Paraller Java Pathfinder für ein Echtzeit-Strategiespiel

Diplomarbeit im Fachbereich Elektrotechnik / Informatik
der Universität Kassel

Abgegeben am: 24. Januar 2008.

vorgelegt von
Heiko Waldschmidt

Betreuer:
Dipl.-Inf. Björn Knafla

Prüfer:
Prof. Dr. Claudia Leopold
Prof. Dr. Gerd Stumme

Fachbereich Elektrotechnik / Informatik
Fachgebiet Programmiersprachen/-methodik
Wilhelmshöher Allee 73
34125 Kassel

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____, Kassel den 24.01.2008.

Inhaltsverzeichnis

1 Einleitung

Pathfinding ist der Teil der Künstlichen Intelligenz (KI) eines Computerspiels, der kürzeste Wege, für alles was ein Spieler oder Teile der KI in einem Computerspiel steuern können (Agenten), berechnet. Diese Berechnungen stellen einen erheblichen Teil des gesamten Rechenaufwands eines Computerspiels dar. Diese Diplomarbeit erläutert paralleles Pathfinding in Java für das Computerspiel am Beispiel jMMORTS. jMMORTS soll mit möglichst vielen Agenten gleichzeitig gespielt werden können, wodurch der Rechenaufwand und der Speicherbedarf für das Pathfinding hoch ist. Daher war es von großer Bedeutung für jMMORTS einen möglichst schnellen Pathfinder zu entwickeln, der mit möglichst geringem Rechenaufwand auskommt.

Es wurde ein (hierarchischer) Pathfinder implementiert. Hierarchische Pathfinder haben einen geringeren Speicherbedarf und geringere Anforderungen an die Rechenleistung als andere Algorithmen (z.B. der A*-Algorithmus). Sie berechnen zunächst "grobe" Wege, die nicht detailliert jeden Punkt auf der Karte des Spiels beinhalten, der auf dem Weg liegt. Der Weg wird auf (Hierarchie-)Ebenen beschrieben, die abstrakter sind als die Karte. Nur die ersten Wegpunkte eines groben Weges sind detaillierte Punkte der Karte, auf denen sich ein Agent entlang bewegen kann. Die Grundalgorithmen des Pathfinding können ausschließlich den kompletten detaillierten Pfad berechnen. Die Berechnungen dieser groben Wege sind wesentlich schneller und verbrauchen weniger Speicher, als die durch die Grundalgorithmen - ein Agent kann sich somit schneller in Bewegung setzen. Weitere detaillierte Wegpunkte können später iterativ (d.h. ein detaillierter Teilweg nach dem anderen) bestimmt werden.

Um die Rechenleistung von Parallelrechnern nutzen zu können und damit für eine möglichst hohe Anzahl an Agenten in möglichst kurzer Zeit Wege berechnen zu können, wurde der Hierarchische Pathfinder parallelisiert. Die Parallelisierung war erfolgreich.

Weiterhin konnten Vorschläge gemacht werden, die beschreiben, wie beim Pathfinding mit Agenten umgegangen werden kann, die eine unterschiedliche Größe besitzen oder sich auf eine andere Art in ihrer Beweglichkeit unterscheiden.

Es werden für das Pathfinding bedeutsame Teile des Computerspiels jMMORTS (in Kapitel 2) und anschließend die Grundlagen für die Parallelisierung und die parallele Programmierung in Java (in Kapitel 3) beschrieben. In Kapitel 4 werden die Grundlagen des Pathfindings erläutert. Die darin vorgestellten Grundalgorithmen des Pathfindings können die in Kapitel 5 genannten Anforderungen des jMMORTS-Spieles bezüglich Berechnungszeit und Speicherverbrauch jedoch nicht erfüllen. Die Vorgehensweise eines Hierarchischen Pathfinders wird genauer in Kapitel 6 erklärt und der verwendete Hierarchische Pathfinder (HPA*-Algorithmus) in Kapitel 7 erläutert.

In Kapitel 8 wird die Parallelisierung des HPA*-Algortihmus beschrieben, dessen Implementierung und Integration in Kapitel 9 folgt.

Kapitel 10 stellt Experimente vor, die mit dem parallelen HPA*-Algorithmus durchgeführt wurden und u.a. dazu dienen diesen zu bewerten.

In den Schlussbemerkung (Kapitel 11) wird die Diplomarbeit zusammengefasst und in einem Ausblick wird beschrieben, wie man mit Agenten umgehen könnte, die der Pathfinder aufgrund ihrer unterschiedlichen Beweglichkeit unterschiedlich behandeln muss. Außerdem behandelt dieses Kapitel Optimierungen, die am Pathfinder durchgeführt werden könnten.

2 Das Computerspiel jMMORTS

jMMORTS steht für java Massive Multiplayer Online Real-Time Strategie. jMMORTS wird am Fachgebiet Programmiersprachen und -methodik im Rahmen studentischer Arbeiten erstellt. Wichtige Merkmale von jMMORTS sind:

Künstliche Intelligenz (KI)

Die **Künstliche Intelligenz (KI)** muss das gesamte intelligente Verhalten, das in einem Spiel vorkommt, berechnen. Sie bestimmt für sämtliche Agenten, die nicht von einem Spieler gesteuert werden, das komplette Verhalten. Auch für Agenten eines Spielers führt sie Berechnungen durch. Wenn ein Spieler einen Agenten bewegen möchte, wählt er diesen aus und teilt ihm für gewöhnlich durch einen Mausklick auf eine bestimmte Position der Karte mit, dass er sich dort hinbewegen soll. Der Spieler berechnet nicht die einzelnen Wegpunkte vom Standpunkt der Einheit zum ausgewählten Ziel, dies macht die KI. Die Berechnung dieser Wege nennt sich **Pathfinding**. jMMORTS besitzt, wie die meisten Computerspiele, eine KI.

Die Mainloop

Das Programm eines Computerspiels sieht schematisch aus wie in Listing 1 dargestellt.

```
startup();
while(nothingAbortsThisGame){
    simulate();
    draw();
}
shutdown();
```

Listing 1: Ein Schema des Programm eines Computerspiels

Zuerst werden KI, Grafiksystem usw. initialisiert. Dann begibt sich das Spiel in eine Endlosschleife, **Mainloop** genannt, solange bis es beendet wird. In jedem Schleifendurchlauf wird ein **Zeitabschnitt** bzw. **Zeitschritt** des Spiels durchgeführt. Dies bedeutet, dass der zeitliche Verlauf des Spiels in Zeitschritte unterteilt wird, in denen alles abläuft, was im Spiel passiert: Der Spieler bekommt Punkte, die Bildschirmanzeige wird aktualisiert, physikalische Vorgänge werden berechnet, usw.. Insbesondere führt die KI (u.a.) Pathfinding durch. jMMORTS und alle anderen bekannten Computerspiele enthalten eine Mainloop.

2,5D Spielwelt

Ein Spiel benutzt eine sogenannte **Spielwelt**. Die Agenten eines Spielers befinden sich in dieser Spielwelt und agieren in ihr. Ein Agent steht in der gleichen Beziehung zur Spielwelt, wie ein Mensch zu dem von Menschen bekanntem Universum.

jMMORTS benutzt eine 2.5D Welt. In 2,5D Welten betrachtet der Spieler die Spielwelt meist aus der Vogelperspektive. Die Dinge, die er sieht, sind in 3D dargestellt. Es gibt aber keinen Punkt auf der Höhenkarte mit mehr als einem Höhenwert. Daher existieren keine Höhlen in Bergen einer 2,5D Spielwelt. Es sei denn, es gibt eine Karte für die Höhle und eine andere für die Bergoberfläche

Die Darstellung erfolgt durch das **Grafiksystem** des Spiels.

Die Karte

Agenten in Computerspielen bewegen sich auf **gerasterten** Karten. jMMORTS benutzt eine Karte, die durch gleichgroße Quadrate **gerastert** ist. Jedem Quadrat wird ein Koordinatenpaar (x,y) zugeordnet. Beim Pathfinding werden diese Quadrate wie **Felder** oder **Punkte** behandelt, die entweder betreten oder auch nicht betreten werden können und die in 8 Richtungen verlassen werden können (siehe Abbildung 1).

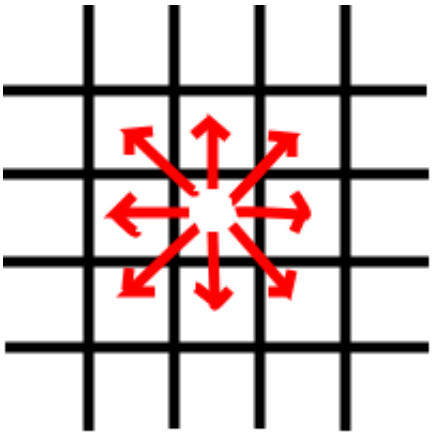


Abbildung 1: Ein Teil einer Karte, die gerastert wurde. Wenn sich eine Einheit auf dem Feld in der Mitte befindet, kann sie auf jedes Feld ziehen, auf das ein roter Pfeil zeigt (die oben genannten 8 Richtungen).

Mögliche und unmögliche Bewegungen von Einheiten

Für alle Einheiten in Computerspielen muss definiert werden, wie sie sich bewegen können. Zu dieser Definition gehört auch die Festlegung der Einheitengrößen, denn nur wenn die Größe definiert ist, kann bestimmt werden, ob für eine

Einheit auf einer Stelle der Karte ausreichend Platz besteht. Nur wenn dies der Fall ist, darf sie dorthin bewegt werden. In jMMORTS sind alle Einheiten kreisrund, damit sie sich auf der Stelle drehen können, und haben einen Durchmesser, welcher der Länge eines oder mehrerer der Quadrate entspricht mit denen die Karte gerastert wurde. Die Länge mehrerer Quadrate wurde als mögliche Größe festgelegt, damit unterschiedlich große Einheiten verwendet werden können.

Einheiten können daher die in Abbildung 2 gezeigten Bewegungen nicht durchführen, da sie auf dem Weg Felder berühren würden, auf denen Hindernisse sind. Eine Diagonalbewegung ist in diesen Fällen in jMMORTS nicht zulässig.

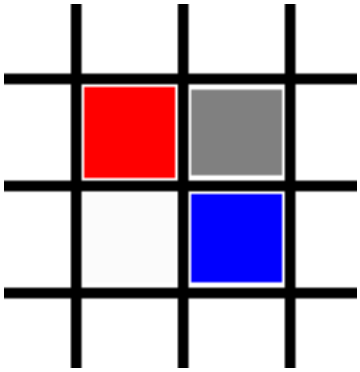


Abbildung 2: Auch hier befindet sich auf dem grauen Feld ein Hindernis und eine Einheit möchte vom roten Feld auf das blaue Feld oder umgekehrt, sie kann nicht den direkten Weg über die Diagonale nehmen.

Höhenkarte

Die Art der Karte ist bei jMMORTS eine Höhenkarte (engl: Heightmap). Eine **Höhenkarte** speichert zu jeder x,y-Koordinate eine zugehörige Höhe ab. Die Oberfläche der Erde kann z.B. dadurch beschrieben werden, dass für jede Kombination aus Längen- und Breitengrad der höchste begehbare Punkte (über Normal Null) angegeben wird.

Offenes Terrain

Das **Terrain** (deutsch: Gelände) kann bei Computerspielen sehr unterschiedlich sein. **Offenes Terrain** wäre z.B. eine Wiese im Gegensatz zu dem Stockwerk eines Gebäudes, das in viele Räume unterteilt ist. Auf einer Wiese kann jeder Punkt über viele verschiedene Wege erreicht werden, während in einem Gebäude Agenten immer durch bestimmte Eingänge gehen müssen, damit ein Raum betreten werden kann. Die Eingänge schränken die Anzahl der Wege stark ein, was das Pathfinding vereinfacht - der Pathfinder für jMMORTS muss ohne diese Einschränkungen auskommen, weil es in jMMORTS auch offenes Terrain geben soll.

j - Java

Das Spiel wird in Java programmiert wird.

MMO - Massive Multiplayer Online

Ein Massive Multiplayer Online Spiel muss von mehreren Spielern (Multiplayer) gleichzeitig in einem Computernetz, möglichst im Internet (Online), gespielt werden können. “Massiv” kann übersetzt werden mit viel(e), groß oder gigantisch. Massiv bezieht sich z.B. auf die Anzahl der Spieler, die Anzahl der Einheiten und die Größe der Karte, diese sollen massiv groß sein. In jMMORTS soll es außerdem möglichst viele unterschiedliche Einheiten geben. Sie sollen sich z.B. in ihrer Größe und ihrer Mobilität unterscheiden können, was für das Pathfinding von Bedeutung ist.

RTS - Real-Time Strategie

Soll ein Spiel Echtzeit (Realtime) Anforderungen erfüllen, muss der Spieler *sofort* eine Rückmeldung vom Spiel bekommen, wenn er eine Änderung veranlasst hat. *Sofort* bedeutet in diesem Fall, dass der Spieler die Zeitspanne zwischen seinem Befehl und der Rückmeldung nicht mehr wahrnehmen kann.

Alles was ein Spieler steuern kann, wird in einem Computerspiel als **Agent** bezeichnet. Spieler von RTS-Spielen benutzen statt dem Begriff Agent den Begriff **Einheit**. Diese Begriffe werden im folgenden synonym verwenden.

In einem RTS handeln alle Spielparteien (auch die vom Computer kontrollierten) gleichzeitig. Ein Spieler befehligt immer eine ganze Reihe von Agenten. Häufig sind dies unterschiedliche Agenten (zum Beispiel ein Panzergrenadier und ein Panzer), die gemeinsam eingesetzt eher zum Ziel des Spieles führen können als einzeln. Es ist notwendig Strategien für das Spiel zu entwickeln (d.h. das Geschehen eine möglichst lange Zeit im voraus zu planen), um es gewinnen zu können.

Randbemerkung

Der Pathfinder, der in dieser Diplomarbeit beschrieben wird, kann in verschiedenen Spielen eingesetzt werden, nicht nur in jMMORTS. Sind die möglichen Bewegungen in einem Spiel anders definiert, soll z.B. nicht mit quadratischen, sondern mit sechseckigen Feldern gearbeitet werden oder soll es eine 3D Spielwelt benutzen, dann müssen Veränderungen am Pathfinder vorgenommen werden.

3 Parallelisierung

Die momentane Weiterentwicklung von Prozessoren führt kaum zu Steigerungen der Taktfrequenz, sondern vielmehr dazu, dass sich immer mehr Kerne in einer CPU befinden. jMMORTS soll mit möglichst vielen Einheiten umgehen

können (siehe Kapitel 2) und deshalb die Leistungsfähigkeit (der Prozessoren) vollständig nutzen. Deswegen soll das Pathfinding parallelisiert werden.

In diesem Kapitel werden Grundbegriffe der Parallelisierung und anschließend Möglichkeiten der parallele Programmierung in Java vorgestellt. Die Parallelisierung eines Pathfinders wird in Kapitel 8 beschrieben.

3.1 Grundbegriffe der Parallelisierung

Die Parallelisierung soll für CPUs durchgeführt werden, die **gemeinsamen Speicher** verwenden, dazu werden **Threads** eingesetzt. Threads können gleichzeitig auf die selben Speicherstellen zugreifen, um Informationen auszutauschen.

Bei der Parallelisierung werden die Berechnungen des sequentiellen Programms in Aufgaben - **Tasks** genannt - aufgeteilt. Diese Tasks sollen parallel bearbeitet werden können und müssen dazu Threads zugeordnet werden, die die Berechnungen durchführen.

Eine Schwierigkeit bei dieser Zuordnung ist das so genannte **Loadbalancing**.

Perfektes Loadbalancing liegt dann vor, wenn alle Prozessorkerne in einem parallelen Teil des Programms zu jedem Zeitpunkt gleich stark und gleichlang ausgelastet sind und ihren Berechnungen demzufolge zum selben Zeitpunkt beenden. Da die Berechnungen möglichst schnell durchgeführt werden sollen, sollten die Prozessoren möglichst voll ausgelastet werden. Der Vorgang in dem versucht wird perfektes Loadbalancing zu erreichen wird ebenfalls Loadbalancing genannt.

Eine weiter Schwierigkeit bei der parallelen Programmierung ist, dass verhindert werden muss, dass mehrere Threads gleichzeitig auf die selbe Speicherstelle zugreifen und einer der Threads die Daten verändert (z.B. kann dann nicht mehr vorhergesagt werden, welchen Wert die anderen Threads gelesen haben - den vor oder den nach der Änderung durch den schreibenden Thread (...)). Um diese Probleme zu beseitigen muss **synchronisiert** werden. Synchronisierung bedeutet, dass die Berechnungen der Threads so aufeinander abgestimmt werden, dass es nicht zu den beschriebenen Fehlern kommen kann. Eine der Synchronisierungsmethoden ist das **Locking**. Beim Locking wird zunächst eine Sperre (engl. Lock) einer Speicherstelle angelegt, dann die Daten dieser Speicherstelle verarbeitet und zuletzt die Sperre wieder entfernt. Der Lock verhindert jegliche Zugriffe von anderen Threads auf die Speicherstelle.

3.2 Parallele Programmierung in Java

Dieser Abschnitt stellt Möglichkeiten der parallelen Programmierung in Java vor, deren Verwendung im parallelen HPA*-Algorithmus wird in Abschnitt 8.1 und Abschnitt 9.2 beschrieben. Die im folgenden erwähnten Klassen befinden sich in java.util.concurrent.*.

In Java [4] können Threads in der Version 6 nicht nur direkt verwendet werden, es gibt nun auch **Threadpools** und **Executor**. Ein Threadpool ist ein Reservoir das mehrere Threads enthält. Ein Executor besitzt einen eigenen

Threadpool und ist dafür zuständig Tasks, die ihm zugewiesen werden auf die Threads im Threadpool zu verteilen. Der Programmierer kann daher anstatt einen Task zu erstellen und diesen direkt einem Thread zuzuordnen, diesen an einen **Executor** übergeben.

Dies hat die folgenden Vorteile:

- Der Programmierer muss nicht herausfinden, wann einer der Threads keinen Task mehr zu bearbeiten hat und wann er diesem einen neuen Task zuteilen kann. Dies geschieht dynamisch durch den Executor.
- Die dynamische Zuordnung erledigt einen großen Teil des Loadbalancings, was dem Programmierer die Arbeit erleichtert.

Java stellt vier Typen von Threadools zur Verfügung (außerdem können eigenen Typen implementiert werden):

- SingleThreadExecutor: Der SingleThreadExecuter besitzt immer genau einen Thread. Sollte dieser sterben, so wird ein neuer erzeugt. Dieser Pool garantiert sequentielle Abarbeitung.
- FixedThreadPool: Der FixedThreadPool besitzt eine maximale Anzahl an Threads. Immer wenn der Pool einen neuen Task zugewiesen bekommt und die maximale Anzahl der Threads noch nicht erreicht ist, wird ein neuer Thread erzeugt. Sobald die maximale Anzahl erreicht ist, wird diese konstant gehalten. Ist kein Thread frei, um einen neuen Task bearbeiten zu können und die maximale Anzahl an Threads bereits erreicht, so wird der Task in eine Warteschlange eingefügt und einem Thread zugeteilt, sobald einer für eine neue Berechnung zur Verfügung steht (weil er seine letzte Aufgabe abgeschlossen hat). Sollte ein Thread sterben, wird ein neuer erzeugt.
- ScheduledThreadPool: Eine Erweiterung des FixedThreadPools, die für periodische Ausführung von Tasks gedacht ist.
- CachedThreadPool: Kann flexibel zusätzliche Threads erzeugen, wenn neue Tasks dem Pool zugewiesen werden, hat aber kein Limit für die Threadanzahl.

