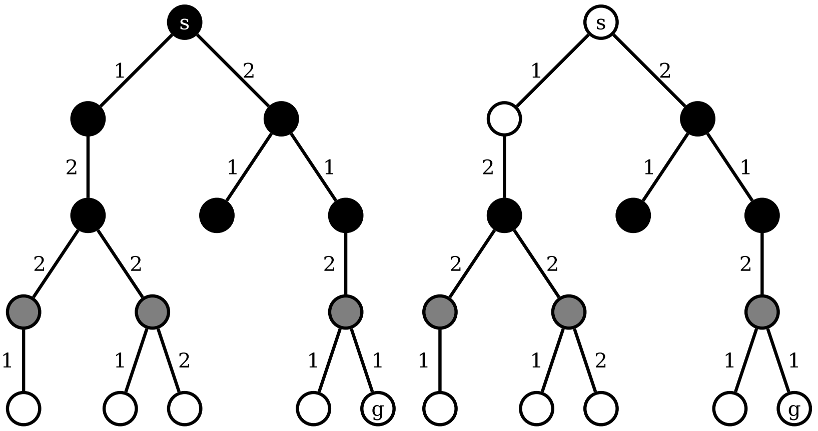
Laut [10] werden während einer Pfadsuche in einem Raster, bei der mehrere Wege zu einem Zielknoten existieren, manche Zustände unter Umständen mehrfach betrachtet, obwohl bereits ein kürzerer Pfad zu den entsprechenden Tiles gefunden wurde. Dieses Problem lässt sich vermeiden, indem die minimalen g-Werte eines Zustands gecached werden, vgl. [14].

Neben mehrfach besuchten Zuständen im Laufe eines Durchgangs, muss IDA* in jeder Iteration die gesamte Arbeit der vorherigen Iterationen wiederholen. Abbildung 3.4 zeigt den Vergleich aus [1] zwischen IDA* und *Fringe-Search*, wobei s den Start- und g den Zielzustand angeben. Schwarze Knoten wurden in der angegebenen Iteration expandiert, graue knoten wurden nur betrachtet aber wegen dem aktuellen f_{Limit} nicht expandiert. Die Kosten zwischen den Zuständen sind an den Kanten angegeben, die Heuristik liefert in diesem Beispiel immer die Anzahl an tiefer liegenden Ebenen. In diesem Beispiel ist sehr schön ersichtlich, wie IDA* die Arbeit der vorherigen Iterationen jedes mal wiederholen muss, während *Fringe-Search* sich die „Grenzknoten“ merkt, die im letzten Durchgang besucht wurden, und nur von diesen Knoten aus weiter sucht.

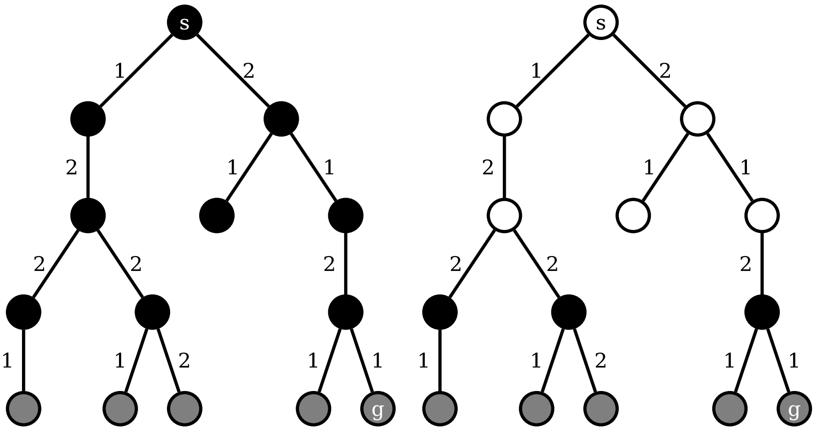
Schließlich expandiert IDA* die Zustände wie bei einer Breitensuche und nicht, wie A*, in einer Prioritäts-Warteschlange, bei der der Zustand mit dem geringsten f-Wert am Anfang steht. Die Autoren in [1] sehen keine Sortierung der expandierten Zustände vor, sondern lediglich die Unterscheidung, ob er Zustand noch in der aktuellen Iteration untersucht werden soll (*Now*-Liste), oder erst in der Nächsten (*Later*-Liste). Es wäre aber möglich die *Later*-Liste in verschiedene *Buckets* aufzuteilen um eine teilweise Sortierung der Liste zu erreichen. Im Extremfall könnte es für jeden f-Wert einen eigenen *Bucket* geben, wodurch die Zustände in der selben Reihenfolge expandiert würden, wie bei A*.



(a) IDA* 1. Iteration (b) Fringe 1. Iteration



(c) IDA* 2. Iteration (d) Fringe 2. Iteration



(e) IDA* 3. Iteration (f) Fringe 3. Iteration

Abbildung 3.4: Vergleich IDA* und Fringe-Search

KAPITEL 4

Eigene Beiträge

4.1 Pfadsuche in einer Triangulation Reduction

Demyen schlägt in [4] vor, den folgenden Wert als Kandidat für $g(s')$ zu verwenden (dabei ist s der Vorgänger-Zustand von s'):

$$g(s') = g(s) + h(s) - h(s') \tag{4.1}$$

Dieser Wert wäre eine gültige Abschätzung für $g(s')$, wenn h eine konsistente Heuristik ist. Der Beweis, dass $g(s')$ nicht überschätzt verwendet die Definition einer konsistenten Heuristik, nämlich, dass für einen Zustand s und einen Folgezustand s' folgendes gelten muss:

$$h(s) - h(s') \leq d(s, s') \tag{4.2}$$

Allerdings ist $d(s, s')$ in einer Triangulation nicht eindeutig definiert, weil nicht im Vorhinein bekannt ist, wo genau ein Dreieck betreten wird. Es kann durchaus vorkommen, dass $h(s)$ eine schärfere Abschätzung ist als $h(s')$ und die Formel (4.1) keine obere Schranke ist, weswegen sowohl für die A*- als auch für die *Fringe-Search*-Variante des *Pathfinders* nur die im Abschnitt 4.1.1 genannten Abschätzungen für die g-Werte verwendet wurden.

4.1.1 Fringe-Search

Die für die *Open*-Liste verwendete Datenstruktur ist wie in [1] vorgeschlagen eine doppelt verkette zyklische Liste, in der das aktuell betrachtete Element und alle Nachfolger davon als *Now*-Liste interpretiert werden, und alle Vorgänger als die *Later*-Liste. Wenn das Ende der Liste erreicht wurde, beginnt eine neue Iteration.

Es gibt keine Unterteilung in Buckets, aber es wurden drei verschiedene Methoden implementiert um die neu expandierten Zustände in die Liste aufzunehmen, siehe Abschnitt 4.2.2.

Der Hauptunterschied zu *Fringe-Search* in einer Raster-Umgebung, besteht darin, dass keine exakten g-Werte für einen Zustand verfügbar sind, sondern diese nur nach unten abgeschätzt werden können, vgl. [4].

Die g- und h-Werte werden anhand derjenigen Kante abgeschätzt über die ein bestimmtes Dreieck betreten wird. Als g-Wert wird das Maximum der folgenden Abschätzungen verwendet:

1. die euklidische Distanz vom Startpunkt bis zur Eintrittskante
2. der g-Wert des Vorgänger-Zustandes plus die in der Abstraktion gespeicherte Abschätzung des Abstands zwischen den beiden Zuständen

Um den h-Wert eines Zustandes zu ermitteln wurden zwei Varianten einer euklidischen Heuristik implementiert. In der simplen Form wird nur einfach der Abstand des Zielpunktes von der Eintrittskante verwendet. Die erweiterte Form berechnet die Abstände von der Eintrittskante zu allen Kanten der Ziel-Dreiecke und verwendet das Minimum davon als h-Wert. Wenn der Startpunkt in einem Level 3 Dreieck liegt, gibt es für bei den Start-Zuständen keine „Eintrittskante“. In diesem Fall wird direkt der Abstand vom Startpunkt verwendet.

Da die g-Werte der Zustände nur Schätzungen sind, ist das in Abschnitt 3.3 angesprochene Caching von bereits betrachteten Zuständen nicht möglich. Denn auch wenn laut Cache das zu diesem Zustand gehörige Dreieck mit einem geringeren g-Wert erreicht werden kann, kann es sein, dass der aktuell betrachtete Pfad in Wahrheit kürzer ist, weil der gespeicherte Wert nur eine Schätzung ist. Also darf der aktuelle Pfad nicht verworfen werden, und es muss auf jeden Fall ein neuer Zustand in die *Open*-Liste aufgenommen werden. Die abgeschätzten g-Werte verhindern generell die Verwendung einer *Closed*-Liste und der in [1] erwähnte Cache entspricht im Prinzip einer *Closed*-Liste.

4.1.2 Funnel-Algorithmus für Liniensegmente

Algorithmus 2 beschreibt eine Modifikation des *Funnel*-Algorithmus, der den kürzesten Weg in einem gegebenen Kanal (*Channel*) zwischen zwei Liniensegmenten berechnet. Der Aufwand einen Pfad in einem Kanal mit n Portalen zu berechnen ist, wie für die Punkt-zu-Punkt-Variante, in $O(n)$. Es wird hier bereits davon ausgegangen, dass der übergebene Kanal nicht leer ist (ein Spezialfall, der schon vor dem Aufruf des *Funnel*-Algorithmus abgefangen wird).

Die Pseudocodes für die Algorithmen 3 und 4 sind nur der Vollständigkeit halber aufgeführt, weil sie eigentlich nur zur übersichtlicheren Darstellung des ei-

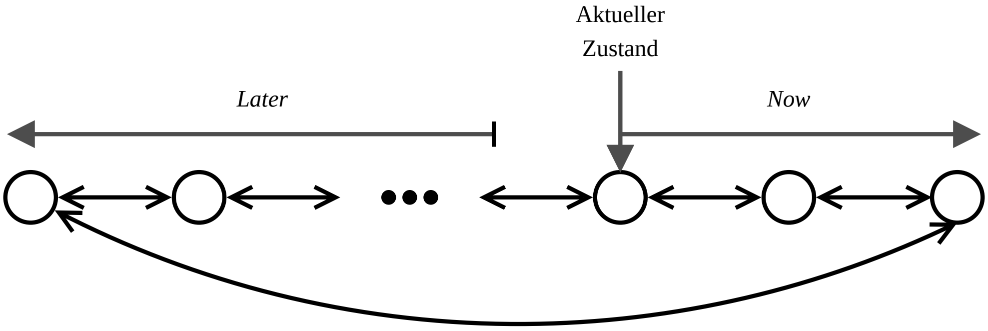


Abbildung 4.1: Fringe Datenstruktur

gentlichen Algorithmus gebraucht werden oder identisch sind mit der Variante für den kürzesten Weg zwischen zwei Punkten.

Es wurde jeweils nur der Fall betrachtet, dass ein Punkt an der linken Seite des Trichters hinzugefügt werden muss. Die Vorgehensweise für das Hinzufügen auf dessen rechten Seite ist analog.

Algorithmus 8 ist eine Anpassung von `ccw()` aus Sedgewicks *Algorithmen in C* ([15], Seite 402f).[1]

---

[1] In Mammoth ist der Punkt $(0, 0)$ in der oberen rechten Ecke der Karte, also muss die Berechnung von $d_{x,1}$ und $d_{x,2}$ in Mammoth umgekehrt werden.

