Vergleich von Pathfinding-Algorithmen

Masterarbeit im Fachbereich Elektrotechnik/Informatik
der Universität Kassel

Abgegeben am: 23.10.2008

vorgelegt von
Heiko Waldschmidt

Betreuer:
Prof. Dr. Claudia Fohry

Prüfer:
Prof. Dr. Claudia Fohry
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 23.10.2008.

Inhaltsverzeichnis

1 Einleitung

In Computerspielen gibt es meist **Agenten**: Alles was Spieler oder die Teile der **Künstlichen Intelligenz(KI)**, die wie Spieler handeln, steuern können, wird als *Agent* bezeichnet. Diesen *Agenten* wird oft (von der *KI* oder einem Spieler) befohlen, sich an einen bestimmten Ort bzw. zu einem Ziel zu begeben. Die Ermittlung des Weges vom Standpunkt des *Agenten* zu dessen Ziel nennt man **Pathfinding** (deutsch: Wegsuche oder Pfadsuche). Das Ziel eines *Pathfinding-Verfahrens* ist einen **kürzesten Weg** zu ermitteln.

Pathfinding ist rechenintensiv und wird während eines Computerspiels häufig benötigt. Die Häufigkeit der *Pathfinding-Anfragen* steigt mit der Anzahl der *Agenten*, die in einem Computerspiel vorkommen. Einige Genres von Computerspielen, insbesondere *Massiv-Multiplayer-Online* Spiele, werden mit immer mehr *Agenten* gespielt. Unter anderem weil ein solches Spiel von immer mehr Spielern gemeinsam gespielt wird und häufig jeder (viele) *Agenten* besitzt. Weiterhin soll sich ein *Agent* in einer möglichst großen Umgebung bewegen können, da dies dem Spieler mehr Möglichkeiten und damit mehr Spielspaß bietet, was letztendlich zu höherem Gewinn für das Entwicklungsunternehmen führt. Ein große Umgebung führt allerdings auch zu einem größeren Suchraum beim *Pathfinding*, was die Suchdauer verlängert. Handelt es sich zusätzlich noch um ein Spiel was Echtzeitanforderungen erfüllen soll, z.B. um schnelle Reaktionen des Spielers zum Element des Spiels zu machen, benötigt dies eine schnelle Ermittlung von vielen *kürzesten Wegen*, deren Start und Ziel sich weit entfernt in einem großen Suchraum befinden können. Hinzu kommen weitere Anforderungen, wie z.B.:

- mit *Agenten* umgehen zu können, die sich auf unterschiedliche Art und Weise (Flugzeuge vs. Fahrzeuge etc.) bewegen und die sich auf unterschiedlichem Gelände unterschiedlich gut bewegen können.
- die Bewegungen einer Einheit möglichst *natürlich* aussehen zu lassen. Selten gibt es nur einen *kürzesten Weg*, welchen davon würde ein Mensch wählen?

Weniger entscheidend ist es hingegen wirklich einen *kürzesten Weg* zu bestimmen, geringfügig längere Wege sind ausreichend. Daher wird oft in den *Pathfinding-Algorithmen* auf Genauigkeit verzichtet, um das *Pathfinding* zu beschleunigen.

Diese Arbeit beschäftigt sich mit folgenden Fragestellungen:

1. Welche Anforderungen gibt es (zusätzlich zu den oben genannten) bei aktuellen Computerspielen?
2. Welche *Pathfinding-Verfahren* gibt es und wie arbeiten diese?
3. Welche *Pathfinding-Verfahren* werden in Spielen eingesetzt?
4. Wie kann ein *kürzester Weg* möglichst schnell gefunden werden?

Der Schwerpunkt liegt dabei auf der Frage: Wie kann ein (nahezu) *kürzester Weg* möglichst schnell gefunden werden?

Diesen Geschwindigkeits-Anforderungen werden **hierarchische Pathfinder** gerecht, die auf einem aussichtsreichen Ansatz basieren - im Gegensatz zu anderen Algorithmen (z.B. dem häufig eingesetzten *A*-Algorithmus*) - daher wurde ein *hierarchischer Pathfinder* implementiert. Die Suche wird im ersten Schritt weniger detailliert durchgeführt - d.h. es wird nicht jeder Punkt des Weges bestimmt, aber mehrere Punkte in zum Teil unterschiedlichen Abständen, die auf dem Weg liegen. In weiteren Schritten wird der Weg dann immer detaillierter bestimmt. Die Suche ist schnell, weil im ersten Schritt ein Suchraum verwendet wird, der wesentlich kleiner (weniger detailliert) ist, als der ursprüngliche und in den weiteren Suchschritten kein Weg mehr um große Hindernisse (z.B. Berge) gesucht werden muss. Dies geschieht bereits im ersten Schritt, in dem dies aufgrund des niedrigeren Detailgrads wenig Aufwand bedeutet.

Aufgrund der Aufgabenstellung erfolgte die Implementierung in Java. Eine genauere Beschreibung eines *hierarchischen Pathfinders* befindet sich in Kapitel 4.

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* (erfolgreich) mit **Java Threads** parallelisiert. Die Implementierung und die Parallelisierung erfolgte bereits während meiner Diplomarbeit, auf der diese Arbeit aufbaut. Ziel verschiedener Optimierungsversuche während dieser Masterarbeit war es, die Berechnungsdauer zu verkürzen:

- Der *hierarchische Pathfinder* benutzt einen **Hierarchiegraphen**. Dieser enthält Wege auf unterschiedlichen Detailstufen. Es gibt zwei Varianten, diese Detailstufen in einer Datenstruktur abzubilden - beide wurden implementiert und verglichen und stellten sich als gleich schnell heraus.

- Der **A*-Algorithmus** arbeitet mit Listen und benötigt immer wieder das kleinste Element einer dieser Listen. Um dieses zu erhalten ist entweder eine sortierte Datenstruktur nötig, die weiß welches Element das kleinste ist, oder es muss eine Suche nach dem kleinsten Element (in einer unsortierten Datenstruktur) durchgeführt werden. Verwendet man eine sortierte Datenstruktur, so muss jedes Element einsortiert werden, jedoch wird nur ein Teil der Elemente noch zur Suche benötigt. Dieses Problem lösen **teilsortierte Listen**: Diese bestehen aus einer sortierten und einer unsortierten Datenstruktur. Elemente, die vermutlich während der Wegsuche noch benötigt werden, werden in den sortierten Teil bzw. die sortierte Datenstruktur eingetragen und die anderen in den unsortierten Teil bzw. die unsortierte Datenstruktur. Ich habe mehrere *teilsortierte Listen* implementiert und mit einer der *teilsortierten Listen* konnte das *Pathfinding* deutlich beschleunigt werden.

- Als alternativer Grundalgorithmus (zum *A*-Algorithmus*) für den *hierarchischen Pathfinder* wurde der **Fringe Search**[6] implementiert. Messungen ergaben, dass der *A*-Algorithmus* in diesem Zusammenhang um ein Vielfaches schneller ist.

- Der Ablauf eines Computerspiels wird in **Frames** (dt.: Zeitabschnitte) eingeteilt. Nicht in jedem *Frame* werden gleich viele und gleich schwere *Pathfinding-Anfragen* gestellt, jedoch kann in einem Frame nur eine begrenzte Anzahl an Berechnungen durchgeführt werden. Es kann geschehen, dass in einem Frame mehr Berechnungen nötig werden als durchführbar sind und in einem vorhergehenden noch Zeit für Berechnungen gewesen wäre. Daher wurde ein **Amortisierungsalgorithmus** entwickelt, der Berechnungen, die in der Zukunft vermutlich benötigt werden, schon in einem vorhergehenden Frame berechnet, wenn während diesem noch Zeit für weitere Berechnungen vorhanden ist.

- Um die nötigen Berechnungen während dem Spielablauf zu verringern, wurde eine Vorverarbeitung entwickelt, die einen Teil der Berechnungen durchführt und die Ergebnisse in einer Datei ablegt. Diese können dann beim Programmstart geladen werden, um Berechnungen einzusparen.

- Es wurden Überlegungen zum Thema **Caching** durchgeführt. Das *Caching* verringert die Anzahl der nötigen Wegsuchen - jedoch nicht so stark wie eine Vorverarbeitung. Allerdings wird weniger Speicher als bei einer Vorverarbeitung benötigt.

Zur Bewertung der Optimierungsversuche wurde eine Experimentierumgebung geschaffen, mit deren Hilfe Wege visualisiert werden können und mit der z.B. die Verteilung von Start- und Zielpunkten auf einer Karte beeinflusst werden kann, um *Caching-E ekte* analysieren zu können.

Diese Arbeit beginnt mit der Vorstellung von Grundlagen von Computerspielen, der Parallelverarbeitung und des *Pathfinding* in Kapitel 2. Kapitel 3 behandelt die oben genannten Fragen eins bis drei. Dieses Kapitel gibt eine Übersicht über aktuelle *Pathfinding-Algorithmen* (Abschnitt 3.1) und analysiert deren Verwendung und Anforderungen an diese in aktuellen Computerspielen anhand von drei Beispielen (Abschnitt 3.2). In das *Pathfinding* in diesen Spielen wird Einblick sowohl durch Beschreibungen von Entwicklern, als auch durch die Sicht eines Spielers gegeben. Dabei stellt sich heraus, dass dort unter anderem *hierarchische Pathfinder* verwendet werden und dass eine der Anforderungen an *Pathfinding-Algorithmen* weiterhin ist: Möglichst viele Wegberechnungen mit möglichst wenig Rechenaufwand in kurzer Zeit durchzuführen.

In den folgenden Kapiteln geht es ausschließlich um die Frage, wie ein kürzester Weg am schnellsten gefunden werden kann. Kapitel 4 stellt die Ergebnisse meiner Diplomarbeit vor: Es beschreibt den **HPA*-Algorithmus** (einen *hierarchischen Pathfinding-Algorithmus*) dessen Implementierung und Parallelisierung, sowie die Ergebnisse der Experimente mit Implementierung und Parallelisierung.

Kapitel 5 betrachtet die oben genannten Optimierungsversuche des *hierarchischen Pathfinders*. Im Anschluss daran (Kapitel 6) wird die Experimentierumgebung vorgestellt, mit deren Hilfe die Optimierungsversuche bewertet wurden. Die zu diesem Zweck durchgeführten Experimente beschreibt Kapitel 7. In den Schlussbemerkungen (Kapitel 8) wird der Inhalt dieser Arbeit zusammengefasst und sowohl ein Ausblick auf weitere mögliche Arbeit als auch ein Fazit gegeben.

2 Grundlagen

Dieses Kapitel stellt Grundlagen von Computerspielen, der Parallelverarbeitung und des *Pathfinding* vor.

2.1 Computerspiele

Im Folgenden werden Elemente von Computerspielen beschrieben. Die in dieser Arbeit vorgestellten *Pathfinding-Verfahren* können in Computerspielen, die diese Elemente enthalten, angewendet werden:

Agent

Alles was Spieler oder die Teile der *Künstlichen Intelligenz*, die wie Spieler handeln, steuern können, wird in einem Computerspiel als *Agent* bezeichnet.

Künstliche Intelligenz (KI)

Die *Künstliche Intelligenz (KI)* muss das gesamte intelligente Verhalten, das in einem Spiel vorkommt, berechnen. Sie bestimmt das Verhalten sämtlicher *Agenten*, die nicht von einem Spieler gesteuert werden. 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 *Agenten* zum ausgewählten Ziel, dies macht die *KI*. Die Berechnung dieser Wege nennt man *Pathfinding*.

Die Mainloop

Das Programm der meisten Computerspiele 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 und vieles mehr initialisiert. Dann begibt sich das Spiel in eine Endlosschleife, **Mainloop** genannt, solange bis es beendet wird. In jedem Schleifendurchlauf wird ein *Frame* des Spiels durchgeführt. In einem *Frame* läuft alles ab, was im Spiel passiert: z.B. bekommt der Spieler Punkte, die Bildschirmanzeige wird aktualisiert und physikalische Vorgänge werden berechnet. Insbesondere führt die *KI Pathfinding* durch.

Die Karte

Die Pathfinding-Verfahren in dieser Masterarbeit beschränken sich auf 2D Karten, die durch gleichgroße Quadrate **gerastert** sind. Auf diesen Karten bewegen sich die *Agenten*. 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.

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. Diese Rückmeldung muss für den Spieler erkennbar in Verbindung mit der veranlassten Änderung stehen (z.B. könnte sich ein Fahrzeug, dem der Spieler ein neues Fahrziel gegeben hat, in Bewegung setzen). *Sofort* bedeutet in diesem Fall, dass der Spieler die Zeitspanne zwischen seinem Befehl und der Rückmeldung nicht mehr wahrnehmen kann.

Für die in dieser Arbeit vorgestellten *Pathfinding-Verfahren* bedeutet dies, dass im oben genannten Beispiel der Beginn des Weges so schnell berechnet werden muss, dass sich das Fahrzeug auch in die richtige Richtung in Bewegung setzen kann. *Pathfinding-Verfahren* für Spiele mit Echtzeitanforderungen müssen diese erfüllen, können aber zusätzlich in Spielen ohne Echtzeitanforderungen eingesetzt werden.

Spieler von RTS-Spielen benutzen statt dem Begriff *Agent* den Begriff **Einheit**. Diese Begriffe werden im Folgenden synonym verwenden.

In einem RTS-Spiel 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.

Anmerkung: Die hier entwickelten Verfahren können auch dann eingesetzt werden, wenn keine Echtzeitanforderungen existieren.

2.2 Parallelverarbeitung

Die Programmierung sollte für Rechner durchgeführt werden, in dem die CPUs **gemeinsamen Speicher** besitzen, da dies der Bauart der aktuellen Mehrkern-Prozessoren entspricht, die im Privatgebrauch zum Spielen Verwendung finden. Auf gemeinsamen Speicher kann jede CPU lesen und schreiben. Da die Aufgabenstellung es forderte die Implementierung in Java zu erstellen, liegt es nahe, dies mit *Threads* zu tun, die Zugriff auf diesen gemeinsamen Speicher haben.

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.

Zur Implementierung der *Tasks* wurden `Callables` verwendet. Ein *Task* wird in einem `Callable` durch die Methode `call` beschrieben. Für `call` kann ein beliebiger Rückgabewert definiert werden. Außerdem kann ein `Callable` Exceptions, die während der Taskausführung aufgetreten sind, nach außen weitergeben.

Das Ergebnis der Bearbeitung eines `Callables` erhält man über ein `Future` - dieses wird bei der Ausführung eines `Callables` erzeugt. Das `Future` kennt nicht nur den Rückgabewert, sobald ein Ergebnis vorliegt, sondern auch den Bearbeitungsstand (erzeugt, einem Pool hinzugefügt, gestartet, bearbeitet) eines *Tasks*. `Future.get()` gibt das Ergebnis zurück, sobald es vorliegt und blockiert den Aufrufer bis zu diesem Zeitpunkt. Außerdem wird eine `ExecutionException` geworfen, wenn der *Task* durch eine Exception beendet wurde, oder eine `InterruptedException`, wenn er während der Ausführung unterbrochen wurde.

Java bietet **Threadpools** an. *Threadpools* bestehen aus einer Menge von *Threads*. Existiert einen Menge von *Tasks*, so kann diese z.B. mit Hilfe eines `ExecutorService` den *Threads* im *Threadpool* zur Bearbeitung zugewiesen werden. Dies hat den Vorteil, dass der `ExecutorService` sich darum kümmert, dass diese *Tasks*

so auf die *Threads* verteilt werden, dass immer dann, wenn ein *Thread* keinen *Tasks* mehr zu bearbeiten hat, dieser einen neuen zugeteilt bekommt. Die Zuteilung wird solange fortgesetzt, wie *Tasks* vorhanden sind. Der Programmierer muss diese Zuteilung daher nicht selbst implementieren und die Zuteilung sorgt dafür, dass die Menge der *Tasks* schnell bearbeitet wird.

Es gibt mehrere *Threadpools*. Ich habe einen `FixedThreadPool` verwendet (Begründung siehe [21]), der eine feste Anzahl an *Threads* enthält.

Ein Beispiel für die Verwendung von `Future`, `ExecutorService` und `Callable` ist in Listing 2 zu sehen. Die Methode `invoke` fordert darin den `ExecutorService` auf, ein `Callable` auszuführen.

```
ExecutorService es = Executors.newFixedThreadPool(numberOfThreads);
Future<ReturnType> f = es.invoke(new Callable<ReturnTyp>{..});
try{
  f.get()
} catch (ExecutionException e){
  // do something
} catch (InterruptedException e){
  //  do something
}
```

Listing 2: Verwendung einen `Callables` mit einem `ExecutorService`

Siehe auch [23] und [24].

2.3 Pathfinding

Alle *Pathfinding-Algorithmen* haben gemeinsam, dass sie einen **Graphen** benutzen und darin **kürzeste Wege** suchen. Abschnitt 2.3.1 erläutert Grundlagen zur Graphentheorie und Abschnitt 2.3.2 erklärt, wie *Pathfinding* in Graphen durchgeführt werden kann.

2.3.1 Graphentheorie

Ein **Graph** ist ein Paar (V, E) endlicher Mengen mit $V \cap E = \emptyset$. Dabei bezeichnet V die Menge der im Graph enthaltenen **Knoten** (engl. vertex) und E die Menge der **Kanten** (engl. edge). Jede *Kante*($K1$, $K2$) verbindet zwei *Knoten* $K1$ und $K2$. *Kanten* von einem *Knoten* zu sich selbst schließe ich in dieser Arbeit aus.
Ich habe ungerichtete, gewichtete Graphen für das *Pathfinding* verwendet: In **ungerichteten Graphen** kann eine *Kante* zwischen *Knoten* a und *Knoten* b sowohl als Weg von A nach B, als auch von B nach A genutzt werden. Die ist die Richtung für die**Kosten** unerheblich. Bei einem **gewichteten Graphen** sind alle *Kanten* mit *Kosten* versehen. Es kann sich zum Beispiel um den Benzinverbrauch oder um die Fahrzeit einer Einheit entlang einer *Kante* handeln.

Ein **Weg** ist eine Folge von *Kanten*, die von einem **Startknoten** S zu einem **Zielknoten** G führen. Die **Weg-Kosten**(S,G) sind die Summe der *Kosten* aller *Kanten* eines Weges. **Kürzeste Wege** von S nach G sind die Wege, für die gilt, dass es keinen Weg von S nach G mit geringeren *Kosten* gibt.

Pathfinding-Algorithmen suchen **einen** *kürzesten Weg* und nicht **den** *kürzesten Weg*, falls es mehrere gleichlange Wege gibt. Um die Suche zu beschleunigen, wird in dieser Arbeit nicht nach einem *kürzesten Weg* gesucht, sondern nach einem Weg, der geringe Abweichungen von einem *kürzesten Weg* haben kann.

2.3.2 Grundalgorithmen des Pathfinding

Dijkstra- und *A*-Algorithmus* sind die **Grundalgorithmen** des *Pathfinding*. Sie bestimmen *kürzeste Wege* zwischen zwei *Knoten* in gewichteten Graphen und bilden die Grundlage für die später vorgestellten Algo-

rithmen. Die Graphen können sowohl gerichtet als auch ungerichtet sein. Zu beiden Algorithmen sollte man folgendes wissen:

- In Echtzeitstrategie-Spielen müssen Wege auf Karten gesucht werden, die *Grundalgorithmen* suchen aber in Graphen. Daher wird ein Graph erstellt, indem für jeden begehbaren Punkt auf der Karte ein *Knoten* erzeugt und dieser mit den *Knoten* der benachbarten (und begehbaren) Punkte verbunden wird.
  Die Suche nach einem *kürzesten Weg* auf der Karte bestimmt zuerst die zugehörigen (Start- und Ziel-) *Knoten* im Graphen und ermittelt anschließend darin den *kürzesten Weg*. Alles, was in dieser Arbeit auf der Karte erklärt wird, lässt sich daher auf den Graphen übertragen und umgekehrt.

- Sowohl der *A*-Algorithmus* 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 Werte müssen zu einem zusammengefasst werden.

- Die *Kosten* dürfen bei beiden Algorithmen nie negativ sein, da die Algorithmen sonst Zyklen (mit negativer Kostensumme) endlos wiederholen würden. In Abbildung 1 ist dies der Fall. Dort gibt es einen Zyklus mit negativer Kostensumme (-1). Sucht ein Pathfinding-Algorithmus dort entlang, so findet er zu jedem der beteiligten Punkte des Zyklus bei jedem Durchlauf durch den Zyklus einen kürzeren Weg.

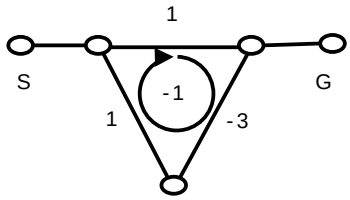


Abbildung 1: Ein Zyklus mit negativer Kostensumme im Graph.

Der Dijkstra-Algorithmus Der *Dijkstra-Algorithmus* [1] sucht *kürzeste Wege* von einem *Knoten* zu jedem anderen *Knoten*. Er kann jedoch abgebrochen werden, wenn ein Weg zu einem gewünschten Zielknoten gefunden wurde.

Das Vorgehen des *Dijkstra-Algorithmus* kann man sich wie folgt vorstellen (siehe auch Abbildung 2): Angenommen es liegt auf jedem Punkt des Spielfeldes eine elektrisch betriebene, analoge Stoppuhr und man 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, wird das Ausschütten von Wasser beendet.

Anschließend sieht man auf die Uhren, die auf den benachbarten Feldern vom Zielfeld liegen, sucht davon die Uhr mit geringsten Zeitspanne 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 geringsten Zeitspanne ausgewählt. 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** (priorisierende Warteschlange) implementiert. Auf diese werden Elemente gelegt, bestehend aus:

- *Knoten* k,

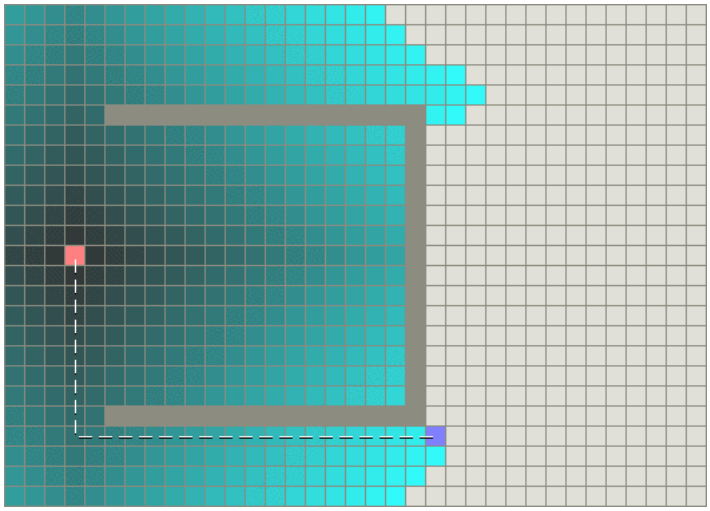


Abbildung 2: Zu sehen ist eine Karte mit einem Startpunkt (rot) und einem Zielpunkt (lila). Mit einer gestrichelten Linie wurde der *kürzeste Weg* eingezeichnet. Ein Hindernis wurde grau dargestellt. Der *Dijkstra-Algorithmus* betrachtet Punkte in der Reihenfolge, die durch die blaue Färbung dargestellt wird. Je dunkler das blau ist, desto früher wird der Punkt betrachtet. In diesem und den (in diesem Kapitel) folgenden Beispielen, wurde eine Diagonalbewegung zur Vereinfachung ausgeschlossen. [2].

- k', wobei k' der **Vorgängerknoten** von k auf dem bekannten *kürzesten Weg* vom S zu k ist
- und den $Weg\text{-}Kosten(S, k)$.

Die *Priority Queue* priorisiert nach den Weg-Kosten.