In Java gibt es außerdem 2 verschiedene Varianten von Tasks, die beide von Threadpools bearbeitet werden können - **Runnable** und **Callable** Tasks (Klassen müssen das jeweilige Interface Runnable oder Callable implementieren). Das Verhalten der Tasks wird in der Methoden run() (bei einem Runnable) und call() (bei einem Callable) definiert. Der Unterschied besteht darin, dass Callables Rückgabewerte besitzen und Exceptions, die während der Taskausführung aufgetreten sind, nach außen weitergeben können (s.u.), während Runnables keine Rückgabewerte besitzen und keine Exceptions weiterreichen können.

Die Rückgabe eines Callables erfolgt über ein **Future**. Wird ein Callable einem Threadpool hinzugefügt, kann der Pool anschließend nach einem Future

befragt werden. Das Future kennt den Bearbeitungsstand bzw. Lebenszykluszustand (erzeugt, einem Pool hinzugefügt, gestartet, bearbeitet) eines Tasks und kennt nach dessen Bearbeitung auch den Rückgabewert oder die entstandene Exception. Ein Thread, der einen Task erzeugt hat und ihn an einen Threadpool übergeben hat, kann daher ein Future befragen in welchem Bearbeitungszustand sich der Task befindet und kann den Rückgabewert abfragen. Dabei wird eine Exception geworfen, wenn eine Exception bei der Taskausführung entstanden ist.

4 Grundlagen des Pathfinding

Alle Pathfinding-Algorithmen haben gemeinsam, dass sie einen **Graphen** benutzen und darin versuchen **kürzeste Wege** zu finden. Abschnitt 4.1 erläuterte Grundlagen zur Graphentheorie und Abschnitt 4.2 erklärt, wie Pathfinding in Graphen durchgeführt werden kann.

4.1 Graphentheorie

Dieser Abschnitt beschreibt was ein Graph ist und welche Arten von Graphen es gibt und was ein kürzester Weg ist.

4.1.1 Graph

Ein **Graph** ist ein Paar endlicher Mengen V und E (wobei V geschnitten E die leere Menge ergibt). Dabei bezeichnet V die Menge der im Graph enthaltenen **Knoten** (engl. vertex) und E die Menge der **Kanten** (engl. edge) des Graphen. Eine Kante verbindet immer zwei Knoten. In den im folgenden betrachteten Graphen verbindet eine Kante niemals einen Knoten mit sich selbst.

Bei einem **gewichteten Graphen** sind alle Kanten mit **Kosten** versehen, die darstellen sollen, wie viel es eine Einheit "kostet", wenn sie von einem Punkt bzw. Knoten zum anderen will. Was genau hinter diesen Kosten steckt, hängt vom Spiel ab. Es kann sich zum Beispiel um den Benzinverbrauch einer Einheit oder um die Fahrzeit handeln.

Weiterhin gibt es **gerichtete Graphen**. In einem gerichteten Graphen haben die Kanten eine Richtung. Eine Kante A beschreibt dann beispielsweise, wie vom Knoten B zum Knoten C gelangt werden kann, aber bietet dann nicht die Möglichkeit von Knoten C zu Knoten B zu gelangen. Soll sich eine Einheit in diesem Graphen auch von C nach B bewegen können, so wird eine zweite gerichtete Kante benötigt.

Ich habe einen ungerichteten gewichteten Graphen für das Pathfinding verwendet. Die Kosten sind daher in beide Richtungen gleich.

4.1.2 Kürzeste Wege

Gesucht wird immer ein **kürzester Weg** von einem Startknoten S des Graphen zu einem Zielknoten G. Der Weg setzt sich aus den Kanten zusammen, an denen

sich eine Einheit entlangbewegen kann, um vom Start zum Ziel zu gelangen.

Es handelt sich dann um einen kürzesten Weg, wenn die summierten Kantengewichte eines Weges von Start zum Ziel kleiner oder gleich den summierten Kantengewichten jedes anderen Weges von Start zum Ziel sind.

Die KI sucht *einen* kürzesten Weg und nicht *den* kürzesten Weg, da es mehrere gleichlange Wege geben kann. In Computerspielen ist es egal, welcher dieser Wege benutzt wird, daher werden die später erläuterten Algorithmen die Suche beenden, sobald sie einen kürzesten Weg gefunden haben.

4.2 Grundalgorithmen des Pathfindings

Die beiden im Folgenden vorgestellten **Grundalgorithmen** des Pathfinding, der Dijkstra- und der A*-Algorithmus, bestimmen kürzeste Wege zwischen zwei Knoten in gewichteten Graphen und bilden die Grundlage für die später vorgestellten Algorithmen. Weil in jMMORTS Wege auf der Karte gesucht werden müssen, die Grundalgorithmen aber einen Graphen benötigen, wird dieser erstellt. Dies geschieht indem für jeden Punkt auf der Karte ein Knoten erzeugt und dieser mit allen Knoten der benachbarten Punkte verbunden wird. Soll ein kürzester Weg zwischen zwei Punkten bestimmt werden, werden zunächst die zugehörigen Knoten im Graphen ermittelt und zwischen diesen wird dann ein kürzester Weg gesucht. Fälle, die auf der Karte erklärt werden, lassen sich damit auf den Graphen übertragen und umgekehrt.

Beide Grundalgorithmen haben Vor- und Nachteile, auf die ich zum Teil erst in Abschnitt 7.4 eingehen werde.

Sowohl der A*- als auch der Dijkstra-Algorithmus betrachten als Kosten immer einen einzelnen skalaren Wert pro Kante. Sie können nicht gleichzeitig mit den Werten für z.B. Benzinverbrauch und Fahrzeit umgehen. Diese beiden Werte müssen zu einem Wert zusammengefasst werden, damit die Algorithmen damit umgehen können. In jMMORTS ist noch nicht festgelegt worden, wie sich die Kosten genau zusammensetzen sollen.

Die Kosten dürfen bei beiden Algorithmen nie negativ sein: Angenommen es gibt eine Kante mit negativen Kosten (AB) zwischen Knoten A und Knoten B. Weiterhin gibt es eine Kante (AC) zwischen A und einem Knoten C und eine Kante (BC) zwischen B und C. Beide Algorithmen versuchen einen kürzesten Weg zu finden, indem sie Kosten minimieren. Ist $AB + AC + BC < 0$ so kann der Algorithmus die Kosten von einem beliebigen Knoten bis zum den Knoten A,B und C dadurch minimieren, dass er sobald er einen dieser Knoten A, B, C erreicht hat, sich im Kreis bewegt (z.B. von A nach B von B nach C von C nach A). Sollte er sich einmal diesen Kreis entlang bewegen, so wird er dies immer wieder tun, da er die Kosten (zu A, B und C) dann noch weiter senken kann. Damit befindet sich der Algorithmus in einer Endlosschleife.

4.2.1 Der Dijkstra-Algorithmus

Angenommen es liegt auf jedem Punkt des Spielfeldes eine elektrisch betriebene analoge Uhr und der Dijkstra-Algorithmus [7] schüttet über dem Startknoten

langsam einen Eimer Wasser aus. Das Wasser verteilt sich daraufhin gleichmäßig in alle Richtungen, kann jedoch von Hindernissen aufgehalten werden. Das Wasser benötigt 1,41 ($\sqrt{2} \approx 1,41$) mal so lange, um ein diagonal benachbartes Feld zu überschwemmen, wie für andere benachbarte Felder. Sobald das Wasser eine Uhr erreicht hat, bleibt diese stehen. Bleibt die Uhr am Zielpunkt stehen, beendet der Algorithmus das Ausschütten von weiterem Wasser.

Anschließend sieht der Algorithmus auf die Uhren, die auf den benachbarten Feldern vom Zielfeld liegen, sucht davon die Uhr mit der frühesten Uhrzeit heraus und speichert dieses Feld nun als (vorletzten) Wegpunkt. Danach werden von diesem Feld aus wieder die Uhren auf den Nachbarfeldern betrachtet und wiederum der nächste Wegpunkt anhand der frühesten Uhrzeit bestimmt. Das ganze wird so lange wiederholt, bis das Startfeld erreicht ist, dann wurde der Weg vollständig rekonstruiert und ein kürzester Weg gefunden.

Der Dijkstra-Algorithmus wird typischerweise mit einer **Priority Queue** (die dafür sorgt, dass ihre Elemente zu jedem Zeitpunkt nach einer Priorität sortiert sind) implementiert oder einer Liste, bei der bei jedem Zugriff das Element mit den geringsten Kosten gesucht werden muss.

Die verwendeten Elemente bestehen aus einem Knoten K, dem (Vorgänger-) Knoten K', der zuletzt auf dem Weg zu K betreten wurde und den aufsummierten Kosten der Kanten (des kürzesten bekannten Weges), die vom Startpunkt bis zu K führen.

Zu Beginn des Algorithmus befindet sich ausschließlich ein Element in dieser Priority Queue, das den Startknoten (sowie keine Vorgänger und Kosten von null) enthält. In einer Schleife wird in jedem Schritt das Element mit den geringsten Kosten aus der Queue gewählt und es wird über alle Kanten iteriert, die sich an diesem Knoten befinden. Für jeden Knoten, der sich am anderen Ende einer der Kanten befindet, wird in die Priority Queue eines der oben beschriebenen Elemente eingefügt, es sei denn:

- ein Element, dass diesen Knoten (als K, nicht als K') enthält, befindet sich bereits in der Priority Queue und es wurde bereits ein kürzerer Weg zu ihm gefunden.
- auf dem Feld befindet sich ein unüberwindliches Hindernis.
- der Fall, der in Abbildung 2 dargestellt wird, trifft zu.

Sobald aus der Priority Queue beim Schleifendurchlauf das Element, das den Zielknoten enthält, gewählt wird, wurde der kürzeste Weg gefunden.

Alles was sich danach noch in der Priority Queue befinden kann, sind Knoten zu denen ein Weg führt, der mehr kostet als der zum Zielknoten. Weil die Kosten für eine Kante niemals negativ sind, können die Wege, die noch gefunden werden nur länger werden. Gäbe es einen kürzeren Weg, wäre er früher gefunden worden und der Algorithmus hätte die Suche schon beendet.

Zuletzt muss noch der Weg zurückgegangen und zurückgegeben werden - dafür wurden die Vorgängerknoten in den Elementen gespeichert.

Das Vorgehen des Dijkstra-Algorithmus ist nicht auf das Ziel (den Zielknoten) ausgerichtet, sondern darauf, alle kürzesten Pfade vom Startknoten aus zu

berechnen. Der Dijkstra-Algorithmus bricht ab, sobald der Zielknoten erreicht worden ist. Der im nächsten Abschnitt beschriebene A*-Algorithmus geht zielgerichteter vor.

4.2.2 Der A*-Algorithmus

Der A*-Algorithmus[5] betrachtet nicht nur die Kosten, die benötigt wurden um zu einem Punkt zu kommen (im folgenden **bisherige Kosten** genannt), sondern er benutzt eine **Heuristik**, um zusätzlich die Kosten von dem Knoten, den er gerade betrachtet bis zum Ziel abzuschätzen (im folgenden **Schätzwert** genannt). Die aus diesen beiden Kosten berechneten **Gesamtkosten** werden verwendet, um zu bestimmen, welcher Knoten als nächstes betrachtet wird - wie beim Dijkstra-Algorithmus ist dies das Element mit den geringsten Kosten. Der A*-Algorithmus läuft mit Hilfe der Heuristik zielgerichteter als der Dijkstra-Algorithmus auf das Ziel zu (siehe auch Abschnitt 4.2.3).

Der A*-Algorithmus arbeitet mit zwei Listen genannt **openList** und **closedList**. Beide Listen dienen dazu Elemente zu speichern, die einen Knoten K, den Vorgänger des Knoten K' (auf dem bekannten kürzesten Weg zu K), die bisherigen Kosten und die Gesamtkosten enthalten. Die openList hat den selben Zweck wie die Priority Queue des Dijkstra-Algorithmus. Zu Beginn des Algorithmus wird ein Element, das den Startknoten enthält, auf die openList gelegt.

Der Algorithmus entfernt in jedem Schritt das Element mit den geringsten Gesamtkosten von der openList (aktuelles Element) und betrachtet alle seine Nachbarknoten (auch hierbei wird berücksichtigt, ob diese betretbar und direkt erreichbar sind (wie in 4.2.1 beschrieben)). Dabei berechnet er die bisherigen Kosten, die Heuristik bis zum Ziel und die Gesamtkosten und erzeugt neue Elemente. Diese legt er auf die openList, es sei denn, er findet auf einer der beiden Listen ein Element, was den gleichen Knoten enthält. Findet er ein solches Element, werden die Kosten darin mit den gerade berechneten verglichen. Sind die neuen Kosten kleiner, wird das alte Element aktualisiert (in beiden Fällen wird kein neues Element erzeugt). Befindet sich das alte Element auf der closedList und wurden seine Kosten aktualisiert, dann wird das Element zurück auf die openList gelegt (und in der closedList entfernt).

Das aktuelle Element wird anschließend auf die closedList gelegt.

Durch das Zurücklegen der Elemente auf die openList bei kleineren Gesamtkosten werden Abweichungen der Schätzung vom tatsächlichen Wert durch den Algorithmus behandelt. Das Element wird noch einmal betrachtet, wenn es nun Kosten besitzt, die kleiner sind als die des kürzesten Weges, denn sobald dieser gefunden wurde, bricht der Algorithmus ab.

4.2.3 Heuristiken

Eine Heuristik für den A*-Algorithmus ist immer eine Abschätzung von einem Knoten bis zum Zielknoten.

Eine Heuristik[1] ist **zulässig**, wenn die Kosten, die von einem Punkt bis zum Ziel geschätzt werden nie größer sind als die tatsächlichen Kosten k. Sie müssen im Intervall [0,k] liegen.

Eine Heuristik (h) ist **monoton**, wenn

- die Kosten nie überschätzt werden
- für jeden Knoten k und jeden Nachfolgerknoten k' gilt:

$$h(k) \leq h(k') + c(k, k') \tag{1}$$

Wobei c(k,k') die tatsächlichen Kosten von k nach k' bezeichnet.

Ein Beispiel für eine monotone Heuristik ist die euklidische Distanz.

Monotone Heuristiken führen immer zu einem optimalen Pfad (wenn es überhaupt einen Pfad gibt). Für einen kürzesten Weg gilt für den Zielknoten k: Gesamtkosten $G(k) = h(k)$ + tatsächliche Kosten $t(k)$. Da die Kosten nie überschätzt werden dürfen, ist $h(k) = 0$ und damit $G(k) = t(k)$. Jeder andere Weg müsste für jeden Knoten a den er auf seinem Weg zum Ziel betritt, um kürzer zu sein, die Gleichung $h(a) + t(a) < G(k)$ bzw. $h(a) + t(a) < t(k)$ erfüllen (auch $h(a)$ darf nicht überschätzt werden). Würden alle Punkte a diese Gleichung erfüllen, so wären sie vor dem Zielknoten betrachtet worden, weil sie niedrigere Gesamtkosten besitzen (dies garantiert der Algorithmus). Da sie aber nicht alle vorher betrachtet worden sind, kann die oben genannte Gleichung nicht für alle Punkte a erfüllt worden sein und damit gibt es keinen kürzeren Weg. Dieser Beweis kann für jeden Punkt auf dem kürzesten Weg angewendet werden, wenn dieser als Zielknoten betrachtet wird. Damit kann gezeigt werden, dass es auch zu diesen Punkten keinen kürzeren Weg gibt.

Weil der A*-Algorithmus zur Bewertung eines Knotens die Gesamtkosten (berechnet aus dem Schätzwert und den bisherigen Kosten) verwendet, um das Ziel zu finden, wird die Orientierung in Richtung Ziel schwächer, wenn die Heuristik (generell) kleinere Werte schätzt und stärker, wenn sie (generell) größere Werte schätzt. Denn in den Gesamtkosten sinkt der Anteil des Schätzwertes, wenn dieser kleiner wird und steigt der Anteil, wenn der Wert größer wird.

Wird der Wert zu stark unterschätzt, untersucht der Algorithmus mehr Knoten und wird langsamer. Ein Spezialfall stellt dabei die Heuristik $h(k) = 0\ \forall k \in Menge\ der\ Knoten$ dar. Wird sie verwendet, verhält sich der A*-Algorithmus genauso wie der Dijkstra-Algorithmus. Der Dijkstra-Algorithmus ist daher ein Spezialfall des A*-Algorithmus.

Wird der Wert überschätzt und damit eine nicht monotone Heuristik verwendet, dann orientiert der A*-Algorithmus sich stärker zum Ziel. Dabei kann es passieren, dass die Zielorientierung so stark wird, das die bisherigen Kosten kaum noch eine Rolle spielen. Der Algorithmus bevorzugt dann Wege mit wenigen Wegpunkten, die allerdings nicht mehr kürzeste Wege sein müssen. Der oben genannte Beweis gilt für diese Heuristiken nicht mehr.

In Spielen sollte bei der Wahl der Heuristik durchaus über eine Heuristik nachgedacht werden, die leicht überschätzt, da die Rechenzeit, die für den A*-Algorithmus benötigt wird, und die zielgerichtetere Suche wichtiger sind als die Genauigkeit der Wege [3].

5 Anforderungen von jMMORTS an Pathfinding-algorithmen

Spiele, die Echtzeitanforderungen, viele unterschiedliche Einheiten, große Karten und offenes Terrain besitzen, stellen hohe Anforderungen an die Pathfindingverfahren (siehe auch 2). Die vorgestellten Algorithmen benötigen unter diesen Bedingungen viel Speicher und viel Rechenleistung. Für eine Einheit, die auf einen 1000 Felder entfernten Punkt zulaufen soll, müssen nach der Berechnung durch den A*- oder den Dijkstra-Algorithmus (den ich in Abschnitt 4.2.1 bzw. 4.2.2 vorgestellt habe) mindestens 1000 Wegpunkte gespeichert werden. Während der Berechnung durch den A*- oder Dijkstra-Algorithmus befindet sich ein Mehrfaches dieser Punkte im Speicher. Außerdem wird es bei vielen Einheiten nicht dazu kommen, dass eine Einheit sofort losläuft, nachdem ein Spieler ihr einen Befehl erteilt hat, weil die Berechnung eines langen Weges mit den Grundalgorithmen lange dauert. Diese Verzögerungen genügen jedoch nicht den Echtzeitanforderungen und sorgen somit für Unmut beim Spieler.

Es sind daher Optimierungen notwendig, um das Spiel dennoch spielbar zu machen. Diese Anforderungen sind viel wichtiger als einen 100% genauen Pfad zu berechnen. Das Pathfinding wurde im Sinne der Anforderungen von jMMORTS verbessert, wenn die Kombination folgender Kriterien gegeben ist:

- Berechnung der Pfade mit geringer Abweichung vom optimalen Pfad
- geringerer Speicherverbrauch
- Erfüllung der Echtzeitanforderungen durch schnelleres Pathfinding

Jene Kriterien sorgen somit für eine Erhöhung des Spielspaßes.

Derartig optimiert sind **Hierarchische Pathfinder**.