Algorithmus 2 funnelAlgorithmLine($s, g, channel$)

Eingabe: Startlinie s, Endlinie g und eine Liste von Kanal-Portalen *channel*
Rückgabe: Der kürzeste Weg von s nach g im von *channel* definierten Kanal

```
 1: $s' \leftarrow \text{truncate}(s, \text{head}(channel))$
 2: $g' \leftarrow \text{truncate}(g, \text{last}(channel))$
 3: $channel \leftarrow channel + g'$
 4: $apex \leftarrow s'$
 5: $path \leftarrow \langle\rangle$
 6: $funnel \leftarrow \langle apex \rangle$

 7: for all $c \in channel$ do
 8:     $left \leftarrow \text{head}(funnel)$
 9:     if $left = apex \wedge apex_{\text{left}} \neq apex_{\text{right}}$ then
10:         if $c_{\text{left}} \neq apex_{\text{left}}$ then
11:             $added \leftarrow \text{addLeft}(c_{\text{left}})$
12:             while $\neg added$ do
13:                 $added \leftarrow \text{addLeft}(c_{\text{left}}, funnel, apex, path)$
14:             end while
15:         end if
16:     else if $c_{\text{left}} \neq left$ then
17:         $added \leftarrow \text{addLeft}(c_{\text{left}})$
18:         while $\neg added$ do
19:             $added \leftarrow \text{addLeft}(c_{\text{left}}, funnel, apex, path)$
20:         end while
21:     end if

22:     $right \leftarrow \text{last}(funnel)$
23:     Analog zu 9-21 mit left und right vertauscht

24:     return $path$ + shortest funnel side
25: end for
```

Algorithmus 3 addLeft($p, funnel,$ **inout** $apex,$ **inout** $path$)

```
if $apex_{\text{left}} = apex_{\text{right}}$ then
    return addLeftPoint($p, funnel, apex, path$)
else
    return addLeftLine($p, funnel, apex, path$)
end if
```

Algorithmus 4 addLeftPoint($p, funnel,$ **inout** $apex,$ **inout** $path$)

Eingabe: Ein Punkt p der an der linke Seite des Trichters $funnel$ hinzugefügt werden soll. Der aktuelle Scheitelpunkt im Trichter ist $apex$ und der Pfad der gerade erstellt wird ist $path$. Der Scheitelpunkt muss bereits zu einen Punkt kollabiert sein.

Rückgabe: Der Punkt p wird zum Trichter hinzugefügt. Dadurch ändert sich unter Umständen der aktuelle $apex$ bzw. kann der $path$ um einen Punkt erweitert werden. Gibt **true** zurück, wenn der Punkt tatsächlich zum Trichter hinzugefügt wurde und **false** sonst.

```
if size($funnel$) = 1 then
    $funnel = p + funnel$
    return true
end if

$f \leftarrow$ head($funnel$)                                  ▷ Das erste Element von links
$s \leftarrow$ head(tail($funnel$))                           ▷ Das zweite Element von links
$o \leftarrow$ orientation($s, f, p$)
$add \leftarrow f = apex \wedge o =$ ClockWise
$add \leftarrow add \vee (f \neq apex \wedge o =$ CounterClockWise$)$
if $add$ then
    $funnel \leftarrow p + funnel$
    return true
end if

$funnel \leftarrow$ tail($funnel$)
if $f = apex$ then
    $path \leftarrow path + apex$
    $apex \leftarrow$ head($funnel$)
end if

return false
```

Algorithmus 5 addLeftLine($p, funnel,$ **inout** $apex,$ **inout** $path$)

Eingabe: Ein Punkt p der an der linke Seite des Trichters $funnel$ hinzugefügt werden soll. Der aktuelle Scheitelpunkt im Trichter ist $apex$ und der Pfad der gerade erstellt wird ist $path$. Der Scheitelpunkt darf noch nicht zu einen Punkt kollabiert sein.

Rückgabe: Der Punkt p wird zum Trichter hinzugefügt. Dadurch ändert sich unter Umständen der aktuelle $apex$ bzw. kann der $path$ um einen Punkt erweitert werden. Gibt **true** zurück, wenn der Punkt tatsächlich zum Trichter hinzugefügt wurde und **false** sonst.

```
if size($funnel$) $= 1$ then
    truncateApex(Left, $p$, $apex$)
    $funnel \leftarrow p + funnel$
    return true
end if

$f \leftarrow$ head($funnel$)                    ▷ Das erste Element von links
if $f = apex$ then
    if truncateApex(Left, $p$, $apex$) then
        return false
    end if
    $funnel \leftarrow p + funnel$
    return true
end if

$s \leftarrow$ head(tail($funnel$))              ▷ Das zweite Element von links
if $s = apex$ then
    $s \leftarrow apex_{\text{left}}$
end if

$o \leftarrow orientation(s, f, p)$
if $o =$ CounterClockWise then
    $funnel \leftarrow p + funnel$
    return true
end if

$funnel \leftarrow$ tail($funnel$)
return false
```

Algorithmus 6 truncateApex($s, p,$ **inout** $\overline{a_1a_2}$)

Eingabe: Der Linien-Scheitelpunkt $apex = \overline{a_1a_2}$, die zu verkürzende Seite s und der Punkt p zu dem verkürzt werden soll.

Rückgabe: Die Linie $\overline{a_1a_2}$ wird so verkürzt, dass der neue entsprechende Endpunkt der nächstgelegene Punkt zu p ist. Fällt dadurch der Scheitelpunkt auf einen Punkt zusammen, wird **true** , andernfalls **false** zurückgegeben.

```
if $s$ = Left then
    $a_1 \leftarrow$ closestPositionOnSegment($\overline{a_1a_2}, p$)
else
    $a_2 \leftarrow$ closestPositionOnSegment($\overline{a_1a_2}, p$)
end if

return $a_1 = a_2$
```

Algorithmus 7 truncate($\overline{ab}, \overline{cd}$)

Eingabe: Linien $\overline{ab}$ und $\overline{cd}$, die sich an einem der Endpunkte berühren

Rückgabe: Eine Linie mit der selben Ausrichtung wie $\overline{ab}$, die aber möglicherweise verkürzt wurde auf die Länge der Projektion von $\overline{cd}$ auf $\overline{ab}$. (Ist das Skalarprodukt von $\vec{ab}$ und $\vec{cd}$ kleiner gleich Null, fällt die zurückgegeben Linie auf einen Punkt zusammen.)

```
if $a = c \lor b = c$ then
    $p \leftarrow$ closestPositionOnSegment($\overline{ab}, d$)
    return $\overline{cp}$
else
    $p \leftarrow$ closestPositionOnSegment($\overline{ab}, c$)
    return $\overline{pd}$
end if
```

Algorithmus 8 orientation(a, b, c)

Eingabe: Die Punkte a, b, c.

Rückgabe: Gibt an ob der Pfad durch die Punkte a, b und c einen Knick im *Uhrzeigersinn* oder im *Gegen-Uhrzeigersinn* macht, oder ob die Punkte *Kolinear* sind.

```
$d_{x,1} \leftarrow b_x - a_x$
$d_{y,1} \leftarrow b_y - a_y$
$d_{x,2} \leftarrow c_x - a_x$
$d_{y,2} \leftarrow c_y - a_y$

if $d_{x,1} \cdot d_{y,2} < d_{y,1} \cdot d_{x,2}$ then
    return Clockwise
else if $d_{x,1} \cdot d_{y,2} > d_{y,1} \cdot d_{x,2}$ then
    return Counter-Clockwise
end if

return Colinear
```

4.1.3 Beispiel 1

Abbildung 4.2 zeigt einen beispielhaften Ablauf des *Funnel*-Algorithmus für Liniensegmente. Die ersten beiden Bilder zeigen den Kartenausschnitt in dem der kürzeste Weg zwischen den beiden grauen Linien gesucht wird, bzw. den ursprünglichen Kanal (wobei der aktuelle Scheitelpunkt gelb, die linken Kanal-Portale blau und die rechten Kanal-Portale grün eingezeichnet sind). Ähnlich wie beim *Funnel*-Algorithmus für Punktobjekte, wird hier das Ziel-Segment als letztes Portal an den Kanal angehängt.

Zuerst werden die Start- und Endsegmente gekürzt indem das erste bzw. das letzte Kanal-Portal darauf projiziert werden (Abbildung 4.2(c)). Die Endpunkte der ersten beiden Portale können ohne weitere Bearbeitung hinzugefügt werden (die Endpunkte des jeweils gerade bearbeiteten Portals sind ausgefüllt). Wenn wie in Abbildung 4.2(f) ein Punkt aus dem Trichter entfernt werden muss, bevor der Endpunkt des neuen Portals hinzugefügt werden kann, wird wieder überprüft, ob der Scheitelpunkt wieder gekürzt werden kann.

Nachdem in Abbildung 4.2(i) das letzte Portal bearbeitet wurde, entspricht die kürzeste Verbindung zwischen den Liniensegmenten der kürzeren der beiden Trichter-Seiten. In diesem Beispiel sind die Start- und Ziel-Segmente parallel, deswegen ist die kürzeste Verbindung eine beliebige Linie zwischen der linken und der rechten Seite des Trichters.

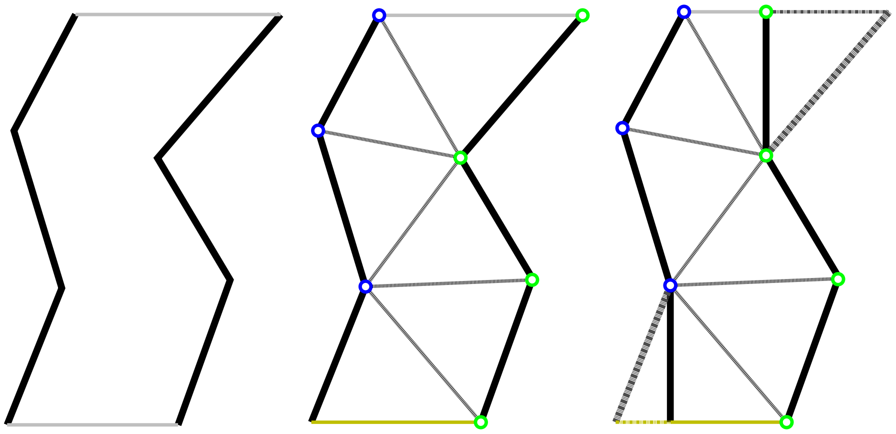
4.1.4 Beispiel 2

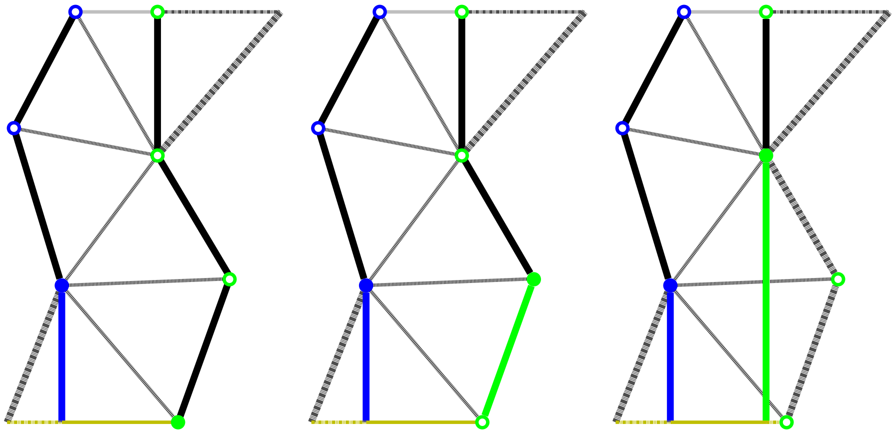
Abbildung 4.3 zeigt ein Beispiel, bei dem der Scheitelpunkt während der Durchführung des Algorithmus von einer Linie zu einem Punkt kollabiert.