Der Algorithmus geht wie im Pseudocode in Listing 3 vor. Die *Priority Queue* wird hierin als **Open List** bezeichnet, da sich darauf die unbearbeiteten Elemente befinden. Die bearbeiteten Elemente werden auf die **Closed List** gelegt, die am Ende des Algorithmus die Elemente enthält, die zur Rekonstruktion des Weges benötigt werden. Die Namen *Open List* und *Closed List* stammen vom *A*-Algorithmus* (siehe 2.3.2). Anmerkung: Die mit (*) gekennzeichneten Zeilen sind dabei nur notwendig, wenn die Datenstruktur der *Open List* (z.B. eine *Java Priority Queue*) ein Element nicht automatisch neu einsortiert, obwohl sich seine Priorität ändert.

```
1  lege (S,null,0) auf OpenList
2  while (Ziel nicht gefunden (optional)
3    & Open List nicht leer) {
4
5    (K, K', Weg-Kosten(S,K)) = entferne Element E mit den geringsten Kosten
6      auf der Open List
7
8    if(K == G) break (optional)
9    foreach (Nachbar N von K) {
10     } if(OpenList enthält N) {
11       if((Wegkosten(S,K) + Kosten der Kante(K,N)) < bekannte Wegkosten(S,N)){
12         entferne Element zu N von OpenList (*)
13         Element zu N = (N, K , Weg-Kosten(S,N) über K) (update)
14         lege Element zu N auf OpenList (*)
15     } else {
```

```
16          NeuesElement EN = (N, K, Wegkosten(S,N) über K)
17          lege EN auf OpenList
18        }
19      }
20      Lege E auf die ClosedList
21    }
22    rekonstruiere den kürzesten Weg
```

Listing 3: Pseudocode des *Dijkstra-Algorithmus*

Der, im nächsten Abschnitt beschriebene, *A*-Algorithmus* geht zielgerichteter vor und kann ausschließlich eine kürzeste Wege-Suche zwischen zwei *Knoten* durchführen.

Der A*-Algorithmus Der *A*-Algorithmus*[2] betrachtet nicht nur die *Kosten* $g(K)$, die benötigt wurden, um zu einem *Knoten* K zu kommen, sondern er benutzt eine **Heuristik**, um zusätzlich die *Kosten* vom *aktuellen Knoten* bis zum Ziel ($h(K)$) abzuschätzen. Die **Gesamtkosten** $f(K) = g(K) + h(K)$ 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* arbeitet mit Hilfe der *Heuristik* zielgerichteter als der *Dijkstra-Algorithmus*.

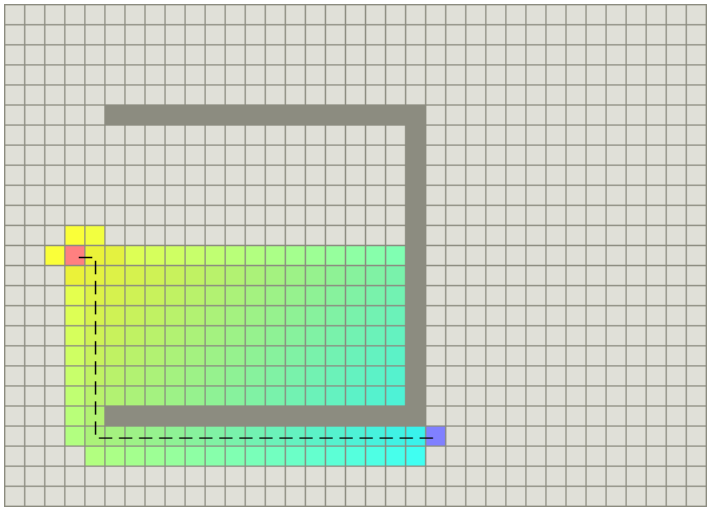


Abbildung 3: Zu sehen ist die Karte aus Abbildung 2 mit den gleichen Start- und Zielpunkten. Auch hier wurde der Weg mit einer gestrichelten Linie eingezeichnet. Die Punkte, die vom *A*-Algorithmus* näher betrachtet wurden, sind mit Farben von Gelb bis Türkis eingefärbt. Je höher der Gelbanteil der Farbe ist, desto eher wurde der Punkt betrachtet. In diesem Fall läuft der Algorithmus immer noch direkt (bzgl. Luftlinie) auf das Ziel zu, betrachtet jedoch dabei Punkte, die zwar nah am Ziel liegen, jedoch weit entfernt vom optimalen Weg sind. Im Vergleich zum *Dijkstra-Algorithmus* (siehe Abbildung 2) sind die Vorteile in diesem Beispiel deutlich geringer, als bei einer Karte ohne Hindernisse, aber immer noch beträchtlich. [2]

Der *A*-Algorithmus* arbeitet mit zwei Listen genannt *Open List* und *Closed List*. Beide Listen dienen dazu, Elemente zu speichern. Diese Elemente enthalten:

- *Knoten* K
- K', wobei K' der *Vorgängerknoten* von K auf dem Weg vom S zu K ist

- $g(K)$
- $f(K)$

Die *Open List* hat den selben Zweck wie die *Priority Queue* des Dijkstra-Algorithmus. Es wird daher das Element mit den geringsten *Gesamtkosten* auf der *Open List* als nächstes betrachtet und anschließend auf die *Closed List* gelegt. Die Elemente auf der *Closed List* werden am Ende des Algorithmus verwendet, um den Weg zu rekonstruieren (analog *Dijkstra-Algorithmus*).

Die Schätzung durch die *Heuristik* führt häufig dazu, dass zu einem *Knoten* K zunächst ein Weg gefunden wird, der nicht ein *kürzester Weg* ist - dies geschieht auch beim *Dijkstra-Algorithmus*. Beim *Dijkstra-Algorithmus* hätte der Weg zum *Knoten* K allerdings *Kosten*, die höher sind als die zu einem zweiten *Knoten* L, der sich auf dem *kürzesten Weg* zu K befindet. L würde damit vor K betrachtet und die *Kosten* von K würden korrigiert, bevor von K aus weiter zum Ziel gesucht wird. Es wird kein Weg von K aus gesucht, solange nicht sicher ist, dass der *kürzeste Weg* zu K bekannt ist.

Dies gilt nicht für den *A*-Algorithmus*: Aufgrund der Schätzung kann gelten: $f(K) < f(L)$. Ist dies der Fall, so wird der Weg von K in Richtung Ziel bestimmt und dann sind nicht nur $f(K)$ und $g(K)$ falsch, sondern auch (mindestens) alle $g(K')$ der Nachbarknoten K' von K, zu denen bisher noch kein Weg gefunden wurde. Gilt auch $f(K') < f(L)$ kann ein nicht minimaler Kostenwert zu weiteren nicht minimalen *Kosten* (bei Nachbarn von K') führen und deshalb benötigt der *A*-Algorithmus* ein Korrekturverfahren.

Der *A*-Algorithmus* inklusive Korrekturverfahren wird in Listing 4 dargestellt.

Es gibt drei Fälle, die der Algorithmus bei der Bearbeitung eines Nachbarknoten unterscheidet:

1. Es wurde noch kein Element erzeugt, das den *Knoten* enthält (Zeile 19).
2. Es gibt ein Element, das den *Knoten* enthält.
   (a) Es befindet sich auf der *Open List* (Zeile 12)
   (b) Es befindet sich auf der *Closed List* (Zeile 5)

Auch hier sind die mit (*) gekennzeichneten Zeilen nur notwendig, wenn dies die verwendete Datenstruktur erfordert (siehe auch Abschnitt 2.3.2).

```
1  while( Ziel nicht gefunden & Open List nicht leer) {
2    (K, K', g(K), f(K)) = entferne Element E
3       mit den geringsten Kosten aus der Open List;
4    foreach (Nachbar N von K) {
5      if(ClosedList enthält N) {
6        (N, N', g(N), f(N)) = Element EN der ClosedList mit (N=N,*,*,*)
7        if ((Weg-Kosten(S,K) + Kosten der Kante(K,N)) < g(N)) {
8          EN = (N, K, neues g(N), neues f(N))
9          entferne EN von ClosedList
10         füge EN zu OpenList hinzu
11       }
12     } else if ( OpenList enthält N){
13       (N, N', g(N), f(N)) = Element EN der OpenList mit (N=N,*,*,*)
14       if((Weg-Kosten(S,K) + Kosten der Kante(K,N)) < g(N)) {
15         entferne EN von OpenList (*)
16         EN = (N, Kante(K,N), g(S,N) über K, f(S,N) über K) (update)
17         lege Element EN auf OpenList (*)
18       }
19     } else {
20       Neues Element EN = (N, Kante(K,N), g(N) über K, f(N) über K)
```

```
21          lege EN auf OpenList
22        }
23      }
24      Füge E zur ClosedList hinzu
25  }
26  rekonstruiere den kürzesten Weg
```

Listing 4: Pseudocode des *A*-Algorithmus*

Fall 1 und Fall 2(a) sind die Fälle, die es auch beim *Dijkstra-Algorithmus* gibt und sie werden äquivalent (wie die Fälle beim *Dijkstra-Algorithmus*) behandelt. Im Fall 2(b) wird festgestellt, dass die bisher berechneten *Kosten* nicht die niedrigsten waren. Daraufhin wird das Element korrigiert, aus der *Closed List* gelöscht und, um eine erneute Bearbeitung herbeizuführen, zurück auf die *Open List* gelegt.

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

Eine *Heuristik*[13] ist **zulässig**, wenn die *Kosten*, die von einem Punkt bis zum Ziel geschätzt werden, nie größer sind als die **tatsächlichen Kosten**. Eine *Heuristik* h ist **monoton**, wenn

- sie zulässig ist und
- für jeden *Knoten* K und jeden Nachfolgerknoten K' gilt:

  (1) $$h(K) \leq h(K') + c(K, K')$$

  Wobei $c(K, K')$ die *tatsächlichen Kosten* von K nach K' bezeichnen.

Ein Beispiel für eine *monotone Heuristik* ist die euklidische Distanz. *Monotone Heuristiken* führen immer zu einem optimalen Weg (wenn es überhaupt einen Weg gibt) - Beweis siehe [21].

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.

Ist der Wert der Schätzung nur ein Bruchteil z.B. der Euklidischen Distanz (eine *monotone Heuristik*), aber nicht null, so gilt für das Beispiel in Abbildung 4 folgendes: $f(C) < f(A)$. Damit wird C vor A behandelt und man hat einen zielorientierten Algorithmus. Da der Wert der *Heuristik* jedoch sehr klein ist, kann $f(B) < f(C)$ sein. B wird dann als kleinstes Element gewählt, obwohl er wahrscheinlich nicht auf dem *kürzesten Weg* liegt. Daraus folgt, dass die Anzahl der untersuchten *Knoten* steigt, wenn man zu stark unterschätzt. Erhöht man hingegen die Schätzungen ausreichend, so gilt $f(B) > f(C)$, da $h(B) >> h(C)$ wird, und der Algorithmus arbeitet zielorientierter.

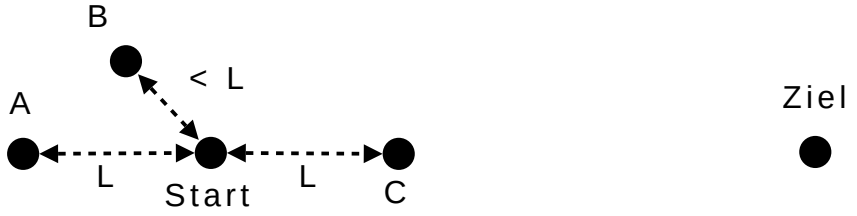


Abbildung 4: In dieser Abbildung sind 5 *Knoten* (*Start*, *Ziel*, A, B und C) dargestellt. Weiterhin sind drei Entfernungen abgebildet. Davon haben zwei die selben *Kosten* L und eine ist kleiner als L.

Das oben aufgeführte Beispiel, in dem $h(k) = 0$ für alle *Knoten* ist, verhält sich 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 [4].

2.3.3 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 ihren **ersten groben Weg** (Frankfurter Straße Kassel - Berlin Flughafen - Shanghai Flughafen). Nun benutzt A eine iteratives Suchverfahren und betrachtet den Weg von Kassel nach Berlin in jedem Schritt 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 etc..

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, so muss A 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 inklusive 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 findet heraus, zu welcher Kreuzung sie als erstes fahren muss.

Die niedrigste *Hierarchieebene*, auf der A schauen müsste, wenn sie diesen Weg in einem Computerspiel 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 im Rahmen meiner Diplomarbeit implementierte *Pathfinder* (der *HPA*-Algorithmus*) verwendet unterschiedlich große Regionen.

Die *Hierarchieebenen* werden durch **Hierarchielevel** gekennzeichnet. Auf *Level* 0 befindet sich die Karte eines Spiels (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 Punkt der Karte. 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 *Kanten* zwischen *Knoten* gilt dies jedoch nicht - z.B. existiert eine Flugverbindung nur dann auf *Level* 0, wenn die Flughäfen direkt nebeneinander liegen. Die *Hierarchieebenen* bilden gemeinsam den **Hierarchiegraphen**.

Der *hierarchische Pathfinder* benutzt den *A*-Algorithmus*, um im *Hierarchiegraphen* zu suchen, und kann somit die ersten Weg**punkte** viel schneller berechnen als ein reiner *A*- oder Dijkstra-Algorithmus* es tun könnte, diese haben nur *Hierarchielevel* 0 zur Verfügung (Ausnahme: Der *hierarchische Pathfinder* benutzt ebenfalls ausschließlich *Hierarchielevel* 0, z.B. weil Start und Ziel direkt nebeneinander liegen). Die Suche kann auf weniger detaillierten *Hierarchieleveln* schneller durchgeführt werden als auf *Hierarchielevel* 0, weil dort weniger *Knoten* existieren. Es müssen nur die ersten *Knoten* des Weges auf *Hierarchielevel* 0 bestimmt werden, damit Einheit A einen Teil der Bewegung durchführen kann. Der reine A*- oder *Dijkstra-Algorithmus* muss immer den kompletten Weg berechnen, weil er sonst nicht weiß, ob es überhaupt einen Weg gibt. Einheit A kann sich mit Hilfe dieser Methode in einem Computerspiel viel schneller in Bewegung setzen und den ersten Teil des Weges antreten.

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*.

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 gespart 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 *Dijkstra-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 Weg auf *Hierarchielevel* 0 berechnet.

Die beschriebenen Vorteile helfen, die in der Einleitung genannten Anforderungen (Echtzeitverhalten, viele Einheiten usw.) zu erfüllen.

3 Verwandte Arbeiten

Dieses Kapitel gibt einen Überblick über vorhandene *Pathfinding-Algorithmen* (Abschnitt 3.1) und Anforderungen von aktuellen Echtzeitstrategiespielen, sowie einen Einblick in die Implementierungen der verwendeten *Pathfinding-Verfahren* (Abschnitt 3.2).

3.1 Pathfinding-Algorithmen

Fast alle Algorithmen, die im Folgenden vorgestellt werden, sind Optimierungen des *A*-Algorithmus* oder gebrauchen diesen. In [2] werden weitere Varianten des *A*-Algorithmus* beschrieben, die für Spiele laut [2] ungeeignet sind und in dieser Arbeit deshalb nicht aufgeführt werden. Die hier vorgestellten Algorithmen konzentrieren sich entweder auf den Punkt *kürzere Berechnungsdauer* oder *dynamische Wegsuche* (d.h. Behandlung von einer sich ändernden Umgebung, während eine Einheit auf dem Weg zum Ziel ist).

Algorithmen mit Fokus auf die Berechnungsdauer

In optimierten Implementierungen des *A*-Algorithmus* werden für die *Open List* Datenstrukturen verwendet, die ein schnelleres *Pathfinding* ermöglichen. Die einfachste und langsamste Implementierung ist eine

verkettete Liste, etwas schneller ist eine *Priority Queue* und noch weiter beschleunigt werden kann die Implementierung durch die Verwendung von **binary Heaps**[2] und **teilsortierten Listen**[2]. *Teilsortierte Listen* wurden auch in dieser Arbeit implementiert (siehe Abschnitt 5.2).

In "Improved Heuristics for Optimal Pathfinding on Game Maps"[7] werden Methoden vorgestellt, die eine Vorverarbeitung der Karte verwenden, um die Ergebnisse der *Heuristik* des *A*-Algorithmus* zu verbessern. Die dort vorgestellte **Dead-End Heuristik** gibt immer ∞ zurück, wenn ein *Knoten* in einem Bereich der Karte liegt, durch den hindurch es keinen kürzesten Weg zum Ziel gibt (Sackgassen). Damit wird vermieden, dass der *A*-Algorithmus Knoten* in diesen Bereichen betrachtet (siehe Abbildung 5). Um diese Bereiche zu bestimmen, erfolgt eine Suche auf einem niedrigen Detailgrad, bevor die eigentliche Suche durchgeführt werden kann. Bei anderen *Heuristiken* werden als Parameter nur zwei Punkte benötigt, um die *Kosten* von einem zum anderen Punkt zu schätzen. Hier wird zusätzlich die Information benötigt, wie oder von wo aus man überhaupt zu dem Punkt gelangt ist, von dem aus zum Zielpunkt geschätzt werden soll. Die ist nötig, um überhaupt feststellen zu können, ob man sich durch einen Bereich der Karten hindurch bewegen würde, oder ob der Weg stattdessen in diesem Bereich beginnt.

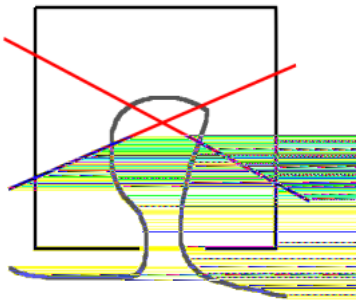


Abbildung 5: Ein Hindernis umschließt einen Bereich der Karte. Dieser Bereich hat nur einen Ausgang. Liegt weder *Start-* noch *Zielknoten* in diesem Bereich, so liegt darin auch nicht der *kürzeste Weg* zwischen diesen Punkten. [7]

Die **Gateway Heuristik** ist eine Alternative zur *Dead-End Heuristik*. In einer Vorverarbeitung werden Bereiche der Karte und Zugänge zu diesen Bereichen bestimmt. Des Weiteren werden die *Kosten* des Weges von jedem Zugang zu jedem anderen berechnet und gespeichert. Die bei der Suche verwendete *Heuristik* setzt sich dann aus einer Schätzung vom Start zum Zugang A von Bereich A', den (gespeicherten) *Kosten* von A zum Zugang B in Bereich B', und den *Kosten* von B zum Ziel zusammen. Da es mehrere Zugänge zu A' und B' gibt, wird die *Heuristik* für alle Kombinationen von A und B berechnet und anschließend das Minimum gewählt. Die Schätzung der *Gateway Heuristik* beruht damit auf Wissen über die Karte und soll damit im Durchschnitt näher an den *tatsächlichen Kosten* liegen als Schätzungen einfacherer *Heuristiken* (Manhattan-Distanz, Euklidischer Abstand). Durch beide *Heuristiken* sollen wesentlich weniger *Knoten* betrachtet werden, als durch die gewöhnlich verwendeten (z.B. die Manhattan-Distanz).

Roadmap-Verfahren[8] arbeiten wie das Verkehrssystem in Deutschland. Ein *Agent* wird auf einem möglichst kurzen Weg auf eine *Autobahn* geführt und bewegt sich auf dieser, bis er sich möglichst nah am Ziel befindet. Zuletzt wird der Weg von der *Autobahn* bis zum eigentlichen Ziel gesucht. Diese *Autobahnen* werden entweder vom Leveldesigner mit in die Karten eingetragen oder durch eine Vorverarbeitung automatisch erzeugt.

In [8] wird beschrieben, wie Wege für Gruppen von *Agenten* so gesucht werden können, dass die *Agenten* auf dem Weg sich ständig nah beieinander befinden und Kollisionen vermeiden.

Eine Variante des *A*-Algorithmus* ist der **iterative deepening A*-Algorithmus**[2][6] (IDA*). Im Gegensatz zum *A*-Algorithmus*, der eine *best-first* Suche durchführt, handelt es sich beim *IDA*-Algorithmus* um eine *depth-first* Suche. Die Suche in Richtung Ziel wird solange durchgeführt, bis dieses erreicht wurde oder ein Schwellwert (bzgl. der Wegkosten) überschritten worden ist - der Schwellwert wird mittels einer *Heuristik* bestimmt. Sobald der Schwellwert überschritten wird, wird die Suche an einem anderen *Knoten* fortgesetzt, zu dem ein Weg führt, der geringere *Kosten* hat als der Schwellwert. Von diesem *Knoten* ausgehend wird eine noch nicht untersuchte Fortsetzung des Weges betrachtet. Kann aufgrund des Schwellenwerts das Ziel nicht gefunden werden, wird der Schwellwert erhöht und der Algorithmus neu gestartet - alle Daten gehen dabei verloren. Der *IDA*-Algorithmus* hat keine *Open List* und spart damit Listenoperationen und Sortiervorgänge, aber führt Suchen mehrfach durch; er soll daher bei großen Karten langsamer sein als der *A*-Algorithmus*.

Fringe Search[6] ist ebenfalls eine *depth-first* Such-Algorithmus und stellt eine Weiterentwicklung des *IDA*-Algorithmus* dar. *Fringe Search* vermeidet es immer wieder vom Start aus zu suchen, indem er die Ergebnisse eines Durchlaufs abspeichert und diese im nächsten Durchlauf verwendet (siehe Abschnitt 5.3). Experimente ergaben laut[6], dass dies schneller als optimierte A*-Implementierungen ist. Ich konnte leider keine *hierarchischen Pathfinder* (siehe Abschnitt 2.3.3) finden, der *Fringe Search* anstelle des *A*-Algorithmus* benutzt. Daher habe ich eine solche Implemetierung selbst vorgenommen (siehe Abschnitt 5.3)

Dynamische Pathfinding-Algorithmen

Die Algorithmen **Dynamic A*-Algorithmus**[2] (D*) und **Lifelong Planning Algorithmus**[2] (LPA*) gehören zu den "dynamischen" Varianten des *A*-Algorithmus*. Sie können Wege teilweise planen ohne die *Kosten* jeder *Kante* zu kennen (s.u.) und können (und müssen) daher ständig ihre Ergebnisse anpassen. Insbesondere können Wege im **Nebel des Krieges** gesucht werden: Ein Punkt der Karte befindet sich im *Nebel des Krieges*, wenn ihn der Spieler noch nie zuvor gesehen hat - d.h. in diesen Spielen haben Einheiten Sichtweiten und die Karte muss erkundet werden. D* berechnet Wege zu Punkten, die im *Nebel des Krieges* liegen, indem er annimmt, dass sie passierbar sind und einen, für diesen Fall festgelegten, Kosten-Wert haben. Wird später herausgefunden, dass eine dieser Annahmen nicht richtig ist, wird der Weg korrigiert. *D** und *LPA** sollen für den Einsatz in Spielen mit vielen *Agenten* zu langsam sein[2] und sich eher zur Robotersteuerung eignen.

D* Lite[10] ist eine Weiterentwicklung des D*-Algorithmus.

Die Algorithmen **Local Repair A*-Algorithmus**[5] (LRA*), **Cooperativ A*-Algorithmus**[5] (CA*), **Hierarchical Cooperativ A*-Algorithmus**[5] (HCA*), sowie **Windowed Hierarchical Cooperativ A*-Algorithmus**[5] (WHCA*) behandeln Kollisionen zwischen *Agenten* bei der Wegberechnung. Der *LRA*-Algorithmus* korrigiert dabei lediglich den Weg sobald eine Kollision auftritt. Die Cooperativen Algorithmen besitzen zusätzliche Datenstrukturen und speichern darin ab, zu welchem Zeitpunkt welches Feld besetzt sein wird. Daher können sie dies bei der Berechnung eines Weges berücksichtigen, so dass es erst gar nicht zu Kollisionen kommt.

Adaptive A*[12] soll ebenfalls schneller sein als *D* Lite*. Dieser Algorithmus verwendet vorherige Suchergebnisse, um bei einer Anfrage bessere Werte für die *Heuristik* zu erzielen. Es erfolgt dabei eine Schätzung "im Dreieck": Die Ecken des Dreieck werden gebildet durch den Ziel*Knoten*, den *Knoten*, von dem aus zum Ziel geschätzt werden soll (*aktueller Knoten*) und als drittes von einem *Knoten*, dessen Entfernung zum *aktuellen Knoten* (durch vorherige Suchergebnisse) bekannt ist. Vom dritten *Knoten* aus werden anschließend die *Kosten* zum Ziel geschätzt und die *Kosten* vom *aktuellen Knoten* zu diesem *Knoten* werden hinzu addiert. Befindet sich der dritte *Knoten* an einer geeigneten Position (z.B. näher am Ziel als der *aktuelle Knoten*), ist diese Schätzung deutlich besser als eine Schätzung vom *aktuellen Knoten* zum Ziel.

*Adaptive A** kann nicht mit sinkenden *Kosten* umgehen (er überschätzt dann möglicherweise), dies kann jedoch **Generalized Adaptive A***[12], welcher eine Erweiterung von *Adaptive A** ist.