6 Hierarchisches Pathfinding

In diesem Kapitel wird das Vorgehen eines Hierarchischen Pathfinders beschrieben. Es ähnelt stark dem eines Menschen: Angenommen Person A möchte von der Frankfurter Straße in Kassel nach Shanghai zum Flughafen. Dann würde A sich zunächst einen Flughafen in der Nähe suchen, von dem aus sie nach Shanghai fliegen kann und findet dabei einen günstigen Flug von Berlin nach Shanghai. A weiß jetzt, dass sie zunächst von Kassel nach Berlin reisen muss, um dann nach Shanghai zu kommen. A kennt also seinen ersten groben Pfad (Frankfurter Straße Kassel - Berlin Flughafen - Shanghai Flughafen). Nun benutzt A iterative Tiefensuche und betrachtet den Weg von Kassel nach Berlin genauer. A stellt fest, dass sie dazu am günstigsten auf die Autobahn A 7 über Auffahrt Auestadion fährt. Der genauere Weg lautet dann: Kassel Frankfurter Straße - Kassel Autobahnauffahrt Auestadion - Berlin Flughafen - Shanghai Flughafen. Als nächstes schaut A nach, wie sie von der Frankfurter Straße zur Autobahnauffahrt Auestadion kommt usw..

Ist A ein Agent in einem Computerspiel, dann kann die KI ihn darüber informieren, dass er den oben angegebenen Weg vor sich hat und ihm sagen, wie er zur Autobahnauffahrt Auestadion kommt. A kann nun losfahren und den Pathfinder am Auestadion fragen, wie genau er nun vom Auestadion zum Berliner Flughafen gelangt.

Was genau tut Person A bei der Suche? Sie betrachtet zunächst eine **Hierarchieebene**, auf der es nur Flughäfen und Flugverbindungen gibt. Wo genau das Flugzeug entlang fliegt interessiert sie dabei nicht, sondern ausschließlich Start- und Zielpunkt, sowie die Kosten zwischen diesen beiden Punkten.

Angenommen es gäbe in Kassel keinen Flughafen und damit auch keine Flugverbindung von Kassel nach Berlin: A muss auf einer anderen (niedrigeren und detailreicheren) Hierarchieebene nach einem Weg zum Berliner Flughafen suchen. Sie sucht auf einer Ebene, auf der es keine Flughäfen und Flugverbindungen gibt, sondern lediglich Autobahnverbindungen mit Auf- und Abfahrten. A findet eine Autobahnverbindung zwischen Kassel und Berlin inclusive Auf- und Abfahrt, sowie Kosten zu dieser Verbindung.

Da A nicht direkt an der Autobahnauffahrt wohnt, muss sie auf einer noch niedrigeren Hierarchieebene weiter nach dem Weg zur Autobahnauffahrt Auestadion suchen. Sie sucht auf dem Stadtplan von Kassel. Darauf gibt es Straßenverbindungen und Straßenkreuzungen. Sie finder heraus, zu welcher Kreuzung sie als erstes fahren muss.

Die niedrigste Hierarchieebene, auf der A schauen müsste, wenn sie diesen Weg in jMMORTS suchen wollte, wäre die Karte des Spiels. Auf dieser Ebene muss der Weg Punkt für Punkt bestimmt werden.

Hierarchische Pathfinder verwenden unterschiedliche Ansätze um ihre Hierarchien zu erzeugen. Die vorgestellte Variante verwendet unterschiedliche Verkehrsknoten (Flughafen, Autobahnauffahrt, innerstädtische Straßenkreuzungen), um die Hierarchieebenen zu bilden. Der in Kapitel 7 vorgestellte Pathfinder verwendet unterschiedlich große Regionen, d.h. er verhält sich etwa so, als suche er erst einen Weg von Deutschland nach China, dann von Hessen nach Berlin (Land Berlin), anschließend von Nordhessen nach Südniedersachsen usw., immer tiefer auf den Hierarchieebenen bis hin zur hochdetaillierten Karte.

Die Hierarchieebenen werden durch **Level** gekennzeichnet. Auf Level 0 befindet sich die Karte von jMMORTS (genauer der Graph, der für jeden Punkt der Karte einen Knoten enthält (s.o.)). Bezogen auf das oben genannte Beispiel wären Straßenverbindungen auf Hierarchielevel 1, Autobahnverbindungen auf Hierarchielevel 2 und Flugverbindungen auf Hierarchielevel 3 (solange keine Landstraßen usw. modelliert werden). Je höher die betrachtete Hierarchieebene ist, desto weniger Knoten sind vorhanden. Auf Level 3 befinden sich nur noch Flughäfen, auf Level 0 hingegen ein Knoten pro $0,25m^2$ (die Auflösung der Spielwelt von jMMORTS). Ein Knoten, der sich auf Level X befindet, befindet sich auch auf allen Leveln kleiner als X, sonst wäre er auf den kleineren Leveln nicht erreichbar. Für Verbindungen zwischen Knoten gilt dies jedoch nicht. Die Hierarchieebenen bilden gemeinsam den **Hierarchiegraph**.

Der Hierarchische Pathfinder benutzt den A*-Algorithmus, um im Hierarchiegraph zu suchen, und kann somit die ersten Wegpunkte viel schneller berech-

nen (wenn er mehr Level als Hierarchielevel 0 benutzt) als ein reiner A*- oder Dijkstra-Algorithmus es tun könnte, die nur Hierarchielevel 0 zur Verfügung haben. Die Suche kann auf weniger detaillierten Hierarchieleveln schneller durchgeführt werden, als auf Hierarchielevel 0, weil dort weniger Knoten existieren. Es muss nur die ersten Knoten des Weges auf Hierarchielevel 0 bestimmen, damit Einheit A einen Teil der Bewegung durchführen kann. Der reine A*- oder Dijkstra-Algorithmus muss immer den kompletten Pfad berechnen, weil er sonst nicht weiß, ob es überhaupt einen Pfad gibt. Einheit A kann sich mit Hilfe dieser Methode in einem Computerspiel viel schneller in Bewegung setzen und den ersten Teil des Weges entlang bewegen.

Der vom Hierarchischen Pathfinder berechnete Weg benötigt zudem weniger Speicher, da er weniger Wegpunkte enthält. Ist A am Auestadion angekommen, können die Wegpunkte bis zum Auestadion gelöscht werden, bevor der Weg nach Berlin genauer berechnet wird. Daher wird nie der komplette detaillierte Weg abgespeichert und der Speicherverbrauch ist somit fast immer geringer als beim A*- oder Dijkstra-Algorithmus: Weil ein Hierarchiescher Pathfinder die Grundalgorithmen zur Suche im Hierarchiegraph benutzt, wird die Suche ausschließlich durch einen Grundalgorithmus durchgeführt, wenn der Weg so kurz ist, das nur Hierarchielevel 0 zu Suche benötigt wird. Sobald A einen genaueren Weg benötigt, kann mit Hilfe der noch vorhandenen Zwischenpunkte iterativ ein genauerer Weg berechnet werden.

Dieses Vorgehen hat einen weiteren Vorteil: In einem Computerspiel kommt es sehr oft vor, dass ein Spieler ein anderes Ziel für eine Einheit bestimmt, bevor sie am vorherigen Ziel angekommen ist. Das führt dazu, dass mit hierarchischem Pathfinding unnötige Berechnungen gesparrt werden. Angenommen die KI hätte den Weg von Kassel nach Shanghai mit dem A*- oder Dijkstra-Algorithmus berechnet und dann entscheidet sich der Spieler am Auestadion, dass A lieber nach Hamburg geschickt werden sollte. Nun hätte die KI den Weg vom Auestadion nach Shanghai durch den A*- oder Dikjstra-Algorithmus vollständig berechnet, ohne dass er gebraucht wird. Der Hierarchische Pathfinder passt sich an das Verhalten des Spielers an und führt weniger unnötige Berechnungen durch, da er nicht sofort den gesamten Pfad auf Hierarchielevel 0 berechnet.

Das die Berechnung durch den A*-Algorithmus viel mehr Aufwand bedeutet, als jene des hierarchischen Pathfinders, zeigt auch das in Abbildung 3 dargestellte Beispiel.

Der A*-Algorithmus läuft in diesem Beispiel zunächst gerade auf das Hindernis zu und breitet sich dann halbkreisförmig solange aus, bis er an einer Seite am Hindernis vorbeikommt. Sobald er das geschafft hat, läuft er vom Punkt neben dem Hindernis gerade auf den Zielpunkt zu. Das bedeutet, dass der A*-Algorithmus in diesem Beispiel sehr viele Punkte betrachtet, bis er einen Weg um das Hindernis herum findet. Das Beispiel ist ein besonders ungünstiger Fall für den A*-Algorithmus und kann genutzt werden, um Vorteile eines Hierarchischen Pathfinders zu zeigen.

Angenommen der Startpunkt und der Zielpunkt wären Stuttgart und Shanghai und Knoten auf der höchsten Hierarchieebene, auf der sich Stuttgart und Shanghai befinden, werden durch schwarze Punkte angegeben (in diesem Bei-

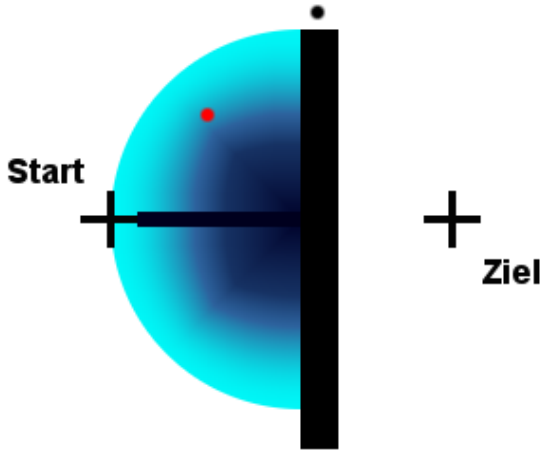


Abbildung 3: Diese Abbildung zeigt, wann welche Punkte durch den A*-Algorithmus betrachtet werden. Das schwarze Rechteck stellt ein Hindernis dar, die betrachteten Punkte vor dem Hindernis wurden blau eingefärbt. Je dunkler ein Punkt ist, desto früher wurde er vom A*-Algorithmus betrachtet (die rechteckige Fläche, die vom Start direkt auf die Mauer zuläuft enthält die zuerst betrachteten Punkte). Weiterhin sind ein schwarzer und ein roter Punkt dargestellt, diese gehören zu einem Hierarchischen Pathfinder. Der schwarze Punkt befindet sich auf einer sehr hohen Hierarchieebene (z.B. der Berliner Flughafen) und der rote Punkt auf einer detaillierteren Ebene darunter (z.B. ein Autobahnkreuz auf der Mitte des Weges).

spiel). Berlin ist der dargestellte schwarze Punkt und befindet sich damit auf der gleichen Ebene. Es existieren keine weiteren Knoten, die auf der höchsten Hierarchieebene für einen kürzesten Weg in Frage kommen. Der hierarchische Pathfinder wird deswegen schnell den Weg Stuttgart-Berlin-Shanghai finden (denn er betrachtet zunächst keine Punkte auf niedrigeren Ebenen). Um die ersten Wegpunkte zu finden, die direkt abgefahren werden können (Level 0 Punkte) bewegt er sich nun eine Ebene niedriger und findet das Autobahnkreuz. Dabei orientiert er sich nun aber schon nicht mehr in Richtung Shanghai sondern in Richtung Berlin (der vom Hierarchischen Pathfinder benutzte A*-Algorithmus verwendet Berlin als Ziel). Eine Ebene darunter orientiert er sich von Stuttgart zum erwähnten Autobahnkreuz. Dieser Weg ist auf der untersten Ebene - den Punkten auf unserer Karte - sehr leicht zu bestimmen. Der Algorithmus kann direkt auf das Autobahnkreuz zulaufen, weil sich dort kein Hindernis befindet. Dies zeigt, dass er nicht nur schneller die ersten Wegpunkte findet, sondern auch, dass er weniger Punkte während der Wegberechnung betrachtet, wenn er den

Weg komplett (überall bis auf die unterste Ebene) berechnet.

Die beschriebenen Vorteile helfen die in Kapitel 2 und 5 genannten Anforderungen (Echtzeitverhalten, viele Einheiten usw.) zu erfüllen.

7 HPA*-Pathfinding

Kapitel 6 hat nicht erklärt, woher ein hierarchischer Pathfinder weiß, wie hoch die Kosten der Level 3 Kante von Kassel nach Berlin sind. Der **HPA*-Algorithmus** (Hierarchical Pathfinding A*-Algorithmus)[2] berechnet diese Kanten während einer Vorverarbeitung der gesamten Karte. Dieser Algorithmus wurde für jMMORTS implementiert.

In diesem Kapitel wird zunächst die Idee, dann die Vorverarbeitung bzw. der Aufbau des Hierarchiegraphs und anschließend das Pathfinding des HPA*-Algorithmus erklärt. Am Ende des Kapitels wird eine optimierte Verwendung der Grundalgorithmen durch den HPA*-Algorithmus vorgestellt und es werden die Nachteile des HPA*-Algorithmus beschrieben.

7.1 Die Idee des HPA*-Algorithmus

Der HPA*-Algorithmus erzeugt **Cluster**, die Punkte auf der Karte, die nahe beieinander liegen, zusammenfassen. Diese Cluster können selbst wiederum zu größeren Clustern zusammengefasst werden, um eine höhere Hierarchieebene zu erzeugen. Die Cluster entsprechen den in Kapitel 6 erwähnten Regionen.

Benachbarte Cluster werden mit Hilfe von (Eingangs-)Knoten - und Kanten zwischen diesen Knoten - miteinander verbunden. Zusätzlich werden Kanten zwischen den Eingangsknoten innerhalb eines Clusters erzeugt. Sie zeigen an, wie der Cluster durchschritten werden kann.

Damit wird es möglich nicht nur Wege, bestehend aus einzelnen benachbarten Punkten auf der Karte zu suchen, sondern es können nun auch Wege von einem Cluster (bzw. Eingangsknoten) zum anderen gesucht werden.

Diese Wege sind weniger detailliert und befinden sich auf höheren Hierarchieleveln (1 und größer).

7.2 Vorverarbeitung

Die Vorverarbeitung dient dazu die Hierarchieebenen zu erzeugen. Es wird zwischen der Vorverarbeitung von Level 1 und höheren Leveln unterschieden.

7.2.1 Vorverarbeitung für Level 1 Graphen

Die Vorverarbeitung beginnt mit der Aufteilung der Karte in gleich große Cluster. Die verwendeten Cluster haben die Form von Rechtecken und ihnen werden die innerhalb eines Rechtecks liegenden Punkte zugeordnet.

In Abbildung 4 ist die für dieses Kapitel verwendete Beispielkarte zu sehen, die in Abbildung 5 mit Clustern der Größe 10 * 10 Felder versehen wurde.

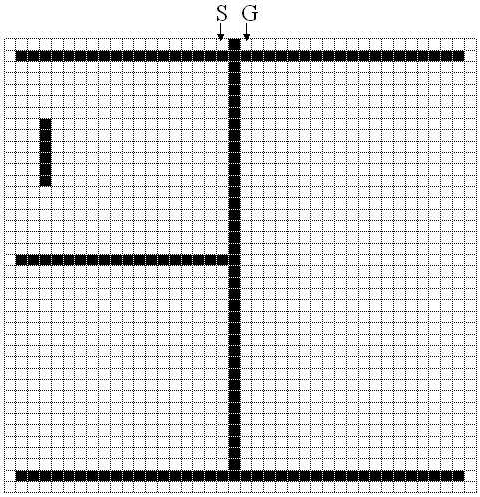


Abbildung 4: Die Beispielkarte[2] mit 160 * 160 Feldern. Die schwarzen Felder sind Hindernisse und die weißen sind passierbar. S markiert den Startpunkt und G den Zielpunkt.

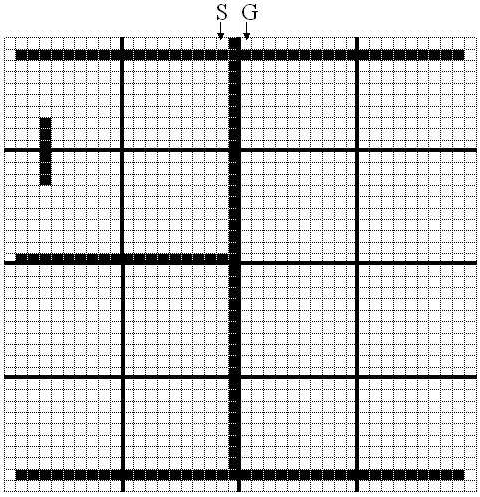


Abbildung 5: Die geclusterte Beispielkarte[2]

Nachdem die Cluster erstellt wurden, werden die Knoten für den Hierarchiegraph erzeugt. Dazu werden zwei Cluster ausgewählt, die aneinander angrenzen. Ihre gemeinsame Grenzfläche wird betrachtet.

Handelt es sich zum Beispiel um beliebige aneinander angrenzende Cluster C1 und C2 und befindet sich C2 rechts von C1, so bilden die rechte Spalte von

C1 und die linke Spalte von C2 die **Grenzfläche**. Diese Grenzfläche wird nun schrittweise von oben nach unten untersucht. Dabei werden immer ein Feld aus C1 und das waagerecht angrenzende aus C2 gemeinsam betrachtet. Sind diese beiden Felder passierbar, so wurde der Anfang eines **Einganges** sowohl in C1 als auch in C2 gefunden. Die schrittweise Untersuchung der Grenzfläche wird fortgesetzt. Der Eingang endet, wenn eines der Felder nicht passierbar ist oder das Ende der Cluster erreicht wurde.

Ist der Eingang klein (in [2] wird eine Maximalbreite von 6 vorgeschlagen, die auch in der Implementierung in jMMORTS verwendet wurde), so wird für jeden Cluster ein Knoten erzeugt, der in der Mitte des Eingangs liegt. Diese Knoten werden durch einer Kante miteinander verbunden. Handelt es sich hingegen um einen breiten Eingang, so werden in jedem Cluster zwei Knoten erzeugt. Jeweils einer am Anfang und einer am Ende des Eingangs. Die benachbarten Knoten werden dann miteinander verbunden.

Ist die Grenzfläche anschließend noch nicht vollständig betrachtet worden, wird sie auf weitere Eingänge hin untersucht.

In Abbildung 6 wird das Ergebnis für vier Cluster links oben in der Beispielkarte gezeigt. Für den Cluster rechts oben in dieser Abbildung wurde bereits der nächste Schritt durchgeführt: Die Kanten zwischen den Eingangsknoten innerhalb eines Clusters wurden erzeugt, um den Hierarchiegraphen zu vervollständigen.

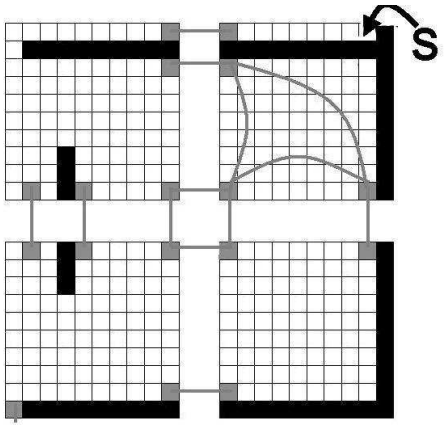


Abbildung 6: Cluster mit erstellten Eingangsknoten und Kanten. Diese sind grau dargestellt.
Anmerkung: Die Kanten wurden z.T. als gerade und z.T. als runde Linien dargestellt. Sie beschreiben ausschließlich, welcher Knoten mit welchem anderen verbunden wurde. Es handelt sich nicht um eine Darstellung des kürzesten Weges zwischen diesen Knoten. [2]

Das Erzeugen dieser Kanten geschieht, indem für jedes Knotenpaar in einem Cluster mit Hilfe des Dijkstra-Algorithmus oder des A*-Algorithmus der

kürzeste Weg berechnet wird. Der Weg kann je nach zur Verfügung stehendem Speicher entweder abgespeichert oder verworfen werden (meist wird er verworfen), die (Weg-)Kosten werden in jedem Fall an der Kante abgespeichert. Die erstellten Kanten bekommen als Maximum- und Minimumhierarchielevel 1 zugeordnet (welchem Zweck dies dient, wird in Abschnitt 7.3 erklärt).

