Parameteroptimierung für die Computerspieleentwicklung

Benjamin S. Hofinger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im November 2016

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 30. November 2016

Benjamin S. Hofinger

Inhaltsverzeichnis

Kurzfassung

Diese Arbeit befasst sich mit den Überlegungen, wie ein Hilfsmittel für die Optimierung von Parametern für Game Designer und Softwareprogrammierer in der Spieleentwicklung aussehen könnte. Es wird erörtert, welche Eigenschaften so ein Tool haben muss und welche Algorithmen dafür in Frage kommen. Es werden zwei dieser Algorithmen ausgewählt und das Tool umgesetzt. Auch die Implementierung wird in dieser Arbeit beschrieben. Zum Testen auf Fehler in der Implementierung wurde ein Problem erstellt und zwei weitere, um für die Spieleentwicklung konkrete Probleme auf Geschwindigkeit, Genauigkeit sowie Umsetzungsdauer zu untersuchen. Eines dieser ist ein Balancing Problem und das andere ein Content-Generation Problem. Schlussendlich werden die Ergebnisse ausgewertet und besprochen.

Abstract

This paper looks at possibilities, how a tool for optimizing parameters in game development could be realized. It is discussing which properties such a tool needs and which algorithms can support those. Two of these algorithms are going to be chosen and implemented. A problem for testing the implementation and two concrete game development problems in balancing and content generation were created. Both algorithms try to solve these problems and their speed, accuracy and speed of implementation will be compared. At the end, the results are evaluated and discussed.

Kapitel 1

Einleitung

In der Softwareentwicklung gibt es für Programme oft Einstellungsmöglichkeiten. Diese reichen von Einstellungen für andere Betriebssysteme bis zu Benutzervorlieben. Und in der Spieleentwicklung gibt es diese auch. Hier sind es sogar Unmengen an Dingen, die eingestellt werden müssen. Und neben den Softwareentwicklern sind es hier vor allem auch Game Designer, die sehr viel Zeit mit dem Finden von richtigen Einstellungen verbringen.

1.1 Game Designer und Softwareentwickler

Der Game Designer wird Ideen für Features und Mechanismen notieren, die in seinen Augen für das jeweilige Spiel sinnvoll erscheinen. Er wird einige davon wahrscheinlich auch schon mit Papierprototypen, Prototypen oder anderweitig ausprobiert haben und feststellen, das hat Potential Spaß zu machen oder das Spiel anderweitig zu verbessern.[1] Der Programmierer muss nun in Zusammenarbeit mit dem Designer besprechen, welche Möglichkeiten es gibt, dieses Feature umzusetzen.[2] In manchen Fällen wird der Programmierer selbst nicht abschätzen können, wie gut die Lösung später funktionieren wird. Für welche Methode er sich auch entscheidet, sie wird verschiedene Vor- und Nachteile haben, die sich auf das Feature auswirken. Während der Entwicklung des Features wird der Programmierer auch einige Rahmenbedingungen festlegen müssen, damit das Feature umgesetzt werden kann. Dies kann unter anderem dadurch passieren, wie das System gestaltet ist oder welcher Algorithmus für das Feature gewählt wurde, aber auch durch die jeweiligen Parameter, die er für verschiedenste Werte festlegt, damit es überhaupt funktioniert. Wenn der Programmierer das Feature nun

[1] Es muss nicht immer mehr Spaß gewünscht sein, manchmal möchte der Game Designer den Spieler nachdenklich stimmen oder ihn Trauer über den Tod eines bestimmten Charakters nachempfinden lassen.

[2] Es kann natürlich auch sein, dass der Game Designer und der Softwareentwickler ein und die selbe Person sind.

fertig gestellt hat, wird der Designer es austesten wollen, ob das Feature so funktioniert, wie er sich das vorgestellt hat. Und häufig wird er erkennen, dass die Grundeinstellungen, die der Programmierer getroffen hat, nicht die sind, die er möchte. Er wird nun versuchen Werte zu finden, die das Spielerlebnis besser machen. Um diese zu finden wird er immer wieder andere Einstellungen verwenden und ausprobieren, ob diese mehr Spaß machen oder sinnvoller sind. Diese Suche kann in bestimmten Fällen lange dauern. Vor allem, wenn der Designer auch durch Wartezeiten gestört wird, wie zum Beispiel durch das erneute Kompilieren des Spieles oder bis er eine bestimmte Stelle im Spiel erreicht, wo er dieses Feature testen kann. Oder indem er auf den Programmierer warten muss, wenn nur dieser bestimmte Werte ändern kann. Oft könnte an diesen Werten noch viel länger getüftelt werden, jedoch ist die Zeit begrenzt, die er damit verbringen kann, da es noch andere Features im Spiel gibt, um die er sich kümmern muss. Er muss also irgendwann die Entscheidung treffen (oder sie wird vom Management für ihn getroffen), wann er aufhört bessere Werte zu finden. Dies kann im Idealfall natürlich dadurch sein, dass er keine besseren Werte mehr findet oder die jetzigen als ausreichend empfindet. Manchmal wird er aber auch nicht zufrieden sein und muss sich dann trotzdem um etwas anderes kümmern.

Es ist in der Spieleindustrie sehr wichtig früh zu wissen, ob eine Idee oder ein Feature Spaß machen, denn dadurch kann rechtzeitig reagiert werden, um Ressourcen von diesem Feature anderen zugute kommen zu lassen. Zum Beispiel werden die geplanten Grafiken doch nicht gebraucht und der Grafiker kann sich anderen widmen oder die Programmierer, die benötigt würden, um dieses Feature komplett fertigzustellen, können sich um etwas anderes kümmern. Deshalb ist es wichtig, dass der Game Designer so schnell wie möglich sagen kann: „Ja, das Feature bleibt.“ oder nicht. Wenn das Feature nun generell nicht passend ist, ist das auch gut so. Aber wenn er bei einem komplexeren Feature keine guten Werte findet, obwohl es welche gäbe, verliert das Spiel ein potentiell gutes Feature und die bisherige Arbeit, die investiert wurde. Es ist also wichtig für ihn, so schnell wie möglich gute Werte zu finden.

Vor allem zu Beginn der Entwicklung des Spieles werden Features getrennt von einander entwickelt. Und wenn diese dann zusammengeführt oder ins Spiel integriert werden, so dass sie sich gegenseitig beeinflussen, dann kann es passieren, dass die eingestellten Werte nicht zueinander passen. Der Designer muss nun abermals Werte finden – für beide Features – damit diese positiv miteinander interagieren. Manchmal sind das nur kleinere Anpassungen, manchmal ist mehr zu tun. Und wieder soll er dies so schnell wie möglich machen, denn es soll herausgefunden werden, ob diese Features zusammenpassen und ein einheitliches Spielerlebnis bieten. Je weiter die Entwicklung fortschreitet, desto mehr Werte können sich gegenseitig beeinflussen.

In der Entwicklung wird oft erst auch später festgestellt, dass ein Feature

umgeändert oder ergänzt werden soll. Es gibt wieder einiges anzupassen, was damit in Berührung steht. Viele Spiele werden nach Erscheinen mit Patches versorgt oder das ein oder andere Addon hinzugefügt. Auch DLC's werden immer häufiger. Viele FreeToPlay-Titel werden sogar kontinuierlich weiterentwickelt, um die Spieler am Spiel zu halten und Dinge hinzuzufügen, die teilweise auch gegen Microtransactions gekauft werden können. Es wird ersichtlich, dass dem Game Designer die Arbeit nicht so schnell ausgehen wird, und ein Teil seiner Arbeit ist das Anpassen von Werten.

1.2 Das Tool

Nun wäre es von Vorteil, wenn der Designer ein Hilfsmittel (Tool) hätte, welches ihm bei dieser Arbeit oder Teile dieser Arbeit unterstützt. So ein Hilfsmittel könnte ihm beim Finden der Werte helfen, in dem es verschiedene ausprobiert. Und natürlich sollte es diese Werte auch gleich testen und bewerten.

Beim Bewerten wird sich das Hilfsmittel schwer tun, denn wie wird Spielspaß gemessen? Dies ist vermutlich eine Frage die Computer noch länger nicht beantworten werden können. Ein Game Designer kann diese Frage nur durch Analyse der eigenen Emotionen und dem Beobachten und Fragen anderer Leute beim Spielen beantworten, wobei die Antwort hier natürlich immer noch subjektiv ist. Was ein Game Designer allerdings auch machen kann, ist, Vermutungen anzustellen, wie der Spielspaß erhöht werden könnte. Diese müssten in objektiven Zielen festgelegt werden. Und diese kann ein Computer überprüfen.

Wenn ein Computer prüfen kann, ob ein Ergebnis besser oder schlechter ist, dann kann der Designer das beste Ergebnis oder einige gute Ergebnisse auswählen und kontrollieren, ob dies tatsächlich mehr Spaß macht oder nicht. Er wird also wahrscheinlich immer noch Testen müssen, nur kann er somit schon eine Vorauswahl treffen und damit Zeit sparen.

Ein solch unterstütztes Testen ist natürlich ein Fortschritt, nur muss der Designer zwischen den Tests die Parameter verändern und dann wieder auf das Testergebnis warten. Dadurch wird er ein Gefühl bekommen, wie er die Werte verändern muss. Stattdessen könnte er auch ein großes Excel-Sheet mit verschiedenen Einstellungen füllen und wenn ihm ein Programmierer hilft, könnten diese eventuell der Reihe nach getestet werden. Während der Tests könnte er sich um etwas anderes kümmern und sobald alle durch sind wieder die besten testen. Hierzu muss er jedoch im Vorhinein die Einstellungen festlegen, was je nach Umfang mühsam sein kann. Wenn nun das Tool eigenständig die Werte in einem bestimmten Rahmen ändern könnte und selbst darauf schließen kann welche Änderungen zu guten Werten führen, wäre ihm diese Arbeit auch noch abgenommen.

Für größere Features kann es sein, dass wirklich ein Programmierer ein

Testsetup dafür schreibt. Nur wird er nicht für die vielen kleineren Features eines schreiben. Deshalb wäre es für den Designer gut, wenn es ein Tool gäbe, das nicht für eine bestimmte Aufgabe geschrieben wird sondern möglich universal eingesetzt werden kann. Auch sollte es möglichst einfach und benutzerfreundlich sein, damit erstens keine lange Einarbeitungszeit benötigt wird und zweitens der Designer auch möglichst viel selbst dort ändern kann, um mit so wenig Hilfe von Programmierern wie möglich auszukommen. Unser utopisches Tool sieht also so aus:

- Es ist für jedes Problem einsetzbar.
- Die Parameter werden selbst ausgesucht.
- Die Probleme werden anhand eines oder mehrere objektiven Kriterien getestet.
- Das optimale Ergebnis wird zurückgeliefert.
- Es benötigt so wenig Zeit wie möglich.
- Es wird keine lange Einarbeitungszeit benötigt.
- Von Programmierern wird so wenig Zeit wie möglich gebraucht.
- Für Designer ist es einfach zu bedienen.

So ein Tool wird es vermutlich längere Zeit nicht geben und ist auch außerhalb der Reichweite eines Masterarbeitsprojektes. Aber es wird versucht, dem so nahe wie möglich zu kommen.

Kapitel 2

Stand der Technik

2.1 Die Kriterien

Nun muss priorisiert werden, was aus der Liste des optimalen Tools am wichtigsten ist. Folgende Prioritäten wurden festgelegt:

1. Es ist für jedes Problem einsetzbar.
2. Die Probleme werden anhand eines oder mehrere objektiven Kriterien getestet.
3. Die Parameter werden selbst ausgesucht.
4. Für Designer ist es einfach zu bedienen.
5. Von Programmierern wird so wenig Zeit wie möglich gebraucht.
6. Es wird keine lange Einarbeitungszeit benötigt.
7. Das optimale Ergebnis wird zurückgeliefert.
8. Es benötigt so wenig Zeit wie möglich.

Aus folgenden Gründen:

1. Es für jedes Problem einzusetzen, wird nicht möglich sein, aber das Tool sollte auf so viele Probleme wie möglich angewandt werden können.
2. Das Testen ist oft der Großteil der Arbeit und sollte vom Tool gemacht werden.
3. Um die beschriebenen Pausen zwischen den Tests zu vermeiden, soll das Tool gleich selbst Werte suchen.
4. Es soll für Designer bedienbar sein und deshalb auch möglichst einfach gehalten sein.
5. Um so mehr Zeit von Programmierern benötigt wird, um das Tool auf ein Problem einzustellen, umso mehr reduziert sich die Einsetzbarkeit, vor allem für kleinere Anwendungsfälle. Jedoch wird es nicht möglich sein ohne Programmierkenntnisse auszukommen, da einige Dinge für jedes Problem spezifisch angepasst werden müssen.

6. Wie bei fünftens, reduziert eine Einarbeitungszeit die Einsatzwahrscheinlichkeit, deshalb sollte sie so gering wie möglich gehalten werden.
7. Aufgrund der vorherigen Punkte nehmen wir in Kauf, dass das Tool für das jeweilige Problem nicht das optimale Ergebnis liefert, so lange es gute Ergebnisse liefert.
8. Auch aufgrund der vorher erwähnten Punkte, wird nicht versucht jedes Problem mit der optimalen Strategie zu lösen, um das schnellste Ergebnis zu liefern. Es muss nur nach einer akzeptablen Zeit ein Ergebnis liefern, denn da das Tool ohne Benutzereingaben einfach rechnet, kann dieser sich um andere Dinge kümmern.

Welche Eigenschaften benötigen die Algorithmen für das Tool? Vorher werden folgende Begriffe erläutert. Die Zielfunktion ist jene Funktion, die überprüft, ob ein Ergebnis besser oder schlechter ist als ein anderes. Sie ist für jedes Problem zu bestimmen. Der Suchraum ist der Raum, in dem Lösungen gefunden werden können. Er kann konvex, nicht konvex und glatt oder nicht glatt sein. Eine Funktion ist konvex, wenn zwischen jedem Punkt auf ihr zu jedem anderen Punkt eine Linie gezogen werden kann, die entweder über oder an ihrem Graphen liegt [22]. Das bedeutet auch, dass das lokale Optimum das globale Optimum ist. „Eine Funktion ist glatt, wenn sie unendlich oft differenzierbar (insbesondere stetig) ist. Die Bezeichnung „glatt“ ist durch die Anschauung motiviert: Der Graph einer glatten Funktion hat keine „Ecken“, also Stellen, an der sie nicht differenzierbar ist [19].“

2.2 Ableitungsfrei

2.2.1 Gradient Ascent und Gradient Descent

Allgemeines

Es könnte von einer simplen Idee ausgegangen werden, welche folgendermaßen aussieht: Wir nehmen die Zielfunktion und berechnen ihre Ableitung. Und mit der Ableitung wäre die Steigung verfügbar, welche uns angibt, welche Parameter uns zu einem Optimum bringen. Die Parameter werden so lange verändert[1], bis wir eine Steigung von 0 haben. Wenn ein Minimum gesucht wird, wird der Algorithmus Gradient Descent genannt, bei einem Maximum heißt er Gradient Ascent. Ein Beispiel könnte wie in [20] aussehen: Es wird das Minimum von

$$\mathrm{f}(x) = x^4 - 3x^3 + 2 \tag{2.1}$$

gesucht, wo von sich die Ableitung

$$\mathrm{f}'(x) = 4x^3 - 9x^2 \tag{2.2}$$

[1] Die Veränderung kann in festgelegten fixen Schritten oder in sich anpassenden variablen erfolgen, dies ist jedoch ein eigenes Problem für sich.

ergibt. Auf diese wird der Gradient Descent Algorithmus aus [20] angewandt:

```
public class GradientDescent() {
  float xOld = 0;
  float xNew = 6;
  float gamma  = 0.01f;
  float precision = 0.00001f;

  public void calculateGradientDescent() {
    while(abs(xNew - xOld) > precision) {
      xOld = xNew;
      xNew = xOld - gamma * derivative(xOld);
    }
    System.out.println("Local minimum occurs at " + xNew);
  }

  private float derivative(float x) {
    return 4*x*x*x - 9*x*x;
  }
}
```

Der Nachteil

Ein großer Nachteil dieses Algorithmus ist, dass die Zielfunktion in einer mathematischen Notation benötigt wird, von welcher auch eine Ableitung gebildet und in angemessener Zeit berechnet werden kann.