Die ersten beiden Bilder zeigen wieder den Kartenausschnitt bzw. den ursprünglichen Kanal. In Abbildung 4.3(c) sieht man wieder wie Start- und End-Segmente angepasst werden, wobei beim Endsegment keine Änderung nötig ist. Die Endpunkte der ersten beiden Portale können dann ohne weitere Änderungen in den Trichter eingefügt werden. Wenn das dritte Portal bearbeitet wird, fällt der Scheitelpunkt von einer Linie auf einen Punkt zusammen.

Nach diesem Schritt versucht, der Algorithmus noch einmal den linken Endpunkt des dritten Portals zum Trichter hinzuzufügen, aber ab diesem Zeitpunkt wird dafür immer die Prozedur `addLeftPoint()` verwendet und der Algorithmus arbeitet ab hier wie der standard *Funnel*-Algorithmus.

Der kürzeste Weg zwischen den beiden Liniensegmenten ist die orange Linie die vom kollabierten Scheitelpunkt ausgeht plus die kürzere Seite des Trichters, in diesem Fall die Rechte.


(a) Kartenausschnitt (b) Original *Channel* (c) Anpassungen


(d) 1. Channelportal (e) 2. Channelportal (f) 3. Channelportal

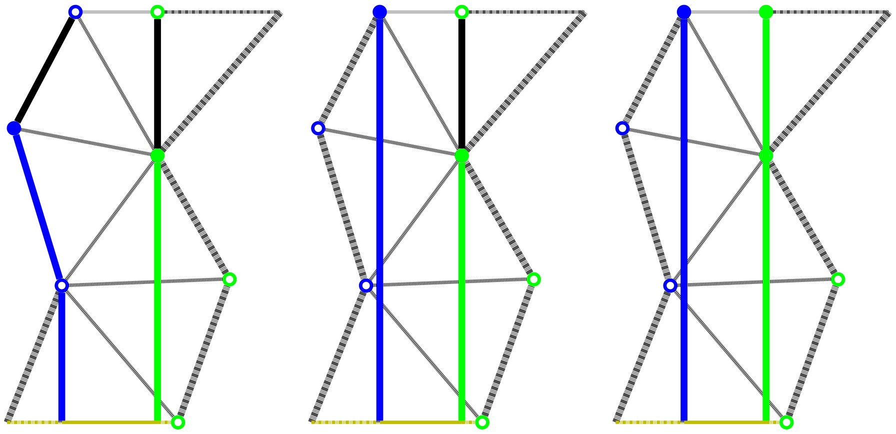
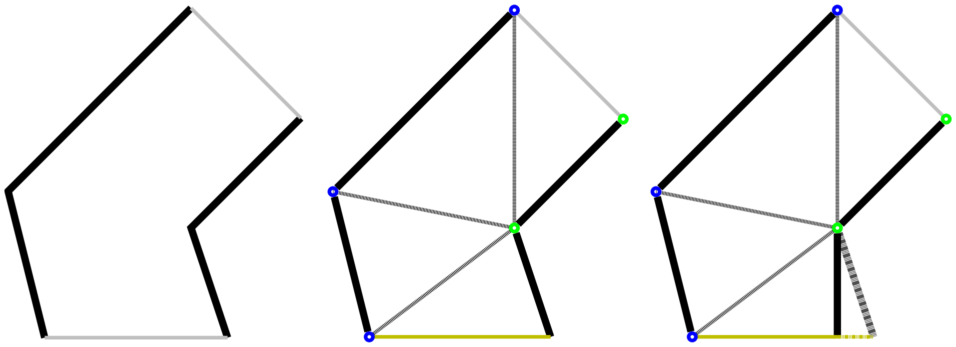
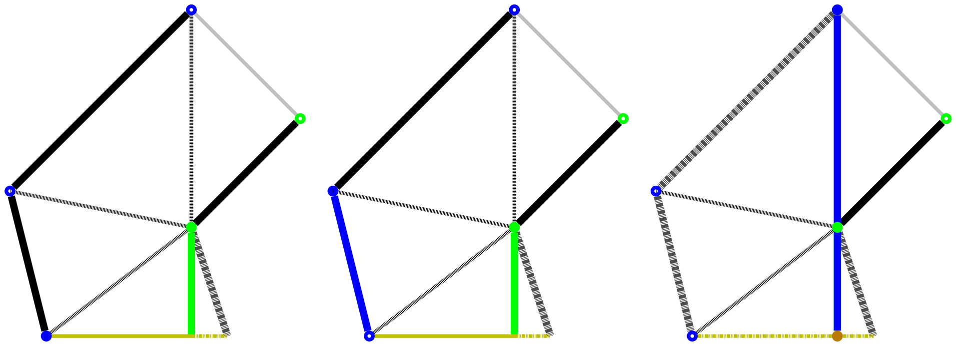

(g) 4. Channelportal (h) 5. Channelportal (i) Letztes Channelportal

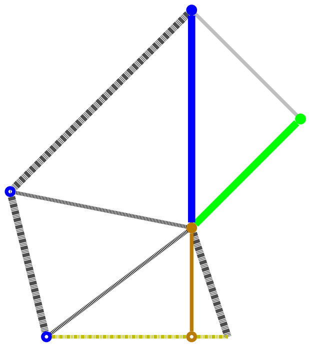
Abbildung 4.2: Beispiel für den Ablauf des Linien-Funnel-Algorithmus



(a) Kartenausschnitt (b) Original *Channel* (c) Anpassungen



(d) 1. Channelportal (e) 2. Channelportal (f) 3. Channelportal



(g) 4. Channelportal

Abbildung 4.3: Beispiel für den Ablauf des Linien-Funnel-Algorithmus

4.2 Das Mammoth MMO-Framework

4.2.1 Pathfinder für eine Triangulation Reduction

Der *Pathfinder* identifiziert zuerst die Dreiecke in denen sich der Start- bzw. der End-Punkt befinden, und versucht dann schrittweise entsprechende Level 3 Dreiecke als Start- bzw. End-Dreiecke zu ermitteln. Befindet sich der Startpunkt in einer Sackgasse, werden die Informationen in der Abstraktion verwendet um das Level 2 Dreieck des Korridors zu ermitteln in den die Sackgasse mündet. Nach diesem Schritt, bzw. wenn der Startpunkt schon in einem Korridor liegt, werden die Level 3 Dreiecke, die die Endpunkte des Korridors bilden, aus der Abstraktion gelesen. Diese werden zu den Start-Dreiecken für die spätere Suche mit A* oder *Fringe-Search*. Falls der Korridor ein Level 3 Dreieck mit sich selbst verbindet gibt es nur einen Startpunkt. Das ist auch der Fall, wenn sich der Startpunkt bereits in einem Level 3 Dreieck befindet.

Die selben Schritte werden für den Zielpunkt durchgeführt um ein oder zwei Zieldreiecke ausfindig zu machen.

Bevor aber das entsprechende Dreieck der nächst höheren Stufe ermittelt wird, wird geprüft, ob sich der Start- und der Zielpunkt in der selben Struktur befinden, weil das eine Suche unnötig macht. Befinden sich beide Punkte im selben Dreieck, werden sie einfach durch eine gerade Linie verbunden, befinden sie sich in der selben Sackgasse, wird der tiefste gemeinsame Vorgänger ermittelt und ein Pfad konstruiert. Befinden sich beide Punkte im selben Ring-Korridor ist ebenfalls keine Suche nötig, es muss nur überprüft werden ob der Weg im Uhrzeigersinn oder im Gegenuhrzeigersinn kürzer ist.

In allen anderen Fällen muss eine Suche durchgeführt werden, auch wenn die Start- und Zielpunkte sich im selben Korridor befinden. Es ist durchaus möglich, dass der kürzeste Pfad zwischen den beiden punkten, aus dem Korridor hinausführt, wie im in Abbildung 4.4 gezeigten Beispiel.

Erzeugung der Triangulation Reduction

Das Mammoth-Framework verwendet das triangle-Programm[2] von Jonathan Richard Shewchuk um aus einer Liste von Polygonen, die die Hindernisse darstellen, eine beschränkte Delaunay-Triangulation zu erstellen, siehe [16]. Der Mammoth-Karteneditor *Mape* dient als Front-End für die Einstellungsmöglichkeiten von triangle. Für die *Triangulation Reduction* ist es wichtig, dass die Eckpunkte der Dreiecke immer auf Hindernissen liegen und die Fläche der Dreiecke nicht zusätzlich beschränkt ist. Auf solche Art eingeschränkte Triangulationen bieten zwar

---

[2] Siehe `http://www.cs.cmu.edu/~quake/triangle.html`.

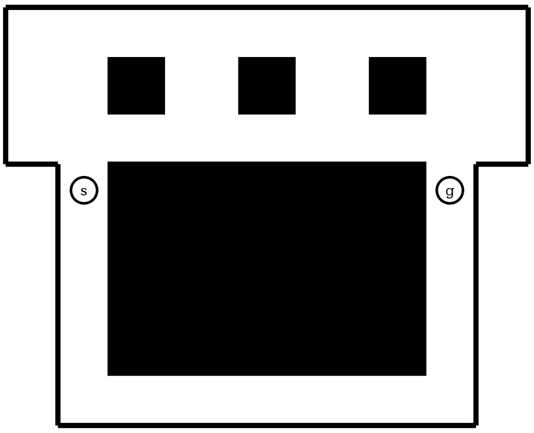


Abbildung 4.4: Beispiel für einen kürzesten Pfad der sich außerhalb des gemeinsamen Korridors befindet

bestimmte Vorteile, da die Zerlegung einheitlicher ist, aber für eine *Triangulation Reduction* dürfen keine „überflüssigen“ Dreiecke existieren, also muss eine Flächenbeschränkung in *Mape* deaktiviert werden.

Mammoth definiert den Wert `PhysicsEngineConstants.THRESHOLD`, der die minimale Differenz zweier Fließkomma-Zahlen angibt, die Mammoth noch als zwei verschiedene Zahlen erkennt. Dieser Wert beeinflusst auch, wie weit zwei Punkte auseinander sein müssen, damit sie von der *Physics Engine* auch tatsächlich als zwei verschiedene Punkte wahrgenommen werden.

Da das triangle-Programm diese Konstante aber nicht kennt, kann es bei komplexen Karten vorkommen, dass triangle Dreiecke erzeugt, dessen Eckpunkte von Mammoth nicht alle unterschieden werden können. Bevor deshalb die Abstraktion durchgeführt wird, stellen folgende Vorbearbeitungsschritte sicher, dass

- redundante Punkt- bzw. Kanten-Objekte, die Mammoth nicht unterscheiden kann, eliminiert und durch ein einziges Stellvertreter Objekt ersetzt werden, bzw. dass
- Dreiecke dessen Eckpunkte von Mammoth nicht unterschieden werden können durch eine einzige Kante bzw. einen einzigen Punkt ersetzt, und die Nachbardreiecke entsprechend angepasst werden.

Danach wird, wie in Abschnitt 3.2.1 beschrieben, der *Most Abstract Graph* erzeugt. Dabei wird zu jedem Dreieck neben seinem Typ auch gespeichert, zu welcher Struktur es gehört, sowie der Abstand und das nächste zu betretende Dreieck seiner Verbindungen.

Ein Level 1 Dreieck hat dabei entweder keine Verbindungen, wenn es sich um einen „Sackgassen-Baum“ handelt, oder eine Verbindung zu einem Level 2 Dreieck.

Ein Level 2 Dreieck hat entweder keine Verbindungen, oder je eine zu den nicht notwendigerweise verschiedenen Endpunkten des Korridors. Ein Level 3 Dreieck hat entweder eine Verbindung zu einem anderen Level 3 Dreieck, wenn das Dreieck eine Verbindung zu sich selbst hat, die in der Abstraktion nicht abgebildet wird, oder drei Verbindungen zu drei anderen Level 3 Dreiecken.