Nun ist die Vorverarbeitung des HPA*-Algorithmus abgeschlossen - der Graph ist in Abbildung 7 zu sehen - es sei denn es soll mehr als ein Hierarchielevel (bzw. Hierarchielevel 1) erzeugt werden. Je mehr Hierarchielevel erzeugt werden, desto schneller kann später der erste Weg gesucht werden. Ob es Sinn macht weitere Level zu erzeugen, hängt zum einen von der Größe der Karte ab (spätestens sobald ein Cluster größer als die Karte wird, macht es keinen Sinn ein weiteres Level hinzuzufügen). Zum anderen davon, ob überhaupt Wege gesucht werden, bei denen Start und Zielpunkt weit genug voneinander entfernt sind, um die Kanten auf hohen Leveln zu benutzen. Befinden sich nahezu alle Wege die im Spielverlauf gesucht werden innerhalb eines Clusters mit dem höchsten erzeugten Hierarchielevel, so wurde ein Level zu viel erstellt. Diese Kanten dienen dazu diese Cluster zu durchqueren und nicht um einen Weg darin zu finden - schließlich verlaufen diese Kanten von einer Stelle am Rand eines Clusters zu einer anderen. Im seltenen Fall, dass von einer Stelle des Randes zu einer anderen ein Weg gesucht wird, können diese Kanten genutzt werden. Jedes Hierarchielevel beinhaltet allerdings zusätzliche Kanten und diese benötigen Speicher. Die Kanten sollten daher nur erzeugt werden, wenn sie auch benötigt werden.

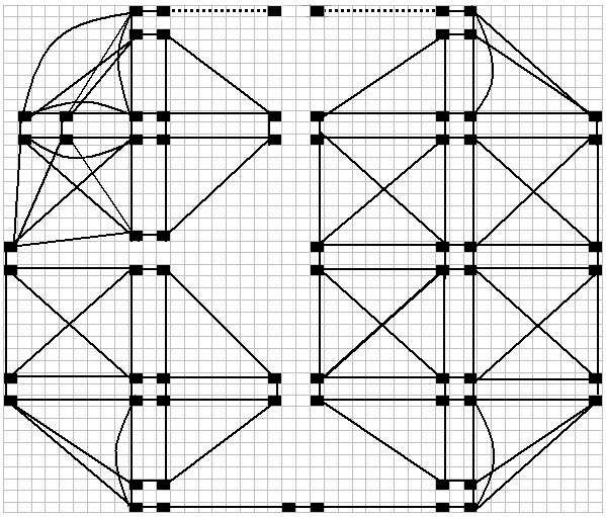


Abbildung 7: Der Hierarchielevel 1 Graph[2] für die Beispielkarte. Die Kanten zum Start und Zielknoten wurden gestrichelt dargestellt. Weder Start- und Zielknoten, noch die Kanten zu diesen Knoten werden in der Vorverarbeitung in den Graphen eingefügt.

7.2.2 Vorverarbeitung für Graphen größerer Level

In Abbildung 8 wird dargestellt, wie Level 2 Cluster gebildet werden. Es werden jeweils 4 Level 1 Cluster zu einem Level 2 Cluster zusammengefasst. Die Knoten dieser vier Level 1 Cluster, die im Grenzbereich des Level 2 Clusters liegen, werden nun Level 2 zugeordnet. D.h. ihr Leveleintrag wird auf 2 gesetzt und sie werden zum Level 2 Cluster hinzugefügt (bleiben jedoch auch in ihrem Level 1 Cluster). Zwischen diesen Knoten werden anschließend neue Kanten gezogen, wenn sie noch nicht durch irgendeine andere Kante (egal welchen Levels) verbunden sind. Die neuen Kanten bekommen als Maximum- und Minimumlevel 2 zugewiesen. Besteht schon eine Kante zwischen zwei Level 2 Knoten wird das Maximumlevel der Kante auf Level 2 gesetzt. Dadurch entsteht eine Bereichsangabe für Kanten - es wird angegeben zu welchen Leveln des Graphen die Kanten gehören (alle Level vom Minimumlevel bis zum Maximumlevel). Das Ergebnis ist in Abbildung 9 zu sehen.

Die gleichen Schritte können angewendet werden, um höhere Hierarchielevel hinzuzufügen.

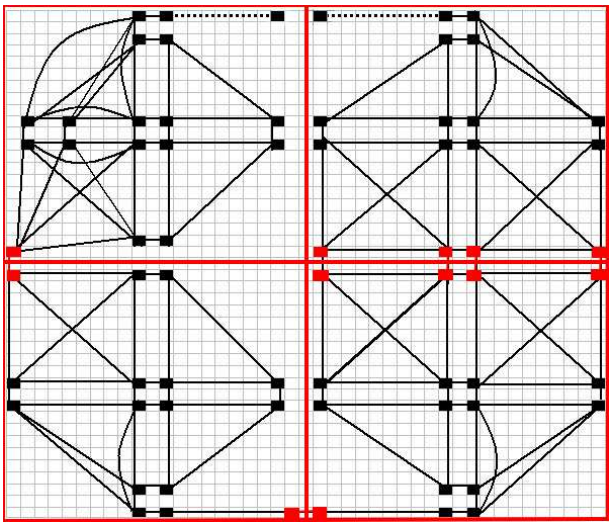


Abbildung 8: Abbildung 7 ergänzt um rote Rechtecke, die Level 2 Cluster darstellen, und mit rot gefärbten Knoten, die auf Level 2 gehoben wurden.

7.3 Der Pathfinding-Algorithmus des HPA*-Algorithmus

In [2] wird beschrieben, dass die ersten Schritte des Pathfindings das Einfügen des Start- und Zielknotens in den Hierarchiegraphen sind. Es wird weder in [2] noch in einer anderen mir bekannten Quelle beschrieben, wie das Einfügen und das spätere Löschen genau geschieht. Die Suche findet laut Beschreibung ausschließlich im Hierarchiegraphen statt und wird ebenfalls nicht genau erklärt. In

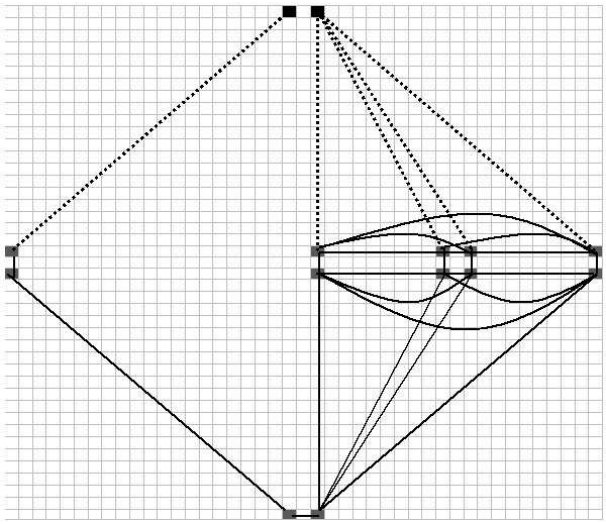


Abbildung 9: Der Level 2 Graph[2].

diesem Abschnitt wird erläutert, wie das Einfügen der Knoten und die Suche des Algorithmus in [2] funktionieren könnten. D.h. die im folgenden beschriebenen Varianten sind eigene Überlegungen.

Das Einfügen der Knoten ist notwendig, um *ausschließlich* im Hierarchiegraphen nach einem kürzesten Weg suchen zu können.

Variante I

In Variante I werden Start- und Zielknoten auf dem höchsten Level eingetragen, das im Hierarchiegraph vorhanden ist. Anschließend wird für jedes Level (außer 0) bestimmt, in welchem Cluster sich Start- und Zielknoten jeweils befinden. Von allen Clustern in denen sich der Startknoten befindet, werden die Eingangsknoten bestimmt und es wird von allen diesen Eingangsknoten eine Kante zum Startknoten erzeugt. Um die Kosten dieser Kanten zu bestimmen, kann einer der Grundalgorithmen verwendet werden. Für den Zielknoten wird das Gleiche durchgeführt.

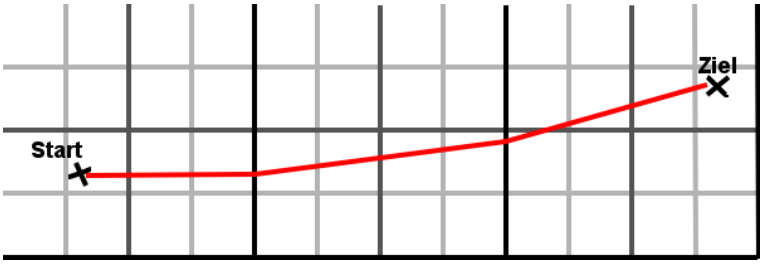
Diese Variante ermöglicht es mit dem A*-Algorithmus vom Start- zum Zielknoten einen Weg zu finden, der sich ausschließlich auf dem höchsten Hierarchielevel befindet und deswegen eine sehr kleine Anzahl an Knoten enthält (im Gegensatz zu Variante II wo es mehr Knoten sein können).

Variante II

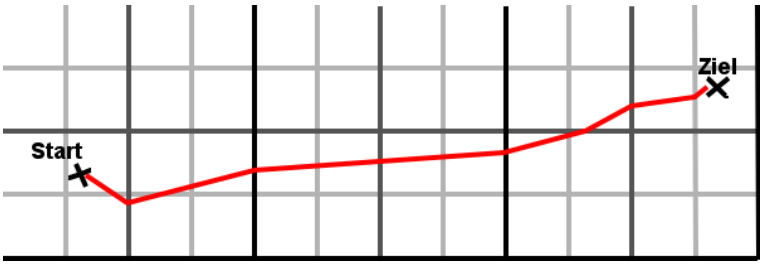
In Variante II werden Start- und Zielknoten ausschließlich auf Level 1 eingefügt, es sei denn es gibt an der Position des Start- oder Zielpunktes schon einen Knoten im Hierarchiegraph - dann wird dieser verwendet. Wurden Knoten eingefügt,

so werden Level 1 Kanten erzeugt, die die eingefügten Knoten mit den Eingangsknoten der Cluster verbinden, in welchen sich die neuen Knoten befinden. Die Kosten dieser Kanten können mit einem der Grundalgorithmen bestimmt werden.

Der Hauptvorteil dieser Variante liegt darin, dass weniger Kanten eingetragen werden müssen (je mehr Level der Hierarchiegraph hat, umso deutlicher wird der Unterschied) und dadurch Rechenzeit gespart wird. Schon für den ersten, möglichst groben Pfad werden dadurch allerdings Level 1 Kanten verwendet, denn der Algorithmus muss erst einen Weg zu einem Knoten finden, an dem sich Kanten höheren Levels befinden. Dies erhöht die Anzahl der Wegpunkte. Zur Verdeutlichung wird der erste Pfad für beide Varianten in Abbildung 10 dargestellt.



(a) Variante I: In diesem Fall wurden zum Start- und zum Zielknoten jeweils Level 3, Level 2 und Level 1 Kanten eingefügt. Der erste grobe Pfad kann mit einer Level 3 Kante beginnen.



(b) Variante II: Hier wurden nur Level 1 Kanten zu Start- und Zielknoten eingefügt. Der erste Pfad muss zuerst eine Level 1 Kante benutzen und ist daher detaillierter und enthält mehr Wegpunkte als bei Variante I.

Abbildung 10: Die Farben haben folgende Bedeutung: schwarz: Clusterlevel 3 Grenzen, dunkelgrau: Clusterlevel 2 Grenzen, hellgrau: Clusterlevel 1 Grenzen.

Variante II berechnet jedesmal Kanten welchen Levels überhaupt benutzt werden dürfen, um zum Ziel zu gelangen. Die geschieht, nachdem der A*- oder Dijkstra-Algorithmus das Element mit den geringsten Kosten aus der Priority Queue geholt hat und bevor er über die Kanten iteriert, die sich an dem Knoten des Elements befinden (siehe Abschnitt 4.2). Der Knoten des Elements mit den geringsten Kosten wird im Folgenden **aktueller Knoten** genannt. Die Benutzung der Kanten muss eingeschränkt werden, damit sichergestellt werden kann,

dass Kanten von hohen Hierarchieleveln überhaupt verwendet werden und der Grundalgorithmus nicht schon bei der Suche nach einem sehr groben Pfad einen Pfad auf Level 1 sucht.

Zum einen muss also ein möglichst hohes Level verwendet werden (siehe Abbildung 11), um die Anzahl der Wegpunkte gering zu halten. Zum anderen darf das Level aber auch nicht zu hoch sein, denn sonst wird der Zielpunkt nicht gefunden (siehe Abbildung 12). Daher gibt es ein **Maximum-** und ein **Minimumlevel**, sowohl als Schranke für die Suche, wie auch als Information an den Kanten. In Abbildung 13 wird beispielhaft dargestellt Kanten welchen Levels verwendet werden dürfen.

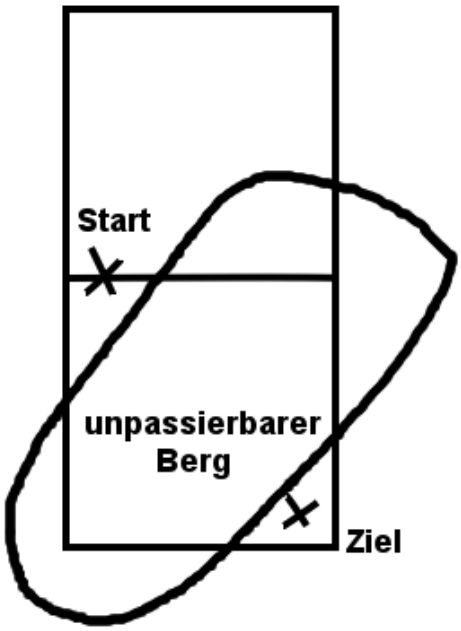


Abbildung 11: Ein Hindernis steht mitten in einem Level 3 Cluster zwischen Start- und Zielpunkt. Das Minimumlevel muss hier 3 sein, was eine zu genaue Betrachtung der linken oberen Ecke des Clusters vermeidet (ausschließlich interne Level 3 Kanten werden untersucht). Es handelt sich in diesem Beispiel nur um einen Teil einer Karte.

Das Berechnen der zu verwendenden Level stellt ein Nachteil von Variante II in Bezug auf die Rechenzeit dar. Diese Berechnungen werden in Variante I nicht benötigt.

Die möglichen Level der Kanten werden auf folgende Art und Weise bestimmt:

1. Das Maximumlevel wird durch zwei obere Schranken festgelegt(wovon nur

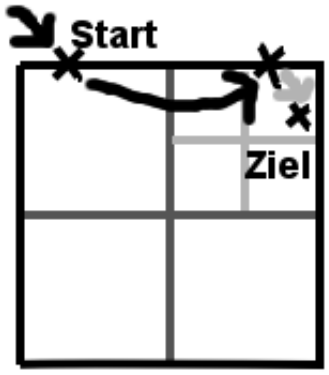


Abbildung 12: Hier wird zuerst über eine Level 3 Kante (durch den schwarzen Pfeil angedeutet) und dann über eine Level 1 Kante (grauer Pfeil) zum Ziel gegangen. Das Minimumlevel kann nicht durch Knotenlevel - 1 berechnet werden, da der Knoten vor dem Zielpunkt ein Level 3 Knoten ist und die Kante zum Zielpunkt hat (Maximum-)Level 1 ($1 \neq 3 - 1$).

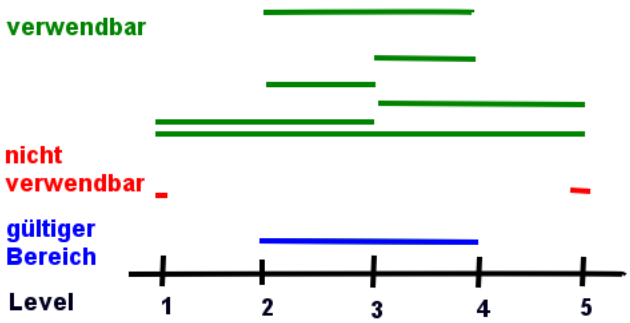


Abbildung 13: Diese Abbildung zeigt, Kanten welcher Level verwendet werden sollen und welche nicht. Nur Kanten die sich im "gültigen Bereich" befinden, sind erwünscht. Die als verwendbar eingezeichneten Kanten befinden sich im erwünschten Bereich und werden zur Pfadsuche benutzt, die anderen nicht. Level 2 stellt hier die untere Schranke der Kantenlevel dar und Level 4 die obere Schranke.

die jeweils niedrigere verwendet wird):

(a) Die erste obere Schranke ist das Level des aktuellen Knotens. Handelt es sich z.B. um einen Level 2 Knoten, gibt es keine Level 3 Kante die mit ihm verbunden ist (sonst wäre das Knotenlevel in der Vor-

verarbeitung auf Level 3 geändert worden).

(b) Die zweite obere Schranke bezieht sich auf die Entfernung des aktuellen Knotens zum Zielknoten. Der Pathfinding-Algorithmus soll keine Flugverbindungen bzw. Kanten hoher Level mehr betrachten, wenn der aktuelle Knoten sich bereits im gleichen Cluster wie der Zielknoten befindet. Es wird also der kleinste gemeinsame Cluster bestimmt und als obere Schranke für das Level der nächsten Kanten festgelegt. Befinden sich der aktuelle Knoten und der Zielknoten nicht in einem Cluster wird das maximale Level des Hierarchiegraphen als obere Schranke gewählt.

2. Das minimale Level der zu berücksichtigenden Kanten ist gleich dem Level des aktuellen Knotens, wenn er sich nicht zusammen mit dem Zielknoten in einem Cluster befindet (in diesem Fall soll ausschließlich das größtmögliche Level verwendet werden, um mit wenigen Knoten zum Ziel zu gelangen). Andernfalls ist es das Level des kleinsten Clusters in dem sich beide Punkte befinden - dann muss die Verwendung des Levels dieses Clusters erlaubt werden, damit der Zielknoten nicht verfehlt wird.

Das berechnete Maximumlevel wird mit dem Minimumlevel jeder Kante verglichen. Ist das Maximum kleiner als das Minimumlevel einer Kante, befindet sich die Kante auf zu hohen Hierarchieleveln und wird nicht verwendet. Außerdem wird das berechnete Minimumlevel mit dem Maximumlevel einer Kante verglichen. Ist das Maximumlevel der Kante geringer, als das berechnete Minimumlevel, so liegt die Kante auf zu niedrigen Hierarchieleveln und wird auch dann nicht verwendet.

Diese Berechnung der Level gilt jedoch nur für die Suche des ersten (groben), kürzesten Weges. Wurde schon ein kürzester Weg gefunden und soll ein detaillierterer Pfad gesucht werden, dann wird das vom Pathfinding berechnete Maximumlevel um 1 gesenkt, um zu garantieren, dass iterativ ein immer genauerer Pfad (auf immer niedrigeren Leveln) berechnet wird - im Folgenden auch *Abstieg* genannt. Würde dies nicht getan, d.h. nicht abgestiegen werden, würde der Algorithmus denselben Pfad zurückgeben, den er beim letzten Mal berechnet hat (das Ergebnis hängt davon an, in welcher Reihenfolge der Algorithmus die Kanten überprüft).