Für kleinere Probleme wird es sich oft nicht rentieren, diese in eine mathematische Form zu bringen und eine Ableitung zu berechnen und bei komplexeren Problemen wird beides zu einer Herausforderung für einen Game Designer führen. Algorithmen die ohne Ableitung auskommen, haben ein großes praktisches Potential, auch ist es eines der wichtigsten, offenen und herausfordernden Gebiete in der Simulationswissenschaft [4]. Das Berechnen der Ableitung war eines der größten Fehlerquellen beim Anwenden dieser Algorithmen [4]:

> [...] we should remember that until relatively recently computing derivatives was the single most common source of user error in applying optimization software (see, [5]).

Von [7] wird das Problem so definiert:

> [...] the optimization of a deterministic function $f : \mathbb{R}^n \to \mathbb{R}$ over a domain of interest that possibly includes lower and upper bounds on the problem variables. We assume that the derivatives of f are neither symbolically nor numerically available [...].

So ein Problem wird als ableitungsfreie Optimierung bezeichnet und Algorithmen die solche Probleme lösen, werden als ableitungsfreie Algorithmen bezeichnet, selbst wenn sie Ableitungen für andere Funktionen als f beinhalten [7]. Es werden also Algorithmen benötigt, die ableitungsfrei sind.

2.3 Global oder Lokal

2.3.1 Hill-Climbing oder Bergsteigeralgorithmus

Allgemeines

Nun könnte von der nächsten einfachen Idee ausgegangen werden, welche einen Wert der Zielfunktion verändert und prüft, ob das Ergebnis verbessert wurde. Dieser Schritt wird so lange wiederholt bis keine besseren Ergebnisse mehr gefunden werden.

Dies wird Hill-Climbing oder Bergsteigeralgorithmus genannt. Der Name legt nahe, das mit diesem Algorithmus ein Maximum gefunden werden soll, durch Umkehren der Zielfunktion kann aber auch ein eigentliches Minimum gefunden werden.

Der Simple Hill-Climbing Algorithmus bewegt sich sofort in die erste Richtung in die eine Verbesserung festgestellt wird, während der Steepest Ascent Hill-Climbing Algorithmus sich in die Richtung mit der größten Verbesserung bewegt. Beide können sich jedoch nur entlang der Achsen bewegen – was zu Zick-Zack Mustern führen kann –, während der Gradient Descent Algorithmus die Ableitung berechnet und sich so direkt Richtung Minimum bewegt [21], siehe Abbildung 2.1.

Der Nachteil

Somit wäre zwar das Problem der Ableitung gelöst, jedoch wird dieser Algorithmus in lokalen Maxima feststecken, wenn der Suchraum nicht konvex ist, wie in Abbildung 2.2 zu sehen ist. Auch das Bilden von Zick-Zack Mustern bei manchen Suchräumen kann den Algorithmus sehr langsam machen.

Es gibt Algorithmen für die lokale und die globale Suche eines Optimums. Eine lokale Suche versucht so schnell wie möglich das ihr nächst gelegene Optimum zu finden, wie der Bergsteigeralgorithmus. Es kann aber sein, dass das gefundene Optimum nur ein lokales und nicht das globale Optimum. Eine globale Suche versucht aus lokalen Optima auszubrechen und an das globale Optimum zu kommen.

Für das Tool wäre es von Vorteil, wenn es das globale Optimum finden würde, deshalb kommen nur Algorithmen infrage, die das ermöglichen.

2.4 Stochastisch oder Deterministisch

2.4.1 Überblick

Deterministische Algorithmen versuchen eine Lösung zu finden, ohne dass sie auf Zufallselemente zurückgreifen und liefern somit immer das selbe Ergebnis, wenn die Eingabe gleich bleibt. Im Gegensatz dazu versuchen stochastische Algorithmen sich dem Optimum durch geschickte Nutzung von

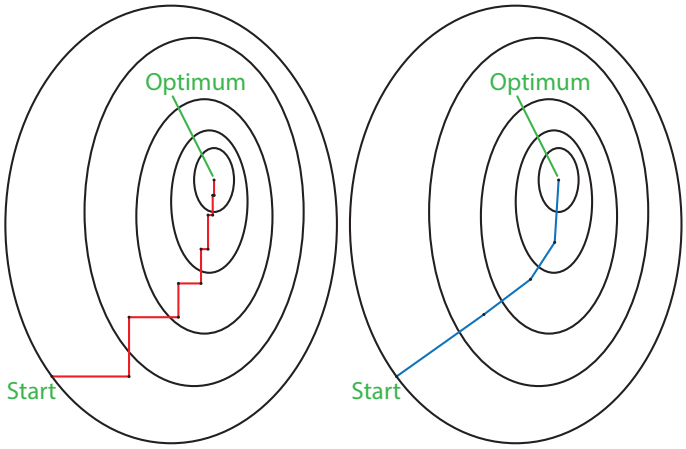


Abbildung 2.1: Der Unterschied zwischen Hill-Climbing (links) und Gradient Descent (rechts), auf einer zweidimensionalen Fläche. Der Hill-Climbing Algorithmus kann sich nur den Achsen entlang bewegen und bildet dadurch Zick-Zack Muster.

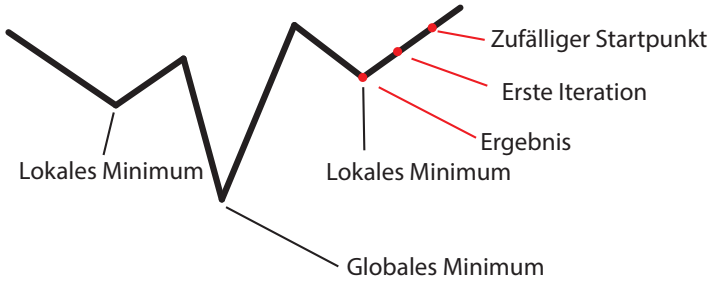


Abbildung 2.2: Angenommen, es wird das globale Minimum dieser zweidimensionalen, nicht-konvexen Funktion gesucht. Eine lokale Suche (wie die Gradient Suchen oder der Bergsteigeralgorithmus) wird nur ein lokales Optimum finden, wenn sie zufällig in seiner Nähe startet, sonst wird sie wie hier gezeigt auf ein lokales Minimum zusteuern.

Zufallselementen zu nähern. Sie liefern deshalb bei gleicher Eingabe unterschiedliche Ergebnisse zurück. Hier ist eine Übersicht über einige dieser Algorithmen aus [7]: Es gibt deterministische globale Suchalgorithmen, zu welchen Lipschitzian-based Partitioning Techniques und Multilevel Coordinate Search (MCS) zählen. Es gibt deterministische globale Suchalgorithmen, zu welchen Lipschitzian-based Partitioning Techniques und Multilevel Coordinate Search (MCS) zählen. Dann gibt es die globalen Model basierte Algorithmen wie Response Surface Methods (RSMs), Surrogate Management Framework (SMF) und Optimization by Branch-and-Fit. Stochastische glo-

bale Suchalgorithmen sind Hit-and-run Algorithms, Simulated Annealing, Genetic Algorithms und Particle Swarm Algorithms. Der Vorteil von stochastischen Algorithmen ist, dass sie vergleichsweise einfacher zu implementieren sind als ihre deterministischen Gegenstücke [7]. Deshalb werden diese genauer betrachtet. Es wäre ganz interessant zu sehen, welche von diesen am Geeignetsten ist, für das Masterprojekt steht aber nur begrenzt Zeit zur Verfügung und es wurde beschlossen, zwei auszuwählen und zu vergleichen. Da unbekannt ist, welcher dieser Algorithmen für das Anwendungsgebiet besser funktionieren wird, wurden der Genetische Algorithmus und der Simulated Annealing Algorithmus gewählt.

Der Simulated Annealing Algorithmus hat den Vorteil, dass er sehr leicht für einen „warmen“ Start verwendet werden kann. Wenn also nach einem Ergebnis gesucht und die Suche unterbrochen wird (zum Beispiel durch ein Abbruchkriterium), kann die Suche erneut dort gestartet werden, wo das beste Ergebnis der vorherigen war. Dies könnte auch ganz nützlich sein, wenn ein Feature nur leicht geändert wird, denn dann ist die neue Lösung eventuell nicht so weit von der alten entfernt.

2.5 Die ausgewählten Optimierungsalgorithmen

2.5.1 Simulated Annealing

Simulated Annealing (Simulierte Abkühlung) wurde von [6] vorgestellt. Die Autoren beschäftigten sich mit statistischer Mechanik und damit was passiert wenn man Materie abkühlt. Wird sie flüssig bleiben oder sich verfestigen? Und wenn sie sich verfestigt, wird sie eine kristalline Struktur bilden? Die meisten makroskopischen Körper befinden sich selten in ihrem Grundzustand. Der Grundzustand eines Körpers ist, wenn er am wenigsten Energie benötigt. Jedoch nähern sich auch makroskopische Körper ihrem Grundzustand an, wenn sie niedrige Temperaturen haben. In der Praxis reicht es jedoch nicht aus, den Körper allein durch eine niedrige Temperatur in seinen Grundzustand zu versetzten. Um zum Beispiel einen einzigen Kristall aus einer Substanz zu erhalten, muss dieser zuerst zu erhitzt werden, damit sie schmilzt. Danach muss sie langsam abzukühlen und sogar lange Zeit in Temperaturen nahe dem Schmelzpunkt verbringen. Denn wenn zu schnell abgekühlt wird, haben die Teilchen keine Zeit sich in ihren Grundzustand zu begeben und es werden kein oder nur ein defekter Kristall gebildet werden [6].

Der Grundgedanke ist, nicht wie beim Hill-Climbing / Gradient Descent Algorithmus, siehe Abbildung 2.2, nur immer in die Richtung einer Verbesserung zu gehen um das Minimum zu finden, da der Algorithmus so in lokalen Minima feststecken könnte. Stattdessen ist es dem Algorithmus auch erlaubt, mit gewisser Wahrscheinlichkeit auch schlechtere Positionen anzunehmen, um solchen Minima zu entkommen. Zu Beginn ist diese Wahr-

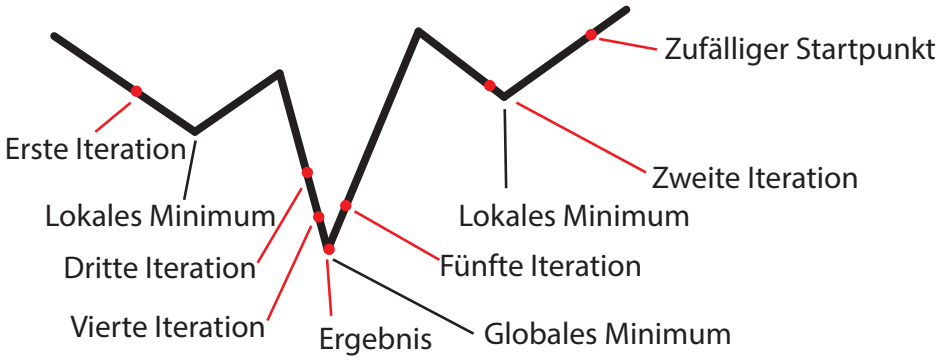


Abbildung 2.3: Durch Simulated Annealing kann aus einem lokalen Minima ausgebrochen werden. Zu Beginn mit einer hohen Temperatur nimmt man auch eher schlechtere Positionen an, bis sie so weit zurückgeht, dass man bessere Positionen erhält. Erreicht die Temperatur Null, bewegt man sich nur noch an besseren Positionen.

scheinlichkeit, aufgrund der hohen Temperatur, recht groß. Und da die Temperatur im laufe der Zeit sinkt, wird die Wahrscheinlichkeit geringer und es werden zusehends nur bessere Schritte zugelassen, siehe Abbildung 2.3. Wie ein Atom, das zu Beginn noch viel Energie hat um aus dem lokalen Minima zu entkommen, mit der Zeit an Temperatur verliert und sich nahe am globalen Minimum ansiedelt.

Der Algorithmus kann so zusammengefasst werden wie in Algorithmus 2.1 ersichtlich ist, wobei es Abwandlungen davon geben kann. Um lange, wahrscheinlich nicht mehr viel beitragende Berechnungen zu unterbinden, kann der Algorithmus auch begrenzt werden. Zum Beispiel durch das Begrenzen der Durchläufe für die while-Schleife, um somit nach einer maximalen Iteration nicht mehr weiter zurechnen. Auch kann es sein, dass schon ein gutes Ergebnis ausreicht und die Suche nach noch besseren Ergebnissen nicht nötig ist. Dies kann erreicht werden indem geprüft wird, ob die Kosten schon eine Grenze unterschreiten. Oft wird auch das beste erreichte Ergebnis mitgespeichert, da bei vorzeitigen Terminieren nicht garantiert wird, dass der letzte Wert der beste gefundene ist.

2.5.2 Genetische Algorithmen

Ein Genetischer Algorithmus benötigt eine Population. Diese Population besteht aus einzelnen Individuen oder „Chromosomen“. Diese können sowohl zufällig als auch heuristisch generiert werden. Ein Individuum ist ein Vorschlag für die Lösung des Problems. Die einzelnen Individuen werden mit der Zielfunktion (hier auch Fitnessfunktion genannt) bewertet. Anschließend folgt die Selektion, bei welcher bestimmt wird, welche Chromosomen

Algorithmus 2.1: Simulated Annealing. Abbruchbedingungen können das Erreichen einer Maximalen Iteration oder einer akzeptablen Lösung sein

```
Temperatur (T) festlegen
Abkühlungsrate (α) festlegen
Einen (zufälligen) Lösungskandidaten erstellen
while Abbruchbedingung nicht erfüllt do
    Die Kosten des Lösungskandidaten (KL) berechnen
    Einen Nachbarn des Lösungskandidaten erstellen
    Die Kosten des Nachbarn (KN) berechnen
    if KL < KN then
        Zum Nachbarn (neue Lösung) wechseln
    else
        Zufällige Zahl (Z) zwischen 0 und 1 generieren
        Akzeptanzwahrscheinlichkeit(KL, KN, T) (AW) ausrechnen
        T = T * α
        if Z < AW then
            Zum Nachbarn wechseln
        else
            Lösung beibehalten
        end if
    end if
end while

function Akzeptanzwahrscheinlichkeit(KL, KN, T)
    return exp((KL - KN) / T)
end function
```

für die Nachkommenserzeugung (Kreuzung) ausgewählt werden. Hierzu gibt es verschiedene Strategien, eine davon ist die Lösungskandidaten proportional zur Performance (oder hier auch Fitness-Wert) auszuwählen. Von den beiden ausgewählten Individuen werden Nachkommen erzeugt, dies können ein oder mehrere sein. Die Nachkommen werden so generiert, dass sie durch die Kombination von Teilen der Eltern entstehen. Daraufhin folgt die Mutation, welche zufällige Änderungen an einem Individuum hervorruft. Dadurch können neue Werte für den Problemraum gefunden werden. Die Wahrscheinlichkeit mit der eine Mutation auftreten kann, wird Mutationsrate genannt. Anschließend wird die jetzige Population mit der Nachkommensgeneration ausgetauscht. Hier zu gibt es auch unterschiedliche Strategien, eine davon ist der Elitismus, welcher einer gewissen Anzahl der besten Individuen erlaubt, automatisch in die nächste Generation zu wechseln. Dieser Vorgang wird mehrmals wiederholt, da ein Genetischer Algorithmus jedoch keine Konvergenz garantieren kann, werden oft auch Abbruchkriterien eingeführt. Diese

können entweder eine gewisse Anzahl an durchlaufenen Generationen, das Finden eines akzeptablen Ergebnisses oder ein Mechanismus zur frühzeitigen Erkennung von Konvergenz sein [1].