Die in der Abstraktion gespeicherten Längen dieser Verbindungen werden mit dem *Funnel*-Algorithmus für Linien-Segmente berechnet, siehe Abschnitt 4.1.2.

Wenn der Abstraktions-Prozess abgeschlossen ist, wird sie in einer Datei gespeichert um die nachfolgenden Ladevorgänge zu beschleunigen.

Implementierungen des Funnel-Algorithmus

Neben dem *Funnel*-Algorithmus für Linien-Segmente, der für die Erstellung der Abstraktion vorgesehen ist, wurde der in [4] beschriebene *Funnel*-Algorithmus für Punkt-Objekte, sowie eine Adapter-Klasse für den bereits vorhandenen modifizierten *Funnel*-Algorithmus für Objekte mit einem Radius größer Null.

Suchalgorithmen

Für die eigentliche Suche im *Most Abstract Graph* wurden der in [4] beschriebene TRA*, sowie ein für die Suche in einer Triangulation modifizierter *Fringe-Search* implementiert. Es wurde versucht möglichst viele Gemeinsamkeiten zu identifizieren und in eine gemeinsame Basisklasse auszulagern, wie zum Beispiel die Erzeugung von Nachfolgezuständen oder die Initialisierung der Heuristik.

4.2.2 Verschiedenes

Die folgenden Algorithmen(-Sammlungen) und Datenstrukturen wurden zwar für die Realisierung von TRA* implementiert, sind aber allgemein genug gehalten, um auch in anderen Bereichen des Mammoth-Frameworks eingesetzt zu werden.

Der GeometryHelper ist eine Klasse mit effizienten Implementierungen für verschiedene geometrische Problemstellungen, wie

- ob der Weg über drei Punkte im Uhrzeigersinn, im Gegen-Uhrzeigersinn, oder in einer Linie verläuft
- ob ein durch drei Punkte definierter Winkel spitz bzw. stumpf ist
- der Abstand zwischen einem Punkt bzw. einer Kante zu einer Kante
- der nächstgelegene Punkt auf einem Liniensegment zu einem anderen Punkt

In den jeweiligen Implementierungen wurde vor allem darauf geachtet, die Anzahl an Aufrufen der trigonometrischen Funktionen und der Wurzel-Funktion möglichst gering zu halten oder ganz zu vermeiden.

Die SplitList ist eine doppelt verkette zyklische Liste mit einem speziell ausgezeichnetem *Split*-Element. Die Implementierung unterstützt effizientes Einfügen am Anfang und am Ende der Liste, sowie direkt vor und nach dem *Split*-Element.

Die Methoden `advance()` und `removeAndAdvance()` verschieben den Zeiger auf das *Split*-Element um eine Position nach rechts bzw. entfernen das aktuelle *Split*-Element und machen dessen Nachfolger zum neuen *Split*-Element. Beide Methoden weisen darauf hin, ob dadurch der Zeiger auf das *Split*-Element über das Ende der Liste hinaus und wieder auf das erste Listenelement verschoben wurde.

Durch den Zeiger auf das *Split*-Element kann auch sehr einfach über die Listenelemente vom *Split*-Element zum ersten bzw. zum letzten Element iteriert werden.

Die *SplitList* wird verwendet um die *Open*-Liste in der *Fringe-Search* Implementierung zu realisieren, wobei das *Split*-Element die Grenze zwischen den *Now*- und *Later*-Listen darstellt, siehe Abschnitt 4.1.1. Außerdem wird die Datenstruktur verwendet um den aktuellen Zustand des Trichters in den Implementierungen des *Funnel*-Algorithmus zu speichern, wobei das *Split*-Element den aktuellen Scheitelpunkt darstellt, siehe auch die Abschnitte 2.4.4 und 4.1.2.

Der LazyCache ist ein Cache der mit einem Aufwand in $O(1)$ gelöscht werden kann. Die Verwendung eines solchen Caches wird in [1] zum Zwischenspeichern von besuchten Zuständen und deren g-Werten empfohlen. Wie in Abschnitt 4.1.1 erwähnt, können diese Werte für eine Suche in einer Triangulation nicht gecached werden, aber der *LazyCache* wird verwendet um die h-Werte von Suchzuständen zwischen zu speichern.

Der TestLauncher ließt die Einstellungen aus der globalen Mammoth-Konfigurationsdatei und startet automatisch eine entsprechende Anzahl an Pfadsuche-Anfragen und speichert die jeweiligen Statistiken in eine CSV-Datei. Es kann entweder einfach die gewünschte Anzahl an Test-Datensätzen angegeben werden, oder eine Anzahl an Fächern und wieviele Datensätze pro Fach erzeugt werden sollen. Im letzteren Fall werden die erzeugten Testdaten nach der Pfadlänge auf die Fächer aufgeteilt, bis für jedes Fach die nötige Anzahl an Datensätzen erzeugt wurde.

Das Mammoth-Framework hatte bereits einen Kommandozeilen-Client, der aber nur für Tests von Netzwerkprotokollen verwendet wurde. Mit dem *TestLauncher* können nun auch Testreihen für Pfadsuche-Implementierungen automatisiert werden.

KAPITEL 5

Tests

In diesem Abschnitt werden die Tests behandelt, mit denen die Suchalgorithmen TRFringeSeach und TRA* verglichen wurden.

Der Aufbau der Tests wird in Abschnitt 5.1 beschrieben und in Abschnitt 5.2 werden die Resultate vorgestellt.

5.1 Aufbau

Die Tests wurden auf fünf verschiedenen Karten durchgeführt, wovon jeweils zwei repräsentativ für Außenbereiche bzw. Innenbereiche sind und eine Karte Merkmale für beide Arten von Bereichen aufweist.

Pro Karte wurden jeweils 2000 Datensätze aufgeteilt auf 100 Intervalle generiert. Die maximale Pfadlänge für die betrachteten Pfade war immer das Maximum aus Kartenhöhe und Kartenbreite. Diese maximale Länge wurde in 100 gleichmäßige Intervalle geteilt, und pro Intervall wurden 20 Datensätze generiert. Um diese Tests durchzuführen wurde der Kommandozeilen-Client des Mammoth-Frameworks um ein entsprechendes Test-Starter-Modul erweitert.

Sämtliche Pfadsuche-Anfragen wurden mit einem Punkt-Objekt ohne Radius und mit einem bedingten Timeout von 2 Sekunden durchgeführt, siehe auch Abschnitt 5.2.1. Nach zwei Sekunden wurde der beste bis dahin verfügbare Pfad zurückgegeben. Es war sehr wohl möglich, dass eine Anfrage länger dauert, wenn innerhalb dieser zwei Sekunden noch kein Pfad gefunden wurde. Für die Graphiken, die sich mit der Verbesserung vom ersten gefundenen bis zum endgültigen Pfad befassen, wurde das Zeitfenster in 10 gleichmäßige Intervalle geteilt.

Alle Tests wurden auf einem Rechner mit zwei 3,16 GHz E8500 Intel Core2 Duo CPUs und 4 GB RAM unter Debian 6.0 im Einzelbenutzer-Modus durchgeführt. Dabei kam der Compiler bzw. die JVM des Java Development Kits 1.6 zur Verwendung.

5.2 Ergebnisse

Die Abbildungen 5.1 und 5.2 zeigen die Gesamt-Laufzeiten bzw. die Laufzeiten des eigentlichen Suchalgorithmus über die verschiedenen Pfadlängen. Während A* die 200 ms Grenze (in beiden Graphiken) nicht überschreitet, kommt es bei *Fringe-Search* schon bei wesentlich kürzeren Pfaden zu Ausreißern, bei denen die reine Suchzeit eine Sekunde beträgt bzw. die gesamte Suchanfrage das Timeout erreicht. Ab etwa 70 % der maximalen Pfadlänge ist das sogar bei etwa zwei Dritteln der Anfragen der Fall.

Dass die reine Suchzeit der Anfragen im letzten Intervall noch immer 200 ms unter dem Timeout liegt, und die Mehrheit dieser Suchanfragen trotzdem vorzeitig abgebrochen wurde, ist ein Hinweis darauf, dass *Fringe-Search* deutlich mehr Pfade testen muss als A*.

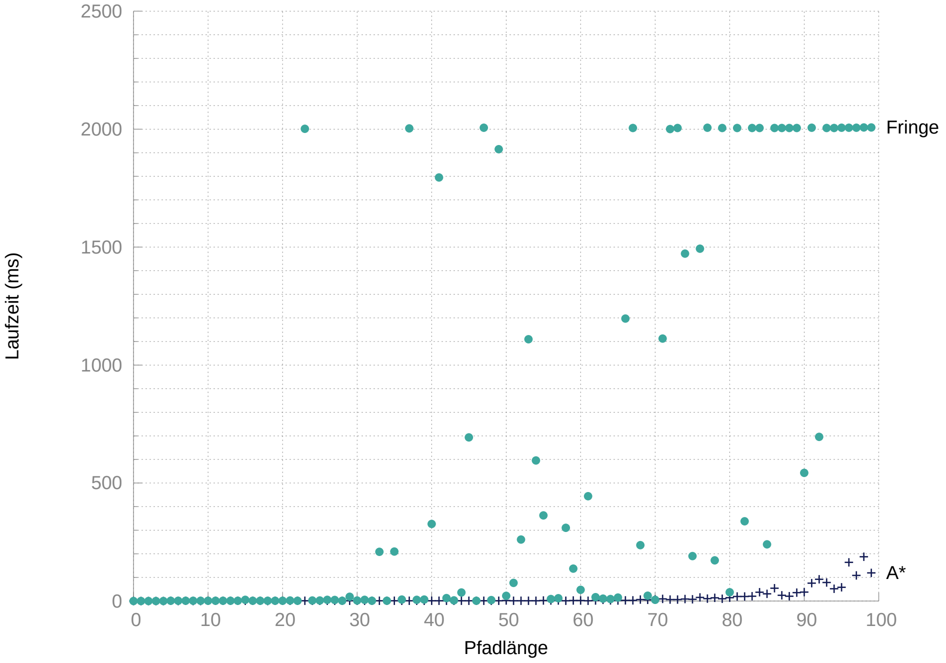


Abbildung 5.1: 90 %-Quantile der Gesamt-Laufzeit nach Pfadlänge

In der Abbildung 5.3 wird die Länge des ersten gefundenen Pfades in Abhängigkeit der Länge des Ergebnispfades betrachtet, und Abbildung 5.4 zeigt die

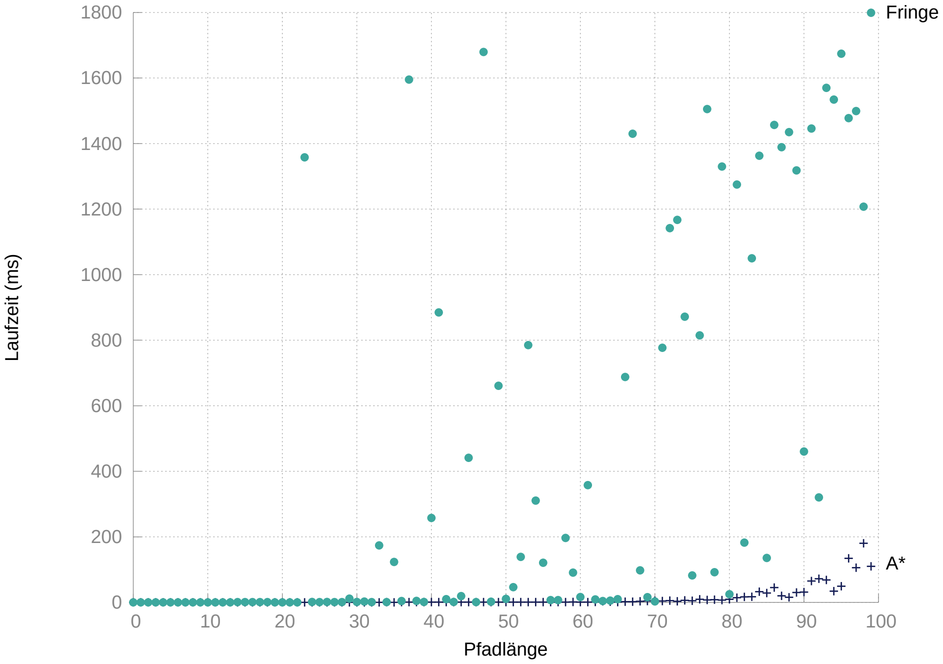


Abbildung 5.2: 90 %-Quantile der Reinen Suchzeit nach Pfadlänge

dafür benötigte Zeit. Beide Algorithmen finden, bis auf einige Ausreißer bei *Fringe-Search*, sehr schnell einen initialen Pfad, wobei die Pfade von A* schon zu Beginn eine höhere Qualität aufweisen.