7.4 Optimierte Verwendung der Grundalgorithmen

Wann wird im HPA*-Algorithmus der Dijkstra und wann der A*-Algorithmus verwendet?

Laut [3] ist der Dijkstra-Algorithmus in bestimmten Fällen schneller als der A*-Algorithmus. Einen dieser Fälle stellt die Vorverarbeitung des HPA*-Algorithmus dar: Das Erstellen der Kanten innerhalb eines Clusters.

Wird der Dijkstra-Algorithmus so benutzt, dass er nicht abbricht, wenn er an einem Zielknoten angekommen ist, sondern so lange läuft, bis er die kürzesten Wege von einem Startknoten zu allen anderen (Eingangs-)Knoten in einem Cluster gefunden hat, dann kann der Dijkstra-Algorithmus schneller sein, als der

A*-Algorithmus. Dieser Teil des Dijkstra-Algorithmus muss dann nur noch einmal pro Knoten durchgeführt werden und nicht mehr einmal pro Knotenpaar. Es müssen anschließend allerdings immer noch alle Pfade rekonstruiert werden (d.h. von jedem (Eingangs-)Knoten außer dem Startknoten muss zum Startknoten zurück gelaufen werden).

Sei E die Anzahl der Eingangsknoten und L die Länge eines Clusters, der zur Vereinfachung quadratisch ist. Dann betragen die Abmessungen $L * L$, und damit ist die maximale Anzahl an Feldern am Rand eines Clusters $4L - 4$ und damit gilt auch $E \leq 4L - 4$.

Für den A*-Algorithmus gilt: Es gibt $E*(E-1)/2$ Paare von Eingangsknoten die verbunden werden müssen und im ungünstigsten Fall läuft der Algorithmus bei einer Wegsuche über alle Punkte im Cluster, was L^2 viele sind. Für jeden Punkt der expandiert wird, wird eine konstante Anzahl an Operationen auf der Priority Queue durchgeführt. Eine dieser Operation kostet $O(log(L^2))$, wobei L^2 die maximale Anzahl an Knoten auf der Queue ist. $O(E * (E-1)/2) \leq O(E^2)$, somit ergibt sich eine Laufzeit von $O(E^2 * L^2 * log(L^2))$.

Für den Dijkstra-Algorithmus ergibt sich auf den gleichen Weg eine Laufzeit von $O(E * L^2 * log(L^2))$, da dieser in der zuletzt beschriebenen Variante einen kürzesten Weg von einem Eingangsknoten zu allen anderen Eingangsknoten in einem Durchlauf sucht.

Expandiert der A*-Algorithmus jedoch nicht jeden Punkt im Cluster, da

- keine Hindernisse im Cluster vorhanden sind
- und der kürzeste Weg damit die Luftlinie ist
- der A*-Algorithmus aufgrund einer guten Heuristik auf das Ziel zuläuft, ohne Knoten abseits des kürzesten Weges zu betrachten

dann hat die Suche für alle kürzesten Wege zur Verbindung der Eingänge nur eine Komplexität von $O(E^2 * L * log(L^2))$. Der A*-Algorithmus ist daher eine bessere Wahl für Cluster mit wenigen Hindernissen und wenigen Eingängen, wohingegen der Dijkstra-Algorithmus somit die bessere Wahl für Cluster mit vielen Hindernissen und Eingängen darstellt.

Ideal wäre es für jede Karten eines Spiels zu untersuchen, welcher Algorithmus besser für die Vorverarbeitung der Karte geeignet ist und daraufhin diesen zu verwenden. Ich habe für die Vorverarbeitung den Dijkstra-Algorithmus gewählt und für das Pathfinding den A*-Algorithmus:

Beim Pathfinding wird immer eine Suche von einem Startpunkt zu einem Zielpunkt durchgeführt. Kürzeste Wege zu anderen Punkten sind nicht von Interesse und daher kann die in diesem Abschnitt beschriebene Variante des Dijkstra-Algorithmus nicht genutzt werden. Die Variante zur Suche nur eines kürzesten Weges ist im Durchschnitt langsamer als der A*-Algorithmus. Für den Dijkstra-Algorithmus gilt dann: $O(E * L^2 * log(L^2))$ für E = 1 entspricht $O(1 * L^2 * log(L^2))$ gegenüber $O(1 * L * log(L^2))$ beim A*-Algorithmus im besten Fall und $O(1 * L^2 * log(L^2))$ im schlechtesten Fall. Weil nicht immer der schlechteste Fall eintritt, ist der A*-Algortihmus im Durchschnitt schneller.

Wie beschrieben hängt die Geschwindigkeit der Algorithmen von der Karte ab. Da noch nicht feststeht, wie die Karten gestaltet sind, kann nicht entschieden werden, welcher der Algorithmen für die Vorverarbeitung die besten Ergebnisse erzielen wird. Ich habe mich entschieden den Dijkstra-Algorithmus für die Vorverarbeitung zu verwenden. Dies hat den Vorteil, dass nun beide Algorithmen jMMORTS zur Verfügung stehen und die Algorithmen bei Bedarf ausgetauscht werden können.

7.5 Nachteil des HPA*-Algorithmus

Der Weg, den der HPA*-Algorithmus berechnet, führt immer über Eingangsknoten (es sei denn Start- und Zielpunkt liegen in demselben Level 1 Cluster). Diese Knoten liegen aber nicht immer auf einem kürzesten Weg. Der HPA*-Algorithmus berechnet daher meist nur einen Weg, der gering von einem kürzesten Weg abweicht. Wie in Kapitel 5 zu lesen war, ist dies ein zu vernachlässigender Nachteil dieses Algorithmus. Eine Möglichkeit zur Pfadverbesserung und Experimente zu Pfadabweichungen sind in [2] nachzulesen.

8 Die Parallelisierung des HPA*-Algorithmus

Das parallelisierte Programm verwendet einen Thread mehr als Prozessorkerne für das Pathfinding eingesetzt werden sollen. Ein Thread ist dabei ausschließlich dafür zuständig die Tasks zu erzeugen und die Ergebnisse auszuwerten (genaueres siehe Kapitel 9) - z.B. wird überprüft ob Exceptions geworfen wurden - und weiterzugeben. Die übrigen Threads führen die Berechnungen während der Vorverarbeitung bzew. einzelne Suchanfragen parallel durch. Jeder Thread soll einen Prozessorkern mit Berechnungen auslasten. Jede einzelne Suchanfrage wird also sequentiell bearbeitet.

In diesem Kapitel wird zuerst die Parallelisierung der Vorverarbeitung und anschließend die Parallelisierung der Pfadsuche beschrieben.

8.1 Vorverarbeitung

In der Vorverarbeitung gibt es laut Kapitel 7 folgende Schritte für das Erzeugen einer Hierarchieebene:

1. Das Erzeugen der Cluster.
2. Das Erzeugen (Level 1) oder Bestimmen (Level 2 und höher) der Eingangsknoten.
3. Das Verbinden der Eingangsknoten innerhalb eines Clusters.

Diese Schritte werden nacheinander durchgeführt. Jeder Schritt wurde einzeln parallelisiert.

8.1.1 Clustererzeugung

Im sequentiellen Fall wird beim Erzeugen der Cluster für jeden Cluster ein Clusterobjekt angelegt. Die Größe dieses Clusters wird durch zwei Kartenkoordinaten festgelegt, die ein Rechteck aufspannen. Diese Koordinaten geben zwei gegenüberliegende Ecken des Rechtecks an und definieren es damit eindeutig.

Weiterhin gibt es eine Matrix, in der alle Cluster eines Levels eingetragen werden. Die Matrix ist so aufgebaut, dass der Cluster, der die Punkte links oben auf der Karte zusammenfasst sich auch links oben in der Matrix befindet. Die Berechnung der Kartenkoordinaten benötigt daher als Parameter die Position des Clusters in dieser Matrix und das Level für das der Cluster erzeugt werden soll. Bei der Erzeugung eines Clusters werden ausschließlich zwei Informationen benötigt:

1. die Position des Clusters in der Matrix
2. das Level des Clusters

Das Erzeugen eines einzelnen Clusters benötigt nie Informationen, die beim Erzeugen irgendeines anderen Clusters verändert werden. Bei der Erzeugung von zwei beliebigen Clustern wird außerdem nie auf eine gemeinsame Speicherstelle geschrieben. Bei jeder Clustererzeugung wird zwar in die Matrix geschrieben, aber nur an eine Stelle und zwar die, in der eine Referenz auf das Cluster angelegt wird. Diese Stelle ist für zwei beliebiger Cluster niemals die selbe.

Dadurch, dass beim Erzeugen von zwei Clustern nie die selben Informationen benutzt werden, kann bei der Parallelisierung der Erzeugung auf Synchronisierung verzichtet werden. Mehrere Threads können gleichzeitig verschiedene Cluster erzeugen.

8.1.2 Das Erzeugen oder Bestimmen der Eingangsknoten

Es war schwieriger Schritt 2 zu parallelisieren als die Schritte 1 und 3: In Schritt 1 werden zu keinem Zeitpunkt Daten zur Bearbeitung von zwei Tasks benötigt(s.o.). Das Gleiche geschieht in Schritt 3 (s.u.). Bei Schritt 2 ist dies jedoch nicht der Fall.

In Schritt 2 bearbeiten bis zu vier Threads die Knotenliste eines Clusters, da bei der Ausführung eines Task die Eingangsknoten in ein Cluster von genau *einem* Nachbarcluster erzeugt werden. Cluster haben allerdings zwei bis vier Nachbarknoten (in den Ecken der Karte sind es zwei, an anderen Stellen des Rands drei und bei allen anderen Clustern vier Nachbarcluster). Es darf nicht passieren, dass mehrere dieser Tasks gleichzeitig bearbeitet werden und gleichzeitig in die Knotenliste geschrieben wird, denn dann kann, wie in Kapitel 3 beschrieben, ein Fehler erfolgen. Daher ist Synchronisation erforderlich, für deren Umsetzung mir zwei Lösungen eingefallen sind:

1. Die Zugriffsmethoden auf die Knotenlisten werden synchronisiert, so dass nicht zur gleichen Zeit zwei Mal die selbe Knotenliste zugegriffen werden. Wollen zwei Threads an einer Knotenliste arbeiten, muss einer warten.

Zugriffe auf die Knotenliste werden aber auch an anderen Stellen des Algorithmus verwendet, z.B. bei der Suche. Möchte man diese Variante der Synchronisation verwenden, muss darauf geachtet werden, dass die Synchronisation nicht andere Teile des Algorithmus verlangsamt (z.B. durch das Setzen eines Locks zur Synchronisation), die auch auf die Knotenliste zugreifen. Diese Variante benötigt mindestens 1 mal pro Clustergrenze das Setzen und Entfernen eines Locks.

2. Das Erzeugen oder Bestimmen der Eingangsknoten wird in vier Schritte eingeteilt. In jedem dieser Schritte werden für jedes Cluster nur die Eingangsknoten in eines der benachbarten Cluster bestimmt (hat ein Cluster weniger Nachbarcluster als vier, so wird es in manchen dieser Schritte nicht betrachtet). Durch diese Einteilung wird niemals in einem der Schritte die Knotenliste eines Clusters von zwei Tasks benötigt. Fügt man eine Barriere nach jedem Schritt ein, so verhindert man das zwei Tasks verschiedener Schritte gleichzeitig bearbeitet werden und diese gleichzeitig auf die selbe Knotenliste zugreifen.
   Der erste der vier Schritte betrachtet die Grenzen zwischen Clusterpaaren der Zeilen 0 und 1, 2 und 3, 4 und 5, usw. der Matrix, in der die Cluster eingetragen wurden. In Schritt 2 werden die Grenzen zwischen Clusterpaaren der Zeilen 1 und 2, 3 und 4, 5 und 6 usw. betrachtet. In Schritt 3 wird das Selbe wie in Schritt 1 nur Spalten statt Zeilenweise durchgeführt. In Schritt 4 geschieht das Ganze dann noch einmal Spaltenweise mit den gleichen Zeilennummer wie in Schritt 2. Jeder dieser Schritte benutzt completeBlockingExecution zur Synchronisation. Der Nachteil dieser Variante ist die Synchronisation durch completeBlockingExecution. Diese kann dazu führen, dass ein Thread noch eine Berechnung durchführen muss und alle anderen nicht arbeiten können, weil sie auf den einen Thread warten müssen.

Ich vermute, dass die Synchronisation der ersten Variante mehr Zeit kostet, als die der zweiten. Die Barrieren der zweiten Variante halte ich für weniger zeitraubend als das Locking von Variante 1. Für diese Vermutung spricht das Verhältniss zwischen der gewöhnlich verwendeten Anzahl an Threads und der Anzahl an Task. Auf einem 4 Prozessorkernrechner und bei einer Karte der Größe 1000*1000 Punkte und 10*10 Punkten großen Level 1 Cluster (dies sind Werte die auch für die Messungen in Kapitel 10 verwendet wurden) gibt es ungefähr 1000*1000/(10*10) = 10000 Task und damit ungefähr 2500 Tasks pro Prozessorkern in einem der Schritte von Variante 2. Selbst wenn es einen Task geben sollte, dessen Bearbeitung 10 mal soviel Zeit benötigt, wie bei einem durchschnittliche Task, so benutzt er doch nur etwa 10/2500 der gesamten verwendeten Zeit eines Prozessorkerns. Wenn dieser Task im ungünstigsten Fall als letztes ausgeführt würde und die Berechnung beginnen würde, während drei Prozessoren bereits die Arbeit für diesen Schritt beenden, so müssen diese höchstens 10/2500 der gesamten Berechnungsdauer warten. Bei höheren Level kann diese Barriere sich stärker auswirken, da die Anzahl der Tasks geringer ist.

Variante 1 nutzt bei jeder Veränderung der Knotenliste Locking und dadurch kann es in dieser Variante geschehen, dass ein Thread warten muss, weil ein anderer eine Knotenliste gesperrt hat. Kommt dies nur zwölf mal auf einem 4 Prozessorkernrechner vor, so entspricht dies dem gleichen Zeitverlust, den die vier Barrieren erzeugen können (maximale Anzahl wartender CPUs an einer Barriere * Anzahl der Barrieren = 3 * 4 = 12). Zwölf Vorkommen bei 10000 Tasks sind leicht zu erreichen.

Aus den genannten Gründen bin ich zu dem Schluß gekommen Varinate 2 zu implementieren. Jeder der vier Schritte wurde für sich betrachtet auf die gleiche Art und Weise parallelisiert, wie die Erzeugung der Cluster.

8.1.3 Das Verbinden der Eingangsknoten innerhalb von Clustern

Für das Verbinden oder Bestimmen der Eingangsknoten gilt das Gleiche wie bei der Clustererzeugung: es kann auf Synchronisierung verzichtet werden. In Schritt 3 kann ebenso Cluster für Cluster vorgegangen werden. Das Verbinden der Eingangsknoten benötigt nur Informationen, die sich in dem Clusterobjekt befinden und bei Level 1 Informationen über die Punkte im Cluster (Punkte der Karte). Die Punkte auf der Karte werden dabei nie verändert und stellen damit kein Problem bei der Paralleisierung dar, die Knoten die zu einem Cluster gehören jedoch schon, denn an die Knoten werden Kanten gefügt. Programmiert man das Verbinden der Eingangsknoten in einem Cluster als ein einzelnen Task, stellt das Verändern der Knoten ebenfalls kein Problem mehr da, weil dann nur noch während dieses einen Tasks die Knoten verändert werden und diese somit nie von 2 Threads verändert werden (denn ein Task wird immer von genau einem Thread bearbeitet).

8.1.4 Die Wahl des Threadpools

Zur Parallelisierung der Vorverarbeitung wurden ausschließlich FixedThreadPools benutzt. Einen SingleThreadExecutor kommt für die Parallelisierung der Vorverarbeitung nicht in Frage, da ein Thread nicht ausreicht, denn es sollen identische Berechnungen mit unterschiedlichen Daten durchgeführt werden und zwar auf so vielen Prozessorkernen, wie auf dem benutzten Rechner vorhanden sind. Dafür werden mindestens so viele Threads gebraucht, wie der Rechner Prozessorkerne besitzt.

Die Tasks sollen nicht periodisch ausgeführt werden. Die Vorverarbeitung wird einmal während des Programmablaufs durchgeführt, daher kommt auch kein ScheduledThreadPool in Frage.

Der CachedThreadPool ist ebenfalls nicht die beste Wahl für die Vorverarbeitung. Die Prozessorkerne sollen ausgelastet werden und dies geschieht bei einer entsprechend großen Karte (für kleine Karten könnte der A*-Algorithmus benutzt werden und niemand bräuchte sich die Mühe zu machen, den hier vorgestellten Algorithmus zu implementieren, weil die Vorteile des HPA*-Algorithmus dann nicht mehr zum Tragen kommen). Die Belastung der Kerne ist in einem

parallelen Abschnitt der Vorverarbeitung sehr gleichmäßig und ein Pool, der seine Größe dynamisch anpasst, ist nicht erforderlich.

Mit einem FixedThreadPool bei dem die Anzahl der Threads gleich der Anzahl der CPUs ist, kann die Leistung eines Parallelrechners ausgeschöpft werden und daher wurde der FixedThreadPool implementiert.

8.2 Pfadsuche

Dieser Abschnitt beschreibt die Suche des parallelen HPA*-Algorithmus. Zuerst werden Probleme beschrieben, die bei der Parallelisierung der Varianten zur Suche im sequentiellen Fall (siehe Abschnitt 7.3) auftreten und dann wird die verwendete Lösung vorgestellt.

Diese Lösung ermöglicht eine Parallelisierung analog zur Vorverarbeitung. Alle Anfragen an den Pathfinder während eines Durchlaufs der Mainloop können anschließend analog zur Parallelisierung der Vorverarbeitung gehandhabt werden.

Zuletzt wird in diesem Abschnitt die Wahl des Threadpools erläutert.

8.2.1 Parallelisierungsprobleme mit den sequentiellen Varianten

Die in Abschnitt 7.3 vorgestellten Varianten zur Suche von kürzesten Wegen sind für die Parallelisierung keine gute Wahl:

Ein und der selbe Hierarchiegraph soll im laufenden Spiel nur einmal existieren, um Speicher zu sparen. Parallel arbeitende Threads müssen daher den Hierarchiegraph gemeinsam verwenden. Das Eintragen der Start- und Zielknoten in den Hierarchiegraphen zur Suche - wie in Abschnitt 7.3 beschrieben - führt dazu dass der Hierarchiegraph durch alle Threads gleichzeitig verändert werden kann. Im parallelen HPA*-Algorithmus werden die Start- und Zielknoten nicht im Graphen eingetragen, um die folgenden Probleme zu vermeiden:

1. Wenn die Start- und Endknoten in den Hierarchiegraphen eingetragen werden sollen, so müssen sie auch wieder daraus entfernt werden. Würden sie nicht entfernt, würde nach einiger Zeit für jeden Punkt der Karte ein Knoten existieren und dies verbraucht zu viel Speicher.
   Das Entfernen der Knoten muss synchronisiert werden, damit kein Thread auf einen Knoten oder eine Kante zugreifen kann, die ein anderer gerade gelöscht hat.