Algorithmus 2.2: Standard Arbeitsweise eines Genetischen Algorithmus aus [1].

Erstelle eine anfängliche Population an Individuen
Evaluiere die Fitness aller Individuen
while Abbruchbedingung nicht erfüllt **do**
    Wähle geeignetere Individuen für die Reproduktion aus und produziere neue Individuen (Kreuzung und Mutation)
    Evaluiere die Fitness neuer Individuen
    Generiere eine neue Population durch das Hinzufügen von einigen „guten“ und dem Löschen einiger „schlechter“ Individuen
end while

2.6 Andere Tools

2.6.1 Engines

Game Designer und Softwareentwickler arbeiten mit Engines, welche sie selbst erstellen oder vorhandene nutzen. Engines stellen gewisse Funktionalitäten zur Verfügung, wie zum Beispiel Box2D, welches Physikberechnungen für 2D Spiele ermöglicht. Andere sind umfangreicher und ermöglichen, ein ganzes Spiel mit der Engine zu erstellen. Logisch wäre also, wenn eine Engine so ein Feature bereitstellen würde. Das ist aber nicht der Fall, und weder kommerzielle Engines wie Unity3D, Unreal4, CryEngine noch open-source Engines wie jMonkeyEngine, Ogre3D, LibGDX oder Slick2D bieten so etwas an. Dabei hätte es einige Vorteile, wenn so ein Feature direkt in der Engine verbaut wäre. Denn ein externes Tool müsste die Werte die es einstellt an die Engine senden und diese müsste die Daten einlesen, die Tests durchführen und nach Berechnung der Zielfunktion das Ergebnis zurückschicken. Das würde zusätzlichen Aufwand für die Benutzung bedeuten, denn so eine Verbindung muss erst einmal umgesetzt werden und auch bedeutet es einen Performanceverlust. Zusätzlich könnte ein Teil an Einarbeitungszeit gespart werden, wenn die Engine ein Userinterface anbietet und das Tool dort integriert werden kann, da der Benutzer schon damit vertraut ist.

2.6.2 Wissenschaftliche Problemlöser

Es gibt wie in [7] beschrieben, verschiedene Softwareimplementierungen von Optimierungsalgorithmen, welche solche Probleme lösen können. Nur sind die meisten davon für erfahrene Benutzer konzipiert, die sich im Bereich der

Optimierung gut auskennen. Dies bedeutet eine längere Einarbeitungszeit von Spieleentwicklern, vor allem wenn die Grundeinstellungen anzupassen sind. Die meisten sind für Matlab verfügbar.

Kapitel 3

Implementierung

3.1 Beschreibung der Implementierung

Für das Tool BAGDAT (Broadly Applicable Game Developer Aiding Tool) wurde die Programmiersprache Java gewählt, welche laut [17] die mit Abstand am häufigsten verwendete Programmiersprache ist, wie auch in Abbildung 3.1 zu sehen ist.

Es gibt in Java drei große freie Engines, zum einen die LWJGL (Lightweight Java Game Library [16]), welche wie der Name sagt, nur das Nötigste bieten möchte. Zum anderen die jMonkeyEngine [14], welche jedoch nur eine Grafikengine sein möchte und zuletzt das LibGDX Framework [15], welches versucht, viele nützliche Features für das Entwickeln von Spielen bereitzustellen, aus denen ausgewählt werden kann. Von diesen würde BAGDAT am

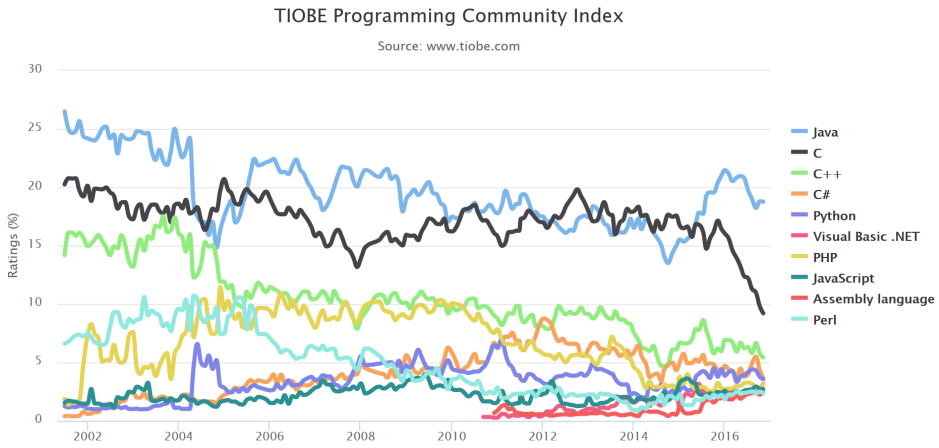


Abbildung 3.1: Dieser Index basiert auf Suchmaschinenanfragen welche auf „+ <language> programming“ lauten, mehr dazu unter [18]. Grafik aus [17].

besten ins LibGDX Framework passen, zunächst, soll es aber als LibGDX Projekt erstellt werden. LibGDX bietet auch eine 2D UI Library, welche für die UI Elemente in BAGDAT benutzt wird.

Es wird hier angemerkt, dass im Tool keine Verwendung von Threads gemacht wird, welche sich vor allem auf die Performance des Genetischen Algorithmus auswirken könnten.

3.2 Beschreibung zur Benutzung des Tools

Das Tool ist in ScreenStates gegliedert, die vom MenuScreenState aus erreichbar sind. Um neue ScreenStates hinzuzufügen muss eine entsprechende <State> Klasse erstellt werden (alternativ kann eine bestehende Beispiel State Klasse kopiert und angepasst werden) und im MenuScreenState ein Eintrag hinzugefügt werden, indem ein neuer Button erstellt und mit einem ClickListener für den neuen MyNewScreenState verbunden wird:

```
private TextButton MyNewStateApproach;
...

@Override
  public void Enter(Agat agat) {
  ...

  MyNewStateApproach = agat.createTextButtonAndAddToStage("Name", x, y,
    changeStateToMyNewStateClickListener());

  ...
}

...

private ClickListener changeStateToMyNewStateClickListener() {
    return new ClickListener() {
      public boolean touchDown(InputEvent event, float x, float y, int
    pointer, int button) {
        agat.stateMachine.ChangeState(MyNewScreenState.getInstance());

        return false;
      }
    };
  }
```

3.2.1 Benutzung des Genetischen Algorithmus in BAGDAT

High-Level Benutzung

Um die eingebaute GA-Variante zu benutzen, muss eine Klasse erstellt werden, die das Interface GAChromosome implementiert. Das Interface sieht folgendermaßen aus:

```
public interface GAChromosome {

  public float[] getAttributes();
  public float getPerformance();

  public void createRandomized();
  public void evaluate();
  public GAChromosome crossover(GAChromosome second);
  public void mutate(float mutationRate);

  public String toString();
}
```

Hierbei wird getAttributes() dazu benötigt, um die Attribute des Individuums abzufragen, getPerformance() soll den abgespeicherten Performance Wert zurückliefern, der in evaluate() berechnet wird. CreateRandomized() wird benötigt, um eine Population mit zufällig generierten Individuen zu erstellen, crossover() und mutate() werden für den Algorithmus benötigt und toString(), welches alle Informationen des Individuums als String liefern soll. Wenn so ein Chromosom erstellt wurde, muss eine Population davon erstellt werden:

```
Population population = new Population(MyGAChromosome.class,
    populationSize, mutationRate);
```

Und diese muss nun wie folgt aufgerufen werden:

```
population.create();
population.searchUntilGoodEnoughOrMaxGen(goodEnoughResult,
    maxGeneration);
```

Somit wird der Algorithmus so lange suchen, bis der Performance-Threshold (float zwischen 0 und 1) oder die maximale Generation (int) erreicht wurden. Alternativ dazu kann auch nur immer eine Generation vorangeschritten werden mit:

```
population.generationStep();
```

Anpassung

Für den Genetischen Algorithmus lassen sich hier die Werte der Populationsgröße und der Mutationsrate einstellen. Beide haben Einfluss darauf wie schnell der Algorithmus konvergiert. Eine große Population bedeutet, dass es schon viele Startlösungen gibt. Dadurch steigt die Wahrscheinlichkeit, dass eine gute Lösung darunter ist, jedoch auch die Rechenzeit pro Generation. Eine geringe Mutationsrate ermöglicht selten zufällige Änderungen, während eine hohe diese fördert. Mutationen ermöglichen es, neue Punkte im Problemraum zu finden.

Low-Level Benutzung

Falls der Benutzer eine andere Variante des GA benutzen möchte, hat er in der Population Klasse Zugang zu den Low-Level Methoden

- create(),
- crossover(),
- mutate(),
- exchange(),
- und evaluate().

Durch dieses Design hat der Benutzer sowohl Flexibilität als auch eine schnelle Möglichkeit den vorliegenden Genetischen Algorithmus zu benutzen.

3.2.2 Benutzung des Simulated Annealing in BAGDAT

Um den eingebauten Simulated Annealing Algorithmus zu benutzen, muss eine Klasse erstellt werden, die das Interface SAEntity implementiert. Das Interface sieht so aus:

```
1 public interface SAEntity {
2
3   public float[] getAttributes();
4   public float getPerformance();
5
6   public void evaluate();
7   public SAEntity getNeighbour();
8
9   public String toString();
10 }
```

Wobei hier wie beim Genetischen Algorithmus, bei getAttributes() die Werte der aktuellen Iteration zurückgegeben werden sollen und bei getPerformance() der abgespeicherte Performance Wert, der in evaluate() berechnet wird. GetNeighbour() soll zwischen den Nachbarn der aktuellen Iteration einen auswählen und zurückgeben. ToString() soll eine Beschreibung im String Format liefern. Nun muss noch ein SimulatedAnnealing Objekt mit folgenden Parametern erstellt werden (der startingGuess ist ein SAEntity):

```
1 SimulatedAnnelaing sa = new SimulatedAnnealing(startingTemperature,
      coolingRate, startingGuess);
```

Und mit folgendem Befehl ausgeführt werden:

```
1 sa.calcResult(goodEnoughResult, maxIteration);
```

Auch hier können einen Akzeptanz-Threshold und ein Threshold für die maximale Anzahl an Iterationen mitgegeben werden.

Abbildung 3.2: Die gesuchte Farbe.

Anpassung

Beim Simulated Annealing können die Starttemperatur, die Abkühlrate und die Startvermutung angepasst werden. Die Starttemperatur für den Algorithmus beträgt meist 1, während die Abkühlrate oft zwischen 0,8 und 0,99 liegt. Die Abkühlrate bestimmt, wie schnell der Algorithmus abkühlt. Je niedriger die Abkühlrate ist, desto eher werden schlechtere Ergebnisse nicht mehr akzeptiert. Die Startvermutung beeinflusst, wo der Algorithmus im Problemraum beginnt, kann also Auswirkung auf dessen Performance haben.

3.2.3 Benutzung von beiden Algorithmen (GA und SA) für das selbe Problem in BAGDAT

Falls der Benutzer beide Varianten für sein Problem anwenden möchte, kann er einfach eine Klasse mit beiden Interfaces, GAChromosome und SAEntity, erstellen. Wenn schon eines davon implementiert wurde, dann müssen für das andere nur mehr die Algorithmus spezifischen Methoden implementiert werden.

3.3 Implementierung der Testprobleme

3.3.1 Farbkugeln

Um zu sehen, ob die Algorithmen richtig funktionieren, wurde ein Testproblem erstellt, bei welchem ein Objekt mit drei Attributen (r, g, b) erstellt wird und diese bestimmte Werte erreichen sollen. Konkret wurden Farbkugeln erstellt, welche einen Rotwert von 0,8 (kann zwischen 0 und 1 liegen), einen Grünwert von 0,4 und einen Blauwert von 1 erhalten sollen, was eine Farbe wie in Abbildung 3.2 ersichtlich ist, ergibt.

Die Bewertung sieht wie folgt aus:

$$r' = 1 - abs(r - 0,8),$$
$$g' = 1 - abs(g - 0,4),$$
$$b' = 1 - abs(b - 1),$$

wobei wenn r', g' oder b' unter 0 fällt, es auf 0 gesetzt wird. Und der Performance,

$$p = \frac{r' + b' + c'}{3}.$$

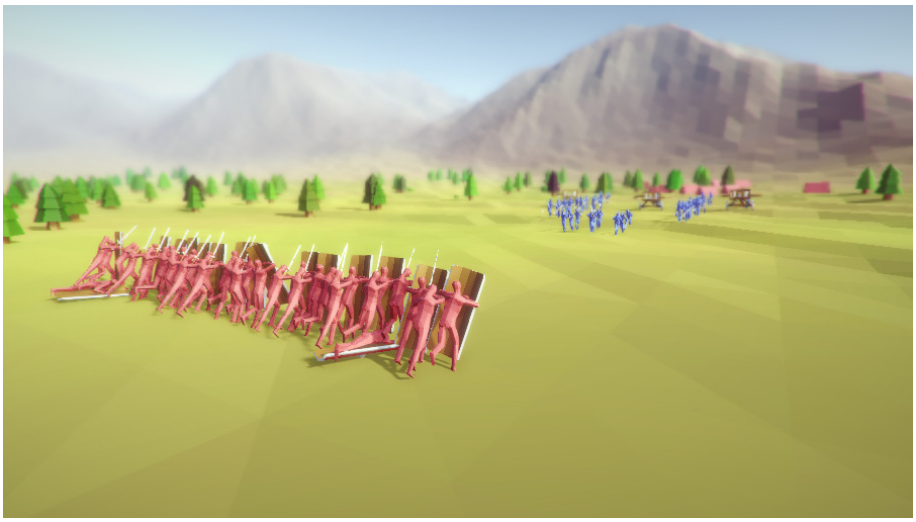


Abbildung 3.3: Eine Szene des Totally Accurate Battle Simulators, aus [9]

3.3.2 Armeezusammenstellung

Allgemeines

Oft geht es in Spielen um Balancing. Deshalb wird ein Balancingproblem untersucht. Eines dieser Probleme könnte so aussehen, dass ein Spieldesigner für ein Strategiespiel Level entwerfen soll, während das Spiel noch in Entwicklungsphase ist. Das bedeutet, dass sich das Balancing des Spiels als auch das Kampfsystem noch ändern können. Wenn er nun ein Level designed, könnte das aufgrund der Änderungen nicht mehr passen und er muss es wieder anpassen. Er kann aber auch nicht nichts tun, da er durch das Ausprobieren Feedback generiert, durch das die Änderungen ja auch teilweise entstehen können. Zum Beispiel, dass ein Einheitentyp zu stark ist und man seine Kampfwerte anpassen muss. Er möchte wahrscheinlich auch wissen, ob das Level überhaupt schaffbar ist und wenn ja wie viel Spielraum der Spieler mit seinen Ressourcen hat, was sich auf die Schwierigkeit des Levels auswirkt.