Fringe Saving A*[9] speichert nach einer Suche *Open* und *Closed List* und nutzt diese für eine Neuberechnung des Weges, wenn auf der Karte ein Hindernis hinzugefügt oder entfernt wurde.

Learning Real-Time A* (LRTA*) ist in der Variante **Priorisierender LRTA*** laut [11] schneller als *D* Lite*, da *D* Lite* komplette Wege berechnen muss und *LRTA** ähnlich wie *hierarchische Pathfinder* schnell die ersten Punkte berechnen kann. *LRTA** ist laut [11] ein dynamischer Wegsuch-Algorithmus, der lernen kann, welcher nächste Punkt sich wahrscheinlich auf dem *kürzesten Weg* befindet.

Wegverbesserung

Hierarchische Pathfinder berechnen oft Wege, die nicht natürlich wirken, weil ein Mensch diesen Weg nicht wählen würde, oder die deutlich länger sind als ein *kürzester Weg*. Daher werden Algorithmen eingesetzt, um die von *hierarchischen Pathfindern* berechneten Wege zu verbessern.

Algorithmen zur Rasterung von Linien aus der Computergrafik, wie z.B. der **Bresenham-Algorithmus** [25] können die Punkte auf einer quadratisch gerasterten Fläche (wie z.B. der Karte) berechnen, die auf einer Geraden liegen. Dies kann laut [4] dazu genutzt werden die Punkte einer geraden zwischen zwei beliebigen Punkten eines Weges zu berechnen. Anschließend kann geprüft werden, ob es sich bei dieser Folge von Punkten, um einen kürzeren Weg(-abschnitt) handelt - verglichen mit dem durch den *hierarchischen Pathfinder* bestimmten. Trifft dies zu, so tauscht man diesen Wegabschnitt aus. Um eine maximale *Wegverbesserung* durch einen solchen Algorithmus zu erzielen, muss er mehrfach - für jedes Paar von Punkten des Weges - durchgeführt werden.

3.2 Anwendungen in Spielen

In Abschnitt 3.1 kann man aufgrund der Algorithmen erkennen, dass die Entwickler sich hauptsächlich mit Geschwindigkeit und Dynamik befassen. Dynamik bezieht sich dabei auf:

- Kollisionen zwischen Einheiten
- Bewegungen von Gruppen
- Veränderungen der Karte

In aktuellen und sich in der Entwicklung befindenden Computerspielen lassen sich diese und weitere Probleme erkennen:

Starcraft 2

Im "Starcraft 2 Live Stage Video" [15] sind unterschiedlich große und unterschiedlich bewegliche Einheiten zu sehen: Auf der präsentierten Karte gibt es mehrere Ebenen, die sich auf unterschiedlichen Höhen befinden. Es wird eine Stelle der Karte gezeigt, an der diese Ebenen aneinander grenzen - die Grenze ist ein senkrechter Abhang. Viele Einheiten können sich an dieser Stelle nicht von einer Ebene zur anderen bewegen - allerdings gibt es eine menschengroße Einheit, die von einer zur anderen Ebene springen kann und eine um ein vielfaches größere Einheit, die diesen Unterschied so überwindet als sei es eine Treppenstufe. Der *Pathfinder* muss daher mit unterschiedlich großen und unterschiedlich beweglichen Einheiten umgehen können.

Medieval II: Total War

In "Medieval II: Total War"[16] gibt es viele Einheiten. Bereits bei seinem Vorgänger soll es möglich gewesen sein 24.000 Einheiten zu simulieren[17]. Dies konnte dadurch erreicht werden[18], dass

- Einheiten immer in Gruppen bewegt werden (alle Gruppenmitglieder dürfen sich nur begrenzt weit voneinander entfernen),
- nur für das Zentrum der Gruppe *Pathfinding* durchgeführt wird

- und eine Gruppe immer eine feste Formation besitzt, in der sie sich bewegt. Befindet sich eine Gruppe in einer festen Formation, so befinden sich die Einheiten in einer Anordnung zueinander, die sie (immer wenn dies möglich ist) einhalten.

Company of Heroes

Während der Entwicklung von "Company of Heroes" leitete die Entwickler das Motto: "nothing can't be destroyed" (deutsch: nichts ist unzerstörbar)[19]. Unter anderem kann folgendes geschehen:

- Panzer können Mauern zerstören, indem sie durch sie hindurch fahren.
- Durch Explosionen können sowohl sämtliche Gebäude, als auch nur Gebäudeteile und kleinere Dinge, wie Zäune, Mauern etc. zerstört werden.
- Der Untergrund kann verändert werden, z.B. durch Explosionskrater.

Zusätzlich können auch neue Dinge auf der Karte entstehen. Z.B. können Stacheldrahtzäune errichtet werden.

Diese Konstruktions- und Destruktionsvorgänge benötigen einen Pathfinding-Algorithmus, der dynamisch arbeiten kann. Für Einheiten wird ständig überprüft, ob es einen neuen *kürzesten Weg* gibt (weil ein Hindernis zerstört worden sein könnte).

Auch in "Company of Heroes" bewegen sich Einheiten in Gruppen, haben jedoch keine feste Formation. Die Gruppe wird dadurch zusammen gehalten, dass es in jeder Gruppe einen Anführer gibt und alle anderen Einheiten der Gruppe sich in seiner Nähe aufhalten müssen (dies wird auch "lockere Formation" genannt). Das *Pathfinding* berechnet dann nur einen Weg für den Anführer, der Zusammenhalt der Gruppe und die Bewegung der anderen Einheiten der Gruppe wird durch einen anderen Algorithmus (der möglichst weniger Rechenleistung benötigt) bestimmt.

"Company of Heroes" kennt unterschiedlich große Einheiten. Es gibt folgende Größen: 0,1,3,7. Die Anzahl der unterschiedlichen Größen wurde scheinbar auf einige wenige festgelegt, um das *Pathfinding* möglichst einfach zu halten. Bewegt sich in dem Spiel ein Squad von Soldaten, dann wird dieser so behandelt, dass er einen Weg bevorzugt, der für eine Einheit der Größe 7 geeignet wäre. Dieser Squad kann Wege, die für eine Einheit der Größe 1 geeignet wären, benutzen, beim *Pathfinding* werden diese nicht als unpassierbar betrachtet, sondern mit zusätzlichen *Kosten* belegt, so dass breitere Wege bevorzugt werden.

Die Kollisionsbehandlung lässt kleine Einheiten größeren aus dem Weg gehen, d.h. ein Feld auf dem sich Soldaten befinden, wird beim *Pathfinding* für einen Panzer nicht als Hindernis angesehen.

Zur Berechnung von Wegen für unterschiedlich große Einheiten gibt es für jede Größe eine Karte und dementsprechend einen extra Graphen.

In "Company of Heroes" sollen Bewegungen natürlich wirken. Insbesondere wird die Ausrichtung einer Einheit berücksichtigt, z.B. wendet ein Fahrzeug, wenn sich das Ziel weit hinter ihm befindet, und fährt dann nicht rückwärts.

Ändert sich das Ziel eines Fahrzeuges, so wird berechnet, ob der vorherige und der neu zu berechnende Weg nah beieinander liegen. Trifft dies nicht zu, wird ein Bremsweg für die Einheit berechnet und von dessen Ende ein Weg zum Ziel. Liegen die Wege stattdessen nah beieinander wird auf den Bremsweg verzichtet.

"Company of Heroes" verwendet *hierarchisches Pathfinding*. Die Vorverarbeitung benutzt einen Algorithmus, der zusammenhängende Bereiche ermittelt, welche die gleiche Schwierigkeit haben und fasst diese dann zu Sektoren zusammen. Auch Hindernisse bilden dabei Sektoren. Sektoren werden bei Änderungen der Karte neu berechnet.

Diese Sektoren beziehen damit die Beschaffenheit der Karte ein, während das von mir verwendete Vorgehen anders arbeitet: Es benutzt eine starre Größe für die *Cluster*, dies ist ein anderer Begriff für Sektoren(siehe Abschnitt 4.2). Ein Vorteil der Variante von "Companie of Heroes" gegenüber der von mir verwendeten ist, dass es einfacher zu bestimmen ist, ob eine Einheit einen Weg durch einen Sektor finden kann oder nicht -

handelt es sich z.B. um einen Sektor aus Feldern, die Teile eines Meeres sind, so ist klar, dass ein Schiff sich hindurch bewegen kann und ein Auto nicht. Ein Nachteil dieser Methode besteht darin, dass Sektoren sehr klein oder sehr groß werden können - z.B. wird für jedes noch so kleine Hindernis ein Sektor gebildet, wodurch sich auf einer Karte mit sehr vielen Hindernissen sehr viele Sektoren befinden können. Im schlimmsten Fall gibt es so viele Sektoren, wie es Felder gibt, dann würde das *Pathfinding* mit einem *hierarchischen Pathfinder* keinen Vorteil mehr bringen, sondern sogar zusätzlichen Aufwand im Vergleich zum *A*-Algorithmus* verursachen. Daher wird das *Pathfinding* auf jeder Karte getestet. Sollte der Rechenaufwand bei einer Karte zu hoch sein, wird sie anschließend von den Level-Designern geändert und die Anzahl der Sektoren reduziert.

Des Weiteren haben Wege in "Company of Heroes" höchstens Längen im Bereich von 30 bis 40 *Knoten*. Vermutlich ist das *Pathfinding* für Wege mit mehr *Knoten* zu langsam und wird daher vermieden.

4 Ergebnisse meiner Diplomarbeit

Während der Diplomarbeit wurde ein *hierarchischer Pathfinder*, der **Hierarchical Pathfinding A*-Algorithmus (HPA*-Algorithmus)**, in Java implementiert und parallelisiert. Außerdem wurden Experimente mit dem *Pathfinder* durchgeführt. Diese Themen werden direkt nach der Vorstellung einiger Implementierungsdetails des *A*-Algorithmus* behandelt.

4.1 Implementierungsdetails des A*-Algorithmus

Dieser Abschnitt beschreibt Implementierungsdetails des *A*-Algorithmus* und beginnt mit einer Vorstellung der, für die *Open List* und *Closed List* verwendeten, `HashMap`:

Eine `HashMap<key, value>`[23] ist eine unsortierte Datenstruktur, die eine beliebig große Menge von Schlüssel-Wert Paaren (englisch: key value pair) enthalten kann. Sowohl Schlüssel als auch Wert müssen Objekte sein. Der Typ von Schlüssel oder Wert kann dabei jeweils auf eine Klasse eingeschränkt werden.

Die `HashMap` kann nach dem zugehörigen Wert zu einem Schlüssel gefragt werden. Der Wert wird im durchschnittlichen Fall in konstanter Zeit zurückgegeben, da aus dem Schlüssel mit Hilfe eines Hashverfahrens eine Speicherstelle berechnet wird, an der der Wert abgelegt wird und wiedergefunden werden kann. Ein bestimmter Schlüssel kann daher nur einmal in einer `HashMap` vorkommen - Werte allerdings mehrfach. Soll ein Schlüssel-Wert Paar eingetragen werden und der Schlüssel existiert bereits in der `HashMap`, so wird der Wert des alten Paares überschrieben.

Die *Open List* und die *Closed List* sind in einer Variante der Implementierung beide vom Typ `HashMap-<Node, NodeRecordAStar>`. Eine Instanz von `Node` ist ein *Knoten* im *Hierarchiegraph*. `NodeRecordAStar` ist die Implementierung der in Abschnitt 2.3.2 vorgestellten Elemente der *Open List* und *Closed List* und hat die folgenden Attribute:

- `costSoFar` → entspricht $g(k)$
- `estimatedTotalCost` → entspricht $f(k)$
- einen *Knoten* des Typs `Node`
- eine *Kante* des Typs `Edge`

Wie man sehen kann wurde in der Implementierung eine Veränderung vorgenommen: Die tatsächlichen Elemente der *Open* und *Closed List* bildet das Tupel `<Node, NodeRecordAStar>`. In Abschnitt 2.3 wurde nur der zweite Teil (`NodeRecordAStar`) des Tupels erwähnt und als Element der *Open* und *Closed List* vorgestellt.

Der *Knoten*, der in der `HashMap` als Key eingetragen wird, ist äquivalent zum *Knoten* in dem zugehörigen `NodeRecordAStar`-Objekt des Tupels. Dadurch kann die `HashMap` genutzt werden, um schnell heraus zu

finden, ob es ein `NodeRecordAStar`-Objekt zu einem bestimmten *Knoten* gibt und man kann dieses Objekt schnell zurück bekommen.

Der Nachteil der `HashMap` ist, dass sie die Suche nach dem kleinsten Element nicht unterstützt und daher über die `HashMap` iteriert werden muss, um dies zu finden.

4.2 Der HPA*-Algorithmus

Abschnitt 2.3.3 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)[3] berechnet diese *Kanten* während einer Vorverarbeitung der gesamten Karte.

4.2.1 Idee

Der HP*A*-Algorithmus* erzeugt **Cluster**, die nahe beieinander liegende Punkte auf der Karte zusammenfassen. Diese *Cluster* können selbst wiederum zu größeren *Clustern* zusammengefasst werden, um eine höhere *Hierarchieebene* zu erzeugen. Die *Cluster* entsprechen den in Abschnitt 2.3.3 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. Dies ermöglicht weniger detaillierte Wege - von *Cluster* zu *Cluster* (genauer: Eingangsknoten zu Eingangsknoten) statt von Punkt zu Punkt - zu suchen.

4.2.2 Hierarchiegraph

Dieser Abschnitt beschreibt den *Hierarchiegraphen* und dessen Erzeugung.

Der *Hierarchiegraph* ist in der Implementierung in zwei Teile aufgeteilt. Es gibt einen Teilgraphen indem sich ausschließlich *Level* 0 befindet, und einen Teilgraph für alle anderen *Level*.

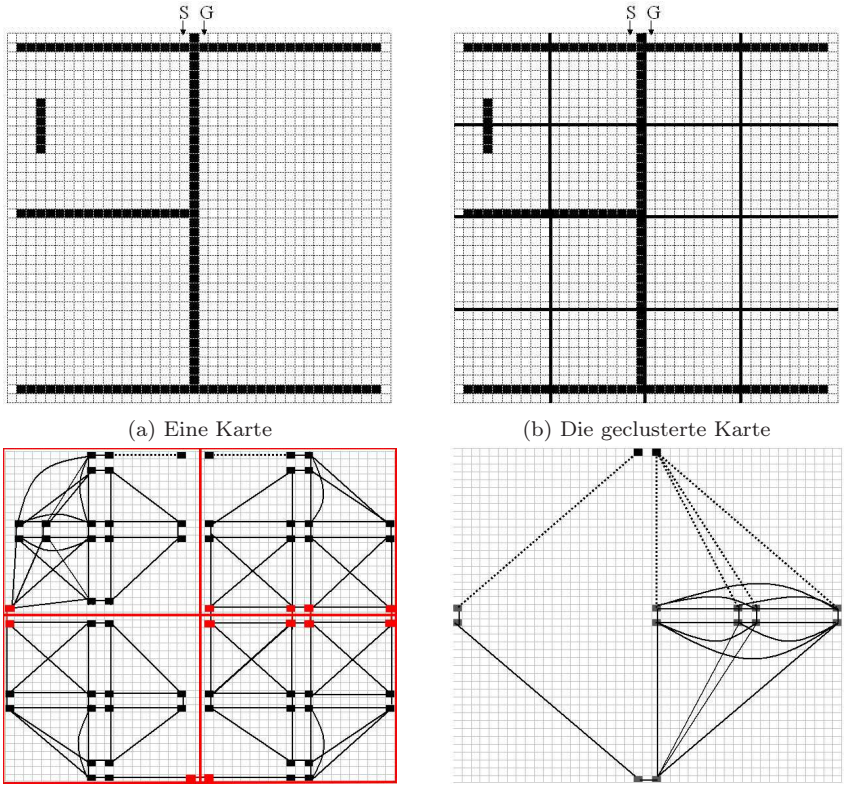
Anmerkung: Es wurde festgestellt, dass der Speicherverbrauch bei großen Karten zu hoch ist, wenn man für jeden Punkt der Karte ein Objekt im Speicher hält. Diese Objekte werden daher bei Bedarf erzeugt und haben nie Assoziationen zu Objekten anderer *Level* (Objekte sind z.B. notwendig, um den Weg zurück zu geben und im Spiel ermitteln zu können welche Einheit auf einem bestimmten Feld steht). Der Typ der *Knoten* auf *Level* 0 ist `Cell` und der *Kanten* auf *Level* 0 `Connection`.

In der Klasse `Clusters` befindet sich der Teil des *Hierarchiegraphen* für *Level* 1 und höher. Dieser Teil beinhaltet die in Abschnitt 4.2.1 genannten Eingangsknoten (Klasse `Node`) und die Kantenobjekte (Klasse `Edge`) die diese verbinden. `Nodes` und `Edges` werden während einer Vorverarbeitung erzeugt, die grob in Abbildung 6 dargestellt wird. Eine detailliertere Beschreibung ist in [21] zu finden.

Jede `Node` besitzt ein `HashSet<Edge>` - dabei handelt es sich um eine Menge, die `Edges` enthalten kann. In jeder `HashSet<Edge>` eines `Nodes` N werden alle `Edges` gespeichert, an deren Ende sich N befindet. Außerdem werden in einem `Edge` die `Nodes` an beiden Enden gespeichert.

Alle `Nodes` können sich auf mehreren *Level* im *Hierarchiegraphen* befinden. Werden mehrere *Level* x *Cluster* zu einem *Level* $x+1$ *Cluster* zusammengefasst, so wird ein Teil der `Nodes` des *Level* x *Clusters* zusätzlich zu `Nodes` des *Level* $x+1$ *Clusters*. Dies wird im *Hierarchiegraphen* dadurch abgebildet, dass in jedem `Node` das *Level* des höchsten *Clusters* gespeichert wird, zu dem der `Node` gehört - d.h. ein `Node` gehört auch immer zu allen kleineren *Leveln*. Somit wird gespeichert, ob es sich bei einer `Node` um eine Straßenkreuzung oder um einen Flughafen (evtl. mit Straßenkreuzung) handelt.

Auch `Edges` haben *Level*: Beispielsweise könnten zwei Flughäfen auch über die Autobahn verbunden sein. Dabei können die Flug- und die Autobahnverbindung die gleichen *Kosten* haben. Ähnlich verhält es sich in der Implementierung: Sind zwei `Nodes` des selben *Level* x *Clusters* zugleich `Nodes` eines *Level* $x+1$ *Clusters*, so gibt es eine Verbindung zwischen ihnen auf *Level* x und eine auf *Level* $x+1$. Dies kann man auch in Abbildung 6 sehen. Die *Level* 2 *Knoten* in Abbildung 6 (c), die sich in der rechten Hälfte auf halber



(a) Eine Karte

(b) Die geclusterte Karte

(c) Der *Level* 1 Graph der Karte, während der Erzeugung von *Level* 2

(d) Der *Level* 2 Graph

Abbildung 6: Diese Abbildung zeigt eine Karte (a) und mehrere Schritte der Vorverarbeitung. In (b) wurde die Karte bereits in 10*10 Felder große *Level* 1 *Cluster* zerlegt. In (c) wurden für diese *Cluster* bereits `Nodes` und `Edges` erzeugt. Außerdem wurden die *Level* 2 *Cluster* als rote Quadrate eingezeichnet. Sie bestehen aus vier *Level* 1 *Clustern*. Einige `Nodes` wurden rot markiert - sie gehören auch zu *Level* 2. In (d) wurden diese `Nodes` verbunden - es wird dort ausschließlich *Level* 2 des *Hierarchiegraphen* dargestellt.

Höhe befinden, sind dort auf *Level* 1 mit *Kanten* verbunden, die sich zusätzlich auf *Level* 2 befinden, wie in Abbildung 6 (d) zu sehen ist.

In jedem `Edge` wird gespeichert, zu welchen *Leveln* es gehört. Dafür hat es die beiden Attribute `Maximumlevel` und `Minimumlevel`.

4.2.3 Pathfinding auf Level 1 und höher

Dieser Abschnitt erklärt das *Pathfinding* im *Hierarchiegraph* zwischen zwei `Nodes`. Dafür wird der *A*-Algorithmus* verwendet.

Der *A*-Algorithmus* muss jedes Mal, wenn er einen *Knoten*, aus der *Open List* entfernt, um den weiteren Weg zu bestimmen, prüfen, *Kanten* welchen *Levels* überhaupt genutzt werden dürfen. Würde er dies nicht tun, so könnte z.B. ein berechneter Weg ausschließlich aus *Level* 1 *Kanten* bestehen, obwohl geeignete *Level* 2 *Kanten* im *Hierarchiegraphen* vorhanden sind und ein möglichst grober Weg angefordert wurde.

Zum einen muss also ein möglichst hohes *Level* verwendet werden (siehe Abbildung 7), 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 8). Daher gibt es ein `Maximum-` und ein `Minimumlevel`, sowohl

als Schranke für die Suche, als auch als Information an den *Kanten* (siehe auch Abschnitt 4.2.2).

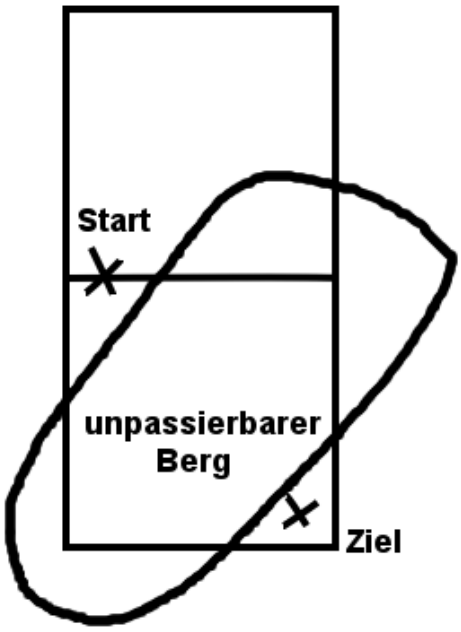


Abbildung 7: Ein Hindernis steht mitten in einem *Level* 3 *Cluster* zwischen Start- und Zielpunkt. Das `Minimumlevel` muss hier 3 sein, was eine zu genaue Betrachtung der linken oberen Ecke des *Clusters* vermeidet (ausschließlich interne *Level* 3 *Kanten* werden untersucht). Es handelt sich in diesem Beispiel nur um einen Teil einer Karte.

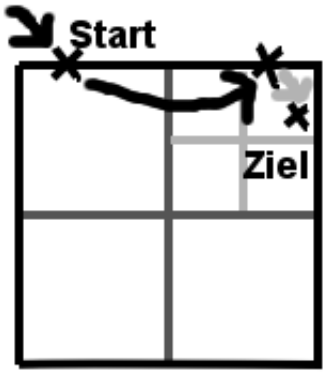


Abbildung 8: Hier gelangt man zuerst über eine *Level* 3 *Kante* (durch den schwarzen Pfeil angedeutet) und dann über eine *Level* 1 *Kante* (grauer Pfeil) zum Ziel. Das `Minimumlevel` kann nicht durch $Knotenlevel - 1$ berechnet werden, da der *Knoten* vor dem Zielpunkt ein *Level* 3 *Knoten* ist und die *Kante* zum Zielpunkt hat (Maximum-)*Level* 1 ($1 \neq 3 - 1$).

Die möglichen *Level* der *Kanten* werden in dem Fall, dass ein *erster grober Weg* gesucht wird auf folgende Art und Weise bestimmt:

1. Das `Maximumlevel` wird durch zwei obere Schranken festgelegt(wovon nur die jeweils niedrigere verwendet wird):
   (a) Die erste obere Schranke ist das *Level* des *aktuellen Knotens* - des *Knotens* für dessen Nachbarn gerade *Kosten* berechnet werden. 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 Vorverarbeitung 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 *aktueller Knoten* und *Zielknoten* sich im selben *Cluster* niedrigeren *Levels* befinden. Es wird deshalb der kleinste gemeinsame *Cluster* bestimmt und dessen *Level* als obere Schranke der nächsten *Kanten* festgelegt. Befinden sich die oben genannten *Knoten* nicht im selben *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, denn nur dann ist sichergestellt, dass der *Zielknoten* nicht verfehlt wird. Bei allen *Kanten* innerhalb des *Clusters* befindet sich das *Clusterlevel* im Intervall $[Minimumkantenlevel, Maximumkantenlevel]$.

Die berechneten `Maximum-` und `Minimumlevel` werden anschließend mit denen der *Kanten* verglichen, um festzustellen, ob eine *Kante* verwendet werden soll.