Um wieviel mehr Zeit die Algorithmen benötigen um vom ersten gefundenen Pfad zum Ergebnis-Pfad zu kommen ist in der Abbildung 5.5 als ein Vielfaches der Laufzeit bis zum ersten gefundenen Pfad dargestellt. Da bei A* ein Großteil der ersten Pfade in einer Millisekunde gefunden wurden, ähnelt diese Kurve bei A* sehr stark der der Gesamt-Laufzeit.

Die Abbildungen 5.8 und 5.9 zeigen jeweils für A* bzw. *Fringe-Search* wieviele Pfade insgesamt getestet wurden, und der wievielte Pfad schließlich als der Beste Pfad erkannt wurde. Bei beiden Suchalgorithmen ist sehr schön sichtbar, dass ein Großteil der Ausführungszeit eigentlich darauf zurück geht, die Optimalität des besten Pfades nachzuweisen. Zu beachten ist, dass die Skalen dieser beiden Graphiken sehr unterschiedlich sind, Abbildung 5.12 visualisiert die Unterschiede in der Größenordnung der getesteten Pfade bei A* und *Fringe-Search*.

Abbildung 5.6 zeigt, um welchen Faktor der Initial-Pfad länger war, als der Ergebnis-Pfad. Bei Abbildung 5.7, die den durchschnittlichen Fortschritt der Algorithmen über den Zeitverlauf zeigt, ist zu beachten, dass ein Datensatz immer den Fortschritt Null hat, wenn die optimale Lösung bereits *vor* dem ersten Über-

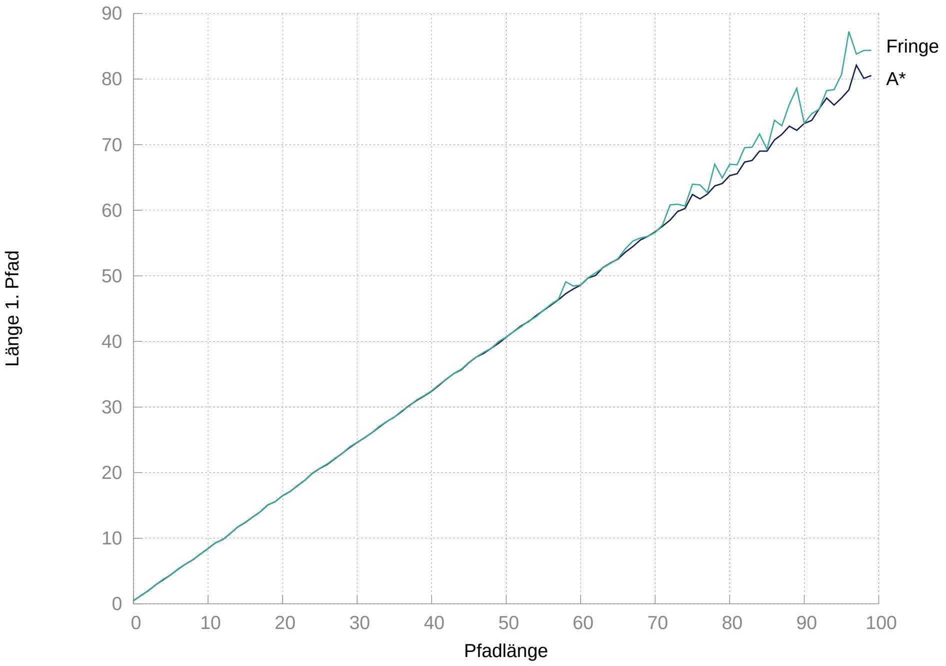


Abbildung 5.3: 90 %-Quantile der Länge des ersten Pfades

prüfungszeitpunkt gefunden wurde. Die tatsächliche Verbesserung ist dann in Abbildung 5.6 ersichtlich. In den Abbildungen 5.7 und 5.6 wird zwar deutlich, dass *Fringe-Search* einen deutlich größeren Nutzen von einer Implementierung als *Anytime*-Algorithmus zieht als A*, allerdings sind die Verbesserungen im Vergleich zur wesentlich höheren Laufzeit verschwindend gering.

In den Abbildungen 5.10 und 5.11 werden schließlich noch die besuchten bzw. expandierten Zustände bei den beiden Algorithmen gegenüber gestellt. Dass die Zahl der tatsächlich expandierten Zustände bei *Fringe-Search* in den meisten Fällen ähnlich ist, wie die Zahl der besuchten Zustände ist eine Erklärung für das generell sehr schlechte Abschneiden von *Fringe-Search* in einer triangulierten Umgebung: Eine der Ideen hinter *Fringe-Search* ist, dass das besuchen eines Zustandes eigentlich sehr günstig ist, und es zu einem Geschwindigkeitsgewinn führt, wenn man zwar viel mehr Zustände besuchen muss, dafür aber auf die Sortierung der *Open*-Liste verzichten kann. Wenn allerdings der Großteil der besuchten Zustände auch expandiert werden muss, fällt dieser eventuelle Vorteil sofort weg.

5.2.1 Modifizierter Funnel-Algorithmus

Es wurde auch eine Testreihe mit dem in [4] vorgestellten modifizierten *Funnel*-Algorithmus durchgeführt, der einen optimalen Pfad durch einen gegebenen Kanal

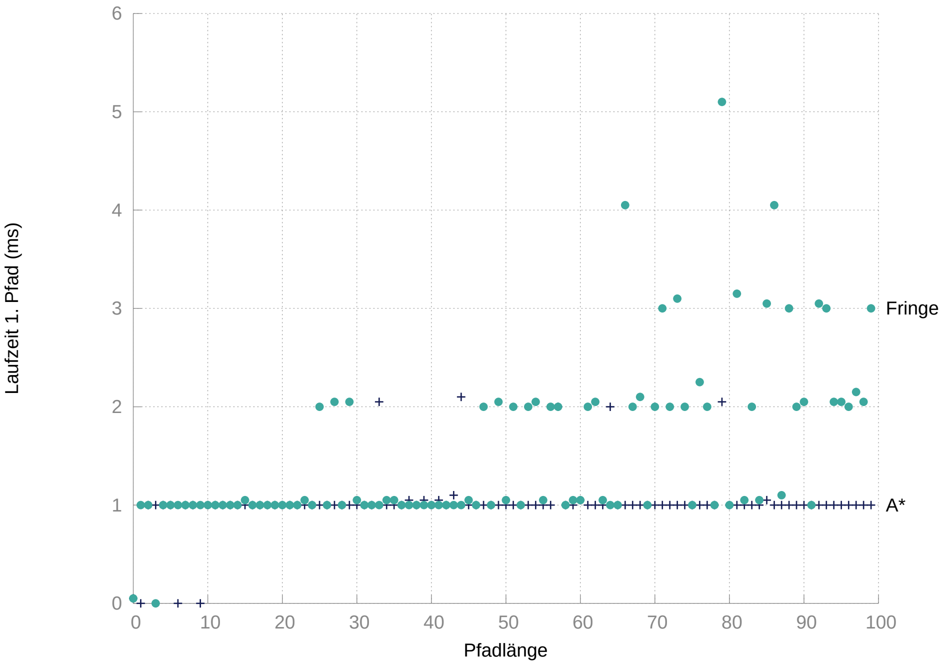


Abbildung 5.4: 95 %-Quantile Laufzeit bis zum ersten Pfad

für ein Objekt mit einem bestimmten Radius bestimmt.

Die Verwendung des modifizierten *Funnel*-Algorithmus ließ allerdings die Laufzeiten bei beiden Algorithmen extrem ansteigen, so dass bereits bei Pfaden mit einer Länge von 40 % der maximalen Pfadlänge die Anfragen nicht mehr im Zeitfenster von 2 Sekunden vollendet werden konnten (Abbildung 5.13).

Sehr starken Einfluss auf die längeren Laufzeiten hat die Anzahl an getesteten Pfaden, die, wie man in Abbildung 5.14 sieht, ebenfalls stark ansteigen. Ohne den modifizierten *Funnel*-Algorithmus kommt A* nicht über 1000 getestete Pfade und auch *Fringe-Search* braucht erst bei den längeren Pfaden deutlich mehr, siehe Abbildung 5.12.

Der Grund dafür ist, dass die Pfade die für ein Objekt mit Radius berechnet werden natürlich länger sind, als die Pfade für Punkt-Objekte. Da die *Triangulation Reduction Abstraction* eine Abstraktion für beliebig große Objekte sein soll, und alle in der Abstraktion gespeicherten Abstände untere Schranken sein müssen, sind die für den Suchgraphen verwendeten Abstände alle für Punkt-Objekte, vgl. [4]. Diese Daten werden bei der Bestimmung der f-Werte verwendet. Die Suche wird erst abgebrochen, wenn der kleinste f-Wert in der *Open*-Liste größer ist, als die Länge des besten bis jetzt gefundenen Pfades. Wenn die tatsächliche Länge der Pfade für Objekte mit Radius, die f-Werte allerdings für Punkt-Objekte berech-

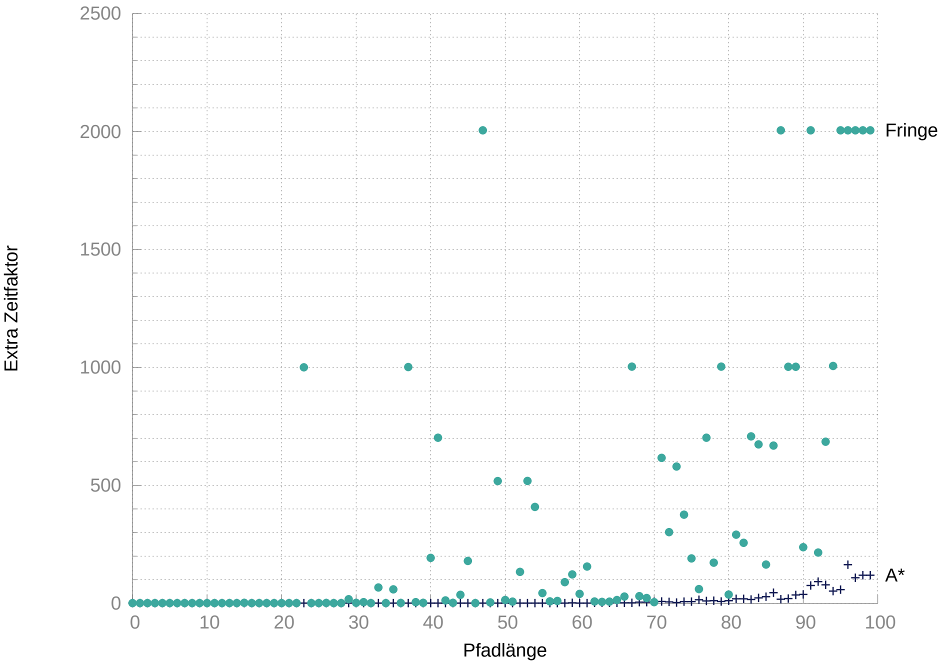


Abbildung 5.5: 95 %-Quantile der benötigten Extrazeit (Faktor)

net werden, muss der Algorithmus ungleich mehr Pfade testen, um ein optimales Ergebnis nachweisen zu können.