In vielen Strategiespielen kommt es häufig auch auf den Einheitenmix der Armee des Spielers an, ob gewonnen oder verloren wird. Denn meist haben bestimmte Einheitentypen einen Vorteil gegen andere. Dies machen sich gute Spieler natürlich zu Nutze und versuchen die bevorzugten Einheitentypen des Feindes zu kontern. Vor allem oft zu Beginn von Einzelspielerkampagnen sendet der Computergegner Armeen aus, deren Einheitenmix sehr einseitig ausfällt, um dem Spieler beizubringen wie er sich am Besten gegen diese wehrt. In Spielen wie Total Accurate Battle Simulator, siehe Abbildung 3.3, geht es nur um die Auswahl der Einheiten und deren Positionierung, be-

Tabelle 3.1: Kosten der Soldatentypen.

	Bogenschütze	*Ritter*	*Speerträger*
Holz	10	4	8
Stahl	2	10	3
Leder	5	0	10

Tabelle 3.2: Kampfattribute der Soldatentypen.

	Bogenschütze	*Ritter*	*Speerträger*
Angriff	6	7	5
Verteidigung	1	5	4
Lebenspunkte	8	20	12
Bonus vs Bogens.	0	4	0
Bonus vs Ritter	0	0	10
Bonus vs Speert.	4	0	0

grenzt durch die Kosten derer Einheiten. Und dies ist oft die Essenz: Wie kann mit den vorhandenen Ressourcen die feindliche Armee möglichst effizient geschlagen werden?

Da die Kampfsysteme in den Strategiespielen unterschiedlich und teils sehr komplex sind und für das Masterprojekt nur begrenzt Zeit zur Verfügung steht, wird ein eigenes einfaches Kampfsystem erstellt, welches jedoch die Grundproblematik enthält.

Das Kampfsystem

Es gibt einerseits die Armee des Feindes und die dem Spieler zur Verfügung stehenden Ressourcen. Ausgehend von den verfügbaren Ressourcen können bestimmte Soldatentypen gekauft werden, die jeweiligen Kosten sind in Tabelle 3.1 ersichtlich. Manche Soldatentypen sind stärker gegen andere, siehe Tabelle 3.2. Eine gute Armee hängt also davon ab wie die feindliche Armee zusammengestellt ist und welche Limitierungen es durch die Ressourcen gibt.

Der Kampf zwischen einzelnen Einheiten ist so abstrahiert:

$$Schaden = Angriff + Bonus - Verteidigung$$

und

$$Lebenspunkte_{neu} = Lebenspunkte - Schaden.$$

Wenn die Lebenspunkte unter 0 fallen scheidet die Einheit aus dem Kampf aus und kann keinen Schaden mehr verursachen.

Der Kampf besteht aus drei sich wiederholenden Phasen, bis eine Seite gewinnt. Beide Armeen verursachen den Schaden zeitgleich und Verluste finden erst am Ende der dritten Phase statt.

1. Bogenschützen Angriff: Verteilen den Schaden gleichmäßig auf Speerträger und Ritter, bis es keine mehr gibt, dann auf Bogenschützen.
2. Ritter Angriff: Verteilt den Schaden gleichmäßig auf alle Soldatentypen.
3. Speerträger Angriff: Verteilt den Schaden gleichmäßig auf Speerträger und Ritter, bis es keine mehr gibt, dann auf Bogenschützen.

Messung der Effizienz

Wenn nun verschiedene Armeekompositionen ausprobiert werden, muss verglichen werden welche sich besser schlägt. Die Effizienz wird mit

$$e = x \cdot 0.5 + y \cdot 0.5 \tag{3.1}$$

festgelegt.

Wenn unsere Armee die feindliche Armee nicht besiegen kann, wird verglichen wie viele Ressourcen der feindlichen Armee vernichtet wurden. Dazu liefert $\mathrm{k}(x)$ die Kosten in Ressourcen der Soldaten zurück. Die besiegten feindlichen Soldaten werden als b bezeichnet, alle feindlichen Soldaten als f. Daraus ergibt sich

$$x = \frac{\mathrm{k}(b)}{\mathrm{k}(f)}. \tag{3.2}$$

Wenn andererseits unsere Armee die feindliche Armee besiegen kann, wird verglichen wie viele Ressourcen unserer Armee übrig geblieben sind. Die überlebenden eigenen Soldaten werden als u bezeichnet, n steht für übrige Ressourcen die nicht für Einheiten genutzt werden konnte und a für die gesamte Ressourcen die zur Verfügung standen. Somit ist

$$y = \frac{\mathrm{k}(u) + n}{a}. \tag{3.3}$$

Somit bedeutet ein e von unter 50% dass die feindliche Armee nicht besiegt werden konnte, je höher e jedoch ist, umso besser hat sich die Armee geschlagen. Ein Wert von über 50% sagt, dass die feindliche Armee besiegt wurde und wie groß der Teil der überlebenden Armee ist. Ein Wert von 100% zeigt, dass die Armee des Spielers die feindliche Armee ohne Verluste besiegt hat, ist also wahrscheinlich schwer zu erreichen.

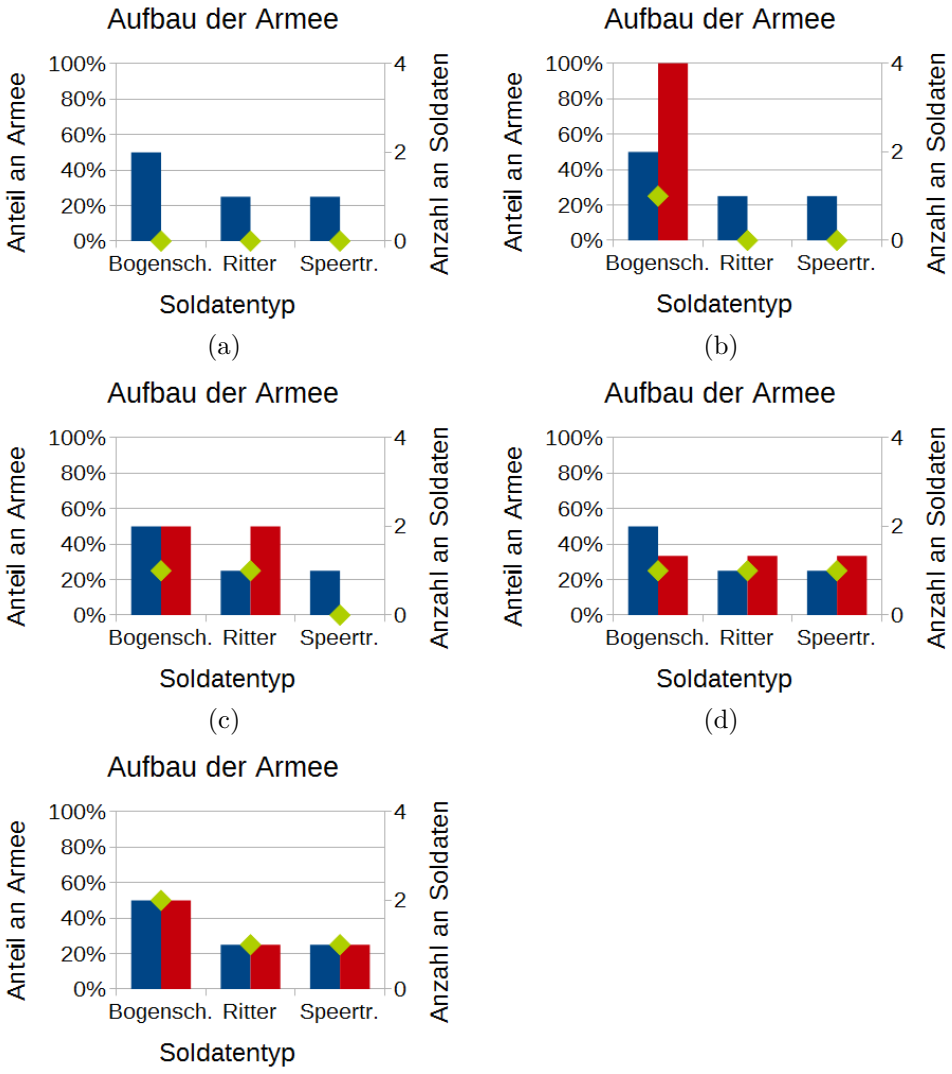


Abbildung 3.4: Blau ist der gewünschte Prozentsatz, rot der aktuelle und der grüne Punkt zeigt die absolute Anzahl des jeweiligen Soldatentyps an. Der gewünschte Prozentsatz ist natürlich für alle Abbildungen gleich. In (a) ist noch keine Einheit zugewiesen, also sind der aktuelle Prozentsatz und die Anzahl der Soldaten 0. In (b) wird der Einheitentyp mit der größten Differenz zwischen gewünschtem und aktuellem Prozentsatz erstellt, also ein Bogenschütze. Mit einem Bogenschützen ist der aktuelle Prozentsatz der 100% Bogenschützen, 0% Ritter und 0% Speerträger. In (c) wird ein Ritter hinzugefügt, also ist der aktuelle Prozentsatz für Bogenschützen und Ritter jeweils 50%. In (d) und (e) ist der weitere Vorgang zu sehen.

Erstellung einer Armee

Um nun eine solche Armee zu erstellen, wird für jeden Einheitentyp ein gewünschter Prozentsatz für den Anteil an der Gesamtarmee festgelegt. Also zum Beispiel 50% Bogenschützen, 25% Ritter und 25% Speerträger. Auch wird ein aktueller Prozentsatz festgelegt, welcher bei allen dreien mit 0% startet. Anschließend wird jeweils die Differenz zwischen aktuellem und gewünschtem Prozentsatz berechnet und für den Einheitentyp mit der größten Differenz wird versucht, einen Soldaten zu erstellen. Wenn zu wenig Ressourcen für diesen Typ vorhanden sind, wird versucht, der mit der nächst größten Differenz zu erstellen und wenn auch dies nicht möglich ist, wird versucht, den letzten zu erstellen. Danach wird der aktuelle Prozentsatz neu berechnet, die benötigten Ressourcen vom Pool abgezogen und wieder versucht, den gewünschten Soldaten zu erstellen, bis irgendwann keine Ressourcen mehr übrig sind. Dadurch wird möglichst eine Armee gebildet, die den gewünschten Prozentsatz hat, jedoch, wenn noch Ressourcen übrig bleiben, diese nutzt um die Armee mit zusätzlichen Soldaten zu verstärken. Teile des Vorgangs sind in Abbildung 3.4 ersichtlich.

3.3.3 Landkartenerstellung und -vergleich

Allgemeines

In Spielen werden auch häufig Inhalte dynamisch generiert, dies geht von Namen für Waffen, die in Spielen wie Diablo gefunden werden, bis hin zu ganzen Universen wie in No Man's Sky. Oft gibt es Faktoren wie diese Inhalte generiert werden und diese beeinflussen die Qualität davon. Deshalb wird ein Problem untersucht, bei dem Inhalte generiert werden.

In vielen Strategiespielen kann die Karte, auf der das Geschehen stattfindet, an die Präferenzen der Spieler angepasst werden, zB. Age of Empires, Age of Mythology, Rise of Nations, Civilization, Sins of a Solar Empire etc. (siehe Abbildung 3.5). Je nach Setting (Weltraum, Planetenoberfläche), gibt es natürlich andere Einstellungen. Es gibt bei erdähnlichen Karten oft Einstellungen wie Landmassen Größe (Pangea, Kontinente, große Inseln, kleine Inseln), Klima (arktisch, humid, trocken) oder Meeresspiegel. Dafür wird oft ein Kartengenerator erstellt, welcher dann mit verschiedenen Werten für die Generation gefüttert wird, um unterschiedliche Ergebnisse zu erzielen. Ob alle gewünschten Terrains damit generiert werden können, wird teilweise auch erst durch das Fertigstellen des Generators und das Anpassen der verfügbaren Parameter ersichtlich.

Kartenerstellung

Für das Masterprojekt wurde ein Kartengenerator erstellt, der Karten generiert, die auf Werte für Höhe, Temperatur und Feuchtigkeit basieren. Das

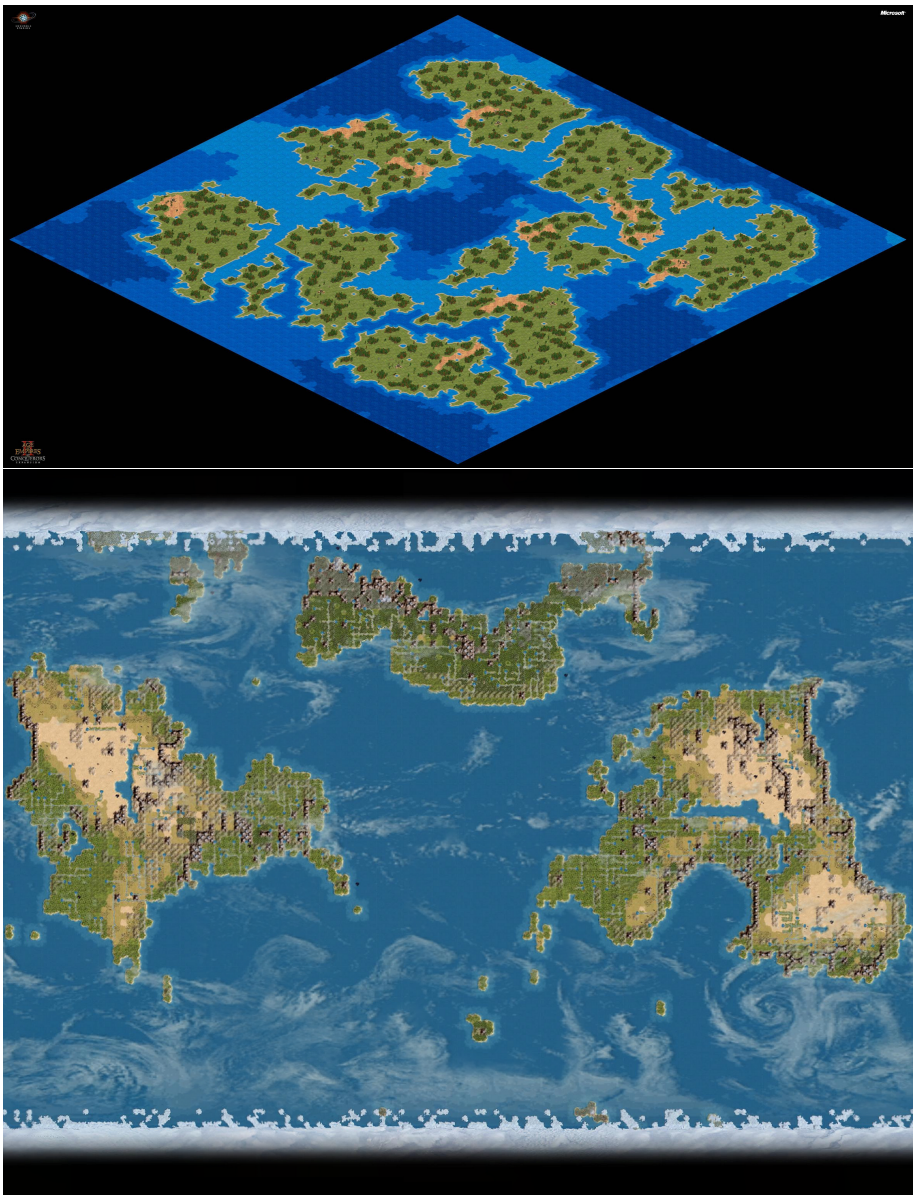


Abbildung 3.5: Generierte Karten von Age of Empires II HD [10] und einem Kartengenerator von vktj für Civilization 4 [11].