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 Weg gesucht werden, dann wird das vom *Pathfinding* berechnete `Maximumlevel` um 1 gesenkt (falls zuvor galt: $Maximumlevel = Minimumlevel$, so wird auch das `Minimumlevel` um 1 gesenkt). Dies garantiert, dass iterativ ein immer detaillierterer Weg berechnet wird - im Folgenden auch **Abstieg** genannt. Würde der *Abstieg* nicht erzwungen, könnte der Algorithmus denselben Weg zurückgeben, den er beim letzten Mal berechnet hat und dann würde nie ein detaillierterer Weg bestimmt.

4.2.4 Pathfinding zwischen Punkten

Beim *Pathfinding* muss folgendes bedacht werden: Das *Pathfinding* soll von einem Startpunkt zu einem Zielpunkt durchgeführt werden, aber nicht jeder Punkt ist im *Hierarchiegraphen* vorhanden (siehe Abschnitt 4.2.2). Das (evtl. temporäre) Einfügen von *Knoten* für Start- und Zielpunkt in den *Hierarchiegraphen* wurde wegen daraus folgenden Probleme bei der Synchronisierung des parallelen *HPA*-Algorithmus* ausgeschlossen (siehe [21]). Aufgrund dessen wurde das *Pathfinding* nach einem *ersten groben Weg* in mehrere Schritte geteilt.

Der Algorithmus sucht sich *Start-* und *Zielknoten* 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 *Hierarchiegraphen* gesucht werden. Der erste und der dritte Teilweg verbinden Startpunkt und *Startknoten* bzw. Zielpunkt und *Zielknoten* und vervollständigen somit den Weg. Dieser ist der *erste grobe Weg* des *HPA*-Algorithmus*. 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 Startpunkt aus zu erreichen. Allen *Knoten*, die erreicht wurden, werden *Kosten* zugeordnet. Diese *Kosten* berechnen sich aus den *Kosten*, die der *Dijkstra-Algorithmus* 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 9).

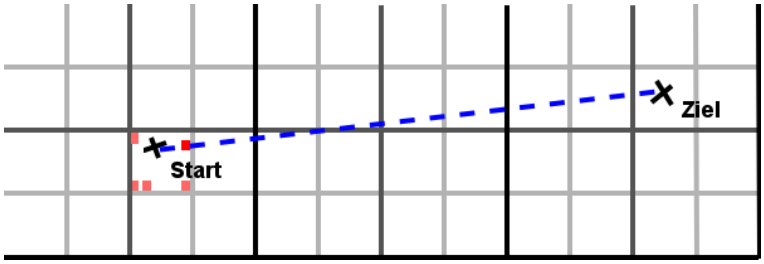


Abbildung 9: Alle roten und rosa Punkte sind Kandidaten für den *Startknoten* (alle *Knoten* des *Level* 1 *Clusters*). Der rote Punkt ist der gewählte Kandidat. Die blaue durchgezogene Linie ist der Weg vom Startpunkt zum *Startknoten* und die gestrichelte Linie deutet die, von der *Heuristik* geschätzte, 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 **Zielknotenkandidaten**. Dieses wird in Abbildung 10 dargestellt. Es ermöglicht die Berechnung eines kürzeren Weges, wenn mit einer Menge von *Zielknoten* (den *Zielknotenkandidaten*) gearbeitet wird, anstatt mit einem festgelegten *Zielknoten* (s.u.).

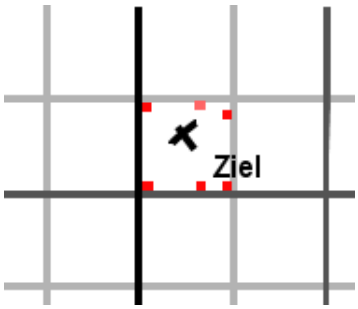


Abbildung 10: Die roten und die rosa Punkte sind Kandidaten für *Zielknoten* (alle *Knoten* des *Clusters*). Nur die roten Punkte sind vom *Zielknoten* aus erreichbar (auf einem Weg, der das *Cluster* nicht verlässt).

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 *Zielknoten* gefunden hat, ist der erste Weg auf dem *Hierarchiegraphen* gefunden worden (hier wird nun davon ausgegangen, dass es sich dabei um den am besten geeigneten *Zielknoten* handelt, was nicht immer stimmt). Ab diesem Zeitpunkt gibt es nur noch einen *Zielknoten* (siehe Abbildung 11).

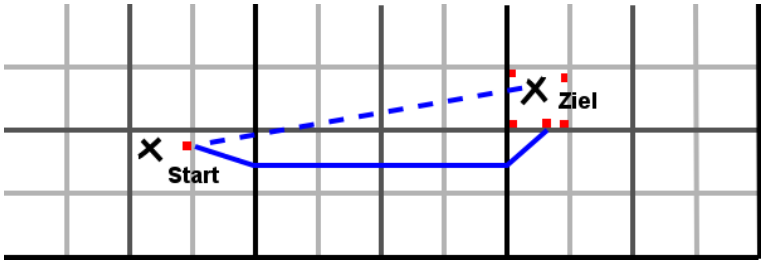


Abbildung 11: Die gestrichelte blaue Linie zeigt die Abschätzung vom *Startknoten* zum Zielpunkt an. Die durchgezogene blaue Linie ist der erste berechnete grobe Weg zwischen dem *Startknoten* und einem der *Zielknotenkandidaten*. Der *Zielknotenkandidat*, zu dem der Weg führt, ist von nun an der *Zielknoten* 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 *Zielknoten* festgelegt wird, desto mehr ist über den *kürzesten Weg* bekannt und damit liegt der gewählte *Zielknoten* mit höherer Wahrscheinlichkeit auf dem *kürzesten Weg*.

Bestimmung des Weges vom Zielknoten zum Zielpunkt

Das Ende des Weges - von *Zielknoten* zu Zielpunkt - wird dem *Dijkstra-Algorithmus* bestimmt (siehe Abbildung 12).

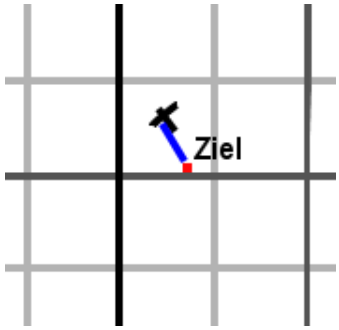


Abbildung 12: Der Weg (blau) zwischen *Zielknoten* (rot) und dem Ziel.

4.3 Parallelisierung

Parallelisiert wurde sowohl die Vorverarbeitung, d.h. die Erzeugung des *Hierarchiegraphen*, als auch das *Pathfinding*.

Das *Pathfinding* wurde so parallelisiert, dass keine Synchronisation der gemeinsam verwendeten Datenstrukturen notwendig ist, da diese während des *Pathfinding* nicht verändert wird. Da der *Pathfinder* für die

Suche von vielen Wegen in kurzer Zeit entwickelt worden ist, wurde nicht die einzelne Wegsuche parallelisiert, sondern es wurde eine zeitgleiche Bearbeitung von mehreren Wegsuchen implementiert.

4.4 Ergebnisse der Experimente

Die Experimente haben gezeigt, dass der *Pathfinder* immer am langsamsten ist, wenn genau zwei *Hierarchieebenen* (zuzüglich der Karte - diese stellt *Level* 0 dar) verwendet wurden. Eine genauere Erläuterung zu diesem Problem und der Versuch einer Lösung befindet sich in Abschnitt 5.1.

Die Parallelisierung war ein Erfolg. Insbesondere die Parallelisierung der Wegsuche führte deutlich schneller zu Ergebnissen und skalierte gut.

5 Optimierung des hierarchischen Pathfinders

Dieses Kapitel behandelt meine Optimierungsversuche am *hierarchischen Pathfinder*. Es wird eine Optimierung der Struktur des *Hierarchiegraphen*, Optimierungen der *Open List* und eine Alternative für den *A*-Algorithmus* vorgestellt. Des Weiteren werden verschiedene Möglichkeiten des *Cachings* und die *Amortisierung* der *Kosten* für *Pathfinding-Anfragen* betrachtet.

5.1 Optimierung des Hierarchiegraphen

In Kapitel 10 (Experimente) meiner Diplomarbeit[21] hat sich herausgestellt, dass das *Pathfinding* bei zwei *Hierarchieebenen* des *hierarchischen Pathfinders* stets langsamer ist als bei einer *Hierarchieebene*:

Der *A*-Algorithmus* iteriert über alle *Kanten*, die sich an einem *Knoten* befinden (siehe Listing 4 Zeile 4). Beim *hierarchischen Pathfinding* kommen einige *Kanten* allerdings nicht in Frage, da sie ein zu hohes oder zu niedriges *Level* haben: Soll der Weg mit möglichst wenigen Knoten auf hohen *Hierarchieleveln* (wenig detailliert) bestimmt werden, so verbraucht das Betrachten der *Kanten* niedrigeren *Levels* unnötig Rechenzeit. Wird hingegen ein detaillierterer Weg gesucht, so benötigt das Betrachten von *Kanten* zu hoher *Level* unnötig Rechenzeit. Der *A*-Algorithmus* im *hierarchischen Pathfinding* sortierte diese nicht benötigten *Kanten* durch die erste Anweisung in der Schleife aus, aber dies kostete weiterhin Rechenzeit. Daher wurde die Datenstruktur / der *Hierarchiegraph* geändert (siehe Tabelle 13):

Struktur vor der Änderung (alt)	Struktur nach der Änderung (neu)
Eine *Kante* kann sich auf mehreren *Leveln* befinden	Eine *Kante* befindet sich auf genau einem *Level*
Maximal eine *Kante* verbindet A und B	**Pro Level** verbindet maximal eine *Kante* A und B
	$\Rightarrow$ Eine alte *Kante*, die sich auf x *Leveln* (der alten Struktur) befindet, wird in x *Kanten* (der neuen Struktur) aufgeteilt
Alle *Kanten*, die von einem *Knoten* abgehen befinden sich in einem *Set*	Alle *Kanten* **eines Levels** befinden sich in einem *Set*. D.h. es gibt einen *Set* pro *Level*.

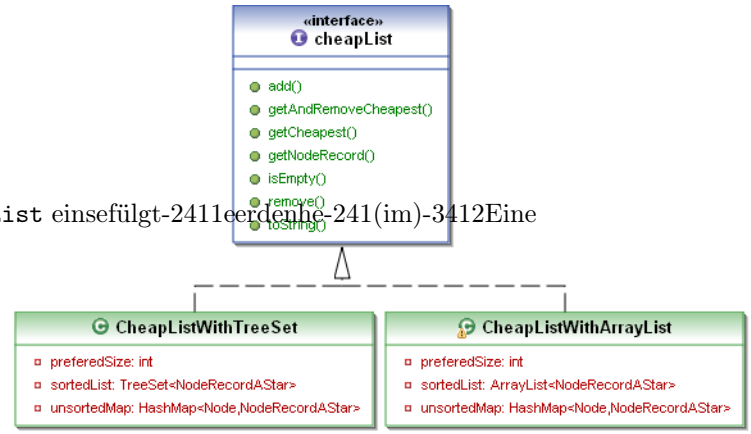
Abbildung 13: Datenstruktur-Gegenüberstellung. A und B sind beliebige *Knoten* des *Hierarchiegraphen*, $A \neq B$.

Die Schleife muss nun nur noch über die *Kanten* iterieren, die sich in den Sets befinden, die zu den *Leveln* gehören, die betrachtet werden sollen. Von Nachteil ist, dass es mehr Kantenobjekte gibt und diese Struktur daher mehr Speicher verbraucht.

5.2 Optimierung der Open List

Um die Suche nach dem kleinsten Element in der *Open List* (siehe Abschnitt 4.1) zu vermeiden kann eine sortierende Datenstruktur verwendet werden. Diese hat den Nachteil, dass sie Rechenzeit für das Einsortieren von *Knoten* benötigt, für die die *Kosten* (vom Start zum *Knoten* und vom *Knoten* zum Ziel) so hoch sind, dass sie wahrscheinlich vom *A*-Algorithmus* nicht weiter betrachtet werden. Eine Verbesserung kann eine Datenstruktur sein, die nur die *Knoten* einsortiert, die niedrige *Kosten* haben. *Knoten* mit hohen *Kosten* werden in einer unsortierten Datenstruktur gespeichert, für den Fall, dass sie doch betrachtet werden müssen [13].

Die *teilsortierte Liste* fügt maximal `preferedSize` viele Elemente in den sortierten Teil ein. Der sortierte Teil wird mit Elementen aus dem unsortierten Teil gefüllt, wenn ausschließlich der unsortierte Teil Elemente enthält. Ein UML-Diagramm von einem Teil meiner Implementierungen ist in Abbildung 14 zu sehen.



`ArrayList` einsefülgt-2411eerdenhe-241(im)-3412Eine

Abbildung 14: UML-Diagramm von zwei der implementierten *teilsortierten Listen*.

Die Klassen `CheapListWithTreeSet`, `cheapListWithArrayList` und `cheapListWithPriorityQueue` enthalten die auch schon in der Diplomarbeit benutzte `HashMap<Node, NodeRecordAStar>`, die den unsortierten Teil der *teilsortierten Listen* dartellt. Die Implementierungen unterscheiden sich durch die verwendeten Datenstrukturen für die sortierte Liste. `CheapListWithTreeSet` benutzt einen `TreeSet`, `cheapListWithArrayList` eine `ArrayList` und `cheapListWithPriorityQueue` eine `PriorityQueue`.

Der `TreeSet` besitzt den Vorteil, dass er eine **Heapstruktur** zur Sortierung verwendet, die nicht alle Elemente wie in einer Liste sortiert, sondern nur darauf achtet, dass das kleinste Element sich immer am Anfang befindet. Von Nachteil ist, dass die höchsten *Kosten* bzw. die *Kosten* des letzten Elements benötigt werden, damit entschieden werden kann, ob ein Element in den `TreeSet` oder die `HashMap` einsortiert werden soll - eine *Heapstruktur* hat aber kein letztes Element. In Java kann auch `TreeSet` nach einem letzten Element befragt werden, vermutlich ist dieser `TreeSet` daher langsamer als eine reine *Heapstruktur*.

Es gibt keinen Befehl der in eine `ArrayList` ein Element in Abhängigkeit eines seiner Attribute (in diesem Fall den *Kosten*)

DSllt in

in die `ArrayList` einsortiert. Dabei kann die `ArrayList` länger werden als vorhergesehen. Das Element mit den höchsten *Kosten* wird anschließend entfernt - dies kann im schlechtesten Fall $n - preferedSize$ mal auftreten.

Die `PriorityQueue` ist ebenfalls eine von Java angebotene Datenstruktur, die allerdings nicht nach dem letzten Element befragt werden kann. Dies muss man selbst implementieren. Muss eine `PriorityQueue` neu gefüllt werden, so wird jedes Element der `HashMap` zunächst der `PriorityQueue` hinzugefügt und dann anschließend das letzte Element aus der `PriorityQueue` entfernt, wenn diese länger als `preferedSize` ist. Anschließend muss erneut das letzte Element mittels Iteration bestimmt werden, was O(`preferedSize`) Schritte benötigt. Die Einsortierung in die `PriorityQueue` geschieht n mal, woraus sich *Kosten* von O(n * `preferedSize`) für die Entfernung und O(n * log `preferedSize`) für das Einsortieren ergeben. Dies ergibt O(n * `preferedSize` + n * log `preferedSize`) = O((n * `preferedSize`) und für `preferedSize` $>$ 2 ergibt sich damit O(n * `preferedSize` + n * log `preferedSize`) $>$ O(n * `preferedSize` + (n - `preferedSize`)) (*Kosten* der `ArrayList` Implementierung). Die Bestimmung des letzten Elements muss zudem nach dem Entfernen (des letzten Elements) oder Hinzufügen eines Elements durchgeführt werden, auch wenn dies nicht durch das Füllen mit Elementen aus der `HashMap` geschieht. Daher vermute ich, das `cheapListWithArrayList` schneller sein wird als `cheapListWithPriorityQueue`.

Eine weitere Implementierung stellt `cheapListWithArrayListAndBuckets` dar. Der sortierte Teil entspricht darin dem der `cheapListWithArrayList` und der "unsortierte" Teil ist nicht *eine* `HashMap<Node, NodeRecordAStar>`, sondern es sind mehrere. Jede `HashMap` dient dazu Elemente mit bestimmten *Kosten* aufzunehmen - z.B. eine `HashMap` für Elemente mit *Kosten* von 1 bis 10, eine zweite für Elemente mit *Kosten* von 11 bis 20 usw.. Damit wird eine Sortierung vorgenommen, die jedoch gröber ist, als die in der `ArrayList`. Sind die *Kosten* eines Elements zu hoch, um in die `ArrayList` einsortiert zu werden, so wird es entsprechend seiner *Kosten* in einer `HashMap` abgelegt.

Diese Struktur hat einen Vorteil und einen Nachteil gegenüber der `cheapListWithArrayList`. Der Nachteil besteht darin, dass es nun länger dauern kann, um zu bestimmen, ob sich ein Element bereits in der `cheapList` befindet, da im schlimmsten Fall in allen `HashMaps` nachgesehen werden muss, statt nur in einer. Der Vorteil ist, dass das Füllen der sortierten Liste mit Elementen des "unsortierten" Teils schneller geht. Es werden zuerst Elemente aus der `HashMap`, die für die Elemente mit den geringsten *Kosten* zuständig ist, einsortiert. Es sei denn, in dieser `HashMap` befindet sich kein Element, dann wird die `HashMap`, in der sich die nächst größeren Elemente befinden, verwendet usw.. Eine Sortierung aller Elemente beim Füllen der `ArrayList` kann daher vermieden werden.

5.3 Alternativalgorithmus zum A*

Der folgende Algorithmus ähnelt dem *Fringe Search* (dt.: Rand, Randgebiet) - siehe Abschnitt 3.1 - und war als Alternative zum *A*-Algorithmus* (im *HPA*-Algorithmus*) gedacht.

Fringe Search soll eine *depth-first Suche* (dt. *Tiefensuche*) sein und arbeitet auf folgende Weise: Zuerst schätzt der Algorithmus die Entfernung vom Start zum Ziel (wofür *Fringe Search* - ebenso wie der *A*-Algorithmus* - eine *Heuristik* verwendet). Das Ergebnis dieser Schätzung wird als Schranke für eine *depth-first Suche* benutzt. Anschließend wird die Suche gestartet und solange durchgeführt, bis das Ziel oder der letzte Weg vom Startpunkt aus (der weniger kostet als durch die Schranke festgelegt wurde) gefunden wurde. Befinden sich die Wegkosten eines *Knoten* oberhalb der Schranke, so wird dort zunächst nicht weiter gesucht. Siehe auch Abbildung 15.

Konnte kein Weg zum Ziel gefunden werden, so muss mit einer höheren Schranke die Suche fortgesetzt werden.

Während der *Tiefensuche* geschieht außerdem folgendes:

- der nächste Wert für die Schranke wird ermittelt; dabei handelt es sich um den kleinsten vorkommenden Kostenwert, der größer ist als die *aktuelle* Schranke.

Abbildung 15: Diese Abbildung zeigt eine Momentaufnahme beim Ablauf des *Fringe Search*. Dargestellt werden *Start-* und *Zielknoten*, der Bereich, der durch die Schranke begrenzt wird und *Knoten*, die sich im *Fringe* befinden.

- es werden die geringsten berechneten *Kosten* zu jedem *Knoten* abgespeichert, gemeinsam mit der *Kante*, über die man (mit diesen *Kosten*) zu dem Punkt gelangt ist. Die dafür zuständige Datenstruktur wird als *Cache* bezeichnet.
- der Algorithmus speichert die *Knoten* im *Fringe*, die betrachtet wurden, aber höhere *Kosten* besitzen als die *aktuelle* Schranke.

Wird die Suche mit einer höheren Schranke fortgesetzt, so wird nicht mehr der *Startknoten* als Ausgangspunkt verwendet, sondern die *Knoten* im *Fringe* werden als Ausgangspunkte verwendet.

Sobald der Algorithmus einen Weg zum Ziel gefunden hat, bricht er die Suche ab und rekonstruiert den Weg mit Hilfe des *Caches*.

Anmerkung: *Fringe Search* findet einen *kürzesten Weg* nur dann mit Sicherheit, wenn eine *monotone Heuristik* verwendet wird. Werden mit Hilfe der *Heuristik* stattdessen die *Kosten* überschätzt, kann es passieren, dass *Fringe Search* einen möglichen, aber keinen *kürzesten Weg* zurück gibt. Der Algorithmus hält den Weg für den kürzesten, den er zuerst gefunden hat.

5.3.1 Überlegungen

Der Algorithmus soll deshalb einen Geschwindigkeitsvorteil gegenüber dem *A*-Algorithmus* haben, da er keine Liste (*Open List*) sortieren muss.

Beim *Fringe Search* ist mir nach weitergehender Überlegung folgendes aufgefallen: Verwendet man den *Fringe Search* nicht nur für Karten, auf denen es ausschließlich passierbare und unpassierbare Felder gibt, sondern auch Felder, die nicht immer mit den gleichen *Kosten* betreten werden können, so wird eine exakte erste Schätzung der Entfernung bis zum Ziel fast nie vorkommen.

Ignoriert man nun diese wenigen Fälle, bei denen die Schätzung exakt wird, kommt man zu dem Ergebnis, dass zunächst alle *Knoten* im Bereich der ersten Schätzung betrachtet werden, ohne das ein Weg zum Ziel gefunden wird. Wird kein Weg gefunden, so macht es aber keinen Unterschied mehr, ob man in diesem Schritte eine *Tiefensuche*, *Breitensuche* oder *best-first Suche* verwendet, denn man betrachtet in jedem Fall alle Elemente.

Nach dem ersten Schritt, wird die Schranke nur minimal erhöht. Eine *Tiefensuche* kann dann innerhalb eines Schleifendurchlaufs gar nicht stattfinden - der Weg wird immer nur um ein Element verlängert. Meiner Meinung nach handelt es sich hierbei nicht um eine *Tiefensuche*, sondern entweder um eine *Breitensuche*, die

durch eine Schranke begrenzt wird oder um eine best-first Suche, die alle gleich guten (bzw. gleich teuren) Wege in einem Schritt behandelt.

Da im Pseudocode des *Fringe Search* eine Datenstruktur verändert wird, während darüber iteriert wird und dies mit Java-Iteratoren nicht möglich ist, habe ich eine *Breitensuche* implementiert, bei der es dieses Problem nicht gibt.

5.3.2 Implementierung

Pseudocode der in diesem Abschnitt beschriebenen Implementierung ist in Listing 5 zu sehen.

Der erste Wert für die Schranke (flimit) wird in Zeile 6 bestimmt und anschließend wird ein `NodeRecordAStar` für den *Startknoten* erzeugt und auf den *Fringe* gelegt. In Zeile 11 beginnt eine Schleife, die so lange läuft, bis ein Weg zum Ziel gefunden wurde oder festgestellt wird, dass dies nicht möglich ist.

Ich habe zwei Objekte - `oldFringe` und `newFringe` - für den *Fringe* verwendet. Beide sind Listen mit Elementen vom Typ `NodeRecordAStar` (siehe Abschnitt 4.1) - diesen habe ich an dieser Stelle wiederverwendet. Über `oldFringe` wird in Zeile 16ff iteriert. Ein `NodeRecordAStar`-Objekt im `oldFringe` wird bearbeitet, wenn seine *Gesamtkosten* die Schranke (flimit) nicht übersteigen. Soll ein *Knoten* zum ersten Mal in dieser Suche zum *Fringe* hinzugefügt werden oder kann von einem *Knoten* aus dem `oldFringe` der Weg nicht fortgesetzt werden, weil seine *Kosten* zu hoch sind, so wird das zugehörige `NodeRecordAStar`-Objekt dem `newFringe` hinzugefügt (Zeile 20 für Objekte aus dem `oldFringe`; Zeile 44-46 für neue *Knoten*).