2. Wenn Start- und Zielknoten im Hierarchiegraphen eingetragen werden, dann können sie auch für die Suche von Pfaden, für die sie nicht Start- und Zielknoten darstellen, verwendet werden. Geschieht dies befinden sie sich in der Mitte eines Pfades. Werden diese Start- und Zielknoten entfernt, um den Speicherverbrauch zu reduzieren, können ungültige Pfade entstehen, was dazu führt, dass auf nicht mehr existente Daten (bzw. Knoten) zugegriffen wird. Sollen diese Zugriffe unterbunden werden, sind ständige Sicherheitsabfragen nötig, die z.B. überprüfen ob ein Knoten noch existiert.

3. Es kann nicht verhindert werden, dass mehrere Threads für die selbe Position auf der Karte gleichzeitig einen Start- und Zielknoten einfügen, es sei denn es wird eine sehr umfassende Synchronisierung eingeführt. Diese Synchronisierung müsste den Zugriff auf ganze Teile des Graphen oder sogar den kompletten Graphen sperren, da während des Einfügens der Zugriff auf alle Knoten unterbunden werden muss, die mit dem neuen Start- oder Endknoten verbunden werden sollen. Sperrt man den kompletten Graphen, so kann in dem Moment der Sperrung nicht mehr parallel gearbeitet werden, was aber durch die Parallelisierung erreicht werden sollte.
   Das *Locking* eines Teilgraphen ist schwierig, weil nicht alle Knoten gleichzeitig gelockt werden können (außer man *lockt* den kompletten Graphen). Soll der Thread die Knoten nacheinander *locken*, so muss verhindert werden dass ein zweiter Thread die gleichen *Locks* (in einer anderen Reihenfolge) setzt, denn dann entsteht ein Deadlock - beide Threads bekommen nicht alle *Locks*, die sie benötigen und warten darauf, dass der jeweils andere sie wieder frei gibt.
   Solange nur ein Cluster betrachtet wird, in das gleichzeitig zwei Knoten eingefügt werden sollen, könnte dies dadurch gelöst werden, dass die Knoten eines Clusters sortiert werden und die Threads nur dann versuchen, den zweiten zu *locken*, wenn sie den ersten schon *gelockt* haben usw. Dann würde derjenige Thread alle *Locks* bekommen, der den ersten Lock bekommt. Tragen aber mehrere Threads gleichzeitig Knoten in benachbarte Cluster ein, so kann es zu Ketten von Blockaden kommen, die nur schwer und kostspielig zu behandeln sind.
   Zusammengefasst bedeutet dies, dass Synchronisierung an dieser Stelle so kompliziert ist oder im Fall des Lockings des gesamten Graphen eine so starke Sequentialisierung darstellt, dass es keine benutzbare Lösung ermöglicht.

4. Wird die Existenz von zwei Punkten an der selben Position nicht verhindert, so muss der Algorithmus Informationen enthalten um entscheiden zu können, ob ein Knoten noch verwendet wird oder nicht. Denn nur wenn er weiß, dass er nicht mehr verwendet wird darf er ihn entfernen. Um dies entscheiden zu können gibt es zwei Möglichkeiten:

   (a) Jeder Knoten enthält eine Liste mit allen Pfaden, die ihn verwenden. Diese Möglichkeit kostet beim Einsatz von massiv vielen Einheiten und damit vielen Pfaden viel Speicher.

   (b) Der Algorithmus muss über alle Pfade iterieren und nach schauen, ob der Knoten, der entfernt werden soll, noch von einem Pfad verwendet wird. Dies kostet bei vielen Einheiten viel Rechenzeit.

   Beide Möglichkeiten sind auf Grund ihrer Kosten schlecht.

8.2.2 Pfadsuchalgorithmus des parallelen HPA*-Algorithmus

In diesem Algorithmus wird auf das Einfügen und damit auch das Löschen von zusätzlichen Knoten in den Hierarchiegraphen verzichtet. Trotzdem werden Knoten benötigt, um im Hierarchiegraphen suchen zu können. Der Algorithmus sucht sich diese (Start- und End-) Knoten in der Nähe des Start- und Zielpunktes, und bestimmt drei Teilwege. Der erste Teilweg geht vom Startpunkt, zum Startknoten, der zweite Teilweg vom Startknoten zum Zielknoten und der dritte Teilweg vom Zielknoten zum Zielpunkt. Der zweite Teilweg kann wie gewünscht im Hierarchiegraph gesucht werden. Der erste und der dritte Teilweg verbinden Startpunkt und Startknoten bzw. Endpunkt und Endknoten und vervollständigen somit den Weg. Der Algorithmus durchläuft folgende Schritte:

Bestimmung des Startknotens - erster Teilweg

Zuerst werden alle Knoten des Level 1 Clusters bestimmt, in dem sich der Startpunkt befindet und anschließend wird versucht diese mit dem Dijkstra-Algorithmus vom Startknoten aus zu erreichen. Alle Knoten, die erreicht wurden, bekommen Kosten zugeordnet. Diese Kosten berechnen sich aus den Kosten, die der Dijkstra-Algortihmus bestimmt hat plus den geschätzten Kosten bis zum Zielpunkt (geschätzt wird auf die gleiche Weise wie beim A*-Algorithmus). Der Knoten mit den geringsten Kosten wird als Startknoten festgelegt (siehe Abbildung 14).

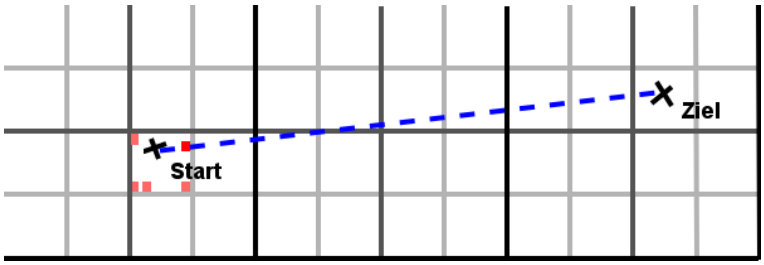


Abbildung 14: Alle roten und rosa Punkte sind Kandidaten für den Startknoten (alle Knoten des Level 1 Clusters). Der rote Punkt ist der gewählte Kandidat. Die blaue durchgezogene Linie ist der Weg vom Startpunkt zum Startknoten und die gestrichelte Linie deutet die von der Heuristik geschätzte weitere Weglänge an.

Bestimmung der Zielknotenkandidaten

In diesem Schritt wird kein bestimmter Zielknoten gesucht, wie bei der Bestimmung des Startknotens, sondern es werden die Knoten gewählt, die sich zusammen mit dem Zielpunkt in einem Level 1 Cluster befinden und die auch vom Zielpunkt aus erreichbar sind. Die Erreichbarkeit kann mit einem der Grundalgorithmen überprüft werden. Die gewählten Knoten sind die Zielknotenkan-

didaten. Dieses wird in Abbildung 15 dargestellt. Es ermöglicht die Berechnung eines kürzeren Pfades, wenn mit einer Menge von Zielknoten (den Zielknotenkandidaten) gearbeitet wird, anstatt eines festgelegten Zielknoten (s.u.).

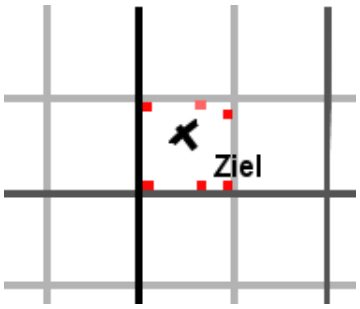


Abbildung 15: Die roten und die rosa Punkte sind Kandidaten für Zielknoten (alle Knoten des Clusters). Nur die roten Punkte sind vom Zielknoten aus erreichbar (auf einem Weg, der das Cluster nicht verlässt).

Bestimmung des Weges vom Start- zu einem Zielknoten

Es wird eine Suche im Hierarchiegraphen durchgeführt. Die Heuristik schätzt die Kosten von dem Knoten, der gerade vom Algorithmus betrachtet wird, bis zum Zielpunkt (nicht zu einem Zielknoten). Sobald der Algorithmus einen Weg zu einem der Endknoten gefunden hat, ist der erste Pfad auf dem Hierarchiegraphen gefunden worden (hier wird nun davon ausgegangen, dass es sich dabei um den am besten geeigneten Endknoten handelt, was nicht immer stimmt). Ab diesem Zeitpunkt gibt es nur noch einen Zielknoten (siehe Abbildung 16).

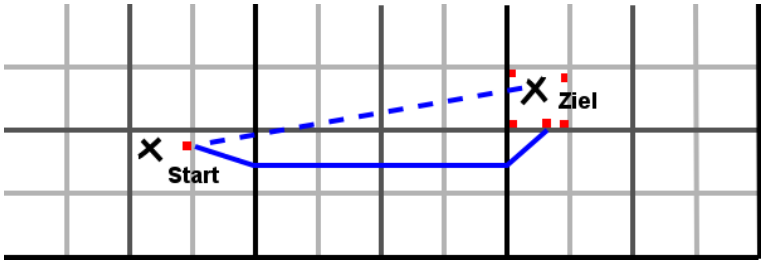


Abbildung 16: Die gestrichelte blaue Linie zeigt die Abschätzung vom Startknoten zum Zielpunkt an. Die durchgezogene blaue Linie ist der erste berechnete grobe Pfad zwischen dem Startknoten und einem der Zielknotenkandidaten. Der Zielknotenkandidat zu dem der Weg führt ist von nun an der Zielknoten des Weges.

Durch die Verwendung einer Menge von Zielknotenkandidaten und der Schätzung zum Zielpunkt anstatt zu einem Zielknoten, wird der letzte Knoten vom

Pathfinding später festgelegt, als wenn er gleich zu Beginn der Suche durch eine Schätzung der Entfernung jedes Knotens bis zum Start berechnet würde. Je später der Endknoten festgelegt wird, desto mehr ist über den kürzesten Weg bekannt und damit liegt der gewählte Endknoten mit höherer Wahrscheinlichkeit auf dem kürzesten Weg.

Dieser Teil arbeitet mit Ausnahme des Vorhandenseins mehrerer Endknoten, ebenso wie in Abschnitt 7.3 beschrieben.

Der Weg wird wie im HPA*-Algorithmus anschließend immer weiter verfeinert. Auch hier wird das Verfahren aus Abschnitt 7.3 verwendet.

Bestimmung des Weges Zielknoten zum Zielpunkt

Sobald das Ende des Pfades von Endknoten zu Endpunkt berechnet werden soll, wird dies mit Hilfe des Dijkstra-Algorithmus getan (siehe Abbildung 17).

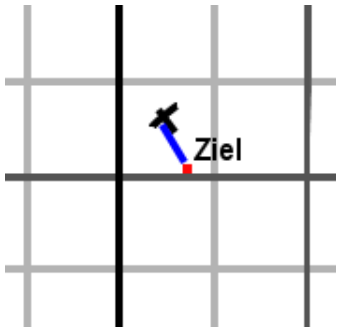


Abbildung 17: Der Weg (in blau) zwischen Zielknoten (in rot) und dem Ziel.

8.2.3 Die Wahl des Threadpools

Die Wahl des Threadpools fällt auch an dieser Stelle auf einen FixedThreadPool. Weil immer alle Pfadanfragen eines Mainloopdurchlaufs gebündelt behandelt werden (siehe Abschnitt 9.2), treffen auch hier die Argumente zu, die bei der Wahl des Threadpools für die Vorverarbeitung erläutert wurden.

9 Implementierung

In diesem Kapitel werden ausgewählte Teile der Implementierung vorgestellt. Es behandelt zunächst die Implementierung der Vorverarbeitung und anschließend die Implementierung der Pfadsuche, sowie die Integration des Pathfinders.

9.1 Vorverarbeitung

Die Implementierungs-Beschreibung der Vorverarbeitung ist entsprechend Abschnitt 8.1 eingeteilt in die folgenden Schritte:

1. Das Erzeugen der Cluster.
2. Das Erzeugen (Level 1) oder Bestimmen (Level 2 und höher) der Eingangsknoten.
3. Das Verbinden der Eingangsknoten innerhalb eines Clusters.

9.1.1 Clustererzeugung

Listing 2 zeigt den Abschnitt des Programmes zur Clustererzeugung.

```
1  Executor myExecutor =
2      Executors.newFixedThreadPool(numberOfThreads);
3
4  ExecutorCompletionService<Void> ecs =
5      new ExecutorCompletionService<Void>(myExecutor);
6
7  for (int i = 0; i < numberOfClusterInY; ++i){
8      for (int j = 0; j < numberOfClustersInX; ++j){
9          ecs.submit(
10             new BuildClusterCaller(
11                 level, map, widthAndHight,
12                 numberOfClustersInX, numberOfClusterInY,
13                 i, j
14             )
15         );
16     }
17 }
18
19 int taskCount = clusterSizeX * clusterSizeY;
20 blockingCompleteExecution(ecs, taskCount, myExecutor);
21
```

Listing 2: Erzeugen und Füllen eines Executors

In Zeile 1f wird ein `Executor` mit einem `FixedThreadPool` erzeugt und anschließend (Zeile 4f) an einen `ExecutorCompletionService` übergeben. Der `ExecutorCompletionService` bietet Methoden zum Umgang mit dem `Executor` und dem Taskpool an, die der `Executor` selbst nicht bereit stellt. Eine davon ist `submit`, die (Zeile 8) einen neuen Task zur Bearbeitung anmeldet. Der erzeugte `ExecutorCompletionService` benutzt `Callables`, die den Rückgabetyp `Void` besitzen, was in Zeile 3 mit angegeben wird. Es wird an dieser Stelle ein `Callable<Void>` statt einem `Runnable` benutzt, da kein Rückgabewert, aber Informationen über evtl. aufgetretene Exceptions benötigt werden, um die Exceptions der Mainloop von jMMORTS mitteilen zu können. Diese muss dann entscheiden, ob sie mit einer fehlerhaft durchgeführten Vorverarbeitung weiterarbeiten will, oder nicht.

Durch die Zeilen 7 und 8 wird über die Matrix, der zu erzeugenden Cluster iteriert. Für jeden Cluster, der zu erstellen ist, wird in Zeile 9 ein Task (ein

Objekt vom Typ `BuildClusterCaller`) erzeugt und an den `ExecutorCompletionService` übergeben.

In Zeile 19 wird die Anzahl der Tasks bestimmt und in Zeile 20 der Methode `blockingCompleteExecution` übergeben, die die Berechnung der Tasks anstößt. Diese benötigt die Anzahl, um festzustellen, wann die Arbeit der Tasks beendet ist. `blockingCompleteExecution` ist in Listing 3 zu sehen.

```
22 private static void blockingCompleteExecution(
23     ExecutorCompletionService<Void> ecs,
24     int taskCount,
25     Executor myExecutor
26     ) throws PathfinderException, InterruptedException{
27
28     ((ThreadPoolExecutor)myExecutor).shutdown();
29     for (int i = 0; i < taskCount; ++i){
30         try {
31             ecs.take().get();
32         } catch (ExecutionException e) {
33             if (e.getCause() instanceof
34                 PathfinderException){
35
36                 throw (PathfinderException)e.getCause();
37
38             } else {
39                 throw new PathfinderException(
40                     "got error which should never happen"
41                 );
42             }
43         }
44     }
45 }
```

Listing 3: Bearbeiten der Tasks und deren Rückgabewerten

In Zeile 27 wird der `Executor` dazu aufgefordert alle vorhandenen Tasks ausführen zu lassen. Außerdem nimmt er nun keine weiteren Tasks mehr an. Anschließend (Zeile 28) wird über die vorhandenen Tasks iteriert.

In Zeile 63 passieren zwei Dinge: Durch `ecs.take()` wird ein `Future` von irgendeinem der Tasks angefordert, das dem `ExecutorCompletionService` übergeben worden ist. `Future.get()` fordert den Rückgabewert des Tasks an (in diesem Fall ein Void-Object, weil es sich um `Callable<Void>`-Tasks handelt) und wirft eine Exception, falls diese bei der Taskausführung aufgetreten ist. Die Behandlung überlassen wir der Mainloop und daher werfen wir die Exception weiter (s.o.).

Grundsätzlich gibt es dieser Stelle zwei mögliche Kategorien von Exceptions, die die Methode `get()` werfen kann:

- Eine InterruptedException wird geworfen, wenn der Task unterbrochen

oder abgebrochen wurde und dürfte hier nicht auftreten, da Interruption nicht verwendet wird.

- ExecutionException - Fehler beim Ausführen innerhalb des Tasks. Diese können durch Programmierfehler oder z.B. nicht (ausreichend) vorhandene Systemressourcen enstehen.

Die Methode `get()` ist blockierend, daher wird `blockingCompleteExecution()` erst dann verlassen, wenn alle Tasks abgearbeitet sind. An dieser Stelle wird dadurch synchronisiert, das der Programmablauf nicht fortgestetzt wird, solange nicht alle Tasks bearbeitet wurden. Diese Art der Synchronisation wird auch Synchronisationsbarriere oder Barriere genannt.

Diese Barriere ist notwendig, damit sichergestellt werden kann, dass das Erzeugen der Cluster beendet ist, bevor ein Eingangsknoten erzeugt wird. Dies wiederum ist notwendig, weil die Eingangsknoten in eine Datenstruktur eingetragen werden, die sich in den Clusterobjekten befindet. Andernfalls würde es zu einem Fehler während des Programmablaufs führen, wenn der Algorithmus versuchen würde Knoten in einem Clusterobjekt anzulegen, das noch nicht vorhanden ist.

9.1.2 Das Erzeugen oder Bestimmen der Eingangsknoten

Die Implementierung jedes einzelnen in Abschnitt 8.2 vorgestellten Schrittes erfolgt analog zur Clustererzeugung.

9.1.3 Das Verbinden der Eingangsknoten innerhalb von Clustern

Dieser Schritt erfolgt analog zur Clustererzeugung.

9.2 Pfadsuche und Integration des Pathfinders

Die Integration des Pathfindings in jMMORTS konnte leider nicht erfolgen, da jMMORTS zum Zeitpunkt meiner Bearbeitung noch nicht weit genug entwickelt war. U.a. wurde die KI entfernt und sollte überarbeitet werden. In die KI hätte der Pathfinder integriert werden sollen und meine Diplomarbeit bezog sich auf das Entwickeln eines Pathfinders und nicht auf die Entwicklung einer gesamten KI. Ich konnte mich jedoch an vorhandene Schnittstellen anpassen oder diese zusammen mit Dennis Keßler, einem anderen Entwickler an diesem Projekt, soweit verändern, dass sich der HPA*-Pathfinder leicht integrieren lassen sollte. Das Pathfinding konnte daher nur mit Hilfe eines Testprogrammes ausgeführt werden und die berechneten Bewegungen wurden auf der Kommandozeile ausgegeben, daher waren leider keine Bewegungen in der jMMORTS GUI zu sehen.

Dieses Kapitel beschreibt die Schnittstellen und einen Teil des Designs bzw. der Arbeitweise, des parallelen HPA*-Algortithmus.

Die Schnittstelle der Pathfinding-Implementierung für jMMORTS befindet sich in der Klasse `HPAStarMapAndMovementImpl`. Um die Vorverarbeitung des

Hierarchischen Pathfinders durchzuführen, muss zum einen der Konstruktor dieser Klasse aufgerufen werden und zum anderen die Methode `init()`, die alle für die Vorverarbeitung nötigen Methoden aufruft. An diese Klasse kann außerdem eine Anfrage zur Berechnung eines Pfades gestellt werden (durch Aufruf der Methode `getPath(...)`). Diese Anfrage führt jedoch nicht direkt zu einer Pfadsuche, sondern die Anfrage wird in der Klasse `HPAStarMapAndMovementImpl` abgespeichert.