Erstellen der Karten funktioniert folgendermaßen:

Für diese drei Attribute werden jeweils eine Noise Map mit Werten zwischen 0 und 1 erstellt. Jede Noise Map basiert auf einem Value Noise und wird mit mehreren Oktaven desselben erstellt, siehe Abbildung 3.6 [12].

Um so einen Value Noise zu generieren, muss die Anzahl an Oktaven angeben werden, sinnvolle Werte für die Kartengenerierung liegen zwischen

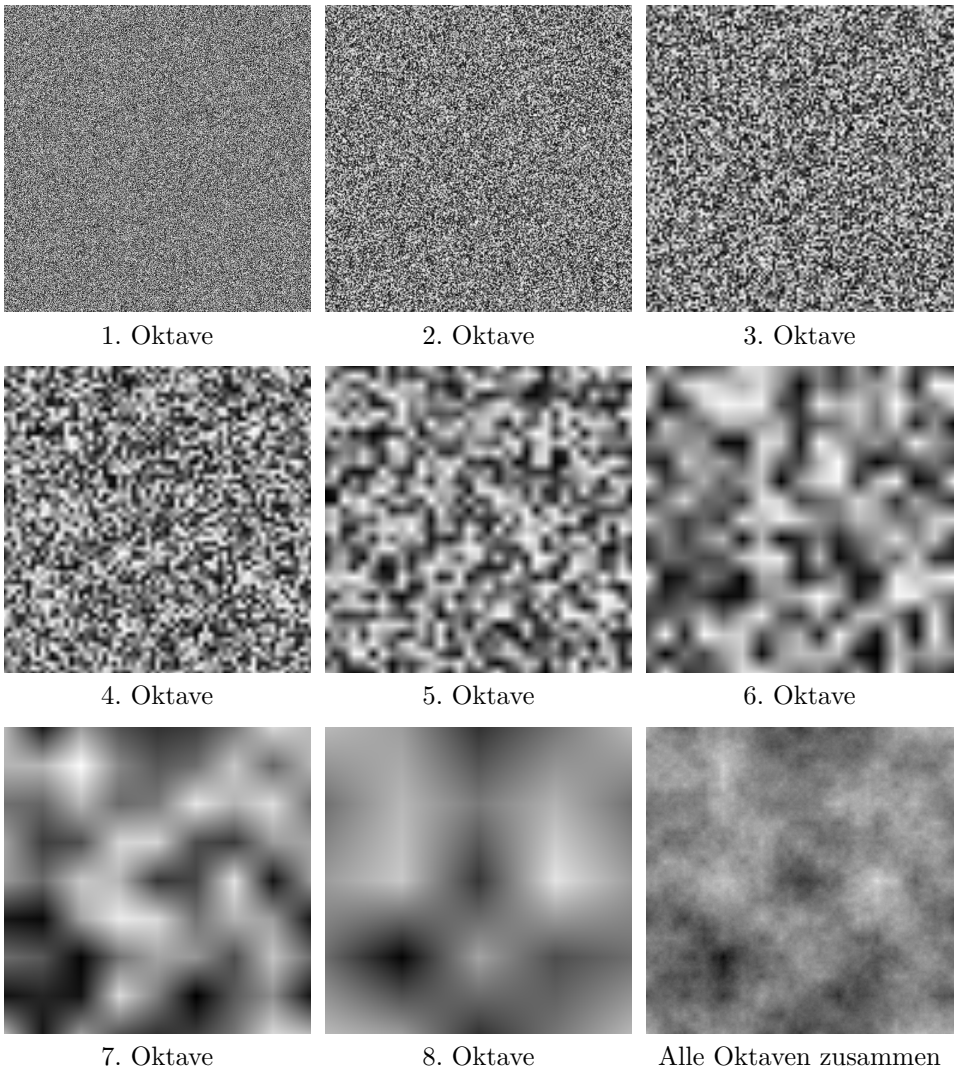


Abbildung 3.6: Die 8 Oktaven des selben Value Noises und ihr Ergebnis.

3 und 12. Für jede Oktave die hinzugefügt wird, wird eine neue Noise Map generiert, welche jedoch ungenauer ist. Statt für jeden Punkt einen Wert zwischen 0 und 1 festzulegen wird nur für jeden $2^{(Oktave-1)}$ Punkt ein Wert berechnet und die Punkte dazwischen linear interpoliert.

Zusätzlich zur Oktave muss auch noch ein Persistenz-Wert angegeben werden, welcher bestimmt, wie viel Gewichtung höheren Oktaven im Gegenzug zu niedrigeren gewährt wird. Ein Persistenz-Wert von 1, bedeutet, dass alle Oktaven eine gleiche Gewichtung haben, ein Wert von 1/4 bedeutet, dass die erste Oktave eine Gewichtung von 1 hat, die zweite von 1/4, die dritte von 1/16, usw. Anhand einer Funktion ist dies in Abbildung 3.7

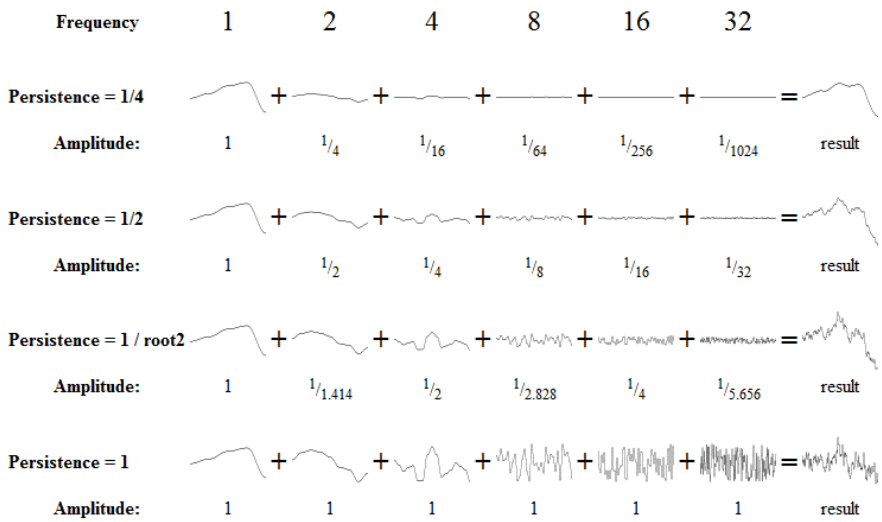


Abbildung 3.7: Hier ist ersichtlich, wie durch das Addieren von mehreren Funktionen (Oktaven) zu einer anderen Funktion, abhängig von der Persistenz unterschiedliche Ergebnisse erzielt werden [8].

zu sehen. Da für Höhe, Temperatur und Feuchtigkeit eine eigene Noise Map erstellt wird, werden jeweils Angaben für die Anzahl an Oktaven sowie der Persistenz benötigt. Für diese Karten wurden Biome wie in Abbildung 3.8 ersichtlich festgelegt. Nun werden die Pixel der Karte eingefärbt, basierend auf der Höhe und dem Biom, welches sich aus Abbildung 3.9 durch die Werte der Temperatur und Feuchtigkeit ergeben. Das Biom legt die Grundfarbe fest, welche mit der Höhe multipliziert wird (höher ist heller), um Schattierungen zu erzielen, siehe Abbildung 3.10. Wenn jedoch der Höhenwert unter einer bestimmten Grenze (Meeresspiegel) liegt, dann wird als Grundfarbe die Wasserfarbe herangezogen.

Kartenvergleich

Basierend auf dem Vergleichsbild (zB. Konzeptzeichnung oder einer händisch erstellten Vorlage) soll der Algorithmus Werte finden, damit der Generator möglichst ähnliche Karten erstellt. Der Vergleich wird durch zwei Methoden durchgeführt, einmal werden 3 2D Histogramme erstellt, siehe Abbildung 3.11. Danach auf ein 3×3 Raster gebinnt, siehe Abbildung 3.12, um ganz grob festzustellen, ob die Bilder ähnliche Farben haben [2]. Weiters werden Punkte berechnet, die sich ergeben, wenn der Kontrastunterschied zwischen benachbarten Pixel zu groß ist, siehe Abbildung 3.13, wie in [3] beschrieben. Die Menge dieser Punkte wird verglichen, um damit ganz grob die Struktur des Bildes zu vergleichen.

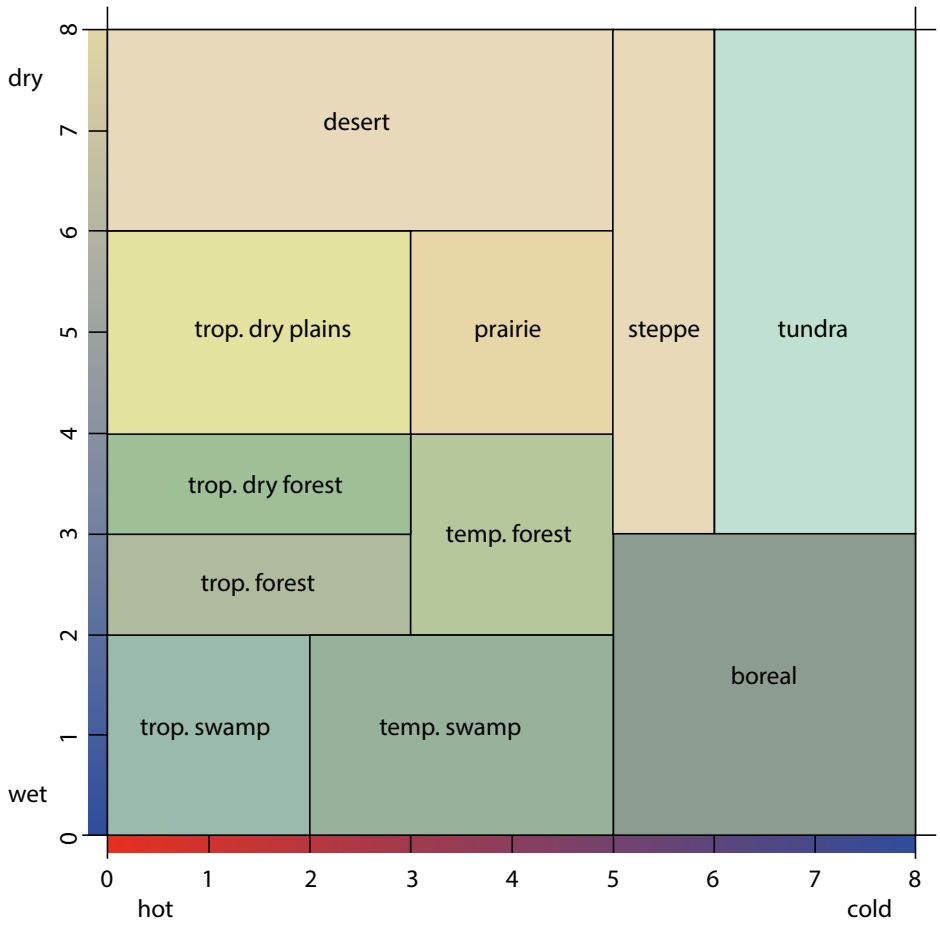


Abbildung 3.8: Einteilung der Biome aufgrund von Temperatur und Feuchtigkeit aus [13].

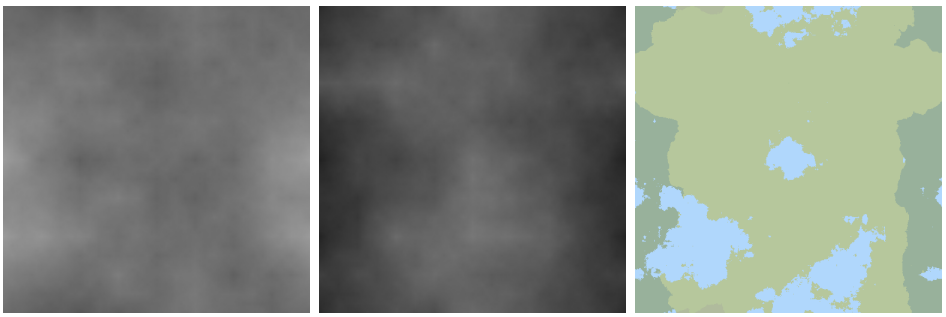


Abbildung 3.9: Zusammenfügen der Temperatur- und der Feuchtigkeitswerte zu Biomen, aufgrund der Tabelle in Abbildung 3.8.

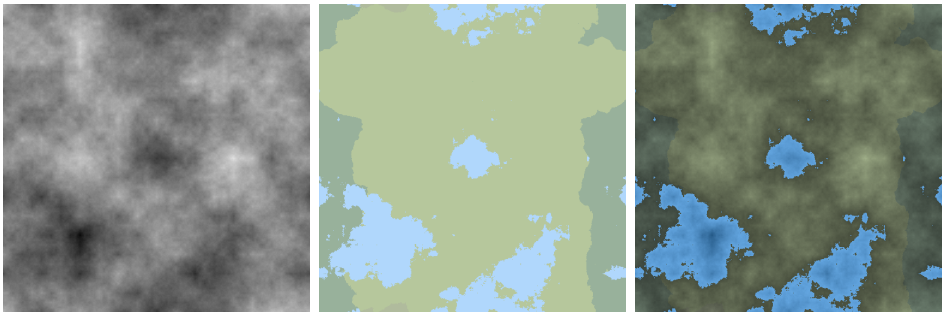


Abbildung 3.10: Zusammenfügen des Höhenwertes mit den Biomfarben zur fertigen Landkarte.

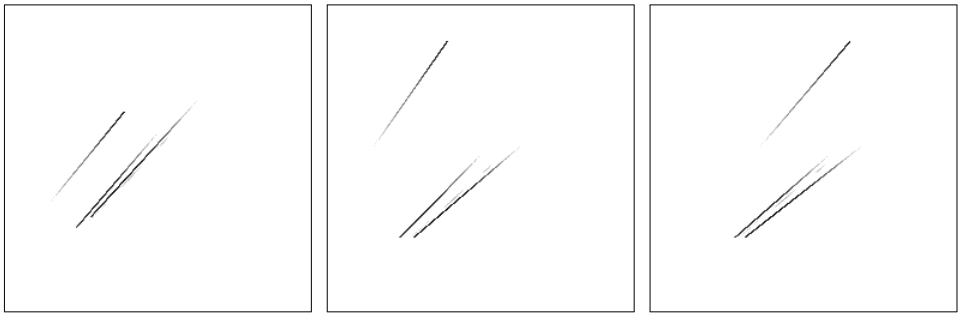


Abbildung 3.11: Die 2D Histogramme der fertigen Landkarte aus Abbildung 3.10.