Objekte aus dem `oldFringe` werden zur Berechnung der nächsten Schranke hinzugezogen, wenn sie oberhalb der bisherigen liegen (Zeile 18f). Sobald über alle Objekte des `oldFringe` iteriert wurde, wird der `oldFringe` durch den `newFringe` überschrieben und der `newFringe` geleert (Zeile 50f). Sofort wenn der Weg gefunden wurde, wird er auf dieselbe Art wie beim *Dijkstra-Algorithmus* und beim *A*-Algorithmus* rekonstruiert.

```
1  Alternativalgorithmus(...){
2    HashMap<Node, NodeRecordAStar> cache
3    ArrayList<NodeRecordAStar> oldFringe
4    ArrayList<NodeRecordAStar> newFringe
5
6    flimit = h(start)
7
8    NodeRecordAStar startNR = (S, 0, h(start), null) und trage
9      dies in den cache und den oldFringe ein
10
11   while( old Fringe nicht leer &
12     Ziel nicht gefunden) {
13
14     fmin = Positiv-Unendlich
15
16     foreach(NodeRecordAStar kr in old Fringe) {
17       k = Knoten aus kr
18       if (f(k) > flimit){
19         fmin = Minimum(fmin, f(k))
20         füge nr zum newFringe hinzu
21         continue
22       }
23
24       if (k == G) break
25
```

```
26          foreach(Nachbarn n von k) {
27
28            g(n) = g(k) + Kosten der Kante(n, k)
29
30            NodeRecordAStar nr = hole NodeRecordAStar des
31              Knotens n aus dem Cache
32
33            if(nr existiert) {
34              g'(n) = g(Knoten in nr)
35
36              if (g(n) >= g'(n)){
37                continue
38              }
39            }
40
41            berechne h(n)
42
43            newNodeRecordAStar = (n,g(n), f(n) = g(n) + h(n), Kante(k,n))
44              und fülle den cache und den newFringe damit
45
46          }
47        }
48        flimit = fmin
49        oldFringe = newFringe
50        leere den newFringe
51      }
52
53      rekonstruiere den kürzesten Weg
54    }
```

Listing 5: Pseudocode des Alternativalgorithmus

Der *Fringe Search* (wie der auch der A*) wurde gegenüber dem Pseudocode in [6] ein wenig verändert (siehe Abschnitt 4.2), damit er in jedem Suchschritt nur *Kanten* betrachtet, die:

- sich entweder auf den möglichst wenig detaillierten *Leveln* des *Hierarchiegraphen* befinden (wenn der *erste grobe Weg* gefunden werden soll)
- oder bei der Suche nach Wegpunkten auf dem detailliertesten *Level* mindestens ein *Level* niedriger sind, um auszuschließen, das derselbe Weg wieder zurück gegeben wird, statt des gesuchten detaillierteren.

5.4 Amortisierung der Kosten für Pathfinding-Anfragen

Der Ablauf eines Spiels wird in *Frames* (Zeitabschnitte) eingeteilt (siehe Abschnitt 2.1). In jedem *Frame* müssen vielfältige Berechnungen durchgeführt werden. *Pathfinding-Anfragen* müssen berechnet werden, die Anzeige muss aktualisiert werden, physikalische Berechnungen müssen durchgeführt werden usw.. Je länger die Summe dieser Berechnungen dauert, desto weniger *Frames* pro Sekunde (*fps*) können berechnet werden. Sind die *fps* zu niedrig, dann kann der Spieler einzelne Bilder sehen oder er kann (auch auf optimalem Terrain) Einheiten sehen, die sich nicht mehr mit gleichem Tempo, sondern ruckartig bewegen (beides wird **ruckeln** genannt).

Nicht in jedem *Frame* wird gleichviel Zeit für die Berechnung von *Pathfinding-Anfragen* benötigt (siehe auch Abbildung 16) - u.a. kann ein Spieler in einem beliebigen *Frame* unterschiedlich vielen Einheiten

den Befehl geben sich zu bewegen. Bei anderen Berechnungen kommt dies ebenfalls vor. Selbst wenn alle Berechnungen über den gesamten Spielverlauf betrachtet (im Durchschnitt) schnell genug erfolgen, so können sie in einem einzelnen *Frame* zu lange dauern, wenn dort überdurchschnittlich viele oder überdurchschnittlich aufwändige Berechnungen durchgeführt werden müssen. Geschieht dies nun in mehreren Teilen des Spiels (Physik, *KI*, Grafik etc.) im gleichen *Frame*, so beginnt das Spiel zu *ruckeln*. Da das *Pathfinding* nichts darüber weiß, wie schwierig die Aufgaben in einem *Frame* für die anderen Teile des Spiels sind, kann man nur versuchen, den Zeitbedarf für jeden Teil des Spiels möglichst konstant zu halten oder ein Maximum nicht zu überschreiten.

Wenn das Ziel einer Einheit nicht geändert wird, bevor sie dort ankommt, dann kennt das *Pathfinding* Berechnungen, die in der Zukunft geschehen müssen, um einen detaillierteren Weg für diese Einheit zu bestimmen. Diese Berechnungen können schon früher durchgeführt werden als nötig, nämlich in einem vorhergehenden *Frame*, in dem das *Pathfinding* die bereitgestellte Zeit für *aktuelle* Berechnungen nicht vollständig benötigt (siehe Abbildung 16). Sollte sich das Ziel der Einheit auf dem Weg ändern, so wurden unnötige Berechnungen durchgeführt - ändert es sich jedoch nicht, so wurde vielleicht ein *Ruckeln* zu einem späteren Zeitpunkt verhindert.

5.4.1 Algorithmus

Der Algorithmus berechnet zunächst die Wege, die im aktuellen *Frame* benötigt werden. Dies tut er auch dann, wenn dies mehr Zeit benötigt, als dem aktuellen *Frame* zur Verfügung steht (Zeitkontingent). Steht hingegen im selben *Frame* anschließend noch Zeit zur Verfügung, werden aus der Menge der vermuteten zukünftigen *Pathfinding-Anfragen* die *Pathfinding-Anfragen* herausgesucht, die innerhalb des nächsten *Frames* benötigt werden. Für diese Anfragen werden dann Berechnungen gestartet. Diese werden abgebrochen, wenn die Berechnungen drohen das Zeitkontingent zu überschreiten. Die bis dahin berechneten Wegpunkte werden an die Einheiten weitergegeben.

Können diese vorgezogenen Berechnungen vollständig durchgeführt werden und wurde das Zeitkontingent noch nicht überschritten, so werden *Pathfinding-Anfragen* herausgesucht, die vermutlich im übernächsten *Frame* berechnet werden müssen usw..

Die *Pathfinding-Anfragen*, die im nächsten *Frame* benötigt werden, werden bestimmt, indem die Länge des Weges, die der Einheit noch zur Verfügung steht, mit ihrer Mindestweglänge verglichen wird. Diese ergibt sich aus ihrer maximalen Geschwindigkeit.

Beispiel: Einheit A kann sich im nächsten *Frame* aufgrund ihrer Geschwindigkeit höchstens 5 Wegpunkte weiter bewegen - daraus ergibt sich eine Mindestweglänge von 5. Im aktuellen *Frame* wurden 6 Wegpunkte berechnet, wodurch eine Berechnung von weiteren Wegpunkten im nächsten *Frame* wahrscheinlich nötig ist, da $6 - Mindestweglänge < Mindestweglänge$. Ist das Zeitkontingent nach den Berechnungen für den aktuellen *Frame* noch nicht erschöpft, so wird anschließend u.a. die Pfadanfrage für Einheit A aus der Menge der vermuteten zukünftigen *Pathfinding-Anfragen* gewählt und diese bearbeitet. Drohen die Berechnungen das Zeitkontingent zu überschreiten, wird die Berechnung der Pfadanfrage für Einheit A angebrochen, es sei denn sie wurde schon erfolgreich beendet.

Ist das Zeitkontingent danach immer noch nicht erschöpft und für Einheit A liegt nun eine Pfadlänge von 10 vor, so ergibt sich wieder mit Hilfe der Mindestpfadlänge, dass im übernächsten Schritt eine weitere Pfadberechnung nötig ist. Eine Pfadanfrage für Einheit A wird damit ein weiteres Mal aus der Menge der vermuteten zukünftigen *Pathfinding-Anfragen* ausgewählt. Dies würde jedoch nicht geschehen, wenn schon eine Pfadlänge ≥ 15 vorliegen würde.

5.4.2 Implementierung

Nach der Erzeugung eines *Threadpools* und eines `ExecutorServices` wird dieser mit den aktuell benötigten Anfragen mit Hilfe der Methode `invokeAll(List<Callable<T>>)` gefüllt. Die Anfragen befinden sich in der

`toDoList`. Die `toDoList` besteht aus `(HPAStar-)Pathfinder-Objekten` - die Klasse `HPAStarPathfinder` ist ein `Callable`.

Anschließend wird eine Schleife über die Liste der `Futures` durchgeführt und die Ergebnisse werden ausgewertet. Ist danach noch Bearbeitungszeit (des *Frames* für das *Pathfinding*) übrig, so wird die `toDoList` mit Anfragen gefüllt, die eventuell in den nächsten *Frames* gestellt würden (diese befinden sich in der `HashMap searches`):

Ob die Berechnungen eines *Frames* in der vorhandenen Zeit ausgeführt werden kann, ist nicht vorherzusagen. Die Berechnungen des aktuellen *Frames* müssen in jedem Fall ausgeführt werden - alle anderen können abgebrochen werden, wenn die Zeit abgelaufen ist; dazu wird die Methode `invokeAll(List<Callable<T>>`, time, timeUnit) verwendet. Diese bricht mit der Bearbeitung der Tasks ab, sobald die Zeit abgelaufen ist. Die Methode `Future.isCancelled()` kann anschließend genutzt werden, um zu überprüfen, ob ein Task beendet werden konnte oder nicht. Ist Das Zeitkontingent danach noch nicht erschöpft, so wird die `toDoList` ein weiteres Mal gefüllt und `invokeAll(List<Callable<T>>, time, timeUnit)` ein weiteres Mal aufgerufen usw.

Die neuen Wegpunkte werden nach jeder erfolgreichen Bearbeitung eines `Callables` an die Einheit weitergegeben.

5.5 Vorverarbeitung der Wege des höchsten Hierarchielevels

Durch Messungen mit meiner Implementierung konnte ich feststellen, dass die Rechenzeit, die während der Wegsuche für die Berechnungen der *ersten groben Wege* benötigt wird, ein Vielfaches im Vergleich zur benötigten Rechenzeit für die Berechnung der nächsten detaillierten Punkte beträgt. Dadurch entstand der Ansatz, nur die Berechnungen der *ersten groben Wege* zu beschleunigen. Ein Teil dieser Berechnungen ist eine Suche in Richtung Ziel auf dem höchsten *Hierarchielevel*. Die Idee war, eine Vorverarbeitung, die alle Wege innerhalb des *höchsten Hierarchielevels* berechnet und abspeichert, so dass diese Wege während des Spiels nicht berechnet, sondern nur geladen werden müssen. Eine Vorverarbeitung des *höchsten Hierarchielevels* benötigt deutlich weniger Speicher als eine innerhalb des niedrigsten (da dort weniger *Knoten* und *Kanten* vorhanden sind); deshalb kann sie je nach vorhandenem Speicher sinnvoller sein als eine komplette Vorberechnung aller detaillierter Wege. Ich möchte daher eine Vorverarbeitung genauer betrachten, die sich auf das *höchste Hierarchielevel* beschränkt. Es ist dabei zu berücksichtigen, dass die Bestimmung des *ersten groben Weges* nicht nur auf dem *höchsten Hierarchielevel* stattfindet. Um die gespeicherten Wege bei der Suche nach dem *ersten groben Weg* nutzen zu können, muss die Suche danach verändert werden. Sie kann in den folgenden Schritten erfolgen:

1. Es wird der *Level* 1 *Knoten* des *Clusters*, in dem sich der Startpunkt befindet, ermittelt, der die niedrigsten *Gesamtkosten* besitzt. Dieser wird als erster *Knoten* des Weges festgelegt.
2. Anschließend wird so lange vom ersten *Knoten* in Richtung Ziel gesucht, bis aus der *Open List* des *A*-Algorithmus* ein *Knoten* des *höchsten Levels* (k1) genommen wird. Statt anschließend die Suche mit dem *A*-Algorithmus* fortzusetzen, wird diese abgebrochen und angenommen, dass k1 zu dem gesuchten Weg gehört; der Weg bis zu k1 wird gespeichert.
3. Schritt eins und zwei werden ein weiteres Mal durchgeführt - allerdings vom Ziel in Richtung Start. Der dabei ermittelte *Knoten* des höchsten *Hierarchielevels* sei k2.
4. Der Weg zwischen den ermittelten *Knoten* (k1,k2) des *höchsten Hierarchielevels* wird aus dem *Cache* geholt.
5. Der Weg wurde nun in fünf Stücken bestimmt, die nun nur noch zusammengesetzt werden.

Der Nachteil bei diesem Verfahren besteht in den Annahmen, dass es sich bei den *Knoten*, die in den Schritten 1 bis 3 gewählt werden, um *Knoten* des gesuchten Weges handelt. Stimmen diese Annahme nicht, wird vom *Pathfinder* ein längerer Weg als Ergebnis geliefert.

Die Berechnungen der *kürzesten Wege* zwischen allen *Knoten* des höchsten *Hierarchielevels* können in einer Vorverarbeitung erfolgen und beim Verkauf eines Spiels auf dem Datenträger gespeichert werden oder beim Laden des Spiels erfolgen.

Im Folgenden werden zwei Algorithmen (mehrfache Anwendung des *A*-Algorithmus* und **Floyd-Algorithmus**) vorgestellt, die für die Berechnung aller Wege des *höchsten Hierarchielevels* eingesetzt werden können. Dabei werden außerdem Datenstrukturen vorgestellt, die für die Speicherung der Wege genutzt werden können. Da es sich hierbei um eine Vorverarbeitung handelt und die Berechnungen nicht während des Spiels durchgeführt werden, ist es nicht zwingend nötig den schnelleren Algorithmus zu ermitteln und zu verwenden. Sollten umfangreiche Vorverarbeitungen durchgeführt werden (viele große Karten) kann es sinnvoll sein den schnelleren Algorithmus zu wählen, um z.B. Rechenkapazitäten während der Entwicklung zu sparen. Ich habe daher nur die mehrfache Anwendung des *A*-Algorithmus* implementiert und möchte die Alternative, den *Floyd-Algorithmus*, nur kurz vorstellen. Eine Ermittlung des schnelleren Algorithmus per Laufzeitvergleich ist nicht möglich, da es nicht möglich ist die Laufzeit des *A*-Algorithmus* im *average case* anzugeben: Diese ist zu sehr von der verwendeten *Heuristik* und der betrachteten Karte abhängig. Der *Floyd-Algorithmus* hat hingegen immer eine Laufzeit von $O(n^3)$.

Außerdem wird die Speicherung der berechneten Wege in eine Datei erklärt.

5.5.1 Mehrfache Anwendung des A*-Algorithmus

Dieser Algorithmus bestimmt jeden möglichen Weg einzeln mit dem *A*-Algorithmus*

einen Knotem in einem *Cluster* zu einem anderen *Knoten* im gleichen *Cluster* geben, dabei handelt es sich dann meist um einen Weg mit wenigen *Knoten*, der oft ein Teilweg eines anderen Weges sein wird. Daher habe ich über die Liste in der äußeren Schleife von rechts nach links und in der inneren Schleife entgegengesetzt iteriert. Somit sollte es häufiger vorkommen, dass Wegberechnungen eingespart werden können.

5.5.2 Floyd-Algorithmus

Der **Floyd-Algorithmus**[27] berechnet die Längen aller kürzesten Wege von jedem *Knoten* zu jedem anderen *Knoten* (er gehört damit zu den All-To-All Shortest Path Algorithmen). Der *Floyd-Algorithmus* speichert nicht die *kürzesten Wege*, sondern nur deren Längen, kann jedoch um die Speicherung dieser erweitert werden und eignet sich dann für die Vorherberechnung der gesuchten Wege.

Der *Floyd-Algorithmus* benötigt eine $N * N$ große Matrix A mit N = Anzahl der *Knoten*, die wie folgt gefüllt sein muss:

$$A_{ij} = \begin{cases} 0 & \text{Falls } i = j \\ +\infty & \text{Falls keine } \textit{Kante} \text{ von i nach j existiert} \\ \text{Länge der } \textit{Kante} \text{ von } i \text{ nach } j & \text{Andernfalls} \end{cases} \tag{2}$$

Der Algorithmus verwendet folgenden Satz: Handelt es sich bei dem Weg von U nach W um einen *kürzesten Weg* und enthält dieser Weg einen *Knoten* V, so sind die darin enthaltenen Wege von U nach V und von V nach W ebenfalls *kürzeste Wege*. Angenommen, man kennt schon die *kürzesten Wege* zwischen allen *Knotenpaaren*, die nur über *Knoten* mit $Index < K$ führen und sucht alle *kürzesten Wege* über *Knoten* mit $Index <= K$ dann gibt es für einen Weg von U nach W zwei Möglichkeiten:

1. Der Weg geht über den *Knoten* K und setzt sich zusammen aus schon bekannten Wegen von U nach K und von K nach W.
2. Es handelt sich um den schon bekannten Weg von U nach W mit $Index <= K$

Der *Floyd-Algorithmus* ist in Listing 7 zu sehen.

```
for k = 1..N {
  for i = 1..N {
    for j = 1..N {
      A_ij = min(A_ij, A_ik + A_kj)
    }
  }
}
```

Listing 7: *Floyd-Algorithmus*

5.5.3 Speicherung der Wege in eine Datei

Die Speicherung der Wege erfolgt in eine Datei nach dem in Abbildung 18 dargestellten Schema.

5.6 Caching

Hier werden die Idee des *Caching*, die Anwendbarkeit in Spielen, die nötige Datenstruktur und Gedanken zur Parallelisierung des *Cachings* erläutert.

5.6.1 Idee

Caching ist das Speichern von Ergebnissen von Berechnungen, um erneutes Berechnen (für identische Parameter) zu einem späteren Zeitpunkt überflüssig zu machen. D.h. es wird Rechenzeit gespart, aber mehr Speicher benötigt. *Caching* wird dann eingesetzt, wenn der Speicherbedarf für eine Speicherung aller Daten (hier: Wege) höher ist, als der verfügbare Speicher.

Um Speicher zu sparen, werden in diesem Fall, wie schon in der Vorverarbeitung in Abschnitt 5.5 geschehen, nur die Wege auf dem höchsten *Hierarchielevel* gespeichert. Das *Pathfinding* funktioniert genauso wie in Abschnitt 5.5. Der Unterschied zur Vorverarbeitung besteht darin, dass weder vor Programmstart noch während des Programmstarts Wege auf dem *höchsten Hierarchielevel* gespeichert werden, sondern dass nach einer Wegberechnung das Ergebnis nicht nur an den *Agenten* weitergeleitet wird, sondern auch in einem *Cache* gespeichert wird. D.h. erst wenn ein zweites Mal ein Weg mit den gleichen *Start-* und *Zielknoten* auf dem *höchsten Hierarchielevel* berechnet werden soll, kann eine Berechnung eingespart werden. Um Speicher den Speicherverbrauch zu reduzieren, wird ein solches Ergebnis nur selten für die gesamte Laufzeit eines Programms im *Cache* gespeichert.

Der *Cache* hat meist eine feste Größe, um den Speicherbedarf zu begrenzen. In diesem Fall kann als Größe eine maximale Anzahl an (speicherbaren) Wegen festgelegt werden.

Angenommen ein *Cache* enthält zufällig ausgewählte 10% aller möglichen Wege des *höchsten Hierarchielevels* und löscht einen einmal gespeicherten Weg nicht, dann würde er im Durchschnitt ein Zehntel des Geschwindigkeitsvorteils der Vorverarbeitung erzielen. Ziel ist es jedoch, eine deutlich größere Geschwindigkeitssteigerung zu erhalten. Damit dies möglich wird, müssen Anfragen an den *Pathfinder* gestellt werden, deren Ergebnisse einen Teil der Wege des *höchsten Hierarchielevels* deutlich häufiger enthalten, als andere. Enthält der *Cache* diese Wege, so ergibt sich ein größerer Geschwindigkeitsvorteil.

Vor und während des Programmstarts steht nicht fest, welche Wege häufiger gebraucht werden als andere, deswegen muss dies während des Spiels ermittelt werden: Immer, wenn ein Weg berechnet wird, wird er zusammen mit einem **Zeitstempel** (z.B. der Nummer des aktuellen *Frames*) im *Cache* gespeichert. Enthält der *Cache* bereits vor der Speicherung die maximale Anzahl an Wegen, die er speichern soll, so wird der Weg mit dem ältesten *Zeitstempel* gelöscht. Wird einer der gespeicherten Wege bei einer Wegsuche verwendet, so wird der *Zeitstempel* des Weges aktualisiert. Oft verwendete Wege werden dadurch seltener aus dem *Cache* entfernt, als wenig verwendete Wege.

Um eine Beschleunigung durch *Caching* erreichen zu können, muss ein weiterer Punkt erfüllt werden - der *Cache* muss groß genug sein: Angenommen 10% der Wege (des *höchsten Hierarchielevels*) werden deutlich häufiger verwendet als alle anderen. Nimmt man weiterhin an, dass diese 10% etwa gleich oft und gleichmäßig oft (also nicht erst 5% der Wege und später andere 5% der Wege) benutzt werden, so sollte der *Cache* etwa so groß sein, dass er diese Wege gleichzeitig enthalten kann. Angenommen ein *Cache* kann nur 1% aller Wege enthalten (bzw. 10% der 10% häufiger verwendeten Wege), so wird er oft Wege aus dem *Cache* entfernen, die wenig später benötigt werden.

Ein zu kleiner *Cache* kann zu einer Verlangsamung des Programms führen, da Speichern und Löschen von Wegen im *Cache* Zeit benötigt, aber der *Cache* keine Beschleunigung mehr erzeugt, wenn die darin gespeicherten Wege kaum genutzt werden können, um Berechnungen zu sparen. Die optimale Größe eines *Caches* sollte daher in Experimenten bestimmt werden. Bei der optimalen Größe wird es sich allerdings immer um einen Kompromiss zwischen Speicherbedarf und Beschleunigung des Programms handeln, denn eine maximale Beschleunigung erhält man dann, wenn alle Wege vorherberechnet und gespeichert werden.

5.6.2 Anwendbarkeit in Spiele

Caching kann nicht in jedem Spiel eingesetzt werden. Gibt es in einem Spiel keine Wege, die (möglichst wesentlich) häufiger verwendet werden als andere, so kann *Caching* nicht verwendet werden. Es gibt RTS-Spiele in denen die Spieler eine Basis (z.B. eine Burg) haben und von dort aus Agenten in Kämpfe gegen Agenten anderer Spielern schicken. Der Start einer Wegsuche ist dann häufig in der eigenen Basis und

das Ziel in der Basis eines anderen Spielers oder an einem anderen strategisch wichtigen Punkt (z.B. einer Brücke). Ein Beispiel für ein solches Spiel ist "Starcraft" der Vorgänger vom oben genannten "Starcraft 2"[15] (wahrscheinlich wird dies auch für "Starcraft 2" gelten) und ein Gegenbeispiel "Medieval II: Total War"[16], dort wird zu Beginn jedes Kampfes erst eine Karte geladen, auf der der Kampf ausgetragen wird und in den Kämpfen selber werden "ähnliche Bewegungen" anders behandelt, nämlich dadurch, dass keine einzelnen Agenten bewegt werden, sondern Gruppen von Agenten (siehe Abschnitt 3.2). Eine solche Bewegung wird selten wiederholt.

5.6.3 Datenstruktur des Caches

Welche Operationen müssen mit einem *Cache* durchgeführt werden?

- Entfernen des Weges mit dem ältesten *Zeitstempel*.
- Hinzufügen eines Weges.
- Aktualisieren von *Zeitstempeln*.
- Suchen von Wegen im *Cache*.

Diese Operationen können mit einer sortierten Datenstruktur umgesetzt werden. Sortiert wird dabei nach den *Zeitstempeln*. Verwendet werden könnte z.B. eine `PriorityQueue`, in sich der Weg mit dem ältesten `Zeitstempel` am Anfang befindet. Von Vorteil wäre es wenn einen solche Datenstruktur eine Update-Funktion hätte, die in der Lage ist einen `Zeitstempel` zu ändern und den Weg, dessen `Zeitstempel` geändert wurde schnell neu einzusortieren.