5.3 Interpretation

In den meisten Abbildungen fällt auf, dass die Ergebnisse bei *Fringe-Search* sehr stark variieren. Wie lange eine Pfadsuche-Anfrage bei *Fringe-Search* dauert, lässt sich also sehr schwer abschätzen und scheint stark von der aktuellen Suchanfrage abzuhängen.

Sowohl die Anzahl an expandierten und besuchten Suchzuständen, als auch die Anzahl der getesteten Pfade weisen darauf hin, dass der Wegfall der *Closed*-List bei *Fringe-Search* viel schwerer wiegt als bei A*. Während bei A* die meisten dieser überflüssigen Zustände weiter hinten in der *Open*-Liste landen und sich die Suche mit der Zeit mehr auf die vielversprechenderen Zustände konzentriert, können ein paar unglücklich expandierte Zustände bei *Fringe-Search* dazu führen, dass die *Open*-Liste explodiert und sich die Suche nicht auf die Zustände mit niedrigem f-Wert konzentrieren kann.

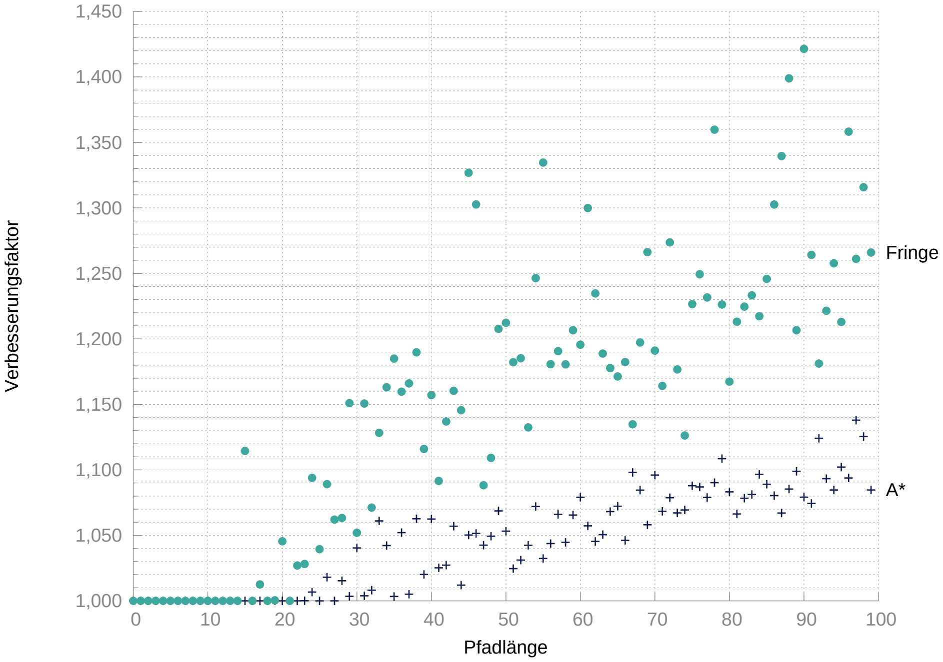


Abbildung 5.6: 90 %-Quantile der relativen Verbesserung zum ersten Pfad

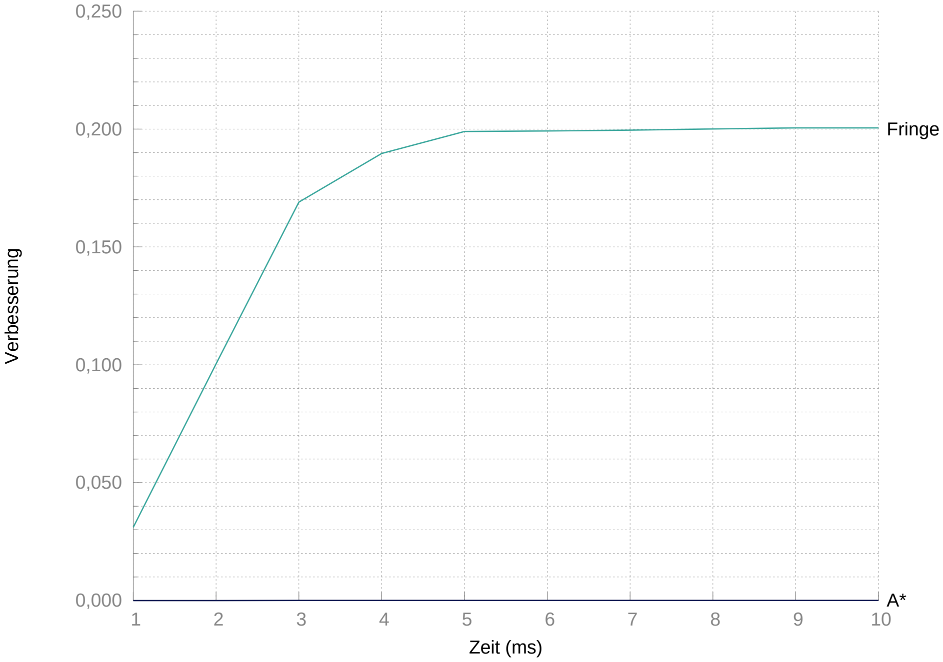


Abbildung 5.7: Durchschnittswerte der relativen Verbesserungen über die Zeit

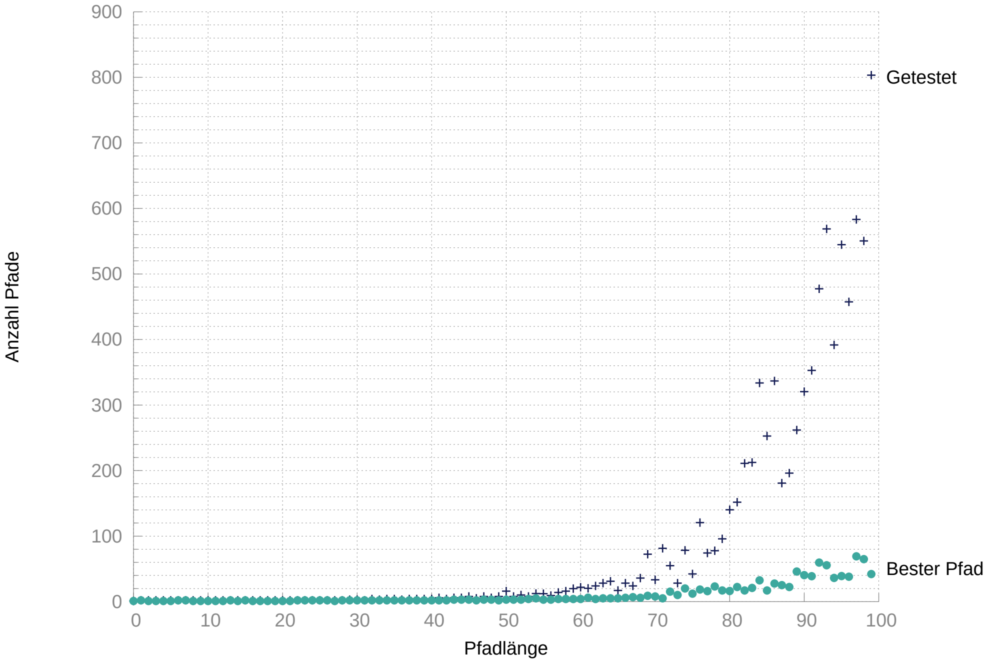


Abbildung 5.8: Vergleich getesteter Pfade zum besten Pfad mit A* (90 %-Quantile)

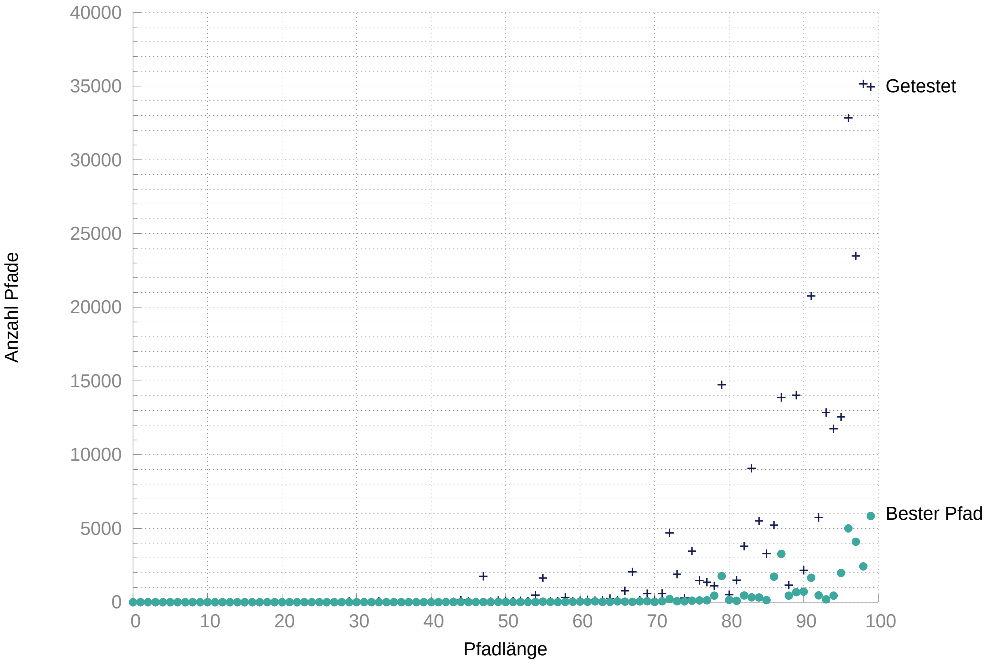


Abbildung 5.9: Vergleich getesteter Pfade zum besten Pfad mit Fringe-Search (90 %-Quantile)