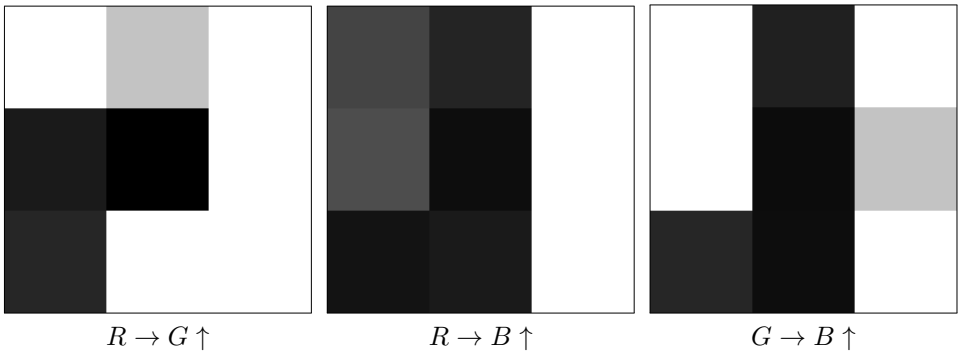


Abbildung 3.12: Die 2D Histogramme aus Abbildung 3.11 gebinnt auf ein 3×3 Grid.

Abbildung 3.13: Das Bild, seine Edgels und beides kombiniert.

Kapitel 4

Ergebnisse

4.1 Testsetup

Die Berechnungen wurden auf einem Acer Aspire VN7-791 durchgeführt, welcher einen Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz (mit 4 Kernen und 8 Threads) Prozessor enthält. Ein Beispieldurchlauf für den Genetischen Algorithmus und das Simulated Annealing sind in Abbildungen 4.1 und 4.2 zu sehen. Es wird für jedes Problem angeführt mit welchen Parametern es getestet wurde und wie die jeweiligen Algorithmus spezifischen Methoden umgesetzt wurden.

4.2 Farbkugeln

4.2.1 Test des Genetischen Algorithmus

Allgemeines

Der Genetische Algorithmus wurde mit einer Populationsgröße von 10, 102 sowie 1000 und einer Mutationsrate von 0,001, 0,01 und 0,1 auf das Problem angewandt. Die Abbruchkriterien waren ein Performance-Threshold von $> 98\%$ und einer Maximalgeneration von 100.000. Die Ergebnisse sind in Tabellen 4.1, 4.2 und 4.3 ersichtlich.

Selektion, Kreuzung und Mutation

Die Selektion findet proportional zur Performance statt, das bedeutet, dass Individuen mit höherer Performance eine höhere Chance haben für die Kreuzung ausgewählt zu werden. Nach der Auswahl wird für jedes Attribut zufällig ermittelt, ob es vom ersten oder zweiten Elternteil übernommen wird. Mit diesen Attributen wird dann ein neues Individuum erzeugt. Für die Mutation wird für jedes Attribut eine Zufallszahl erstellt und wenn diese unter

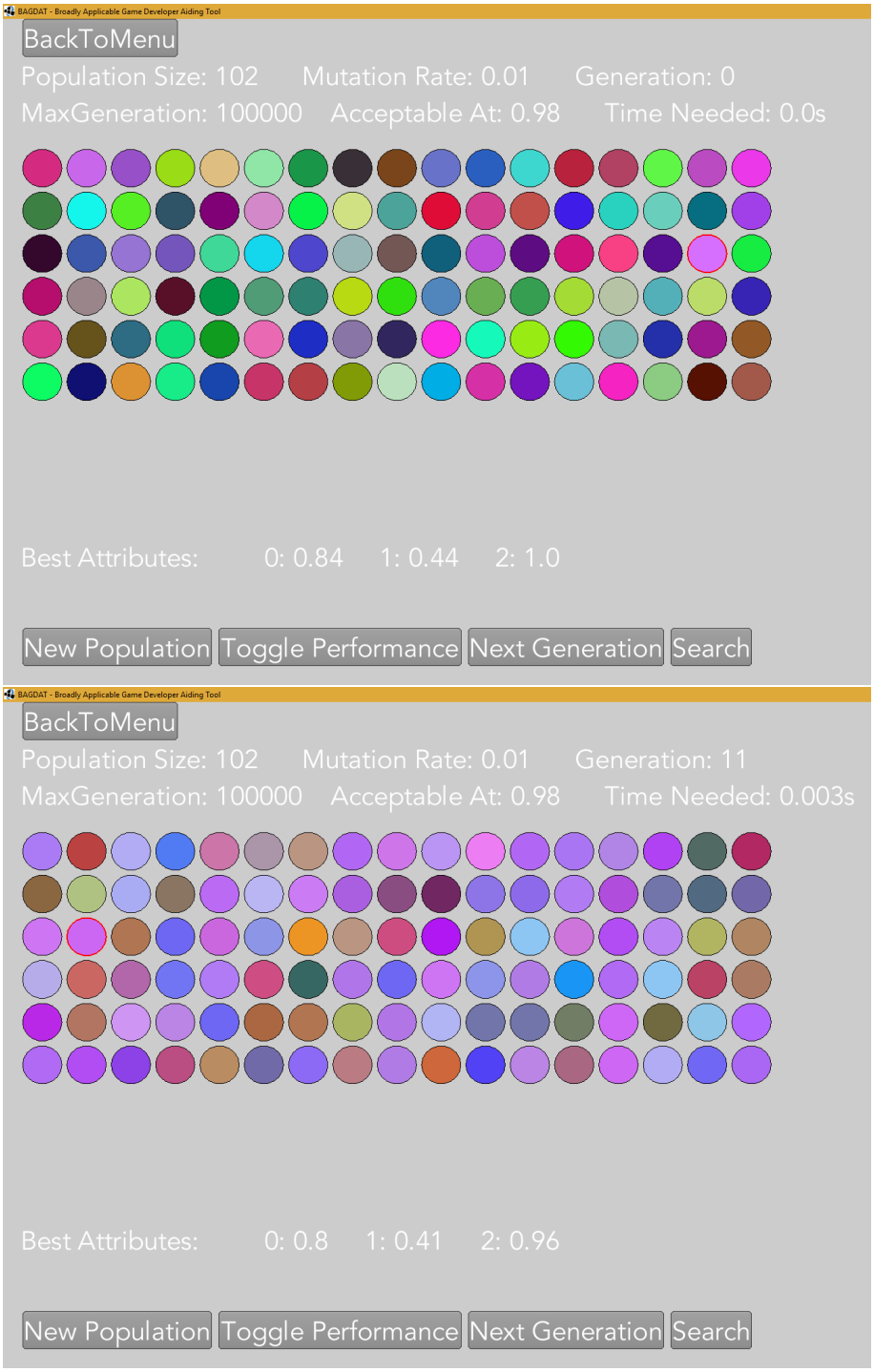


Abbildung 4.1: Ein Beispieldurchlauf des Genetischen Algorithmus in BAGDAT, die rot umrahmte Kugel ist die, mit den besten Attributen. Gesucht wurde eine Kugel mit den Werten (0,8 0,4 1).

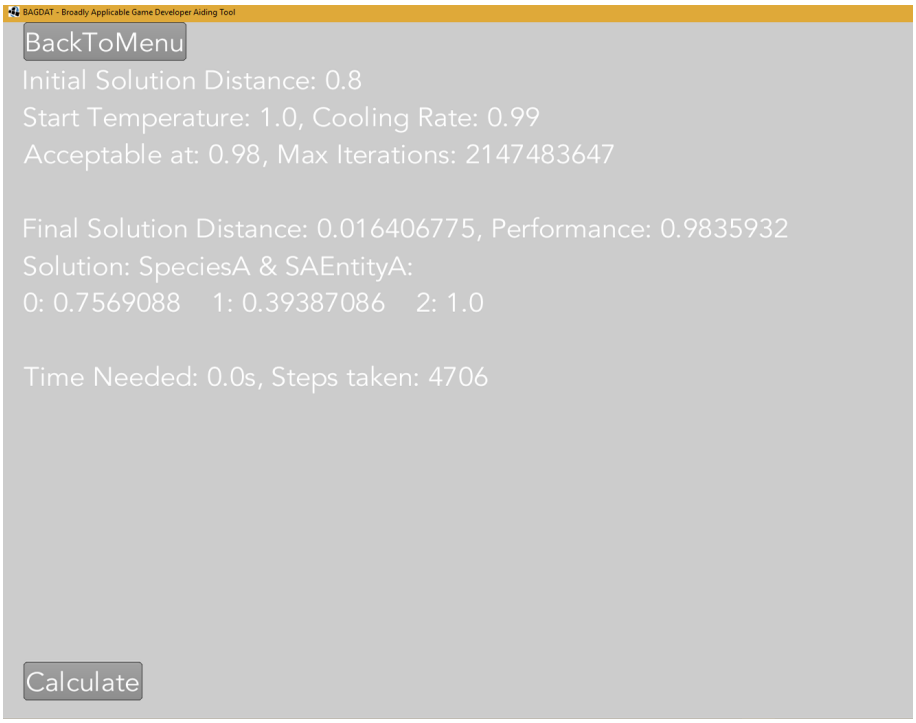


Abbildung 4.2: Ein Beispieldurchlauf des Simulated Annealing in BAGDAT.

Tabelle 4.1: GA Test Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Mutationsrate	0,001		
Populationsgröße	10	102	1000
Anzahl an Generationen	42.780,131	2.027,055	4,264
r	0,801	0,797	0,800
g	0,400	0,397	0,401
b	0,997	0,997	0,995
Performance (p)	0,997	0,997	0,998
Zeit in ms	79,767	64,747	6,141

der Mutationsrate liegt, wird es durch eine zufällig generierte Zahl zwischen 0 und 1 ersetzt.

Tabelle 4.2: GA Test Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Mutationsrate	0,01		
Populationsgröße	10	102	1000
Anzahl an Generationen	5.040,337	145,968	4,320
r	0,800	0,799	0,800
g	0,399	0,397	0,399
b	0,996	0,998	0,998
Performance (p)	0,998	0,998	0,998
Zeit in ms	9,517	4,460	6,183

Tabelle 4.3: GA Test Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Mutationsrate	0,1		
Populationsgröße	10	102	1000
Anzahl an Generationen	702,775	145,968	4,320
r	0,800	0,799	0,797
g	0,403	0,405	0,399
b	0,997	0,999	0,999
Performance (p)	0,997	0,997	0,998
Zeit in ms	1,418	1,081	6,499

4.2.2 Test des Simulated Annealing

Allgemeines

Das Simulated Annealing wurde mit einer Starttemperatur von 1 und einer Abkühlungsrate von 0,8, 0,9 sowie 0,99 und einer Startvermutung von $r = 0,5$, $g = 0,5$, $b = 0,5$ (die für dieses Problem neutralste Startvermutung, da die Distanz zu allen Werten am geringsten ist), $r = 0$, $g = 0,2$, $b = 0,4$ (eine zufällige Vermutung, da es nicht immer möglich ist eine gute Startvermutung zu treffen) und $r = 0$, $g = 1$, $b = 0$ (der für diesen Fall schlechtest mögliche Vermutung) gestartet. Die Abbruchkriterien waren ein Iterations-Threshold von 2.147.483.647 (Integer) und einem Performance-Threshold von ebenfalls $> 98\%$. Diese Ergebnisse sind in Tabellen 4.4, 4.5 und 4.6 zu sehen.

Tabelle 4.4: SA Test Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Start Vermutung	(0,5 0,5 0,5)		
Abkühlrate	0,8	0,9	0,99
Anzahl an Iterationen	863,231	3.625,346	3.763,017
Performance (p)	0,952	0,986	0,986
Zeit in ms	0,242	1,469	0,653

Tabelle 4.5: SA Test Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Start Vermutung	(0 0,2 0,4)		
Abkühlrate	0,8	0,9	0,99
Anzahl an Iterationen	876,933	4.033,913	3.738,575
Performance (p)	0,948	0,986	0,986
Zeit in ms	0,243	1,629	0,666

Tabelle 4.6: SA Test Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Start Vermutung	(0 1 0)		
Abkühlrate	0,8	0,9	0,99
Anzahl an Iterationen	904,667	3.668,300	3.847,641
Performance (p)	0,946	0,986	0,986
Zeit in ms	0,251	1,471	0,672

Finden eines Nachbarn

In diesem Fall wurde das Finden eines zufälligen Nachbarn so implementiert: Nachbarn sind Farbfelder mit einem veränderten Attributswert. Es wurde eine Veränderungsgröße von bis zu 0,2 festgelegt. Es gibt also sechs Nachbarn, drei mit je einem Attribut erhöht um bis zu 0,2 und drei mit je einem Attribut verringert um bis zu 0,2. Wenn ein Attribut dadurch die Grenzen 1 oder 0 über oder unterschreiten würde, wird es auf die jeweilige Grenze festgelegt. Durch das Erstellen einer Zufallszahl zwischen 1 und 6 wird ein bestimmter Nachbar ausgewählt, welcher dann verglichen wird.

4.3 Armeezusammenstellung

Wie schon in Abschnitt 3.3.2 besprochen, wird für die Erstellung einer Armee die Aufteilung in Bogenschützen, Rittern und Speerträgern jeweils in Prozent angegeben. Da dabei insgesamt immer hundert Prozent herauskommen muss, sind diese Werte von einander abhängig.

4.3.1 Test des Genetischen Algorithmus

Allgemeines

Der Genetische Algorithmus wurde mit einer Populationsgröße von 10, 102 sowie 1000 und einer Mutationsrate von 0,001, 0,01 und 0,1 auf das Problem angewandt. Die Abbruchkriterien waren ein Performance-Threshold von $>$ 90% und einer Maximalgeneration von 10. Die Maximalgeneration wurde auf 10 beschränkt, da bei 1000 und 100 schon ersichtlich war, dass das Ergebnis immer fast das optimale ist, welches bei 86% liegt. Da der Performance-Threshold jedoch bei $>$ 90% lag, wurde nie abgebrochen, deshalb wurde um unnötige Rechenzeit zu sparen das Limit von 10 Generationen eingeführt. Die Ergebnisse sind in Tabellen 4.1, 4.2 und 4.3 ersichtlich.

Selektion, Kreuzung und Mutation

Die Selektion findet auch hier proportional zur Performance statt, das bedeutet, dass Individuen mit höherer Performance eine höhere Chance haben für die Kreuzung ausgewählt zu werden. Nach der Auswahl wird für jedes Attribut zufällig ermittelt, ob es vom ersten oder zweiten Elternteil übernommen wird. Da bei dieser Problemstellung die Attributswerte jedoch zusammenhängen und sie einen Gesamtwert von 100% bilden müssen, wird anschließend durch die Summe aller Attribute dividiert und auf 100 multipliziert, um diesen Gesamtwert zu garantieren. Mit diesen Attributen wird dann ein neues Individuum erzeugt. Für die Mutation werden alle Werte zufällig neu erzeugt.

4.3.2 Test des Simulated Annealing

Allgemeines

Auch hier wurde das Simulated Annealing mit einer Starttemperatur von 1 und einer Abkühlungsrate von 0,8, 0,9 sowie 0,99 getestet. Die Startvermutung war 34% Bogenschützen, 33% Ritter und 33% Speerträger (die für dieses Problem neutralste Startvermutung, da die Distanz zu allen Werten am geringsten ist). Die Abbruchkriterien waren ein Iterations-Threshold von 2.147.483.647 (Integer) und einem Performance-Threshold von ebenfalls $>$ 90%. Diese Ergebnisse sind in Tabelle 4.10 zu sehen.