5.6.4 Parallelisierung

Es gibt zwei prinzipiell unterschiedliche Möglichkeiten *Caching* parallel durchzuführen:

1. Jeder *Thread* besitzt einen eigenen *Cache*.
2. Mehrere *Threads* benutzen einen gemeinsamen *Cache*.

Im ersten Fall können die Inhalte jedes *Caches* völlig unterschiedlich sein. Es wird keine Synchronisierung durchgeführt. Dies führt entweder dazu, dass mehr Speicher für das *Caching* benötigt wird, nämlich Anzahl der Threads $*$ Speicherbedarf für einen *Cache* oder die einzelnen *Caches* werden verkleinert. Eine Verkleinerung der *Caches* kann dann durchgeführt werden, wenn es möglich ist eine Karte so aufzuteilen, dass jeder *Thread* nur für einen Teil der Karte zuständig ist (z.B. *Thread* 1 behandelt nur *Pathfinding-Anfragen* von Gebiet A nach Gebiet B oder umgekehrt, *Thread* 2 nur von Gebiet A nach Gebiet C usw.). Dies funktioniert allerdings nicht mehr, wenn aufgrund der Spielsituation sehr oft Agenten von Gebiet A nach Gebiet B (und umgekehrt) bewegt werden und auf anderen Teilen der Karte kaum Bewegungen stattfinden. Dann würde ein *Thread* sehr viel berechnen und die anderen fast nichts. Eine solche Situation kann in einem Spiel durchaus auftreten, z.B. im o.g. "Starcraft", wenn zwei Spieler jeweils eine Basis besitzen und sich immer wieder gegenseitig auf demselben Weg Agenten entgegen schicken.

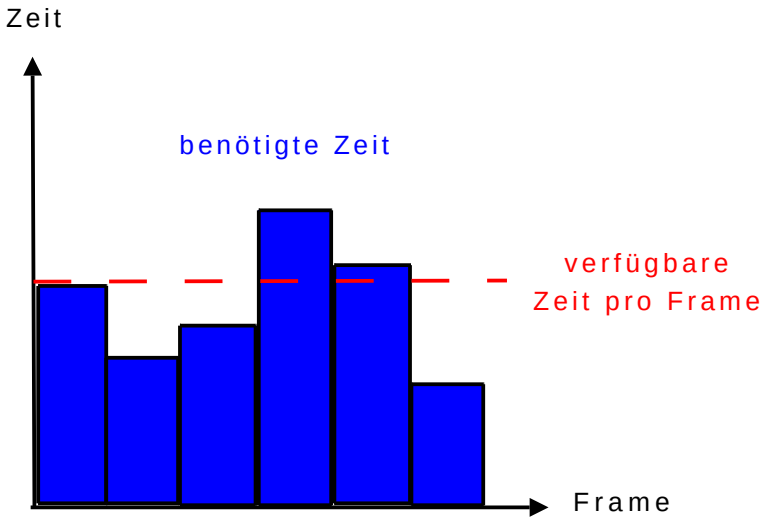
Benutzen mehrere *Threads* einen *Cache*, dann muss der Zugriff auf diesen synchronisiert werden. Es darf nicht passieren, dass ein *Thread* einen Weg aus dem *Cache* löscht, während ein anderer gerade seinen `Zeitstempel` aktualisiert. Nach jeder Wegsuche schreibt jeder *Thread* einmal in den *Cache*: Entweder er aktualisiert einen `Zeitstempel` oder er fügt einen Weg hinzu und löscht evtl. einen anderen. Je mehr *Threads* Zugriff auf den *Cache* haben, desto häufiger kann ein *Thread* nicht weiterarbeiten, weil ein anderer gerade den *Cache* benutzt. Je weniger Zeit eine Wegsuche benötigt und je länger die Zugriffe auf den *Cache* werden, desto häufiger muss ein *Thread* warten, bis er Zugriff auf den *Cache* bekommt.

Das Problem kann evtl. verringert werden dadurch, dass man Zeitabschnitte definiert, in denen ausschließlich aus dem *Cache* gelesen werden darf und Zeitabschnitte in denen er aktualisiert (Aktualisierungsphase)

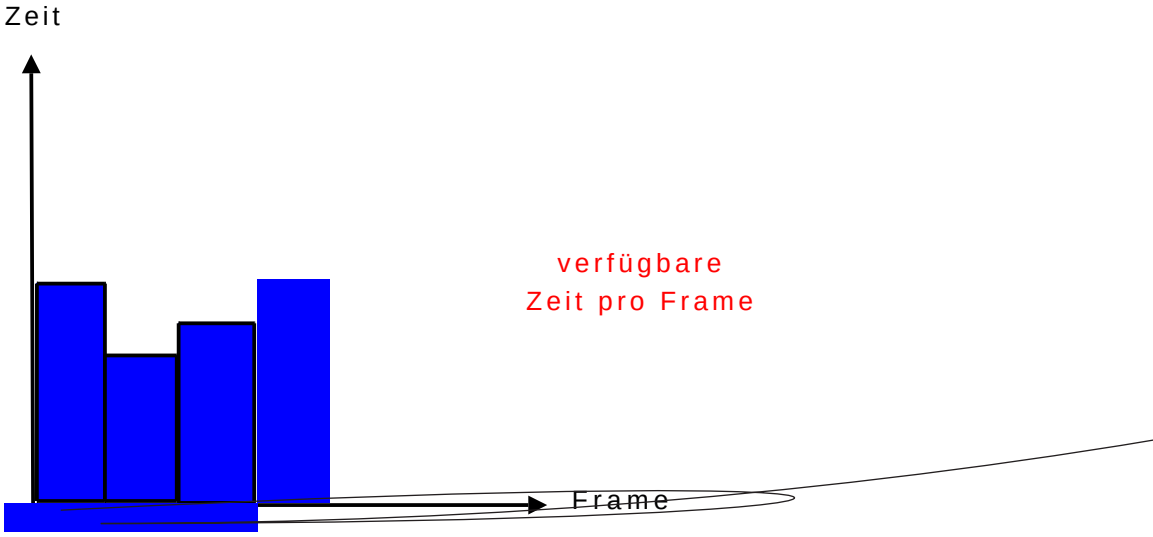
werden darf. Dazu müsste jeder *Thread* Schreibzugriffe sammeln und während der Aktualisierungsphase am Stück ausführen. Werden mehrere Zugriffe am Stück ausgeführt, so sinken die Kosten für die Synchronisierung, da seltener Operationen durchgeführt werden müssen, die einem *Thread* exklusiven Zugriff aus den *Cache* geben.

Eine Folge dieser Vorgehensweise kann sein, dass ein *Thread* einen Weg berechnet, obwohl ein anderer schon das Ergebnis hat und noch nicht in den *Cache* schreiben konnte oder auch, dass ein Ergebnis im *Cache* genutzt wird, was schon gelöscht werden sollte. Diese Effekte lassen sich möglicherweise ausgleichen.

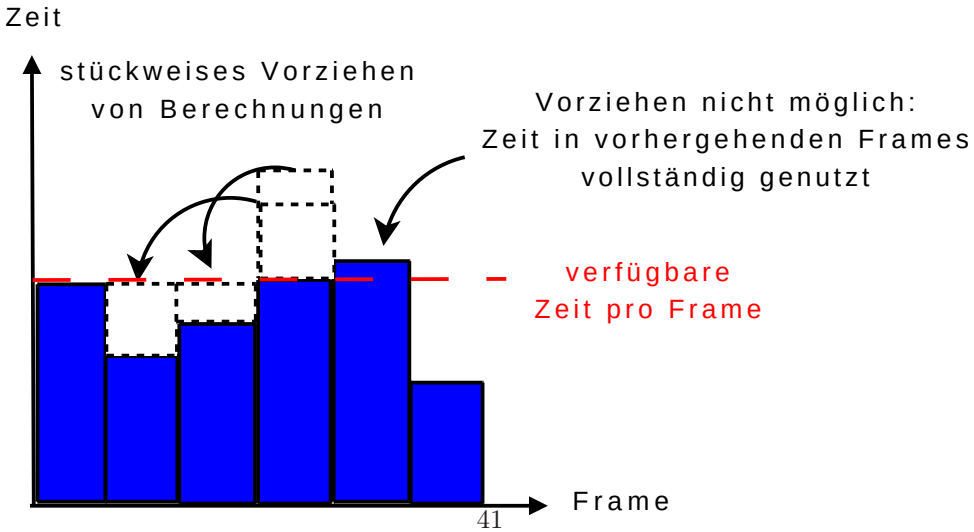
Es ist zu beachten, dass nicht so viele Schreibzugriffe gesammelt werden dürfen, dass ihre Ausführung so lange dauert, dass die nächsten Pfade erst so viel später wieder berechnet werden können, dass es zu einem *ruckeln* des Spiels kommt (siehe auch Abschnitt 5.4).



(a) Vergleich: Zeitbedarf vs. vorhandene Zeit

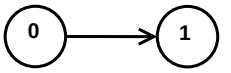


(b) Vorziehen von Berechnungen

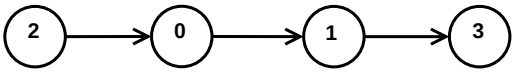


(c) Stückweises und unmögliches Vorziehen von Berechnungen

Abbildung 16: In diesen Balkendiagrammen ist ein Balken pro *Frame* (Schleifendurchlauf der *Mainloop*) zu sehen, der anzeigt, wie viel Zeit benötigt wird, um die nötigen Berechnungen (des *Pathfindings*) durchzuführen. Eine gestrichelte Linie zeigt an wie viel Zeit pro *Frame* zur Verfügung steht.



(a) Simpler Weg - zwei direkt durch eine *Kante* verbundene *Knoten*



(b) Weg über mehrere Zwischenknoten

Zielknoten / *Startknoten*	0	1	2	3
0	-	1		1
1		-		3
2			-	0
3				-

(c) Beide Wege in einer Tabelle gespeichert - Abbildung 17a mit grün und Abbildung 17b mit blau gekennzeichnet.

Abbildung 17: Beispielwege, die in einer Tabelle gespeichert wurden.

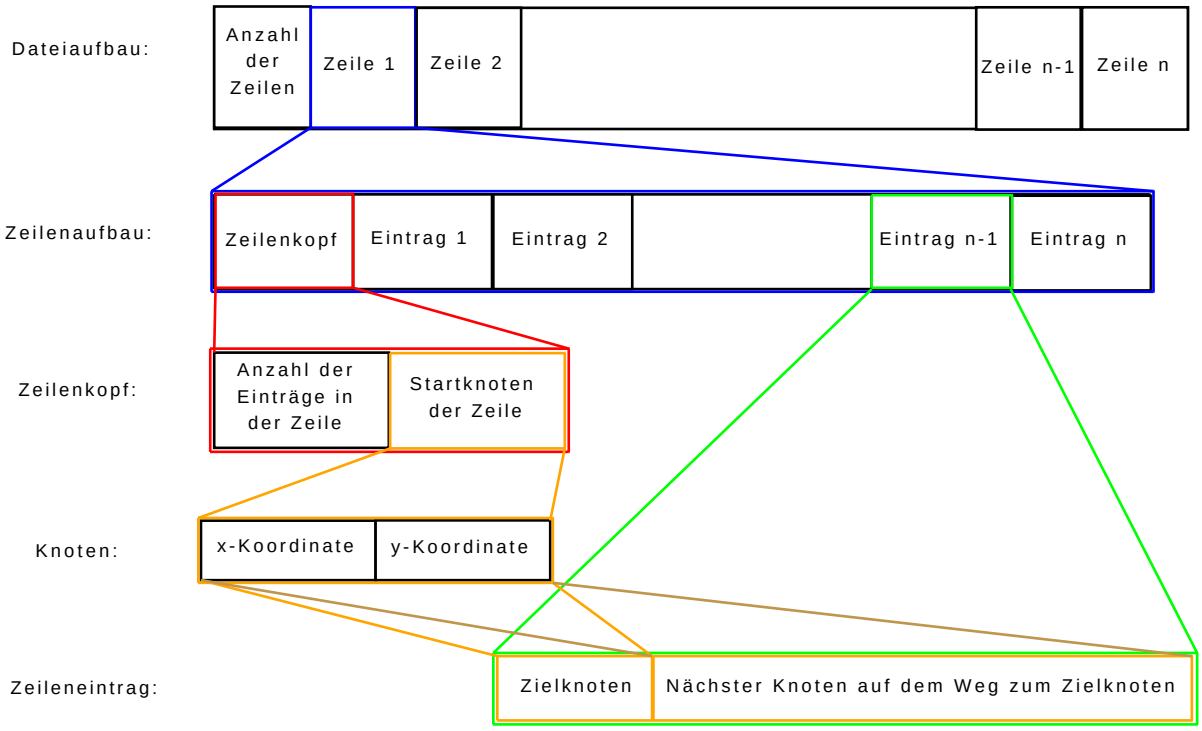


Abbildung 18: Die Daten werden wie hier dargestellt in einer Datei gespeichert.

6 Experimentierumgebung

Im Rahmen dieser Arbeit wurde eine Experimentierumgebung entwickelt. Diese dient dazu:

- den *Hierarchiegraphen* und Wege zu visualisieren, um damit unter anderem das Debugging zu erleichtern. Mit Hilfe einer Visualisierung kann schnell geprüft werden, ob der *Hierarchiegraph* korrekt erzeugt wurde und ob es sich bei einem vom *Pathfinder* berechneten *kürzesten Weg* auch um einen solchen handelt.

- die Durchführung von Experimenten zu den Optimierungsversuchen in Kapitel 5 zu ermöglichen. Derartige Experimente sind nötig, um Veränderungen in der Geschwindigkeit des *Pathfinders* feststellen zu können.

Für die Durchführung von Experimenten werden zu allererst *Pathfinding-Anfragen* benötigt, welche die Experimentierumgebung erzeugen kann. Außerdem bietet sie die Möglichkeit ein oder mehrere Experimente mehrfach mit den gleichen zufällig erzeugten *Pathfinding-Anfragen* durchzuführen. Dies gewährleistet, dass Experimente reproduzierbar und vergleichbar sind. Sollen z.B. zwei *teilsortierte Listen* verglichen werden, so ist es notwendig, dass bei Messungen mit beiden Listen die gleichen Suchanfragen gestellt werden. Ist dies nicht der Fall, so sagen die Ergebnisse des Experiments nichts über die *teilsortierten Listen* aus, da bei Messungen mit einer der Listen deutlich aufwändigere Berechnungen durchgeführt worden sein können, als bei Messungen mit der anderen Liste.

Um die *Amortisierung* (siehe Abschnitt 5.4) testen zu können, ist eine Simulation eines Spielablaufs notwendig. Zu diesem Zweck kann die Experimentierumgebung nicht nur beim Start eines Experiments *Pathfinding-Anfragen* erzeugen, sondern auch in späteren *Frames* - auch dies ist reproduzierbar und damit vergleichbar. Des Weiteren wurde bei den Einheiten eine Funktion hinzugefügt, die eine Bewegung simuliert. D.h. jede Einheit wird in jedem *Frame* entlang des berechneten Weges entsprechend ihrer Geschwindigkeit bewegt. Dabei wird ihre Position verändert und der zurückgelegte Teil des Weges wird gelöscht. Somit verringert sich die Anzahl der bekannten Wegpunkte der Einheit in jedem Schritt und dadurch werden *Pathfinding-Anfragen* nach weiteren Punkten ausgelöst. Die Simulation wurde nur bei den Tests verwendet, die die Wirkung der *Amortisierung* untersuchen sollten.

Die Experimentierumgebung benutzt einen Zufallsgenerator (java.util.Random), um Start- und Zielpunkte für die *Pathfinding-Anfragen* zu generieren. Der Zufallsgenerator erstellt Pseudozufallszahlen und benutzt einen sogenannten **seed** (englisch: Saat, Keim) - einen Wert der benutzt wird, um die Pseudozufallszahlen zu berechnen. Legt der Programmierer einen *seed* für den Zufallsgenerator fest, dann wird immer wieder die gleiche Folge von Pseudozufallszahlen erzeugt. Eine solche Festlegung wurde vorgenommen und stellt sicher, dass immer dann, wenn x *Pathfinding-Anfragen* erzeugt werden es sich um die gleichen Anfragen handelt (dies macht die Experimente vergleichbar).

Die Experimentierumgebung kann außerdem die Start- und Zielpunkte so generieren, dass sie sich jeweils innerhalb eines definierten Bereiches der Karte befinden. Ein solcher Bereich ist der Einfachheit halber quadratisch, maximal so groß wie die Karte und minimal so groß wie ein Punkt der Karte. Der Bereich für die Startpunkte ist genauso groß, wie der für die Zielpunkte, kann sich allerdings in einem anderen Bereich der Karte befinden (genauere Einstellungen waren nicht nötig). Dies habe ich beim Debugging genutzt, um auf einer Karte (mit wenigen Hindernissen) die Weglängen beeinflussen zu können und um zu prüfen, ob ein Weg von einem gewählten Bereich in einen anderen gewählten Bereich korrekt berechnet wird und um eine Abhängigkeit der optimalen `preferedSize` von der Weglänge zu prüfen (siehe Abschnitt 7.2.3).

Es gibt zwei Funktionen zur Verteilung innerhalb der Bereiche, die genutzt werden können. Bei der einen verteilen sich die Zufallszahlen gleichmäßig über den gewählten Bereich und bei der anderen handelt es sich um eine zweidimensionale Normalverteilung - das bedeutet, dass aus dem Zentrum des Bereichs "zufällig" mehr Punkte gewählt werden als am Rand. Diese Möglichkeit ist für Experimente mit *Cachingverfahren* gedacht und kann genutzt werden, um herauszustellen wie oft Wege aus dem *Cache* verwendet werden können, wenn in einigen Bereichen der Karte gehäuft Wege beginnen oder enden.

Die Experimentierumgebung kann mit und ohne grafische Benutzeroberfläche (**GUI** - englisch: graphical user interface) genutzt werden. In jedem Fall werden folgende Parameter benötigt:

- die Anzahl der gewünschten *Pathfinding-Anfragen*
- die Größe der Bereiche in denen Start- und Zielpunkte liegen sollen
- x,y-Koordinaten des jeweils am nähesten am Ursprung des Koordinatensystems (0,0) liegenden Eckpunkt des Bereichs
- die Funktion, welche genutzt werden soll, um die Verteilung der Punkte zu berechnen (Normalverteilung oder gleichmäßige Verteilung)

Die Variante ohne *GUI* dient zum Messen von Laufzeiten und hat einen weiteren Parameter (den die Variante mit *GUI* nicht hat): die Anzahl der *Frames*, die durchgeführt werden sollen. Diese *Frames* werden an einem Stück durchgeführt und das Programm beendet sich anschließend. Die Parameter müssen in dieser Variante in einer Textdatei angegeben werden.

6.1 GUI

Die *GUI* (siehe Abbildung 19) besteht aus zwei Teilen: Im linken Teil befindet sich unter anderem die Möglichkeit Parameter und die Anzeigen der *GUI* einzustellen, sowie die Berechnungen zu starten (Button "Start") oder ein Reset durchzuführen (Button "Reset"). Das Reset ermöglicht eine neue Suche, ohne dass das Programm neu gestartet werden muss - die Parameter können dabei jedoch verändert werden.

Es wird immer nur ein *Frame* ausgeführt, danach muss auf den Button "Tick" geklickt werden, damit der nächste *Frame* durchgeführt wird. Daher eignet sich diese Variante der Experimentierumgebung nicht zum messen der Berechnungsdauer mehrerer Schritte einer Wegsuche, aber dazu die Ergebnisse der Wegsuche schrittweise zu visualisieren.

Im rechten Teil der *GUI* wird immer die Karte dargestellt. Dabei werden (wiederum zur Vereinfachung der Implementierung) allerdings nur die unpassierbaren Punkte angezeigt (siehe Abbildung 19). Der *Hierarchiegraph* wird angezeigt, wenn die Checkbox "V.-Kanten An/Aus" ausgewählt ist (siehe Abbildung 20). Die Anzeige erfolgt für die ausgewählten *Level* über die Checkboxen "Level1", "Level2" und "Level3" (die Anzeige von weiteren *LevelN* wurde nicht benötigt). Außerdem können die berechneten Wege dargestellt werden (siehe Abbildung 21 und Abbildung 22). In Abbildung 21 ist ein erster grober Weg zu sehen und in Abbildung 22 der gleiche Weg einige Berechnungsschritte später und detaillierter.

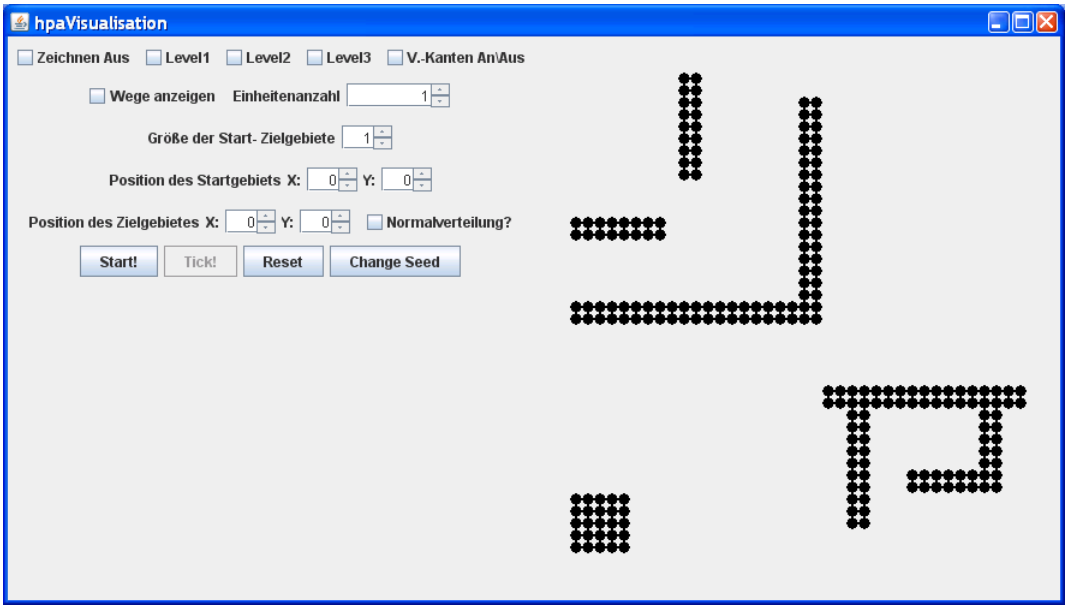


Abbildung 19: Die *GUI* der Experimentierumgebung. In der linken Hälfte können Einstellungen vorgenommen werden. In der rechten Hälfte wird die Karte dargestellt. Ein schwarzer Punkt ist unpassierbar, passierbare wurden nicht farbig dargestellt.

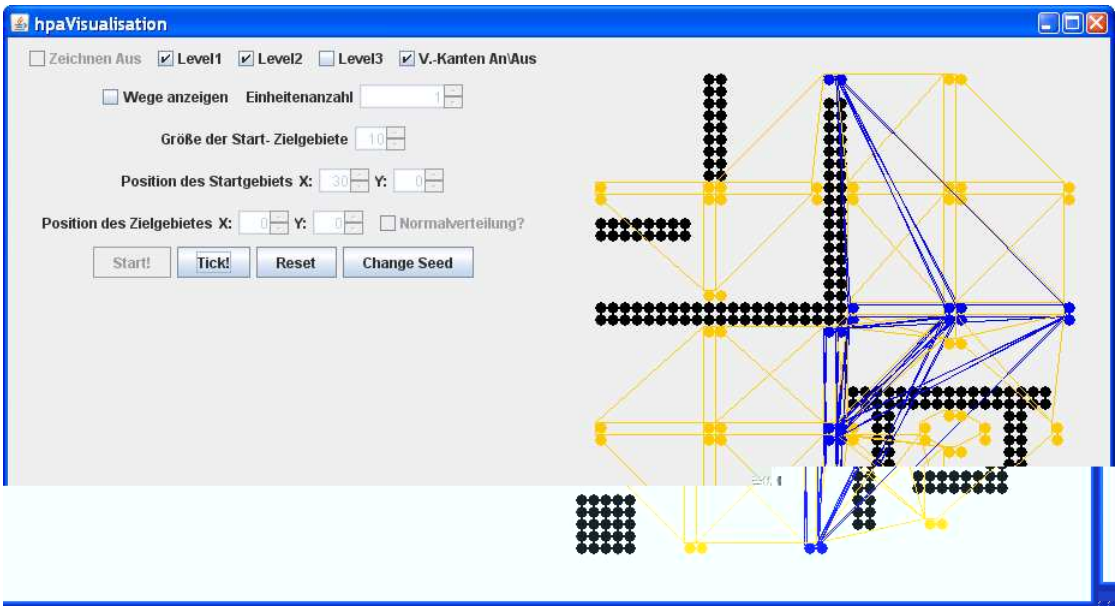


Abbildung 20: In dieser Darstellung ist in der rechten Hälfte nicht nur die Karte zu sehen, sondern es wurden auch *Knoten* und *Kanten* des *Hierarchiegraphen* dargestellt. *Hierarchielevel* 1 in orange und *Hierarchielevel* 2 in blau.