Die Methode `tick` der Klasse `HPAStarMapAndMovementImpl` soll einmal pro Mainloopdurchlauf aufgerufen werden und stößt dann die Bearbeitung der Sucanfragen an. Die Vorteile der iterativen Vorgehensweise können durch dieses Verfahren weiter genutzt werden (s.u.). Außerdem soll sich diese Art der Bearbeitung laut dem Betreuer dieser Diplomarbeit und Leiter des Projekts jMMORTS Björn Knafla mit der "Ahead Simulation" (einer Berechnung dessen, was in der Zukunft geschehen wird) von jMMORTS gut vereinbaren lassen, was nicht bei allen Ideen für die Integration der Fall war.

Gibt eine Einheit eine identische Suchanfrage zwei mal hintereinander ab, so wird beim Aufruf von `tick` nicht mehr nach einem komplett neuen Pfad gesucht, sondern es wird der zuletzt berechnete Pfad gewählt und dort iterativ nach den nächsten detaillierten Wegpunkten gesucht. Diese identischen Anfragen können genutzt werden, um `HPAStarMapAndMovementImpl` mitzuteilen, dass eine Einheit weitere Wegpunkte auf ihrem Pfad benötigt.

Würden diese identischen Anfragen nicht verwendet werden und würde `HPAStarMapAndMovement Impl` auch nicht auf anderem Weg mitgeteilt werden, ob weitere Wegpunkte berechnet werden müssen, so müsste bei jedem `tick` für jede Einheit ein detaillierterer Pfad berechnet werden. Dabei ist es unerheblich ob die Wegpunkte benötigt werden oder nicht. `HPAStarMapAndMovemtImpl` müßte immer davon ausgehen, dass neue Punkte benötigt werden. Wäre dies nicht der Fall, könnte es geschehen, dass Einheiten vor erreichen des Ziels stehen bleiben, da die nächsten Wegpunkte noch nicht berechnet wurden.

Eine Berechnung für jede Einheit bei jedem `tick` würde die in Kapitel 6 beschriebenen Vorteile der iterativen Vorgehensweise zu nichte machen. Pfade für langsame Einheiten würden zu schnell detailliert berechnet werden. Damit befänden sich mehr Daten als nötig im Speicher und es würden wieder unnötige, vollständige Pfadberechnungen für Einheiten durchgeführt werden, deren Ziel sich auf dem Weg doch ändert. Die Überprüfung, ob eine Suchanfrage zum zweiten Mal hintereinander gestellt wurde, ermöglicht es auf einfache Art und Weise die iterative Vorgehensweise umzusetzen.

Für jede *neue* Pfadanfrage wird ein neuer Pathfinder erstellt, es sei denn die Pfadanfrage wurde schon einmal gestellt. Dies ist nötig, wwil der Pathfinder den zuletzt für diese Anfrage berechneten Weg und das Ziel des Weges speichert.

`HPAStarMapAndMovementImpl` kennt zwei Datenstrukturen, die zur Bearbeitung der Anfragen dienen. In `HashMap<LogicEntity, HPAStarPathfinder> searches` wird abgespeichert, welcher Pathfinder für welche Einheit (`LogicEntity` bzw. `Entity`) zuständig ist und in `ArrayList<HPAStarPathfinder> toDoList` wird gespeichert, ob der Pathfinder beim nächsten Aufruf von `tick` eine oben genannte Pfadberechnung durchführen soll.

9.2.1 getPath() im Detail

`getPath` benötigt als Parameter die Einheit, für die ein Pfad gesucht werden soll, sowie das gewünschte Ziel der Einheit.

`GetPath` unterscheidet drei Fälle:

1. Es gibt bereits einen Sucheintrag für die per Parameter übergebene Einheit und ihr Ziel ist immer noch dasselbe. D.h. es muss iterativ vorgegangen werden, um weitere Wegpunkte zu bestimmen und der Pathfinder, der für die Einheit zuständig ist, wird auf die `toDoList` gesetzt. Diesen Fall behandelt Zeile 2-10 in Listing 4.

2. Es gibt einen Sucheintrag für die Einheit, aber dieser hat ein anderes Ziel als das per Parameter übergebene. `kill` (Zeile 13) löscht dann die Einheit in `searches`. Anschließend wird ein neuer Pathfinder erzeugt (durch `getPathfinder`) und die neue Suchanfrage in `searches` und `toDoList` eingetragen.

3. Im letzten Fall (Zeile 19-23) gibt es noch keine Anfrage für diese Entity und daher wird diese wie im zweiten Fall erzeugt.

```
0  public void getPath(LogicEntity entity, Vector3f target){
1      PathFinder finder;
2
3      if (searches.containsKey(entity)){
4          finder = searches.get(entity);
5
6          if (((HPAStarPathfinder) finder).getPath()
7              .targetCell.getCellCoordinates()
8              .equals(target)){
9
10             toDoList.add((HPAStarPathfinder)finder);
11
12         } else {
13             kill(entity);
14             finder = getPathfinder(entity, target);
15             searches.put(entity, finder);
16             toDoList.add((HPAStarPathfinder)finder);
17         }
18
19     } else {
20         finder = getPathfinder(entity, target);
21         searches.put(entity, finder);
22         toDoList.add((HPAStarPathfinder)finder);
23     }
24 }
```

Listing 4: Die Prozedur getPath(LogicEntity entity Vector3f target)

9.2.2 tick() im Detail

`tick` wird in Listing 5 dargestellt und soll eingesetzt werden, um einmal pro Mainloopdurchlauf die Pfadsuche aufzurufen. Zu Beginn wird, wie in der Vorverarbeitung schon gezeigt, ein `Executor` angelegt und mit einem Tasks pro Anfrage aus der `toDoList` gefüllt. Auch der Pathfinder wurde zur Parallelisierung als `Callable` implementiert. In diesem Fall hat das `Callable` einen Rückgabewert - ein Object vom Typ `HPAStarPathfinderCallResult`. Es umfasst eine Referenz auf die Entity, für die eine Pfadanfrage durchgeführt worden ist, sowie eine boolsche Variable (`gotNewPath`), die angibt, ob ein Pfad gefunden wurde.

```
0  public void tick(){
1      int nThreads = 2;
2      HPAStarPathfinderCallResult result;
3      int taskCount = toDoList.size();
4      ArrayList<Vector3f> pathPoints;
5      Vector3f target;
6      Executor myExecutor =
7          Executors.newFixedThreadPool(nThreads);
8      ExecutorCompletionService<HPAStarPathfinderCallResult>
9          ecs = new ExecutorCompletionService
10         <HPAStarPathfinderCallResult>(myExecutor);
11
12     for (PathFinder finder : toDoList){
13         ecs.submit((HPAStarPathfinder)finder);
14     }
15
16     ((ThreadPoolExecutor)myExecutor).shutdown();
17     for (int i = 0; i < taskCount; ++i){
18         try {
19             result = ecs.take().get();
20
21             if(result.gotNewPath){
22                 pathPoints = ((HPAStarPathfinder)searches
23                     .get(result.entity))
24                     .getPath().getCellVectorList();
25                 target = ((HPAStarPathfinder)searches
26                     .get(result.entity))
27                     .getPath().target
28                     .getCellCoordinates();
29                 result.entity.handleEvent(
30                     new PathReadyEvent(pathPoints, target)
31                     );
32
33             } else {
34                 kill(result.entity);
```

```
35                  result.entity.handleEvent(
36                      new PathReadyEvent(null, null)
37                  );
38              }
39          } catch (InterruptedException e){
40              e.printStackTrace();
41          } catch (ExecutionException e){
42              e.printStackTrace();
43              if (e.getCause() instanceof
44                  IntelligentPathfinderException){
45                  LogicEntity entity =
46                      ((IntelligentPathfinderException)
47                      e.getCause()).entity;
48                  kill(entity);
49                  entity.handleEvent(
50                      new PathReadyEvent(null, null)
51                  );
52              } else {
53                  // this can never happen
54              }
55          }
56      }
57      toDoList.clear();
58  }
```

Listing 5: Die Prozedur tick()

Nach der Erzeugung der Tasks wird wie in der Vorverarbeitung der `Executor` aufgefordert die Arbeit zu beenden. Anschließend werden in einer Schleife die Ergebnisse der einzelnen Tasks bearbeitet.

Zeile 21 enthält eine Fallunterscheidung:

1. Ein Pfad wurde gefunden. Die Wegpunkte und das Ziel werden bestimmt und es wird ein neues `PathReadyEvent` erzeugt, das die Wegpunkte und das Ziel enthält. Die Methode `handleEvent` der `Entity` wird anschließend aufgerufen und bekommt das `PathReadyEvent` übergeben.

2. Es wurde kein Pfad gefunden (z.B. weil der Zielpunkt in unpassierbarem Gelände liegt). Die `Entitity`, für die ein Pfad berechnet werden sollte, wird dann durch `kill` aus `searches` gelöscht. Auch in diesem Fall bekommt die `Entity` ein Event und zwar ein Event, welches zwei mal `null` enthält. Dieses Event ist ein Signal an die `Entity`, dass der gewünschte Pfad nicht berechnet werden konnte.

Weiterhin können bei der Ausführung Exceptions auftreten, die behandelt werden müssen. Mit `InterruptedExceptions` wurde nicht gearbeitet, diese sollten daher nicht auftreten. Falls dies doch der Fall sein sollte wird die Exception

ausgegeben und weitergearbeitet (hier kann das Exceptionhandling je nach Bedarf von jMMORTS noch angepasst werden).

Tritt eine `ExecutionException` auf, so sollte es sich dabei immer um eine `IntelligentPathfinderException` handeln, da im Pathfinder alle Exceptions zunächst gefangen werden und in einer `IntelligentPathfinderException` gekapselt werden, die zusätzlich die Informationen enthält, bei der Ausführung der Pfadsuche welcher Entity die Exception aufgetreten ist. Ebenso wie bei einer Suchanfrage, bei der kein Pfad gefunden wurde, wird auch gehandelt, wenn eine `IntelligentPathfinderException` auftritt. Die Suchanfrage wird gelöscht und die `Entity` bekommt ein Event mit `null`-Parametern übergeben.

Am Ende von `tick` wird die `toDoList` geleert, damit im nächsten Durchlauf der Mainloop `HPAStarMapAndMovementImpl` wieder unterscheiden kann, für welche `Entitys` sie in diesem Durchlauf Pfadsuchen anstoßen soll.

10 Experimente

Die Messungen wurden auf einem dual-core dual-processor AMD Opteron 2 GHz, mit 2 GB RAM durchgeführt.

Die Karten die für jMMORTS momentan verwendet werden, sind Graustufenbilder. Bisher wurde definiert, dass absolut schwarze und absolut weiße Pixel nicht betretbare Felder darstellen und alle anderen betretbar sind. Weitere Festlegungen wurden noch nicht getroffen.

In den Experimenten wird der Begriff **Speedup** verwendet. Laut Definition ist dies die Laufzeit des sequentiellen Programmes geteilt durch die Laufzeit des parallelen Programmes mit einer festgelegten Anzahl Threads. Der Speedup gibt damit an, wieviel schneller das parallele Programm bei einer festgelegten Anzahl an Threads ist, als das sequentielle Programm. Ich habe das Wort Speedup verwendet, jedoch statt der Laufzeit des sequentiellen Programmes die Laufzeit des parallelen Programmes von einem Thread verwendet.

Gemessen wurde die sogenannte **WallClockTime**, d.h. die Zeit, die zwischen zwei Zeitpunkten der Programmausführung vergangen ist.

Die Absolutwerte, die in den nachstehenden Tabellen zu finden sind, wurden auf folgende Art und Weise bestimmt: Es wurden elf Messungen durchgeführt und zwar durch einen Programmstart.

Da zu beobachten war, dass die erste Messung sich zum Teil stark von den anderen unterschied, habe ich die erste Messung nie verwendet. Die Java Virtual Machine (die ein Programm ausführt) scheint bei der ersten Messung Dinge zu tun, die bei den weiteren nicht mehr nötig sind. Evtl. wird während des Durchlaufs der ersten Messung noch eine Optimierung durchgeführt.

Von den anderen zehn Messungen wurde immer der beste und der schlechteste Wert ignoriert und aus den verbleibenden acht Werten, wurde der Durchschnitt gebildet. Dieser Durchschnitt ist der in den folgenden Kapiteln angegebene Absolutwert.

Weil Speicher und CPUs des Testrechners nicht zu 100% zur Verfügung standen und die WallClockTime verwendet wurde, können Ergebnisse gemes-

sen werden, die z.B. auf Grund des Schedulings besonders gut oder besonders schlecht sind (die Zeit läuft weiter, auch wenn das Programm gerade nicht von den Prozessoren ausgeführt wird). Ein weiterer Grund für stark vom Durchschnitt abweichende Werte könnte der Java Garbadge-Collector sein. Daher habe ich den Durchschnitt aus mehreren Werten gebildet und den kleinsten und den größten Wert vorher entfernt.

Bei Messungen, bei denen mehrere Einheiten verwendet wurden, wurden deren Positionen und deren Ziele zufällig erzeugt, allerdings wurde in allen Messungen dieselbe Folge von Zufallszahlen verwendet.

Anmerkung: Die Absolutwerte sind nur innerhalb eines Abschnitts dieses Kapitels vergleichbar, da ich nicht immer die gleiche Heuristik verwendet habe. Ich habe nach einigen Messungen bessere Heuristiken gefunden - es wurde ausschließlich der Wert der (schlechtere) Heuristik mit einem konstanten Faktor multipliziert - und wollte die Messungen nicht wiederholen. Innerhalb einer Messreihe wurde selbstverständlich immer dieselbe Heuristik verwendet.

Experiment 1: Speedup Vorverarbeitung

Gesucht wurde der Speedup bei der Vorverarbeitung einer Karte mit unterschiedlich vielen CPUs. Diese Karte ist eine Wiese. Die Speedups bei anderen Karten sind sehr ähnlich (die Absolutwerte können jedoch höher oder niedriger sein) und daher werden keine Speedups für die Vorverarbeitung von anderen Karten aufgeführt. Die Ergebnisse der Messreihe für die Vorverarbeitung der Wiese befinden sich in Abbildung 18.

Speedup	2 CPUs	3 CPUs	4 CPUs	ms 1 CPU
Level 1 Clustererzeugung	0,67	0,9	0,95	18
Level 1 Knotenerzeugung	1,66	1,95	0,98	232
Level 1 Kantenerzeugung	1,53	1,79	1,73	18158
Level 2 Clustererzeugung	0,75	1,2	1	6
Level 2 Knotenbestimmung	1,64	1,68	1,57	74
Level 2 Kantenerzeugung	1,71	2,49	3,14	1211
Level 3 Clustererzeugung	4	2	2	4
Level 3 Knotenbestimmung	1,37	1,66	1,85	48
Level 3 Kantenerzeugung	1,73	2,78	3,52	3669
Level 4 Clustererzeugung	2	2	2	2
Level 4 Knotenbestimmung	1,58	1,65	1,65	38
Level 4 Kantenerzeugung	1,72	2,47	3,56	9161
Level 5 Clustererzeugung	1	1	1	1
Level 5 Knotenbestimmung	1,48	1,79	1,62	34
Level 5 Kantenerzeugung	1,8	2,76	3,5	30313

Abbildung 18: Die Tabelle gibt den Speedup für die in Kapitel 8.1 genannten Schritte für Level 1 bis 5 an. Die letzte Spalte enthält die Absolutwerte für die Berechnung durch eine CPU in Millisekunden.

Bei der Clustererzeugung sind sehr seltsame Werte zu sehen. Dies kann zum einen daran liegen, dass sich die Werte an der Grenze der Messgenauigkeit (von Javas Methode `System.currentTimeMillis()`) befinden, die bei einer Millisekunde liegt. Zum anderen daran, dass die Clustererzeugung - so wie auch die Werte zeigen - kaum Rechenaufwand erzeugt. Diese Zeiten werden wahrscheinlich stärker von den Speichervorgängen beeinflusst, als von den Rechenzeiten. Da die Zeiten sehr gering sind, ist die Parallelisierung kaum von Vorteil und im Falle von Level 1 sogar nachteilig. Ich würde beim Einsatz des Pathfinders an dieser Stelle auf die Parallelisierung verzichten.

Die Messwerte der Knotenerzeugung bzw. Knotenbestimmung sind nicht so extrem niedrig, dass hier von ungenauen Werten ausgegangen werden kann, aber die Speedups sind trotzdem sehr gering. In einem Fall der Knotenerzeugung auf Level 1, ist der Speedup bei 4 Threads kleiner 1. Dieser Schritt benötigt dort länger als bei einem Thread.

Die Knotenerzeugung auf Level 1 unterscheidet sich von der Knotenbestimmung auf anderen Leveln zum einen dadurch, dass hier die Karte benutzt wird, um die Knoten zu bestimmen. Zum anderen werden bei anderen Leveln die Knoten der Cluster in dem darunterliegenden Level verwendet. Weiterhin werden bei der Knotenerzeugung (Knoten-) Objekte angelegt, während bei der Knotenbestimmung nur Variablen, in welchen die Level der Knoten gespeichert werden, verändert werden. Der schlechte Speedup ist daher vermutlich auf Cacheeffekte zurückzuführen.

Weil auch dieser Teil parallelisiert nur wenig Vorteile bringt - sowohl der Speedup, als auch die Laufzeit sind sehr gering - kann darüber nachgedacht werden auf die Parallelisierung zu verzichten. Dies könnte davon abhängig gemacht werden, ob mehrere Teile von jMMORTS gleichzeitig initialisiert werden sollen, oder nicht.

Sowohl bei der Knotenerzeugung, als auch bei der Clustererzeugung, sind die Werte in jedem Fall so gering, dass es sich nicht lohnt zusätzliche Entwicklungszeit in eine bessere Beschleunigung zu investieren.

Die Kantenerzeugung dauert für jedes der Level erheblich länger, als die anderen beiden Schritte. Sie hat mit Ausnahme der Erzeugung für Level 1 einen sehr guten Speedup. Die Parallelisierung kann daher bei der Kantenerzeugung als erfolgreich angesehen werden. Weil es sich hierbei um die mit Abstand rechenintensivsten Schritte handelt, kann die Parallelisierung der Vorverarbeitung insgesamt als Erfolg betrachtet werden.

Experiment 2: Skalierung bei steigender Einheitenanzahl

Experiment 2 soll die Skalierung des Such-Algorithmus von unterschiedlich vielen Einheiten überprüfen. Die Einheiten, ihre Position und ihr Ziel wurden vor den Messungen festgelegt. Der Startpunkt einer Messung ist kurz vor dem ersten *tick()*. Es wurden 10 *ticks* durchgeführt und der Endpunkt der Messungen ist kurz nach dem 10ten *tick*.

Die Zahlen in Abbildung 19 zeigen, dass die benötigte Zeit mit dem gleichen Faktor steigt, wie die Anzahl der Einheiten. Es ist möglich, dass bei der Messung

Anzahl Einheiten	Suchdauer
50	1951
500	21056
5000	194642

Abbildung 19: Bedeutung der Anzahl der Einheiten für die Suchdauer.

für 500 Einheiten, die Zufallszahlenfolge für die Suche schwierigere Aufgaben erzeugt hat und es daher eine kleine Verschlechterung der Zeit gibt.

Experiment 3: Suchdauer bei unterschiedlich vielen Hierarchieebenen

Dieses Experiment soll untersuchen, welchen Vorteil es hat mehr Hierarchielevel zu erzeugen, um einen Pfad schneller zu suchen. Daher wird bei der Suche nur mit einer CPU gearbeitet und es wird ein Weg für einen bestimmten Start- und einen bestimmten Zielpunkt gesucht.