Tabelle 4.7: GA Armee Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Mutationsrate	0,001		
Populationsgröße	10	102	1000
Anzahl an Generationen	10,000	10,000	10,000
Bogenschützen	0,305	0,318	0,322
Ritter	0,211	0,206	0,213
Speerkämpfer	0,484	0,476	0,465
Performance (p)	0,861	0,861	0,861
Zeit in ms	1,791	24,527	255,346

Tabelle 4.8: GA Armee Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Mutationsrate	0,01		
Populationsgröße	10	102	1000
Anzahl an Generationen	10,000	10,000	10,000
Bogenschützen	0,317	0,325	0,314
Ritter	0,217	0,214	0,212
Speerkämpfer	0,465	0,462	0,474
Performance (p)	0,861	0,861	0,861
Zeit in ms	1,673	17,066	195,167

Tabelle 4.9: GA Armee Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Mutationsrate	0,1		
Populationsgröße	10	102	1000
Anzahl an Generationen	10,000	10,000	10,000
Bogenschützen	0,330	0,318	0,320
Ritter	0,197	0,216	0,191
Speerkämpfer	0,472	0,467	0,490
Performance (p)	0,861	0,861	0,861
Zeit in ms	1,836	19,836	183,507

Tabelle 4.10: SA Armee Ergebnisse, die Werte sind Durchschnitt von 1000 Durchläufen.

Start Vermutung	(0,34 0,33 0,33)		
Abkühlrate	0,8	0,9	0,99
Anzahl an Iterationen	2.665,615	5.530,396	58.893,587
Performance (p)	0,842	0,841	0,841
Zeit in ms	88,416	175,338	1.841,281

Finden eines Nachbarn

In diesem Fall wurde das Finden eines zufälligen Nachbarn so implementiert: Nachbarn sind Armeen mit zwei veränderten Attributswerten. Da die drei Werte zusammenhängen und sie nicht unabhängig voneinander sind, ist es nicht möglich nur einen Wert zu ändern. Deshalb wird ein Nachbar so generiert, dass er einen Wert erhöht und einen anderen verringert. Die Höhe der Veränderung ist 5% außer einer der beiden Werte würde dadurch eine Grenze überschreiten, dann ist es nur die Differenz zur Grenze. Zum Beispiel eine

- Startlösung (0,34 0,33 0,33) mit diesen Nachbarn
- Nachbar1 (0,39, 0,28 0,33),
- Nachbar2 (0,39 0,33 0,28),
- Nachbar3 (0,29 0,38 0,33),
- Nachbar4 (0,29 0,33 0,38),
- Nachbar5 (0,34 0,38 0,28),
- Nachbar6 (0,34 0,28 0,38).

Und am Ende wird durch das Erstellen einer Zufallszahl zwischen 1 und 6 ein bestimmter Nachbar ausgewählt, welcher dann verglichen wird.

4.4 Landkartenerstellung und -vergleich

Wie in Abschnitt 3.3.3 kurz angerissen, werden einige Werte benötigt, um eine Karte zu erstellen. Nämlich die Anzahl der Oktaven für die Höhe, Temperatur und Feuchtigkeit, welche aus Ganzzahlen bestehen und sich die sinnvollen im Bereich von 3-12 befinden. Als auch der Persistenz für diese drei Punkte welche sinnvolle Werte zwischen 0,2 und 1 produziert. Sowie des Meeres-spiegels, welcher zwischen 0 und 1 liegen kann, mit 0 ohne Wasser und mit 1 nur Wasser.

4.4.1 Test des Genetischen Algorithmus

Allgemeines

Der Genetische Algorithmus wurde mit einer Populationsgröße von 10, 102 sowie 1000 und einer Mutationsrate von 0,001, 0,01 und 0,1 auf das Problem angewandt. Die Abbruchkriterien waren ein Performance-Threshold von $>$ 98% und einer Maximalgeneration von 10. Die Ergebnisse sind in Tabellen 4.11, 4.12 und 4.13 ersichtlich.

Beim Testen muss gesagt werden, dass nicht immer eine Karte erstellt und verglichen wurde, da dies bei einer Zufallskarte ja auch ein Glückstreffer sein könnte, sondern drei und ihr Durchschnittswert wird berechnet. Es wurden aus Zeitgründen nicht mehr als 3 genommen, es könnten aber natürlich 10 oder 100 generiert werden. Dies erhöht die Wahrscheinlichkeit, dass die gewählten Parameter öfters die gewünschten Karten erstellen und nicht manchmal eine andere.

Selektion, Kreuzung und Mutation

Die Selektion findet auch hier proportional zur Performance statt, das bedeutet, dass Individuen mit höherer Performance eine höhere Chance haben für die Kreuzung ausgewählt zu werden. Nach der Auswahl wird für jedes Attribut zufällig ermittelt, ob es vom ersten oder zweiten Elternteil übernommen wird. Mit diesen Attributen wird dann ein neues Individuum erzeugt. Für die Mutation wird für jedes Attribut eine Zufallszahl erstellt und wenn diese unter der Mutationsrate liegt, wird es durch einen passenden Zufallswert ersetzt.

4.4.2 Test des Simulated Annealing

Allgemeines

Das Simulated Annealing wurde mit einer Starttemperatur von 1 und einer Abkühlungsrate von 0,8, 0,9 sowie 0,99 getestet. Die Startvermutung war

- Oktaven Höhe: 6
- Persistenz Höhe: 0,6
- Oktaven Temperatur: 6
- Persistenz Temperatur: 0,6
- Oktaven Feuchtigkeit: 6
- Persistenz Feuchtigkeit: 0,6
- Meeresspiegel: 0,5.

Die Abbruchkriterien waren ein Iterations-Threshold von 100 und einem Performance-Threshold von $>$ 98%. Diese Ergebnisse sind in Tabelle 4.14 zu sehen.

Tabelle 4.11: GA Karten Ergebnisse, die Werte sind Durchschnitt von 10 Durchläufen.

Mutationsrate	0,001	
Populationsgröße	10	50
Anzahl an Generationen	10,00	10,00
Oktaven Höhe	7,00	6,60
Persistenz Höhe	0,62	0,70
Oktaven Temperatur	6,00	6,10
Persistenz Temperatur	0,59	0,66
Oktaven Feuchtigkeit	4,00	5,60
Persistenz Feuchtigkeit	0,46	0,69
Meeresspiegel	0,30	0,20
Performance (p)	0,97	0,97
Zeit in ms	35.321,60	214.170,50

Finden eines Nachbarn

In diesem Fall wurde das Finden eines zufälligen Nachbarn so implementiert: Nachbarn sind Kartengeneratoren mit einem veränderten Attributswert. Für die Oktaven gibt es nur Ganzzahlen von 3-12, bei der Persistenz gibt es Werte zwischen 0,2-1 und für den Meeresspiegel gibt es Werte zwischen 0-1. Für die Oktaven wurde eine Veränderungsgröße von 1 und für die anderen Attribute von 0,2 festgelegt. Es gibt also vierzehn Nachbarn, sieben mit je einem Attribut erhöht und sieben mit je einem Attribut verringert. Wenn ein Attribut dadurch seine Grenzen über oder unterschreiten würde, wird es auf die jeweilige Grenze festgelegt. Durch das Erstellen einer Zufallszahl zwischen 1 und 14 wird ein bestimmter Nachbar ausgewählt, welcher dann verglichen wird.

Tabelle 4.12: GA Karten Ergebnisse, die Werte sind Durchschnitt von 10 Durchläufen.

Mutationsrate	0,01	
Populationsgröße	10	50
Anzahl an Generationen	10,00	10,00
Oktaven Höhe	6,40	6,50
Persistenz Höhe	0,82	0,70
Oktaven Temperatur	5,20	5,40
Persistenz Temperatur	0,64	0,66
Oktaven Feuchtigkeit	6,60	7,40
Persistenz Feuchtigkeit	0,47	0,64
Meeresspiegel	0,19	0,21
Performance (p)	0,96	0,97
Zeit in ms	36.397,60	186.926,20

Tabelle 4.13: GA Karten Ergebnisse, die Werte sind Durchschnitt von 10 Durchläufen.

Mutationsrate	0,1	
Populationsgröße	10	50
Anzahl an Generationen	10,00	10,00
Oktaven Höhe	6,70	6,00
Persistenz Höhe	0,88	0,63
Oktaven Temperatur	5,70	6,10
Persistenz Temperatur	0,47	0,64
Oktaven Feuchtigkeit	5,70	5,50
Persistenz Feuchtigkeit	0,82	0,79
Meeresspiegel	0,20	0,30
Performance (p)	0,97	0,97
Zeit in ms	42.208,80	198.387,40

Tabelle 4.14: SA Karten Ergebnisse, die Werte sind Durchschnitt von 100 Durchläufen.

Start Vermutung	(6 0,6 6 0,6 6 0,6 0,5)		
Abkühlrate	0,8	0,9	0,99
Anzahl an Iterationen	100,00	100,00	100,00
Performance (p)	0,93	0,93	0,94
Zeit in ms	34.136,90	34.354,47	34.116,65

Kapitel 5

Diskussion

5.1 Was zeigen die Ergebnisse?

5.1.1 Farbkugeln

Genetischer Algorithmus

Wie in Abbildung 5.1 ersichtlich wird, benötigen kleinere Populationen bei einer geringen Mutationsrate für dieses Problem deutlich länger um ein ähnlich gutes Ergebnis zu erzielen. Dies könnte darauf zurückzuführen sein, dass es bei einigen Generationen zu keiner beachtlichen Veränderung der Population kommt, da sich viele Individuen schon sehr ähnlich sind und keine neuen Impulse von Außen hinzukommen. Es zeigt sich auch, dass höhere Mutationsraten kleinere Populationen enorm schneller machen, von 80 ms auf 10 ms und weiter auf unter 2 ms bei der kleinsten Population und bei der mittleren von 65 ms auf 4,5 ms. Auf große Populationen zeigen hohe Mutationsraten kaum eine Auswirkung, die Geschwindigkeit bleibt bei der größten Population stets um die 6,1 ms, während die mittlere Population auf 4,5 ms verharrt. Es muss auch erwähnt werden, dass sich hier zwar die Zeit für die Ergebnisse unterscheiden, die Ergebnisse selbst aber relativ gleich gut bleiben, mit Schwankungen von unter 0,1 %.

Simulated Annealing

Wie in den Diagrammen aus Abbildung 5.2 zu sehen ist, führt ein Abkühlungsfaktor von 0,8 zu einem rascheren, jedoch ungenaueren Ergebnis. Dies liegt daran, dass die Temperatur mit diesem Abkühlungsfaktor am schnellsten sinkt ($Temperatur' = Temperatur \cdot Abkuehlungsfaktor$) und dadurch der Algorithmus aufgrund der Temperaturgrenze terminiert. Bei einem Abkühlungsfaktor von 0,9 kann um einiges länger gerechnet werden, bevor die Temperaturgrenze erreicht wird und durch die längere Rechenzeit können bessere Ergebnisse gefunden werden.

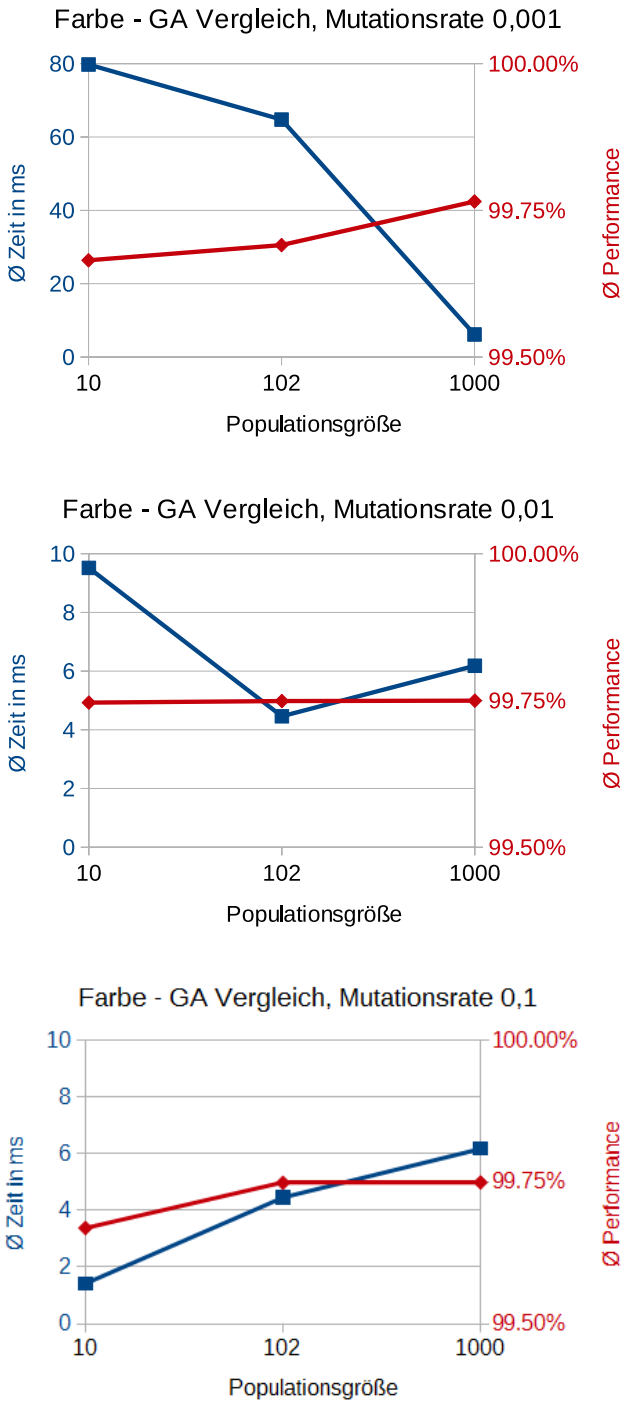


Abbildung 5.1: Vergleiche der GA's mit unterschiedlichen Mutationsraten und Populationsgrößen für die Farbkugeln.

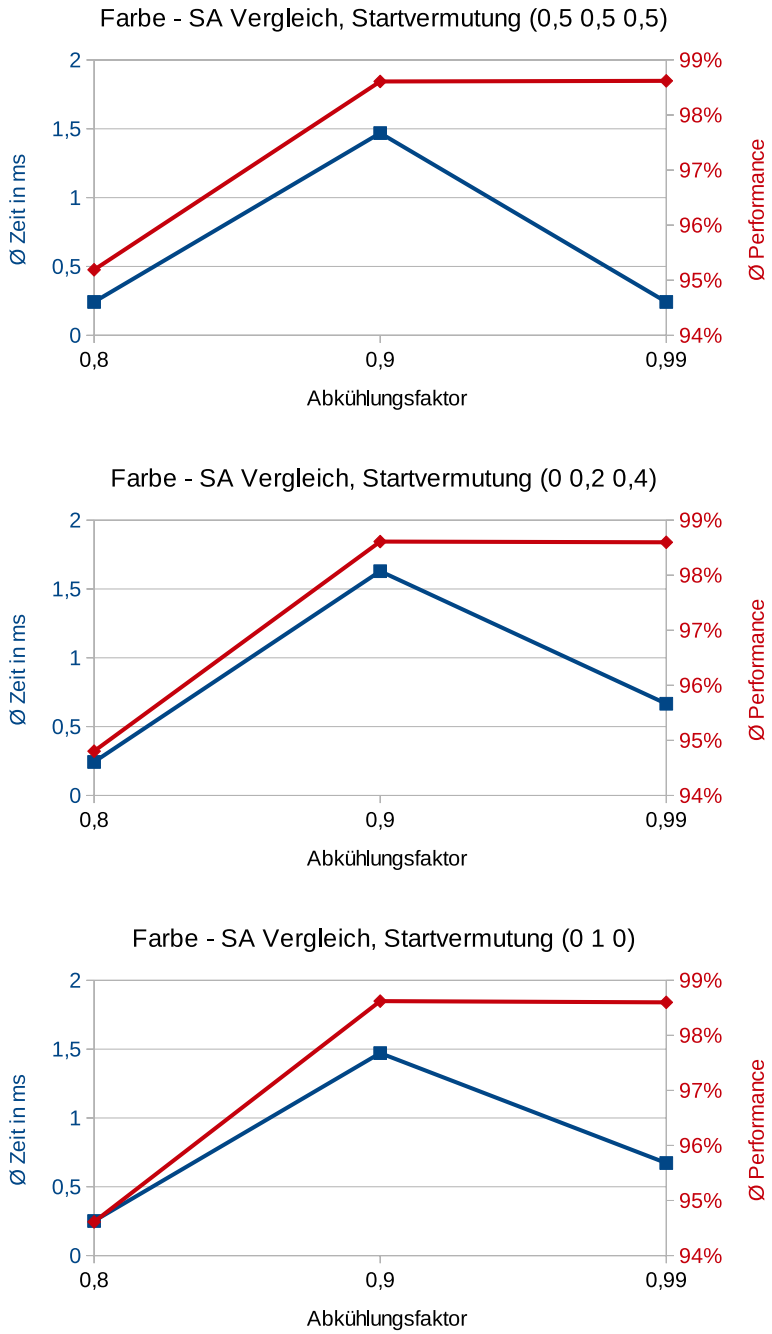


Abbildung 5.2: Hier werden die Unterschiede des Simulated Annealing Algorithmus ersichtlich, wenn die Abkühlrate und die Startvermutung geändert werden.