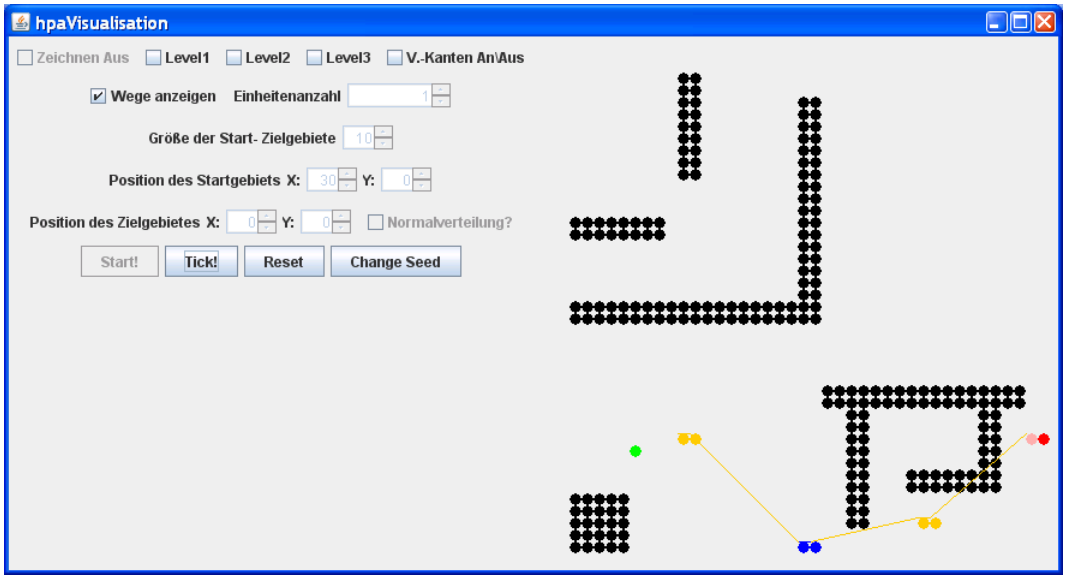


Abbildung 21: Hier ist ein erster grober Weg zu sehen. *Knoten* und *Kanten*, die zu diesem Weg gehören sind orange (*Level* 1) und blau (*Level* 2) dargestellt. Der Startpunkt ist rot, der Zielpunkt grün. Der in diesem Fall einzige berechnete detaillierte Wegpunkt (außer Start- und Endpunkt) ist in pink dargestellt.

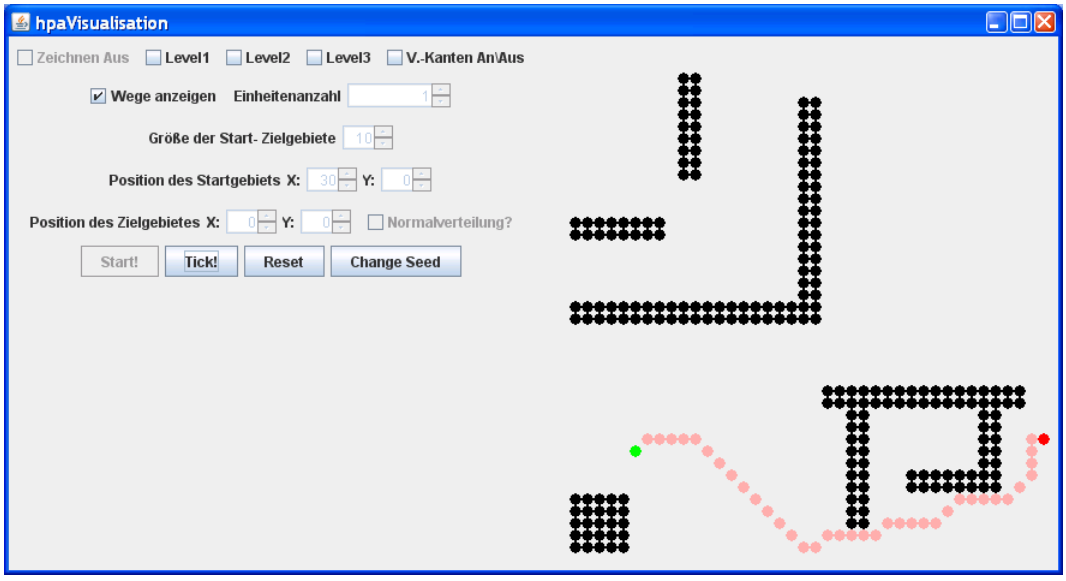


Abbildung 22: Abgebildet ist ein detaillierte Weg mit identischen Start- und Zielpunkte wie in Abbildung 21. Die Bedeutung der Farben entspricht denen aus Abbildung Abbildung 21

7 Experimente

Die Experimente dienen dazu zu bestimmen wie schnell die einzelnen Implementierungen bzw. Optimierungen des *hierarchischen Pathfinders* sind. Für die Experimente wurden die folgenden Testsysteme verwendet:

1. dual-core dual-processor AMD Opteron 2 GHz, 2 GB RAM
2. Intel T2400 (Centrino Duo, 1,83 GHz), 1,5 GB RAM

Die Experimente wurden sequentiell durchgeführt. Es wurde stets *Hierarchielevel* 0 bis 3 erzeugt.

7.1 Optimierung des Hierarchiegraphen

Bei diesem Experiment wurden zwei Karten der Größe 1.000*1.000 Punkte verwendet. Auf der einen Karte befindet sich kein Hindernis, in der Mitte der anderen Karte eine Mauer. Diese verläuft von oben nach unten, fast bist zu den Rändern der Karte. Auf beiden Karten wurde eine Einheit nahe des linken Randes in der Mitte der Karte platziert und ihr ein Zielpunkt nahe des rechten Randes in der Mitte der Karte zugewiesen, so dass diese Einheit auf der Karte mit der Mauer einen Weg darum benötigte. In meiner Diplomarbeit habe ich Messungen dazu für die Implementierung des *Hierarchiegraphen* durchgeführt, die mit *Kanten* arbeiten, welche sich auf mehreren *Hierarchieleveln* befinden (siehe Abschnitt 4.2.2 und Abschnitt 5.1). Die Ergebnisse sind in Tabelle 23 und Tabelle 24 zu sehen. Darin ist zu erkennen, dass der *hierarchische Pathfinder* bei zwei *Hierarchieebenen* am langsamsten ist - wünschenswert wäre allerdings, dass die Berechnungsdauer mit steigender Anzahl an *Hierarchieleveln monoton* fällt. Dieses Experiment hat untersucht, ob durch die Änderung des *Hierarchiegraphen* siehe Abschnitt 5.1 ein solches Verhalten erreicht wird. Messungen zeigten, dass dies nicht der Fall ist. Außerdem ist der *Pathfinder* mit beiden Implementierungen gleich schnell.

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

Abbildung 23: Suchdauer einer Einheit für den *ersten groben Weg* auf einer Karte mit einer langen Mauer.

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

Abbildung 24: Suchdauer einer Einheit für einen *ersten groben Weg* auf einer Wiese.

7.2 Optimierung der Open List

Die Experimente dieses Abschnitts dienen zur Ermittlung der schnellsten *Open List* Implementierung. Bevor dies jedoch angegeben werden kann, muss für die implementierten *teilsortierten Listen* ermittelt werden, bei welcher `preferedSize` sie jeweils am schnellsten sind.

Die verwendete Karte hat eine Größe von 1.000*1.000 Punkten - darauf befanden sich unterschiedliche Hindernisse. Es wurden jeweils 1.000 Einheiten erzeugt. Dabei wurde immer dieselbe Folge von zufällig gewählten Start- und Zielpunkten verwendet - mögliche Punkte waren alle betretbaren Punkte der Karte. Es wurde immer ausschließlich die Berechnung des *ersten groben Weges* durchgeführt.

7.2.1 Optimale preferedSize für ArrayList und TreeSet

In diesem Experiment wurde die Berechnungsdauer in Abhängigkeit der `preferedSize` untersucht. Die Ergebnisse mit Testsystem 2 sind in Abbildung 25 zu sehen. Die Kurven haben beide einen Tiefpunkt.

Ist die `preferedSize` kleiner als im Tiefpunkt, so werden `ArrayList` bzw. `TreeSet` häufiger leer (als im Tiefpunkt) und müssen neu gefüllt werden, dies kostet Zeit. Der Vorteil durch die Sortierung der Elemente mit niedrigen *Kosten* ist hingegen gering, da nur sehr wenige Elemente überhaupt sortiert werden. Das neu Füllen kostet in diesem Bereich mehr Zeit, als das Sortieren in `ArrayList` bzw. `TreeSet` spart und dadurch wird das Programm langsamer (als im Tiefpunkt). Ist die `preferedSize` im Gegensatz dazu größer als im o.g. Bereich, so steigt der Aufwand für das Einsortieren eines einzelnen Elements in den sortierten Teil. Außerdem wird beim letzten Auffüllen des sortierten Teils bei einer Wegsuche mit steigender `preferedSize` eine steigende Anzahl an Elementen einsortiert, die nicht mehr benötigt werden. Beides zusammen lässt bei einer `preferedSize` größer als im Tiefpunkt die Berechnungsdauer des Programms steigen.

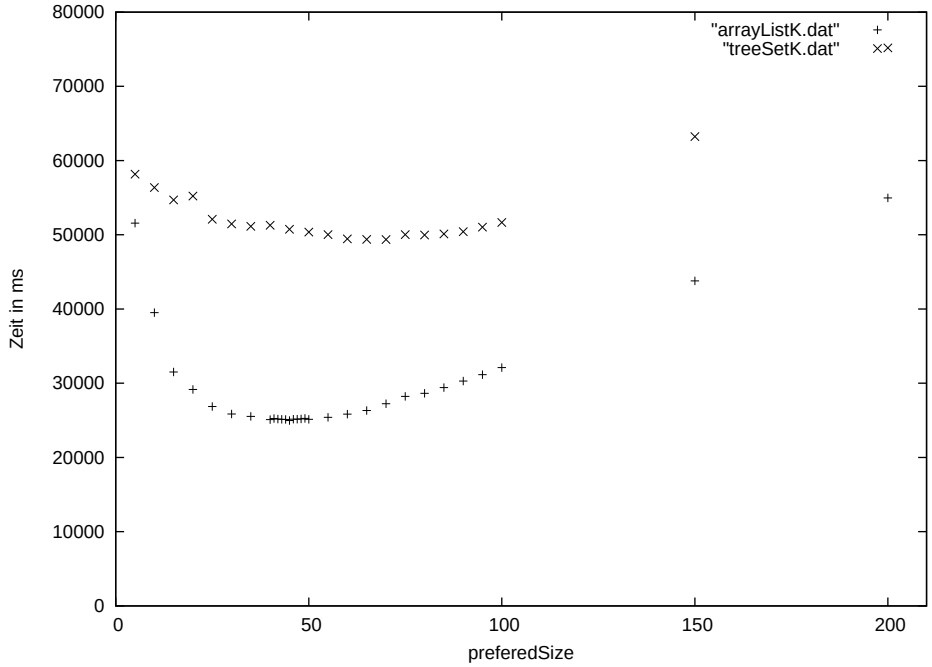


Abbildung 25: Berechnungsdauer in Abhängigkeit der `preferedSize` bei `cheapListWithTreeSet` und `cheapListWithArrayList` auf Testsystem 2.

Anmerkung: Es wurde zusätzlich ein Test auf Testsystem 1 durchgeführt (siehe Abbildung 26). Dabei entstand eine große Streuung der Werte bei gleicher `preferedSize`. Eine so große Streuung war auf Testsystem 2 nicht vorhanden. Es lässt sich jedoch erkennen, dass die Kurven einen ähnlichen Verlauf haben, wie auf Testsystem 2. Weitere Messungen konnten anschließend nicht mehr auf Testsystem 1 durchgeführt werden, da dies nicht mehr zur Verfügung stand.

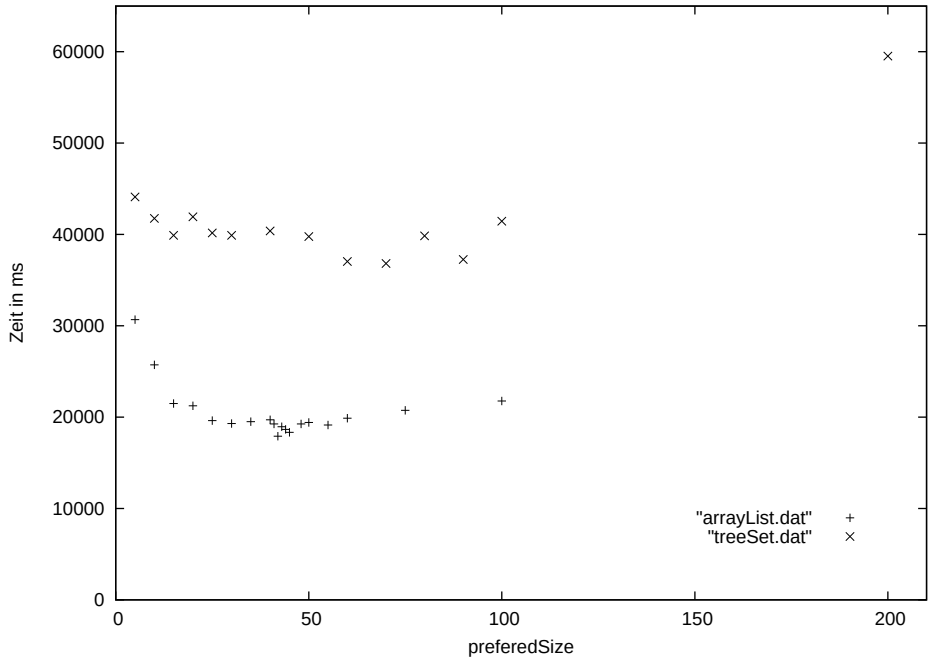


Abbildung 26: Berechnungsdauer in Abhängigkeit der `preferedSize` bei `cheapListWithTreeSet` und `cheapListWithArrayList` auf Testsystem 1.

7.2.2 Optimale preferedSize für die PriorityQueue

Ich habe die optimale `preferedSize` der `cheapListWithPriorityQueue` mit Testsystem 2 bestimmt. Diese Tests haben gezeigt, dass diese Implementierung schneller wird, wenn die `preferedSize` kleiner wird. Je größer der Einfluss der `HashMap` und je kleiner der der `PriorityQueue`, desto schneller wird die Implementierung. Weitere Schlussfolgerungen siehe Abschnitt 7.2.5

Vorhandene *Hierarchieebenen*	Suchdauer in ms
5	976.031
15	1.628.657
30	1.901.031

Abbildung 27: Berechnungsdauer in Abhängigkeit der `preferedSize` bei `cheapListWithPriorityQueue`.

7.2.3 Prüfung auf eine Abhängigkeit zwischen preferedSize und Weglänge

Es sollte ein möglicher Zusammenhangs zwischen der gesuchten Weglänge und der `preferedSize` bei `cheapListWithArrayList` und `cheapListWithTreeSet` geprüft werden. Dazu wurden die Werte aus Abschnitt 7.2.1 mit Testsystem 2 verwendet, sowie weitere Messungen vorgenommen. Bei den Tests in Abschnitt 7.2.1 wurden die Start- und Zielpunkte zufällig auf die Karte verteilt, so dass beliebige Entfernungen (Luftlinie, maximal also 1.000 aufgrund der Kartengröße) zwischen Start- und Zielpunkt vorkommen konnten. Bei den neu durchgeführten Messungen wurden die Startpunkte auf einen willkürlich gewählten Bereich beschränkt und die Zielpunkte auf einen anderen willkürlich gewählten Bereich. Das einzig Entscheidende hierbei war, dass diese Bereiche so gewählt wurden, dass die minimale Entfernung (Luftlinie) erhöht wurde. Dies erhöht auch die

minimalen Weglängen. Die minimale Entfernung lag dadurch bei 400 Punkten und damit wesentlich höher, als in Abschnitt 7.2.1, wo sie 0 betrug. Die Ergebnisse sind vergleichend für `cheapListWithArrayList` in und `cheapListWithTreeSet` zu sehen.

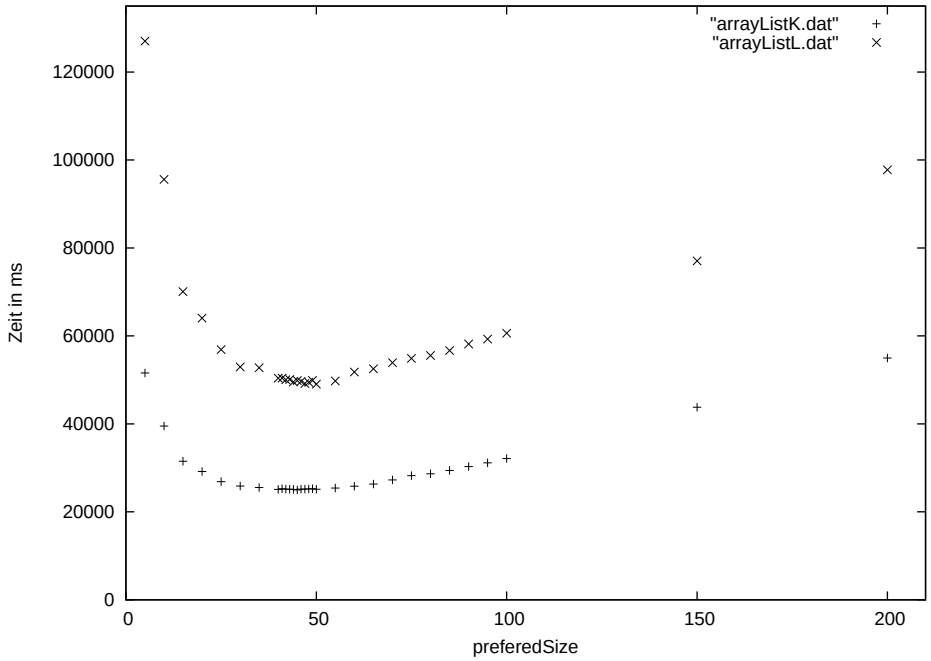


Abbildung 28: Vergleich zwischen Messungen der `cheapListWithArrayList` mit Weglängen mit Minimum 0 (arrayListK.dat) und Minimum 400 (arrayListL.dat).

Als Ergebnis lässt sich feststellen, dass die Berechnungsdauer für Wege mit höheren minimalen Weglängen steigt. Der Tiefpunkt ist jeweils bei derselben `preferedSize` und die Beträge der Steigung sind bei minimaler Entfernung von 400 größer. Bei Messungen von ausschließlich kurzen Weglängen ist zu erwarten, dass die Steigung sinkt: Wenn in die *Open List* bei einer Wegsuche beispielsweise nur 50 Punkte eingefügt werden, dann macht es keinen Unterschied, ob die `preferedSize` 50 oder 100 beträgt, denn bei einer Menge von 50 Punkten werden sich auch in einer Datenstruktur mit einem maximalen Volumen von 100 nur 50 Punkte befinden. Daraus folgt: je größer die Weglängen desto wichtiger wird es die optimale `preferedSize` zu verwenden.

7.2.4 Optimale preferedSize bei Verwendung von Buckets

Hier wird die `cheapListWithArrayListAndBuckets` behandelt. Auch hierbei konnte ich wie bei der `cheapListWithArrayList` einen ähnlichen Kurvenverlauf erahnen (es wurden nur wenige Werte gemessen - warum siehe Abschnitt 7.2.5). Ich habe 2 verschiedene Verteilungen getestet, um die Elemente den Buckets zuzuteilen. In beiden Varianten wurde als Maßstab die Schätzung vom Start zum Zielpunkt (h(S)) verwendet und die Elemente wurden ihren *Gesamtkosten* (f(k)) entsprechend eingeordnet.

1. Das Kostenintervall für ein Bucket ist immer gleich groß (Beispiel für Intervall 1 bis 3: [0,h(S)];]h(S),2*h(S)];]2*h(S),3*h(S)]).

2. Das Kostenintervall für ein Bucket wächst ständig mit dem Faktor 2. (Beispiel für Intervall 1 bis 3: [0,h(S)];]h(S), 2*h(S)];]2*h(S),4*h(S)])

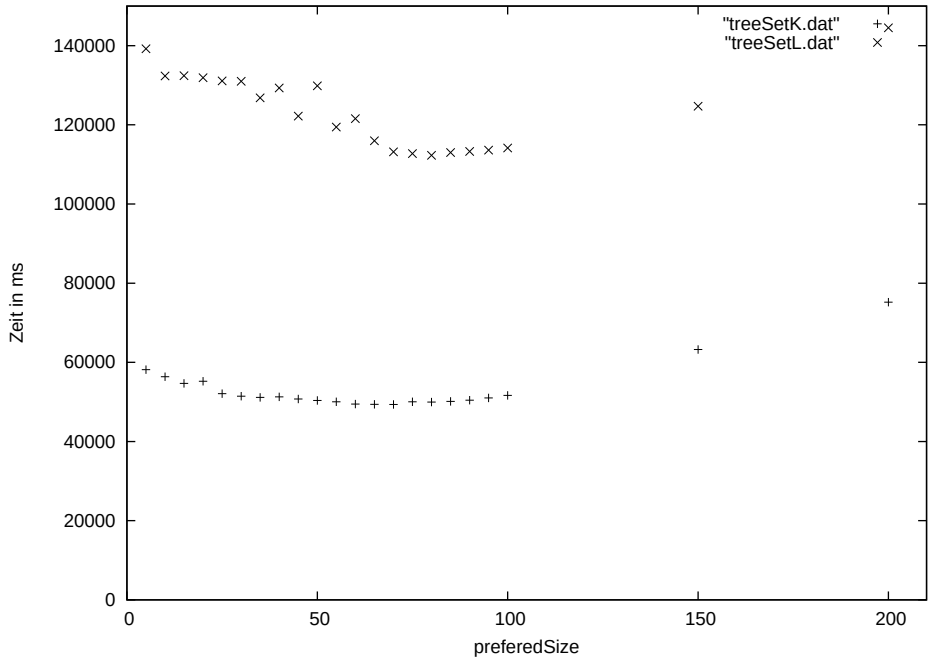


Abbildung 29: Vergleich zwischen Messungen der `cheapListWithTreeSet` mit Weglängen mit Minimum 0 (arraySetK.dat) und Minimum 400 (treeSetL.dat).

Variante 2 diente dazu, insbesondere die Sortierung von vermutlich relevanten *Knoten* zu vereinfachen, weil zum einen die Buckets dafür durch ein kleineres Intervall weniger *Knoten* enthalten; zum anderen, um Buckets größer zu machen und so die Prüfung, inwiefern sich ein Element in der `cheapListWithArrayList-AndBuckets` befindet, zu beschleunigen. Dies kann von Vorteil sein, da die Überprüfung, ob sich ein Element in einer `HashMap` A befindet, die doppelt so groß ist wie die `HashMaps` B und C möglicherweise mehr Zeit benötigt, als für beide `HashMaps` B und C.

7.2.5 Vergleich der Open List Implementierungen

Auf Testsystem 2 wurden die Implementierungen der *Open List* verglichen. Dabei wurde jeweils die beste `preferedSize` verwendet. Die Ergebnisse befinden sich in Tabelle 30. Da die Messungen zeigten, dass die `cheapListWithArrayList` mit Abstand die schnellste Implementierung ist, wurde bei `cheapListWithPri-orityQueue` und `cheapListWithArrayListAndBuckets` auf detaillierte Messungen und deren Darstellung verzichtet. Des Weiteren zeigen sie, dass das *Pathfinding* mit `cheapListWithArrayList` mehr als doppelt so schnell ist, als mit einer `HashMap`.

Implementierung	Zeit in s
`HashMap`	58
`cheapListWithArrayList`	25
`cheapListWithTreeSet`	49
`cheapListWithArrayListAndBuckets`	32
`cheapListWithPriorityQueue`	976

Abbildung 30: Vergleich der *Open List* Implementierungen.

7.2.6 Untersuchung der Geschwindigkeit der PriorityQueue

Dieses Experiment hatte das Ziel, herauszufinden, warum die Suche bei Verwendung einer `cheapListWithPriorityQueue` wesentlich länger dauert als mit `cheapListWithArrayList`.

Für die folgenden Messungen mit beiden `CheapList` Implementierungen wurde eine `preferedSize` von 45 eingestellt. Dies ist eine gute Einstellung für die `cheapListWithArrayList`, aber eine schlechte für die `cheapListWithPriorityQueue`. Dieser Wert wurde gewählt, um möglichst gut herausstellen zu können, bei welchen Funktionen die `cheapListWithPriorityQueue` schlechter abschneidet als die `cheapListWithArrayList`. Bei einer besseren `preferedSize` für `cheapListWithPriorityQueue` werden sich die Messwerte der einer `HashMap` annähern, da die Sortierung durch die `PriorityQueue` dann kaum noch genutzt wird.