Die Karte, der Startpunkt und der Endpunkt des Weges auf einer Wiese sind in Abbildung 20 zu sehen, die Messergebnisse in Abbildung 21.

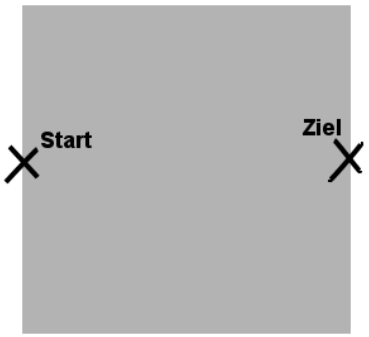


Abbildung 20: Eine Ebene mit einem Startpunkt auf der einen und einem Endpunkt auf der anderen Seite

Für eine Karte mit einer langen Mauer (zu sehen in Abbildung 22), werden die Messergebnisse in Abbildung 23 dargestellt.

Die Messungen zeigen sehr ähnliche Werte für beide Karten. Weitere Messungen haben gezeigt, dass die Ergebnisse sich stark unterscheiden können, wenn die Heuristik verändert wird (indem sie mit einem konstanten Faktor multipliziert wird). Suchanfragen auf diesen beiden Karten könnten dazu genutzt werden, eine gute Heuristik zu finden. Für eine Wiese scheint die in diesem Fall verwendete Heuristik sich nicht zu eignen, weil der Weg auf der Wiese bei niedrigen Hierarchieleveln mit Hilfe einer guten Heuristik schneller gefunden werden

Vorhandene Hierarchieebenen	Suchdauer in ms
1	433
2	527
3	470
4	367
5	255

Abbildung 21: Suchdauer einer Einheit für einen ersten groben Pfad auf einer Wiese.

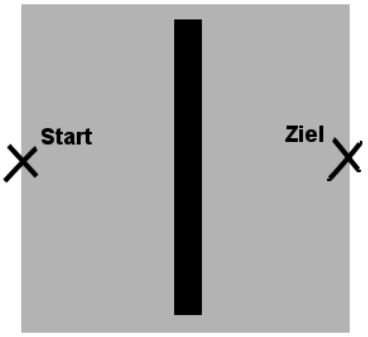


Abbildung 22: Eine Ebene auf der sich eine lange Mauer befindet, um die die Einheit herum möchte.

Vorhandene Hierarchieebenen	Suchdauer in ms
1	413
2	513
3	413
4	357
5	288

Abbildung 23: Suchdauer einer Einheit für den ersten groben Pfad auf einer Karte mit einer langen Mauer

sollte, als dies bei einer Karte mit einer großen Mauer der Fall wäre.

Es fällt auf, dass die Suchzeit bei zwei Hierarchieleveln im Vergleich zu einem steigt. Dies ist nicht wie erwartet. Eine Kurve bei der bei steigenden Leveln die Zeiten monoton fallen, würde den Erwartungen entsprechen und verdeutlichen, dass es sich lohnt, die Hierarchieebenen zu verwenden.

Der Versuch, den Verlauf der Kurve in einen monoton fallenden zu verändern, indem die Heuristik geändert wurde, führte zu dem Ergebnis, dass die Kurve sich veränderte. Allerdings dauerte die Suche bei zwei Hierarchieebenen stets am längsten.

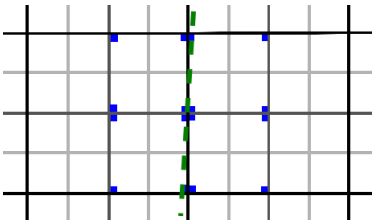
Daraufhin habe ich den Algorithmus zählen lassen, wie oft er Elemente auf die openList legt und über wie viele Kanten er iteriert. Dies habe ich mit verschiedenen Heuristiken durchgeführt.

Dabei konnte festgestellt werden, dass die Anzahl der Kanten, über die iteriert wurde, bei 2 Hierarchieebenen deutlich höher war, als bei einer Hierarchieebene. Außerdem wurden bei fast allen Heuristiken bei zwei Hierarchieebenen am häufigsten Knoten auf die openList gelegt.

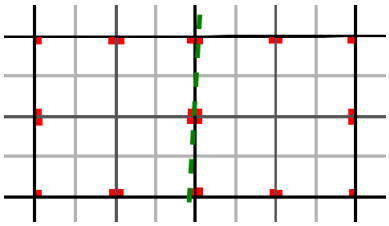
Dieses wurde mit einem Profiler genauer betrachtet. Der Profiler zeigte an, dass der parallele HPA*-Algorithmus in den Methoden `next()`, `hasNext()` und `nextEntry()` der Java-Iteratoren bei zwei Hierarchieebenen deutlich mehr Zeit verbraucht hat, als bei einer. Diese Methoden benötigten einen großen Teil der Zeit.

Da die openList noch nicht in Form einer sortierten Liste implementiert wurde (und in dieser daher sehr oft nach dem kleinsten Element gesucht werden muss), ist es nicht weiter verwunderlich, dass der Algorithmus länger benötigt, wenn er mehr Knoten untersuchen muss.

Warum beim Pathfinding, das zwei Hierarchieebenen benutzt, häufiger Knoten auf die openList gelegt werden, als bei einem Pathfinding mit nur einer Hierarchieebene zeigt Abbildung 24.



(a) In diesem Fall steht der Vorverarbeitung eine Hierarchieebene zur Verfügung.



(b) Hier existiert auch eine Level 2 Hierarchieebene.

Abbildung 24: Die Karte, die geclustert wurde ist eine Wiese. Die gestrichelte Linie ist ein Ausschnitt einer direkten Verbindung vom Start zum Ziel. Die Knoten (dargestellt als kleine Rechtecke) die sich unter der Linie befinden, gehören zu einem der kürzesten Wege und werden daher vermutlich expandiert. Durch diese Expandierung werden die blauen Level 1 bzw. die roten Level 2 Knoten auf die openList gelegt. Es ist zu erkennen, dass in der rechten Abbildung mehr Knoten sind, als in der linken.

Die Anzahl der Kanten, über die iteriert wird, steigt, weil ein Knoten mit einem höheren Level (auf einer Wiese) mit wesentlich mehr Kanten verbunden ist, als ein Knoten eines niedrigeren Levels. Einige Kanten werden zwar aufgrund dessen, dass sie nicht das gewünschte Level besitzen nicht näher betrachtet, doch scheinbar löst dies nicht vollständig das Problem, da die Methoden der Java-Iteratoren zu viel Zeit benötigen.

Um das Pathfinding zu beschleunigen sollte versucht werden, die Datenstruk-

tur so zu verändern, dass Kanten auf falschen Leveln gar nicht erst betrachtet werden, d.h. dass nicht einmal ein Aufruf einer Iteratormethode für diese Kanten durchgeführt wird.

Experiment 4: Speedup bei der Suche von mehreren Einheiten

In diesem Experiment wurde die Zeit gemessen, die für die Suche von 100 Einheiten auf einer Wiese von einer unterschiedlichen Threadanzahl benötigt wurde. Die Positionen und die Ziele der Einheiten wurden zufällig erzeugt (es wurden aus Vergleichbarkeitsgründen immer dieselben Zufallszahlen verwendet). Die Messungen wurden für 3 und 5 Hierarchieebenen durchgeführt. Die Speedupergebnisse sind sehr ähnlich und daher gebe ich in Abbildung 25 nur die Ergebnisse für 3 Hierarchieebenen an.

Anzahl der Threads	Speedup
2	1,92
3	2,84
4	3,81

Abbildung 25: Speedup bei der Pfadsuche für 100 Einheiten.

Dieser Speedup ist sehr gut und ich sehe die Parallelisierung der Suche als sehr erfolgreich an.

Experiment 5: Unterschiedliche Kartengrößen

Experiment 4 diente dazu die Auswirkungen von unterschiedlich großen Karten auf die Vorverarbeitung und die Suche zu überprüfen. Die verwendeten Karten enthielten keine Hindernisse.

Experiment 5a - Vergleich der Vorverarbeitungsdauer

In diesem Experiment wurde die Dauer der Vorverarbeitung für verschieden große Wiesen gemessen und verglichen. Die Ergebnisse sind in Abbildung 26 zu sehen.

Eine Karte in der zweiten Spalte besitzt die vierfache Fläche, wie die vorhergehende. Es konnte daher erwartet werden, dass die Zeiten sich ebenfalls von einer Karte zur nächst größeren etwa vervierfachen. Weil diese Vermutung zutrifft, oder noch günstigere Werte bei größeren Karten gemessen wurden, ist nicht davon auszugehen, dass die benötigte Rechenzeit bei großen Karten schnell steigt. Das könnte sich allerdings ändern, wenn die Karte nicht mehr in den Speicher passt. Dies ließ sich allerdings nicht testen, da Java größere Karten nicht laden wollte.

	250*250	500*500	1000*1000
Level 1 Clustererzeugung	5	7	18
Level 1 Knotenerzeugung	14	34	158
Level 1 Kantenerzeugung	430	1514	6975
Level 2 Clustererzeugung	2	3	6
Level 2 Knotenbestimmung	7	15	45
Level 2 Kantenerzeugung	29	101	404
Level 3 Clustererzeugung	1	1	2
Level 3 Knotenbestimmung	5	11	38
Level 3 Kantenerzeugung	68	243	1034

Abbildung 26: Benötigte Rechenzeit der Vorverarbeitungsschritte bei unterschiedlichen Kartengrößen.

Experiment 5b - Vergleich der Suchdauer

In diesem Experiment wurde eine Pfadsuche für eine Einheit durchgeführt, die immer auf der gleichen Position stand und stets das gleiche Ziel hatte. Wie erhofft waren die Suchzeiten gleich und der bei einer größeren Karte größere Hierarchiegraph sorgte für keinen messbaren Unterschied bei der Pfadsuche der Einheit.

Anmerkung: Es wurde darauf verzichtet Messungen der Suchdauer für unterschiedlich lange Wege durchzuführen, weil diese entscheidend von der verwendeten Heuristik abhängen und nicht nach einer optimalen Heuristik gesucht werden sollte.

11 Schlussbemerkungen

Dieses Kapitel enthält eine Zusammenfassung der Diplomarbeit und einen Ausblick auf die weitere Arbeit am Pathfinder.

11.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit, wurde das Spiel jMMORTS vorgestellt und seine Anforderungen an das Pathfinding beschrieben. Sie umfassen den Umgang mit vielen Einheiten auf einer großen Karte in Echtzeit und waren zu hoch, um sie mit dem Dijkstra- oder dem A*-Algorithmus erfüllen zu können. Deshalb wurde ein Hierarchischer Pathfinder basierend auf dem HPA*-Algorithmus implementiert und um ihn weiter zu beschleunigen, wurde er parallelisiert.

Dazu wurden zunächst Grundlagen der Parallelisierung erklärt und anschließend die Parallelisierung und die Schwierigkeiten dabei beschrieben. Zuletzt wurden die Implementierung und die Integration vorgestellt.

In Experimenten wurde nachgewiesen, dass die Parallelisierung erfolgreich war und der Einsatz von mehreren Hierarchieebenen zu einer Beschleunigung

führen kann. Desweiteren wurden auch Probleme des sequentiellen Algorithmus aufgedeckt, die einer Beschleunigung bei der Verwendung von zwei oder drei Hierarchieebenen entgegen stehen. Ideen zur Lösung dieses Problems bieten einen Ansatz für die zukünftige Arbeit am Pathfinder.

11.2 Ausblick

Dieser Abschnitt beschreibt mögliche Optimierungen des parallelen HPA*-Algorithmus und Ideen zum Umgang mit Einheiten, die vom Pathfinding unterschiedlich behandelt werden müssen, weil sie sich in Größe oder Beweglichkeit unterscheiden. Die Ideen wurden nicht implementiert.

11.2.1 Optimierungen

Folgende z.T. schon in den vorhergehenden Kapiteln genannten Punkte können zu einer schneller Pfadberechnung oder zu "besseren" Pfaden führen:

- Es könnten Experimente durchgeführt werden, um eine für jMMORTS besonders geeignete Heuristik zu finden.
- Die verwendeten Datenstrukturen für die openList und die closedList der Grundalgorithmen könnten verbessert werden.
- Wenn möglich sollte das in Experiment 3 beschriebene Problem, das eine Beschleunigung beim Einsatz von zwei oder drei Hierarchieebenen verhindert, beseitigt werden. Dazu sollte eine Datenstruktur gefunden werden, die weniger Aufrufe der Java-Iteratoren verursachen.
- Damit die Bewegungen natürlicher wirken, könnte ein Algorithmus zur Pfadglättung implementiert werden. Gegenwärtig wird ein Cluster durch die vom HPA*-Algorithmus gefundenen Wege nie diagonal verlassen, obwohl Einheiten sich per Definition diagonal bewegen können.

11.2.2 Behandlung unterschiedlich beweglicher Einheiten

In einem Computerspiel können Einheiten(-typen) unterschiedliche Kosten für die gleiche Kante haben (z.B. Flugzeug vs. Fahrzeug vs. Mensch in einem Wald).

Damit der HPA*-Algorithmus damit umgehen kann, sind Änderungen im Algorithmus erforderlich.

Ich habe folgende Ideen mit diesem Problem umzugehen, wobei die Vorverarbeitung in allen Fällen für jeden Einheitentyp durchgeführt werden muss:

1. Es wird ein Graph pro Einheitentyp erzeugt. Der Algorithmus bestimmt nach einer Pfadanfrage den zum Einheitentyp gehörenden Graphen und führt das Pathfinding darauf aus.
2. An jeder Kante wird nicht nur ein Kostenwert gespeichert, sondern es wird eine Tabelle von Kosten abgespeichert, so dass sich der HPA*-Algorithmus immer die Kosten des Einheitentyps heraussuchen kann.

In beiden Fällen können "ähnliche Einheitentypen" zusammengefasst werden. Gibt es z.B. verschiedene Flugzeugtypen, so können all diese vom Pathfinder gleich behandelt werden, denn sie können sich zwar in der Geschwindigkeit unterscheiden, werden jedoch durch Gelände gleichermaßen nicht behindert. In diesem Fall führt die Pfadberechnung zum gleichen Ergebnis, da esw sich nicht ändert, wenn alle Kosten mit einem konstanten Wert (der Geschwindigkeit) multipliziert werden. Falls eine Einheit schneller oder langsamer ist als eine andere, aber sich sonst im Bewegungsverhalten nicht unterscheidet, dann ist der kürzeste Weg für beide Einheiten identisch und wird nur mit unterschiedlicher Geschwindigkeit zurückgelegt.

Welche dieser Ideen die bessere ist, sollte durch Tests überprüft werden, die ich nicht durchführen konnte, da jMMORTS bisher nur eine Einheitentyp unterstützt.

Bei Verwendung mehrere Rechner kann Variante 1 weiter optimiert werden, indem verschiedene Graphen auf verschiedenen Rechnern gespeichert und die Anfragen entsprechend zugewiesen werden. Ich vermute, dass dieses Verfahren recht effizient ist, wenn der Anteil, der Pfadanfragen für einen Einheitentyp (oder eine Zusammenstellung von Einheitentypen) im Verhältnis zu den gesamten Anfragen annähernd konstant ist.

11.2.3 Behandlung unterschiedlich großer Einheiten

Abbildung 27 zeigt, wie eine große Einheit auf der Karte dargestellt werden kann. In diesem Fall belegt die Einheit eine Fläche von 2 mal 2 Feldern. Die Position der Einheit wird, durch die Festlegung der benötigten Fläche für den Einheitentyp und ein definiertes Feld innerhalb der Fläche (in der Abbildung Positionsmarker genannt) gespeichert. Diese Art der Positionsspeicherung benötigt weniger Speicher, als eine Speicherung jedes Feldes auf dem die Einheit steht.

Wird diese Art der Speicherung benutzt, so kann der HPA*-Algorithmus das Pathfinding zunächst so durchführen, als würde die Einheit nur 1 Feld belegen und sich auf dem Feld des Positionsmarkers befinden. Der HPA*-Algorithmus muss allerdings zusätzlich das gesamte überquerte Gelände aller Felder überprüfen (in Abbildung 27 mit "B" markiert). Für den Fall, dass sich das Gelände der Felder unterscheidet, ist eine Vorschrift zur Berechnung der Kosten notwendig (z.B. stets die Kosten des schwierigsten Geländes).

Andere Teile müssen nicht geändert werden, mit Ausnahme der Aspekte, die in Abschnitt 11.2.2 genannt wurden. Der Weg kann für eine 2 mal 2 große Einheit ein anderer sein, als der für eine 1 Feld große Einheit und hat dementsprechend andere Kosten. Deshalb müssen auch hier Tabellen an die Kanten angefügt oder zusätzliche Graphen erstellt werden.

Alternativ könnte das Pathfinding für große Einheiten optimiert werden. Das bedeutet, die kleinsten Felder, die das Pathfinding betrachtet, sind die, auf welche die größten Einheiten im Spiel passen. Die Bewegungen innerhalb eines dieser Felder kann für kleinere Einheit durch Steering [9] berechnet werden.

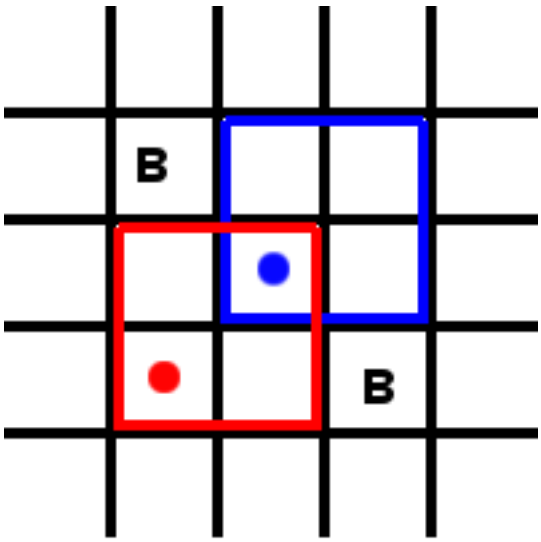


Abbildung 27: Schwarz dargestellt sind Begrenzungen der Felder. Der rote Rahmen wird von einer großen Einheit belegt, die sich zur Position des blauen Rahmens bewegen will. Mit “B” sind die Felder gekennzeichnet, die frei sein müssen, damit die Bewegung erfolgen kann. Die Punkte sind Positionsmarker.

Literatur

[1] Steve Rabin, "Ai Game Programming Wisdom I", 2002, Charles River Media Inc., Hingham, Massachussets

[2] Adi Botea, Martin Müller und Jonathan Schaeffer, "Near Optimal Hierarchical Pathfinding", 2004, Journal of Game Development, volume 1, issue 1, S.7-28, University of Alberta, Edmonton, Alberta

[3] M.R. Jansen and M. Buro, "HPA* Enhancements", AIIDE, 2007, Stanford USA

[4] Brian Goetz et al., "Java - Concurrency in Practice", 2006, Pearson Education Inc. / Addison Wesley, Stoughton, Massachusetts

[5] Amit's A* Pages

`http://theory.stanford.edu/~amitp/GameProgramming/`

[6] Ian Millington, "Artificial Intelligence For Games", 2006, Morgan Kaufmann, Kapitel 4 S.203-300

[7] Dijkstra-Algorithmus

`http://mandalex.manderby.com/d/dijkstra.php`

[8] Dijkstra-Algortihmus

`http://www.informatik.uni-leipzig.de/lehre/Heyer0001/AD2-Vorl7`

[9] Craig W. Reynolds, "Steering Behaviors For Autonomous Characters", 1999, The proceedings of the 1999 Game Developer Conference