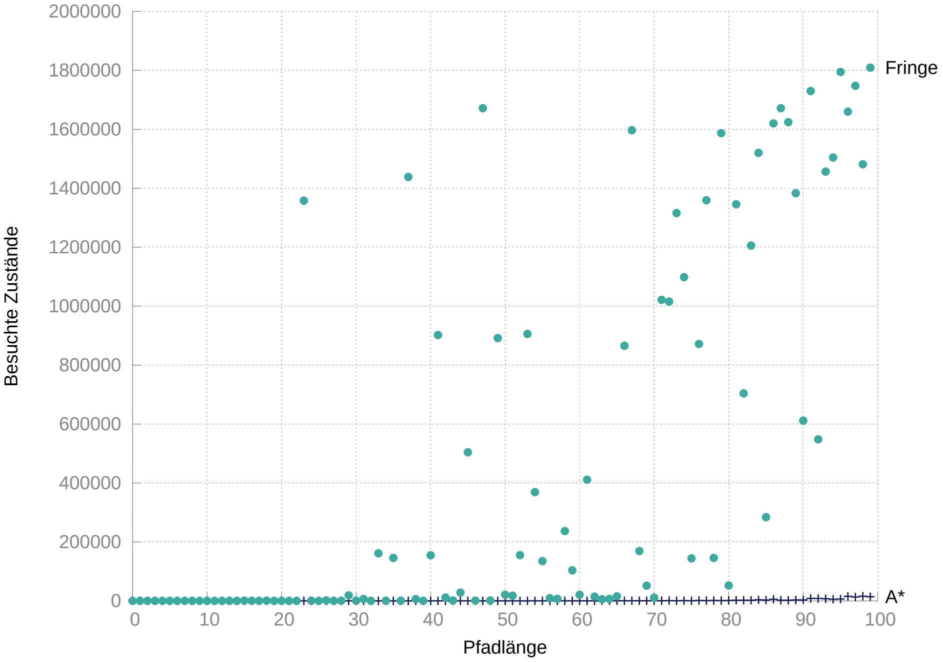


Abbildung 5.10: 90 %-Quantile der besuchten Zustände

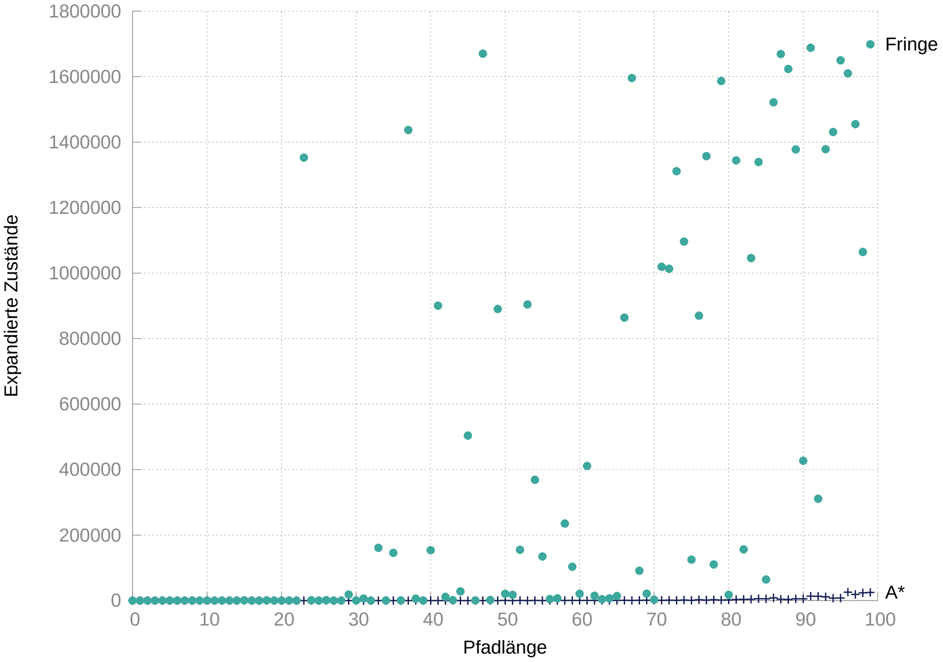


Abbildung 5.11: 90 %-Quantile der expandierten Zustände

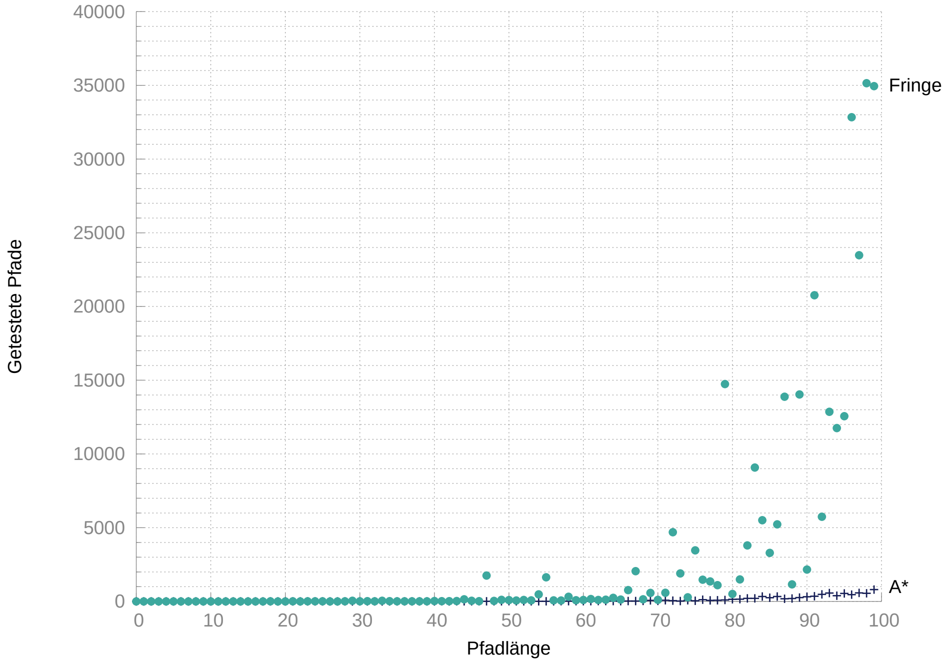


Abbildung 5.12: 90 %-Quantile der Anzahl an getesteten Pfaden

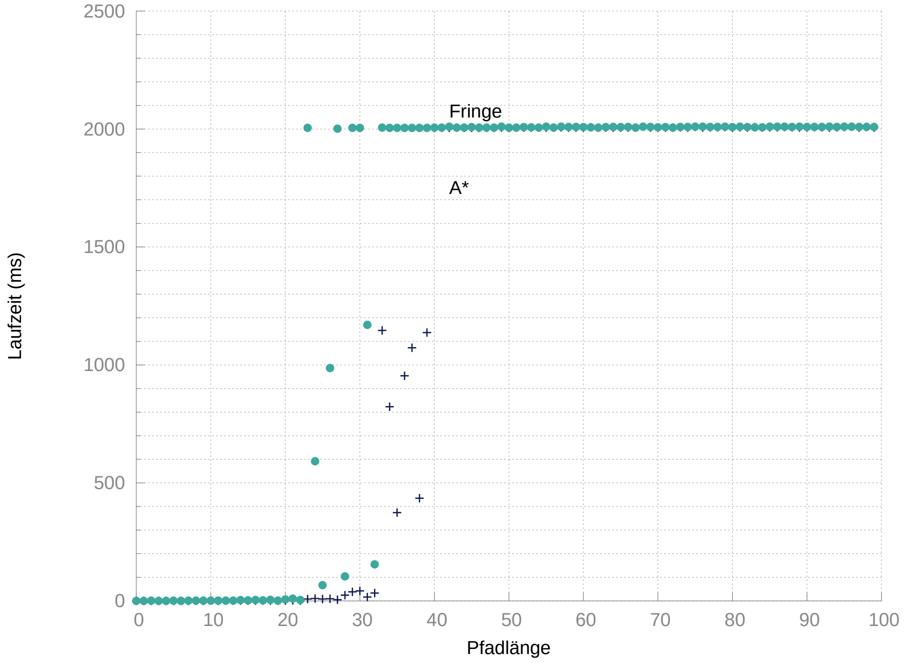


Abbildung 5.13: 90 %-Quantile der Laufzeit (MF)

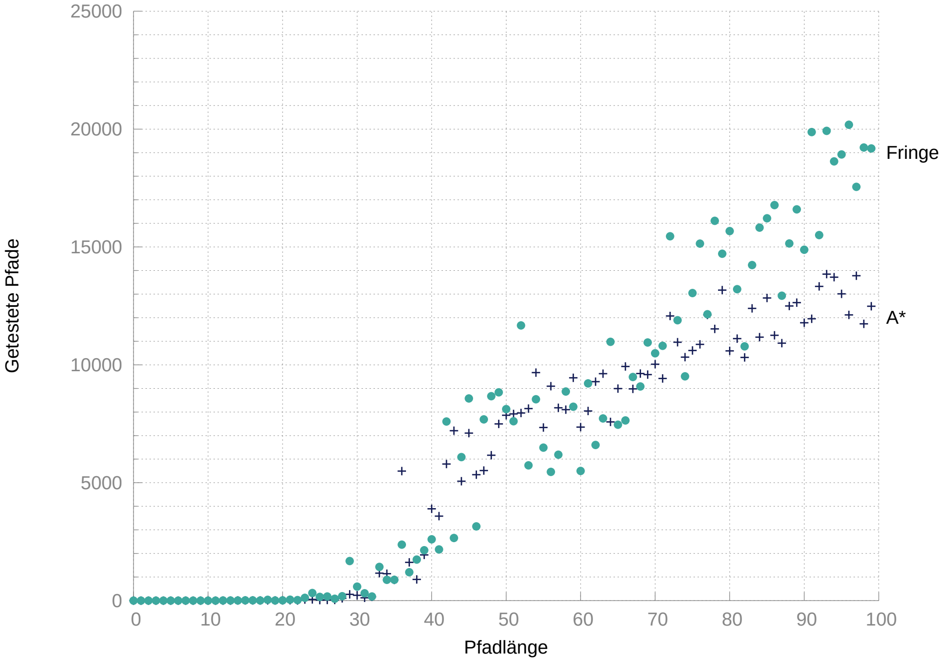


Abbildung 5.14: 90 %-Quantile der Anzahl an getesteten Pfaden (MF)

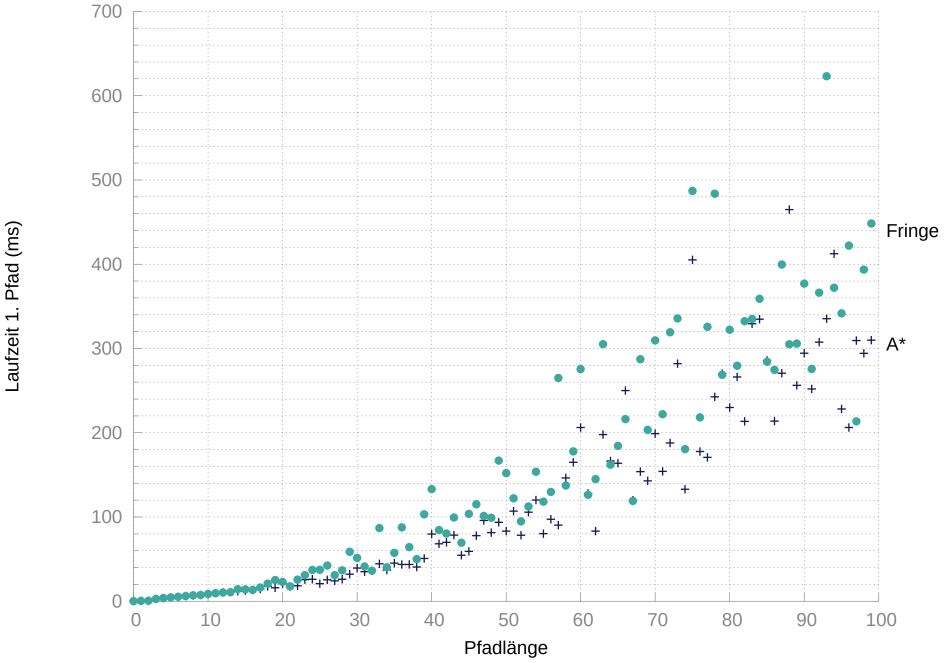


Abbildung 5.15: 90 %-Quantile der Laufzeit bis zum ersten Pfad (MF)

KAPITEL 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Das Hauptaugenmerk dieser Arbeit war eine möglichst ausführliche Zusammenfassung über die Methoden eine Spiel-Umgebung, so am Computer zu repräsentieren, dass eine effiziente Pfadsuche möglich ist, und die Implementierung eines Pfadsuche-Algorithmus, der mit einer TR im MMO-Research-Framework Mammoth arbeitet.

Für den Aufbau der Abstraktion wurde ein Konvertierungs-Algorithmus implementiert, der das Ergebnis des `triangle` Programms[1] in eine Triangulation umwandelt, die für eine *Triangulation Reduction* in Mammoth verwendet werden kann. Weiters wurde eine Variante des *Funnel*-Algorithmus entwickelt und implementiert, die die kürzeste Verbindung zwischen zwei Liniensegmenten in einem Kanal berechnet, um eine möglichst genaue Abschätzung der Längen von Korridoren in der Abstraktion zu ermöglichen.