Weiterhin wurden `NodeRecordAStar`-Objekte erstellt, deren Daten ($g(k)$, $f(k)$, *Knoten*) von einem Zufallsgenerator erzeugt wurden. Diese Daten haben daher nichts mit irgendeiner mir bekannten Karte gemeinsam.

Es wurden jeweils 50 Mal

1. dieselben 30 `NodeRecordAStar`-Objekte zu beiden Listen hinzugefügt.
2. 1.000 `NodeRecordAStar`-Objekte zu beiden Datenstrukturen hinzugefügt und nur die 50 kleinsten entfernt, womit ein Füllen des sortierten Teils erzwungen wurde.
3. 1.000 `NodeRecordAStar`-Objekte zu beiden Datenstrukturen hinzugefügt und anschließend Elemente in den Datenstrukturen gesucht.
4. 1.000 `NodeRecordAStar`-Objekte zu beiden Datenstrukturen hinzugefügt und die Elemente 5 bis 30 (sortiert nach *Kosten*) aus beiden entfernt.
5. 1.000 `NodeRecordAStar`-Objekte zu beiden Datenstrukturen hinzugefügt und die Elemente 30 bis 5 (sortiert nach *Kosten*) aus beiden entfernt.

Dies führte zu den Werten in Tabelle 31. Dabei fällt auf, dass die `cheapListWithArrayList` beim hinzufügen von Elementen und beim Füllen von Elementen wesentlich langsamer ist. Beim Suchen von Elementen ist beides etwa gleich schnell, die meisten Elemente befinden sich in der `HashMap` und daher ist der Unterschied gering. Schneller ist sie hingegen dann, wenn Elemente entfernt werden. Dies wirkt sich dann besonders stark aus, wenn die Elemente weit vorne entfernt werden - dies geschieht bei der Entfernung der Elemente 5 bis 30. Meine Implementierung der Wegsuche ist unter Verwendung der `cheapListWithArrayList` zusammen genommen schneller ist als mit `cheapListWithPriorityQueue`. Daher gehe ich davon aus, dass wesentlich häufiger ein Element aus dem sortierten Teil entfernt wird, als dass eine Füllung des sortierten Teils mit Elementen des unsortierten benötigt wird. Allerdings kann dies alleine nicht die starken Unterschiede der Implementierungen erklären. Ein Blick in die Implementierung der `ArrayList` und der `PriorityQueue` lieferte auch keine Erklärung dafür.

Beschreibung	μs mit ArrayList	μs mit PriorityQueue
30 Objekte hinzugefügt	7.849	5.904
Füllen	312.064	3.881
Suchzeit	74.827	72.694
Entfernen von 5 bis 30	297	693
Entfernen von 30 bis 5	507	728

Abbildung 31: Zeitbedarf der Funktionen `cheapListWithArrayList` vs. `cheapListWithPriorityQueue`.

7.3 Alternativalgorithmus zum A*

Der Alternativalgorithmus zum A* erwies sich als deutlich langsamer als der *A*-Algorithmus* selbst.

7.4 Amortisierung der Kosten für Pathfinding-Anfragen

Ein Experiment sollte zu diesem Zweck zeigen, dass die *Amortisierung* zu einer besseren Verteilung der Berechnungen über die *Frames* führt. Die Implementierung funktionierte, aber: Die Berechnung von *ersten groben Wegen* benötigten ein Vielfaches an Zeit im Verhältnis zu den Berechnungen der detaillierten Wegpunkte. Die Berechnungen von *ersten groben Wegen* können nicht vorgezogen werden, da diese in vorhergehenden *Frames* noch nicht bekannt sind. Ein Vorziehen der Berechnungen der detaillierten Wegpunkte ändert aufgrund des Verhältnisses kaum etwas bezüglich der Verteilung über die *Frames*. Die *Amortisierung* bringt daher keinen Vorteil.

7.5 Vorverarbeitung des höchsten Hierarchielevels

Diese Experimente wurden mit einer 160*160 Punkte großen Karte auf Testsystem 2 durchgeführt.

Das erste Experiment soll ermitteln, wie viele *Pathfinding-Anfragen* während der Vorverarbeitung durch die *intelligente* Speicherung gespart werden können. Im zweiten der beiden Experimente soll die Geschwindigkeit des *hierarchischen Pathfinders* in Verbindung mit der Vorverarbeitung des *höchsten Hierarchielevels* bestimmt werden. Außerdem soll dieses mit der Geschwindigkeit ohne Vorverarbeitung verglichen werden. In beiden Experimenten wird einen Abhängigkeit vom höchsten zuvor erzeugten *Hierarchielevel* untersucht. D.h. der *Hierarchiegraph* wurde mehrfach erzeugt und zwar mit unterschiedlich vielen *Hierarchieleveln*.

7.5.1 Gesparte Wegberechnungen während der Vorverarbeitung

Tabelle 32 zeigt, dass durch die Art der Speicherung (siehe Abschnitt 5.5.1) bei einem niedrigeren *Hierarchielevel* prozentual mehr Berechnungen eingespart werden können. Wollte man die mehrfache Anwendung des *A*-Algorithmus* mit dem *Floyd-Algorithmus* vergleichen, müsste daher berücksichtigt werden, wie viele *Hierarchielevel* verwendet werden.

Hierarchielevel	berechnete Wege in Prozent	berechnete Wege absolut
1	18	190.650
2	40	175.556
3	47	425.66

Abbildung 32: Diese Tabelle zeigt den Prozentsatz der Wege, die während der Vorverarbeitung berechnet werden mussten, in Abhängigkeit vom *Hierarchielevel*. Beispielsweise mussten in der Vorverarbeitung, die mit einem *Hierarchiegraphen* durchgeführt wurde, der nur *Hierarchielevel* 1 enthält, 18 Prozent der Anfragen berechnet werden. 82 Prozent der Berechnungen konnten durch die *intelligente Speicherung* eingespart werden.

7.5.2 Geschwindigkeit der Wegberechnung

Es wurden jeweils 10.000 Wegsuchen bearbeitet. Als *Open List* wurde die `HashMap` verwendet.

In Tabelle 33 wird die Berechnungsdauer für 10.000 Wegsuchen angegeben. Der *Hierarchiegraph* wurde für ein, zwei oder drei *Hierarchielevel* erstellt. Die Wege wurden mit und ohne Vorverarbeitung zur Vorberechnung von Wegen gesucht.

Während die Berechnung ohne Vorverarbeitung mit steigendem, höchsten *Hierarchielevel* schneller wird, wird die Berechnung mit Vorverarbeitung bei sinkendem *höchsten Hierarchielevel* schneller. Start- und Zielpunkt befinden sich bei einem niedrigem *Hierarchielevel* im Durchschnitt "näher" am ersten gespeicherten *Knoten* des Weges, als bei einem hohen *Hierarchielevel*. Je höher das höchste *Hierarchielevel* ist, desto mehr Berechnungen sind notwendig, um zum *Start-* und *Zielknoten* eines geeigneten gespeicherten Weges zu gelangen. Zwei Schritte der Suche ermitteln einen Weg vom *Level* 1 *Knoten* mit den geringsten *Kosten* in der Nähe von Start- und Zielpunkt zu dem *Knoten* auf dem *höchsten Hierarchielevel* mit den geringsten *Kosten* zu Start- und Zielpunkt. Würde man diese Schritte in der Implementierung entfernen, so wäre die Suche bei einem *Hierarchielevel* noch etwas schneller. Die Suche mit Vorherberechnung von Wegen mit einem *Hierarchielevel* ist etwa 1,7 mal schneller als die Suche ohne Vorherberechnung mit 3 *Hierarchieleveln*.

höchstes Hierarchielevel	Wegsuche ohne Vorverarbeitung	Wegsuche mit Vorverarbeitung
1	20.229	10.510
2	19.906	10.828
3	17.516	11.948

Abbildung 33: Diese Tabelle gibt die Dauer der Wegsuche mit und ohne Vorverarbeitung in ms für die *Hierarchielevel* 1 bis 3 an.

Die Zeit für die Suche mit Vorverarbeitung ist die Summe aus der Zeit für:

- Die Suche des Weges vom Startpunkt zu einem Knoten des *höchsten Hierarchielevels*.
- Die Suche des Weges vom Zielpunkt zu einem Knoten des *höchsten Hierarchielevels*.
- Die Konstruktion des Weges des *höchsten Hierarchielevels* aus den Daten in der gespeicherten Tabelle.
- Das Zusammenfügen der Teilwege.
- Das Suchen von Wegen, die nicht mit Hilfe der gespeicherten Tabelle berechnet werden, da Start- und Zielpunkt zu nah beieinander liegen.

Aus den Messungen ergibt sich, dass diese Summe so hoch ist, dass keine größere Beschleunigung erfolgte.

7.6 Caching

Das *Caching* konnte aus Zeitgründen nicht mehr untersucht werden. Es kann jedoch davon ausgegangen werden, dass mit *Caching* prinzipiell ein geringerer Geschwindigkeitsvorteil erreicht werden kann, als durch o.g. Vorverarbeitung. Hinzu kommt noch, dass beim *Caching* in einem parallelen Algorithmus zusätzliche Kosten durch die Synchronisation entstehen - die bei dem Laden von Ergebnissen der Vorverarbeitung nicht notwendig sind. Da es sich bei der Vorverarbeitung "nur" um eine Beschleunigung mit dem Faktor 1,7 handelt, vermute ich, dass *Caching* im parallelen Fall zu einem so geringen Geschwindigkeitsvorteil führt, dass sich der Einsatz nicht lohnt.

8 Schlussbemerkungen

Die Schlussbemerkungen fassen die Ergebnisse zusammen, geben einen Ausblick über weitere mögliche Untersuchungen und ziehen ein Fazit dieser Arbeit.

8.1 Zusammenfassung

Diese Arbeit hat die folgenden Fragestellungen behandelt:

1. Welche Anforderungen gibt es (zusätzlich zu den oben genannten) bei aktuellen Computerspielen?
2. Welche *Pathfinding-Verfahren* gibt es und wie arbeiten diese?
3. Welche *Pathfinding-Verfahren* werden in Spielen eingesetzt?
4. Wie kann ein *kürzester Weg* möglichst schnell gefunden werden?

In Kapitel 2 wurden die Grundlagen von Computerspielen, des *Pathfinding* und der Parallelverarbeitung vorgestellt.

Kapitel 3 gab Antworten auf die Fragen 1 bis 3. Es wurde darin ein Überblick über aktuelle *Pathfinding-Verfahren* gegeben und am Beispiel "Companie of Heroes" gezeigt, dass *hierarchische Pathfinder* in Computerspielen eingesetzt werden. Außerdem stellte sich heraus, dass auch in der professionellen Spieleentwicklung die Geschwindigkeit von *Pathfinding-Verfahren* weiterhin ein zentrales Thema bei der Entwicklung ist und Bedarf für schnellere Algorithmen besteht - jedenfalls ließ dies die Begrenzung der Menge an *Knoten* pro Weg auf 30 bis 40 in "Companie of Heroes" vermuten.

Die Entwicklungen von aktuellen *Pathfinding-Verfahren* und die Anforderungen bei "Companie of Heroes" zeigten, dass *dynamisches Pathfinding* ein weiterer Schwerpunkt der Entwicklung ist.

Kapitel 4 beschrieb einen *hierarchischen* inklusive Implementierung und Parallelisierung und gab die Ergebnisse von Experimenten mit diesem wieder. Es wurde erklärt, dass *hierarchisches Pathfinding* schneller ist, als *Pathfinding*, welches ausschließlich mit dem *A*-Algorithmus* arbeitet: Eine Suche auf einer weniger detaillierten *Hierarchieebene* kann aufgrund des geringeren Suchraums schneller durchgeführt werden, da Hindernisse bereits auf dieser Ebene umgangen werden und somit die folgenden detaillierteren Suchschritte deutlich weniger Rechenaufwand bedeuten.

In Kapitel 5 wurden verschiedene Versuche vorgestellt, um den *hierarchischen Pathfinder* zu optimieren. Bei diesen Versuchen handelte es sich um eine alternative Datenstruktur für den *Hierarchiegraphen*, eine Alternative zum *A*-Algorithmus* - den Fringe Search - *teilsortierte Listen*, eine Vorverarbeitung zur Berechnung der Wege auf dem *höchsten Hierarchielevel*, eine *Amortisierung* und *Caching*. Um diese Versuche auswerten zu können, wurde eine Experimentierumgebung erstellt, die in Kapitel 6 beschrieben wurde. Die Experimente (siehe Kapitel 7) führten zu den folgenden Ergebnissen:

- die Datenstrukturen für den *Hierarchiegraphen* sind gleich schnell.
- der *hierarchische Pathfinder* ist schneller, wenn der *A*-Algorithmus* als Grundalgorithmus verwendet wird (im Vergleich zum *Fringe Search*).
- die optimale `preferedSize` konnte für mehrere *teilsortierte Listen* bestimmt werden. Es stellte sich heraus, dass die Kombination aus `ArrayList` und `HashMap` zu den schnellsten Berechnungen führte.
- die *Amortisierung* führte nicht zu einer ausgeglicheneren Verteilung, der Berechnungen über die *Frames*, da die Berechnung des *ersten groben Weges* deutlich länger dauert, als die Bestimmung der detaillierten Wegpunkte. Außerdem kann die Berechnung des *ersten groben Weges* nicht vorgezogen werden, da die Anfragen in den vorhergehenden *Frames* nicht bekannt sind.
- die Vorverarbeitung der Wege auf dem *höchsten Hierarchielevel* konnte erfolgreich durchgeführt werden und führte in einem Experiment zu einer Beschleunigung der Berechnungen.

8.2 Ausblick

Dieser Ausblick beschreibt Ideen für weitere Untersuchung zum Thema *Pathfinding*:

- dynamisches Pathfinding: In Kapitel 3 hat sich herausgestellt, das die Spieleentwickler Interesse an *dynamischem Pathfinding* haben. Eine Möglichkeit *Pathfinding* weiter zu untersuchen wäre daher, *dynamische Pathfinding-Algorithmen* zu untersuchen. Insbesondere könnte man analysieren, wie ein *hierarchischer Pathfinder* in einer dynamischen Umgebung genutzt werden kann bzw. welche Änderungen am *HPA*-Algorithmus* notwendig sind, um ihn in einer solchen Umgebung einzusetzen.

- Wegqualität: *hierarchische Pathfinder* berechnen häufig Wege, die nicht natürlich wirken (siehe Abschnitt 3.1). Es wäre daher interessant, den *gesamten* Zeitaufwand für Berechnung eines Weges zu bestimmen und sowohl Wegqualität als auch Zeitaufwand in Abhängigkeit der verwendeten *Hierarchielevel* zu ermitteln. Der *gesamte* Zeitaufwand besteht aus der Zeit für die Berechnung eines *hierarchischen Pathfinders* und einer *Wegverbesserung* oder eines *Steering-Algorithmus*. Diese Ergebnisse könnte man zusätzlichmit denen anderer *Pathfinding-Algorithmen* vergleichen.

- außerdem lohnt eine genauere Untersuchung *teilsortierter Listen* **ohne feste Grenze**

8.2.1 Teilsortierte Listen ohne feste Grenze

Ich habe nur Experimente mit *teilsortierten Listen* durchgeführt, die eine feste Grenze für den sortierten Teil hatten. Wurde der sortierte Teil der Liste länger als die `preferedSize`, so wurden Elemente aus ihm entfernt. Es ist alternativ möglich keine Elemente aus dem sortierten Teil zu entfernen, wenn dieser länger als die `preferedSize` wird. Eingefügt werden die Elemente genauso, wie bei *teilsortierten Listen* mit einer festen Grenze. Wird ein Element in den sortierten Teil eingefügt, weil es geringere *Kosten* hat als ein schon darin befindliches Element und hat die Länge des sortierten Teils die `preferedSize` bereits erreicht, so wächst der sortierte Teil. Wird der sortierte Teil leer, so wird er mit `preferedSize` vielen Elementen aus dem unsortierten Teil aufgefüllt. Der Nachteil dabei ist, dass es keine maximale Größe für den sortierten Teil gibt und damit die *Kosten* für das Einfügen in diesen Bereich steigen. Es ist schwer abzuschätzen, welche Längen der sortierte Teil dabei erreicht. Ich gehe davon aus, dass dies auch von der Karte und der *Heuristik* abhängt.

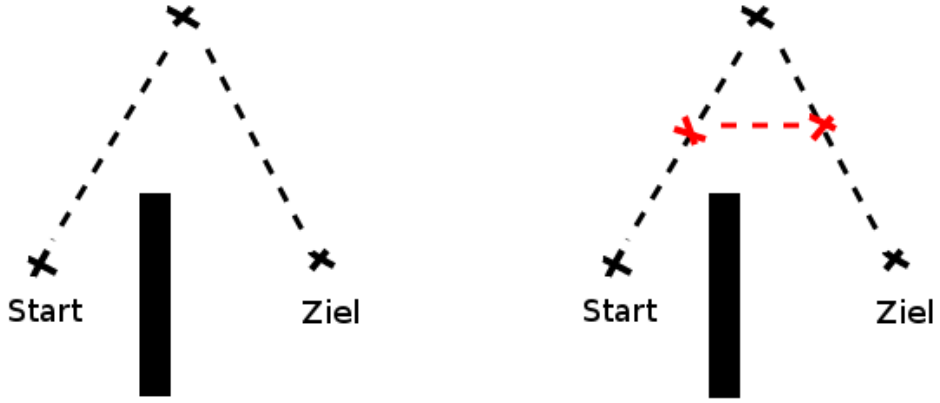
Die Implementierung bedeutet in diesem Fall einen geringen Aufwand, es müssten "nur" Experimente durchgeführt werden, um den Einfluss durch die Änderung zu bestimmen.

8.3 Fazit

Aufgrund der hier erzielten Ergebnisse würde ich folgenden *Pathfinder* implementieren, um eine möglichst hohe Geschwindigkeit zu erzielen: Einen parallelen *hierarchischen Pathfinder* mit nur zwei *Hierarchieleveln* - der Karte und einem *Hierarchielevel* darüber. Dieser *Pathfinder* soll in einer Vorverarbeitung Wege vorherberechnen und die Ergebnisse sollen auf einem Datenträger gespeichert und beim Laden des Programms oder der Karte ebenfalls geladen werden. Weiterhin sollen die Wege bereits weitestgehend während der Vorverarbeitung der Wege mit einem Algorithmus zur Rasterung von Linien verbessert werden.

Die Wege, die durch meine bisherigen Implementierungen berechnet wurden, wirken häufig nicht natürlich - ohne eine *Wegverbesserung* würde ich einen solchen *Pathfinder* nicht in einem professionellen Spiel einsetzen. Die *Wegverbesserung* kostet erneut Zeit und verlangsamt das *Pathfinding*. Wird eine *Wegverbesserung* auf Wege angewendet, sich die ausschließlich auf einem hohen *Hierarchielevel* befinden, so kann diese den Weg nur geringfügig verbessern, da auf diesem *Hierarchielevel* viele Details nicht bekannt sind. Abbildung 34 zeigt, dass eine *Wegverbesserung* angewendet auf den *Level* 2 Weg zu keiner Verbesserung führt, angewendet auf den *Level* 1 Weg jedoch schon.

Wird die *Wegverbesserung* schon während der Vorverarbeitung angewendet und werden dann die verbesserten Wege gespeichert, so spart dies während des Spiels nochmals Zeit. Geschieht dies für *Hierarchielevel* 1



(a) *Wegverbesserung* auf *Level* 2 (b) *Wegverbesserung* auf *Level* 1

Abbildung 34: Dargestellt wird eine Karte, auf der sich eine Mauer befindet. Mit schwarzen Kreuzen markiert sind alle *Level* 2 *Knoten*, die sich in diesem Bereich befinden. Über diese *Knoten* verläuft ein Weg, schwarz gestrichelt dargestellt. Auf diesem befinden sich zwei *Level* 1 *Knoten* (rote Kreuze), zwischen denen sich kein Hindernis befindet. Der Weg wäre kürzer, würde er direkt von einem *Level* 1 *Knoten* zum anderen laufen und der *Level* 2 *Knoten* oberhalb des Hindernisses nicht verwendet würde.

sind die Wege so natürlich wie mit einer Vorverarbeitung möglich (eine Speicherung aller möglichen Wege auf *Level* 0 würde zu viel Speicher benötigen, würde aber zu noch genaueren Wegen führen). Außerdem können die Algorithmen vereinfacht werden, wenn zur Karte nur *Hierarchielevel* 1 hinzugefügt wird, was zum einen zu verständlicherem Programmcode und zum anderen zu schnelleren Berechnungen mit *Hierarchielevel* 1 führen wird. Noch natürlicher können die Wege laut [14] dann wirken, wenn ein **Steering-Algorithmus**[26] verwendet wird. Die detaillierten Wege benötigen dann keine Berechnungen mehr durch *Pathfinding*, sondern die *Level* 1 *Knoten* werden an einen *Steering-Algorithmus* weitergegeben. Dies kann man sich so vorstellen, als würde man auf jeden *Level* 1 *Knoten* des Weges Magneten legen, die den *Agenten* anziehen.

Ich würde nur dann eine Variante mit mehr als zwei *Hierarchieleveln* verwenden, wenn sich herausstellen sollte, dass auch dann noch sehr schnell oder sogar schneller natürlich wirkende Wege bestimmt werden können und der Speicherverbrauch geringer ist, wenn auf der Grundlage von *Knoten* höherer *Level Steering* verwendet wird.

Literatur

[1] Dijkstra-Algorithmus, http://mandalex.manderby.com/d/dijkstra.php

[2] Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/

[3] 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, Canada

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

[5] David Silver, "Cooperative Pathfinding", 2005, AIIDE S.117-122, Edmonton, Canada

[6] Yngvi Björnson et al., "Fringe Search: Beating A* at Pathfinding on Game Maps", 2005, Reykjavik University, Reykjavik, Iceland

[7] Yngvi Björnson, Kári Halldórson, "Improved Heuristics for Optimal Pathfinding on Game Maps", 2006, Reykjavik University, Iceland

[8] D. Nieuwenhuisen et al, "Automatic Constuction Of Roadmaps For Path Planning In Games", 2004, Utrecht University, The Netherlands

[9] Xiaoxun Sun, Sven Koenig, "The Fringe-Saving A* Search Algorithm - A Fesibility Study, 2007, In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), S.2391-2397

[10] Maxim Likhachev, "Incremental Heuristic Search in Games: The Quest for Speed*", 2006, In Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), S.118-120

[11] D. Chris Rayner et al., "Prioritized-LRTA*: Speeding Up Learning via Prioritized Updates", 2006, Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search, Boston, Massachusetts, S.43-49

[12] Xiaoxun Sun, Sven Koenig, William Yeoh, "Generalized Adaptive A*", 2008, In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)

[13] Steve Rabin, "Ai Game Programming Wisdom I", 2002, Charles River Media Inc., Hingham, Massachussets, Kapitel 3 und 4

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

[15] Blizzard Entertainment, "Starcraft 2 Live Stage Video", 2007,
http://www.gamevideos.com/video/id/11671

[16] SEGA GmbH, "Medival II: Total War", http://www.totalwar.com

[17] Interview zu "Medieval: Total War", http://de.videogames.games.yahoo.com/pc/review/medieval-total-war-das-interview-9ee2d3.html

[18] "Flanking Total War's AI: 11 Tricks to Conquer for Your Game", http://aigamedev.com/reviews/total-war-ai

[19] Chris Jurney and Shelby Hubick, "Dealing with Destruction: AI From the Trenches of Company of Heroes", 2007, Game Developer Conference

[20] John O'Brien and Bryan Stout, "Embodied Agents in Dynamic Worlds", 2007, Game Developer Conference

[21] Heiko Waldschmidt, “Paralleler Java Pathfinder für ein Echtzeit-Strategie Spiel”, 2008, Diplomarbeit, Universität Kassel

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

[23] Sun Microsystems Inc., "Java Platform, Standard Edition 6 - API Specification", 2006, http://java.sun.com/javase/6/docs/api/

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

[25] J. E. Bresenham, “Algorithm for computer control of a digital plotter”, 1965, IBM Systems Journal, Jahrgang 4, Nr.1, S.25-30

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

[27] TU Kaiserslautern, “Floyd-Algorithmus”, http://www-bisor.wiwi.uni-kl.de/orwiki/index.php/Floyd_Algorithmus