Es wurden verschiedene geometrische Algorithmen sowie zwei Datenstrukturen implementiert, die nicht nur für die Pfadsuche in einer *Triangulation Reduction* nötig sind, sondern die in verschiedenen Modulen des Mammoth-Frameworks einsetzbar sind: Algorithmen, die die Orientierung bzw. die Abstände zwischen Punkten bzw. Kanten bestimmen, sowie die Datenstrukturen `SplitList` und `LazyCache`. Auch musste die Architektur des Pfadsuche-Moduls von Mammoth erweitert werden, um die Verwendung von *Anytime*-Algorithmen und ein Timeout für Suchanfragen zu ermöglichen.

Ein guter Teil der Arbeit bei der Pfadsuche in einer *Triangulation Reduction* besteht daraus, die den Start- und Zielpunkten nächstgelegenen Entscheidungs-

---

[1] `http://www.cs.cmu.edu/~quake/triangle.html`

punkte zu finden, die dann zu den Start- und Zielknoten für die Suchalgorithmen werden. Bei manchen Suchanfragen muss gar keine Suche auf dem Graphen durchgeführt werden, aber wenn es zu einer Suche im Graphen kommt wurden zwei verschiedene Suchalgorithmen implementiert: `TRAStar` und `TRFringeSearch`.

Um die beiden Suchalgorithmen zu vergleichen wurde, der Kommandozeilen-Klient des Mammoth-Frameworks um ein Testmodul erweitert, mit dem, durch die Konfigurations-Datei gesteuert, ganze Testreihen mit verschiedenen Parametern und auf verschiedenen Karten automatisiert durchgeführt werden können.

Verglichen wurde bei diesen Tests die Performanz der beiden Suchalgorithmen `TRAStar` und `TRFringeSearch`, um festzustellen, ob *Fringe-Search* für eine Anwendung in einer Triangulation geeignet ist.

6.2 Weitere Punkte

Die folgenden Punkte sind während der Implementierung der oben genannten Punkte aufgefallen, und könnten in zukünftigen Arbeiten genauer untersucht werden.

Verschiedene Abbruch-Kriterien für den *Anytime*-Pfadsuche-Algorithmus. In der jetzigen Form stellt der implementierte `TRPathFinder` eine Möglichkeit zur Verfügung, die Suche abzubrechen, wenn von dem zu befüllenden Pfad-Objekt die `Path.cancel()`-Methode aufgerufen wird. Zur Zeit ist das nur der Fall, wenn eine Suchanfrage die maximale Suchzeit überschreitet, aber es wäre durchaus möglich, ein Modul zu integrieren, dass die Rechenzeit aller anfallenden Aufgaben überwacht und den Pfadsuche-Anfragen je nach Auslastung mehr oder weniger Zeit zur Verfügung stellt.

Die Pfadsuche läuft immer mindestens so lange, bis ein erster Pfad gefunden wurde, oder festgestellt wurde, dass kein Pfad existiert. Danach wird in regelmäßigen Abständen überprüft, ob die Suche abgebrochen, oder fortgesetzt werden soll, bis eine beweisbar optimale Lösung gefunden wurde.

Aus den Testdaten ist ersichtlich, dass der größte Fortschritt eher am Anfang der Suchzeit gemacht wird, während die restliche Zeit bis zum Timeout hauptsächlich dem Beweis dient, dass der aktuell gefundene Pfad tatsächlich optimal ist.

Interessant wären also Experimente mit verschiedenen Abbruchkriterien für die Pfadsuche, zum Beispiel über die relative Verbesserung der Pfadlänge in den letzten Durchgängen.

Wie in [12] erwähnt, kann es durchaus Sinnvoll sein, die Optimalität zu vernachlässigen, solange die Ergebnis-Pfade natürlich und intelligent aussehen, und der *Anytime*-Algorithmus könnte sich in vielen Fällen die Überprüfung eines Großteils der Pfade ersparen, wenn der resultierende Pfad nicht beweisbar optimal sein muss.

Vermeidung des modifizierten *Funnel*-Algorithmus Für den Großteil der Tests aus dem vorherigen Kapitel kam der *Funnel*-Algorithmus für Punkt-Objekte zum Einsatz und nicht die modifizierte Version, die den Radius des zu bewegenden Objektes beachtet, damit sie nicht an den Ecken hängen bleiben. Die Tests haben gezeigt, dass durch die Verwendung des modifizierten *Funnel*-Algorithmus die Laufzeiten für die Suchanfragen extrem ansteigt. Eine Möglichkeit das zu vermeiden, wäre den normalen *Funnel*-Algorithmus für Punkt-Objekte zu verwenden und die Pfade lokal zu „reparieren“. Allerdings gibt es im Mammoth-Framework gegenwärtig noch keine Möglichkeit die Bewegung eines Objekts direkt zu beeinflussen. Objekte werden bewegt indem ihnen ein `Path`-Objekt zugewiesen wird, das sie dann strikt ablaufen.

Zusätzliche Daten in der Abstraktion In seltenen Fällen kann es in der Gegenwärtigen Implementierung dazu kommen, dass der g-Wert eines Suchzustandes überschätzt wird. Das passiert dann, wenn zwei benachbarte Level-3-Dreiecke sich in einem Punkt, oder sogar entlang einer Kante, berühren. In diesem Fall ist die einzige untere Schranke, die für den Abstand der beiden Dreiecke zueinander verwendet werden kann Null. Um Kanten mit einem Gewicht von Null zu vermeiden, wird der Verbindung in diesem Fall ein Gewicht von `PhysicsEngineConstants.THRESHOLD` zugewiesen. Dabei handelt es sich um die minimale Distanz, die zwei Koordinaten zueinander haben müssen, damit Mammoth sie als verschieden betrachtet. Da es jetzt sein kann, dass in einem kompletten Pfad, der Abschnitt in dem diese beiden Dreiecke passiert werden, tatsächlich die Länge Null hat, weil der *Funnel*-Algorithmus für Punkt-Objekte verwendet wird, während die Abschätzung für den g-Wert aber mindestens den Wert `PhysicsEngineConstants.THRESHOLD` hat, kann der g-Wert in manchen Fällen überschätzt werden.

Dies ließe sich vermeiden, wenn in der Abstraktion zu den Korridoren, die Summe der Winkel zwischen den passierten Kanten gespeichert würde und der Suchalgorithmus den Winkel zwischen der Eintritts- und der Austrittskante bestimmen würde. Wie in [4] dargestellt, wäre der Radius des zu bewegenden Objekts multipliziert mit der Summe dieser Winkel eine untere Schranke für die Distanz zwischen den beiden Level 3 Dreiecken.

Die aktuelle Implementierung der Suchalgorithmen vermeidet den direkten Zugriff auf die Daten der Dreiecke und teure Winkeloperationen, und arbeitet mit abstrahierten Suchzuständen, die nicht viele Informationen über die tatsächliche Geometrie der Spielwelt enthalten.

Wie oben erwähnt, kann der Wegfall der Optimalitäts-Einschränkung die Effizienz der Pfadsuche erhöhen für den Preis eines geringen Qualitätsverlustes der Ergebnispfade, aber man könnte überprüfen, ob die Elimination dieser seltenen Überschätzungen, den nötigen Zusatzaufwand wert wären.

Literaturverzeichnis

[1] Yngvi Björnsson u. a. „Fringe search: beating A* at pathfinding on game maps“. In: *Proceedings of IEEE Symposium on Computational Intelligence and Games (CIG’05).* Hrsg. von Graham Kendall und Simon Lucas. Colchester, Essex, UK, 2005, S. 125–132.

[2] Adi Botea, Martin Müller und Jonathan Schaeffer. „Near optimal hierarchical path-finding“. In: *Journal of Game Development.* Hrsg. von Michael van Lent. Bd. 1. Hingham, Massachusetts, USA: Charles River Media, 2004, S. 7–28.

[3] Douglas Demyen und Michael Buro. „Efficient triangulation-based pathfinding“. In: *Proceedings of the 21st national conference on Artificial intelligence (AAAI’06).* Bd. 1. Boston, Massachusetts, USA: AAAI Press, 2006, S. 942–947.

[4] Douglas Jon Demyen. „Efficient Triangulation-Based Pathfinding“. Magisterarb. Edmonton, Altberta, Canada: Department of Computing Science, University of Alberta, 2007.

[5] Edsger W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* 1 (1959), S. 269–271.

[6] Scott D. Goodwin, Samir Menon und Robert G. Price. „Pathfinding in Open Terrain“. In: *Proceedings of International Academic Conference on the Future of Game Design and Technology.* Hrsg. von Jay Rajnovich und Brian Winn Mike Katchabaw. London, Ontario, Canada, 2006.

[7] Peter Hart, Nils Nilsson und Bertram Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968). Hrsg. von John N. Warfield, S. 100–107.

[8] M. Renee Jansen und Michael Buro. „HPA* Enhancements“. In: *AIIDE’07: Conference on Artificial Intelligence and Interactive Digital Entertainment.* Hrsg. von Jonathan Schaeffer und Michael Mateas. Menlo Park, California, USA: AAAI Press, 2007, S. 84–87.

[9] Richard E. Korf. „Depth-first Iterative-Deepening: An Optimal Admissible Tree Search“. In: *Artificial Intelligence* 27 (1985), S. 97–109.

[10] Richard E. Korf und Weixiong Zhang. „Divide-and-conquer frontier search applied to optimal sequence alignment“. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. Hrsg. von Henry A. Kautz und Bruce W. Porter. AAAI Press / The MIT Press, 2000, S. 910–916.

[11] Marc Lanctot, Nicolas Ng Man Sun und Clark Verbrugge. „Path-finding for Large Scale Multiplayer Computer Games“. In: *Proceedings of the 2nd Annual North American Game-On Conference (GameOn’NA 2006)*. Monterey, California: Eurosis, 2006, S. 26–33.

[12] Ian Millington. *Artificial Intelligence for Games*. Elsevier, 2006.

[13] Amit Patel. *Amit’s A* Pages*. Jan. 2014. URL: `http://theory.stanford.edu/~amitp/GameProgramming/`.

[14] Alexander Reinefeld und T. A. Marsland. „Enhanced Iterative-Deepening Search“. In: *IEEE Transactions on Pattern Analysis Machine Intelligence* 16.7 (1994). Hrsg. von David A. Forsyth, S. 701–710.

[15] Robert Sedgewick. *Algorithmmen in C*. Addison Wesley Longman Verlag, 1992.

[16] Jonathan Richard Shewchuk. „Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator“. In: *Applied Computational Geometry: Towards Geometric Engineering*. Hrsg. von Ming C. Lin und Dinesh Manocha. Bd. 1148. Springer-Verlag, 1996, S. 203–222.

[17] Nathan R. Sturtevant und Michael Buro. „Partial Pathfinding Using Map Abstraction and Refinement.“ In: *AAAI’05: Proceedings of the 20th National Conference on Artificial intelligence*. Hrsg. von Manuela M. Veloso und Subbarao Kambhampati. AAAI Press / The MIT Press, 2005, S. 1392–1397.

[18] Paul Tozour. „Search Space Representations“. In: *AI Game Programming Wisdom*. Hrsg. von Steve Rabin. Bd. 2. Hingham, Massachusetts, USA: Charles River Media, 2002, S. 85–113.

[19] Peter Yap. „Grid-Based Path-Finding“. In: *Advances in Artificial Intelligence, Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence (AI 2002)*. Hrsg. von Robin Cohen und Bruce Spencer. Bd. 2338. Lecture Notes in Computer Science. Calgary, Canada: Springer-Verlag, 2002, S. 44–55.