Vergleich

Wenn nun beide Algorithmen verglichen werden, ist zu sehen, dass der Genetische Algorithmus (mit einer Mutationsrate von 0,1 und Populationsgröße von 10) mit der Zeit von 1,08 ms und einer Performance von $99,73\%$ ein wenig schneller und genauer als der Simulated Anneling (mit einem Abkühlungsfaktor 0,9 und der neutralen Startvermutung von (0,5 0,5 0,5)) mit einer Zeit von 1,4 ms und einer Performance von $98,6\%$ ist.

5.1.2 Armee

Genetischer Algorithmus

Auffällig ist die Ähnlichkeit der Diagramme aus Abbildung 5.3. Fast alle Durchgänge erzielen ein Ergebnis um die 86,14%. Dieser Wert ist vermutlich einer der höchstmöglichen Werte, da es keine Verbesserungen gibt, wenn die Population größer ist oder die Anzahl an Generationen erhöht wird. In Vortests wurde die Anzahl an Generationen auf 1000 festgelegt, um die Rechenzeit zu beschränken, da der Algorithmus hier nicht durch den Performancethreshold von $> 90\%$ terminieren kann. Da aber Tests mit 100 und 10 Generationen fast die selbe Performance lieferten, nur mit einer viel kürzeren Rechenzeit, wurde die Grenze von 10 Generationen eingeführt, um Rechenzeit zu sparen. Und warum ist nun die Rechenzeit für größere Populationen exponentiell? Da bei jeder 10. Generation die Berechnung gestoppt wird, berechnet eine Population von $10 \cdot 10 = 100$ Individuen, während eine Population von $102 \cdot 10 = 1020$ und eine von $1000 \cdot 10 = 10.000$ Individuen berechnet. Die exponentielle Rechenzeit kommt also nur von dem exponentiellen Wachstum aufgrund der Populationsgröße mit der Anzahl an Generationen. Dieses Problem lässt sich schon mit einer kleinen Population lösen, am schnellsten mit einer Mutationsrate von 0,01 in 1,7 ms und je 1,8 ms mit den anderen.

Simulated Annealing

Wie in Abbildung 5.4 zu sehen ist, benötigt der Simulated Annealing Algorithmus länger, je langsamer er abkühlt, da er – solange die Temperaturgrenze nicht unterschritten wird – weiter rechnet und hier der Performancethreshold auch nicht erreicht werden kann. Interessant ist hier, dass die Performance bei langsamer Abkühlung abnimmt, wenn auch nur marginal.

Vergleich

Bei einem Vergleich der beiden Algorithmen ist zu erkennen, dass der Genetische Algorithmus (mit einer Mutationsrate 0,1 und Populationsgröße von 10) mit einer Berechnungszeit von 1,8 ms und einer Performance von $86,1\%$ gegen den Simulated Annealing (mit einem Abkühlungsfaktor von 0,8) mit

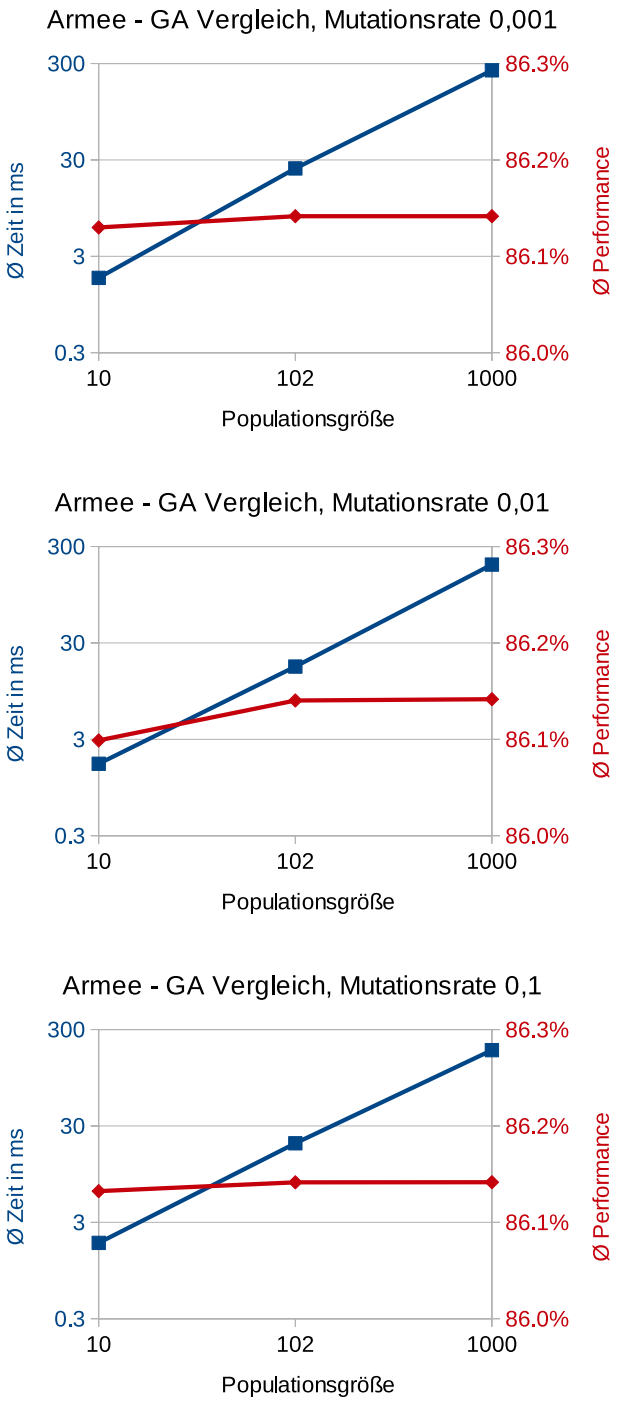


Abbildung 5.3: Vergleiche der GA's mit unterschiedlichen Mutationsraten und Populationsgrößen für die Armeezusammenstellung.

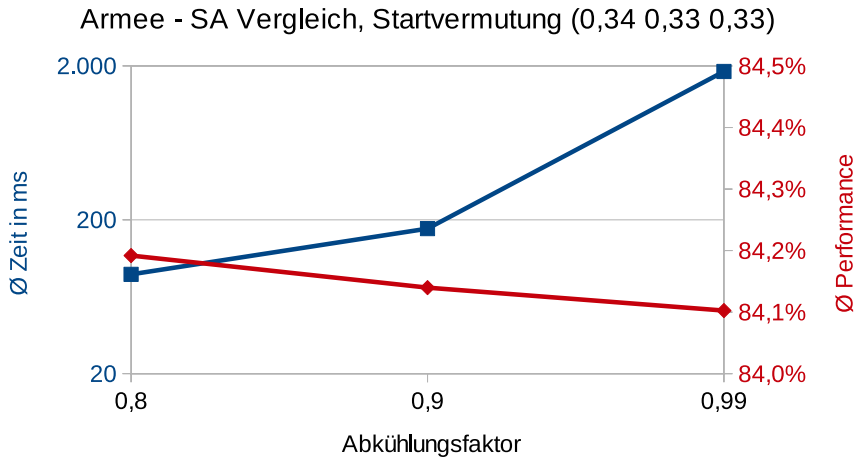


Abbildung 5.4: Zu sehen sind hier die Unterschiede des Simulated Annealing bei unterschiedlichen Abkühlungsfaktoren bei der Armeezusammenstellung.

einer Zeit von 88,4 ms und einer Performance von 84, 2% besser abschneidet, Wobei hier der Genetische Algorithmus aufgrund der Beschränkung auf 10 Generationen für eine geringere Rechenzeit einen klaren Vorteil hatte. Dass das Simulated Annealing jedoch trotz unbeschränkten Iterationen keine bessere Performance erzielt, ist interessant.

5.1.3 Landkarten

Genetischer Algorithmus

Auch bei diesem Problem wurde der Performancethreshold von $> 98\%$ nicht erreicht. Dies könnte jedoch auch daran liegen, dass aufgrund der langen Rechenzeit das Limit von 10 Generationen beibehalten wurde. Die Anzahl an Individuen ist mit $10 \cdot 10 = 100$ und $50 \cdot 10 = 500$ um den Faktor 5 größer, die Rechenzeit schwankt jedoch um den Faktor 4, wie in Abbildung 5.5 zu sehen ist. Wie auch schon zuvor ist eine niedrigere Mutationsrate für größere Populationen förderlich. Hier zeigt sich, dass eine kleine Population mit einer hohen Mutationsrate die genauesten Ergebnisse erzielt, bei einer Rechenzeit von 42,2 Sekunden, während eine kleine Population mit geringer Mutationsrate ein ungenaueres, jedoch schnelleres – mit 35,3 Sekunden – erzielt.

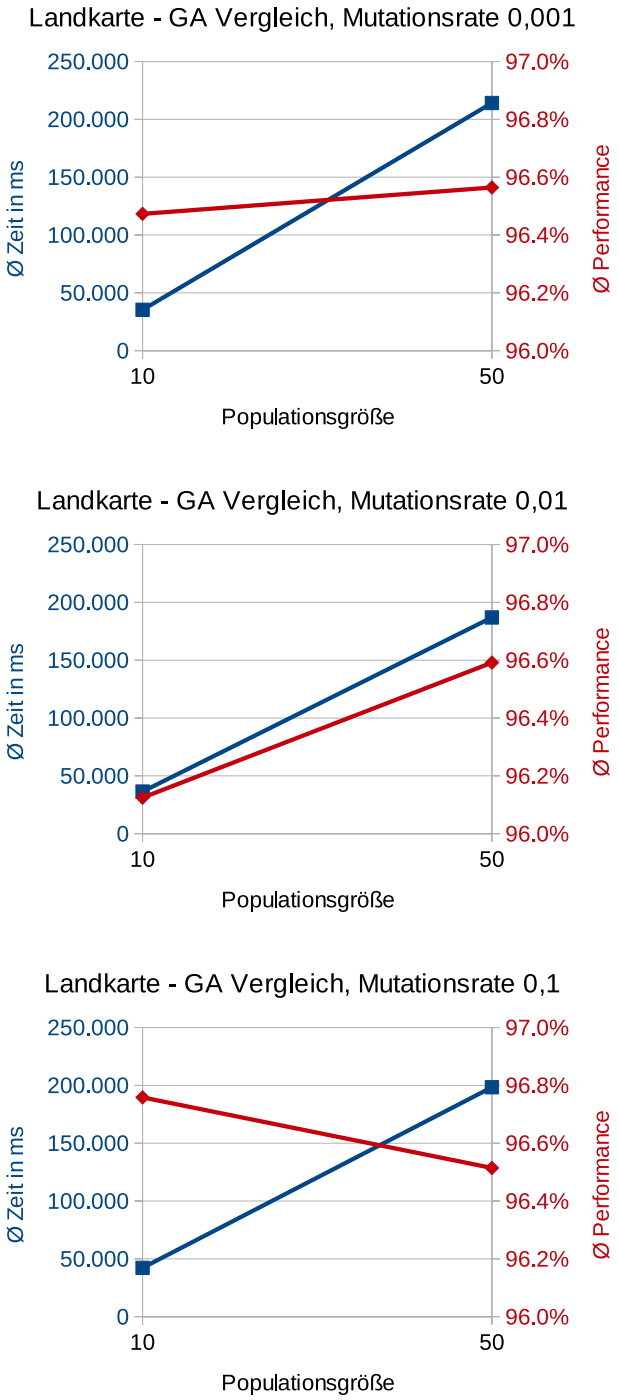


Abbildung 5.5: Vergleiche der GA's mit unterschiedlichen Mutationsraten und Populationsgrößen für die Landkartengenerierung.

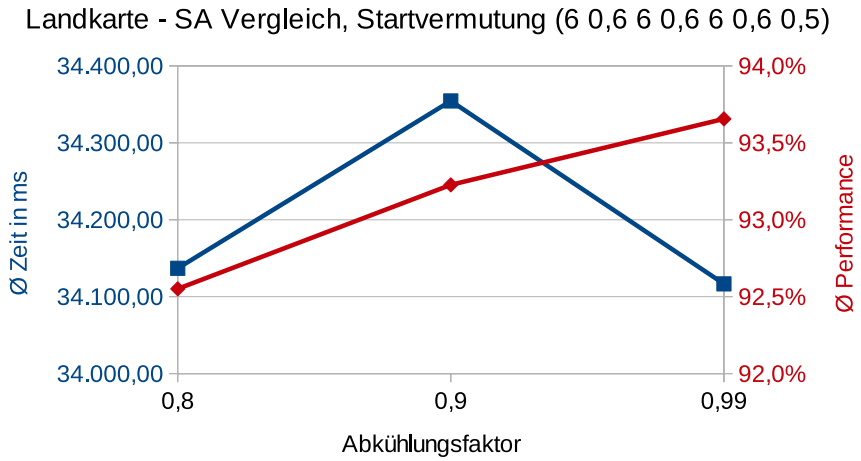


Abbildung 5.6: Dieses Diagramm zeigt die Unterschiede des Simulated Annealing bei unterschiedlichen Abkühlungsfaktoren für das Landkartengenerieren.

Simulated Annealing

Wie in Abbildung 5.6 zu erkennen ist, nimmt die Performance mit höherem Abkühlungsfaktor zu. Dies könnte hier auch daran liegen, dass der Algorithmus aufgrund der Rechenzeitbeschränkung nur 100 Iterationen durchgehen kann. Und für dieses Problem ist es wahrscheinlicher, innerhalb der 100 Iterationen einen guten Wert zu treffen, wenn auch manchmal schlechtere Werte akzeptiert werden, um anschließend bessere zu finden. Die Rechenzeit ist zwischen 34,1 Sekunden und 34,4 Sekunden auch relativ ähnlich.

Vergleich

Auch hier stellt sich die Frage, wie die Unterschiede in Geschwindigkeit und Genauigkeit der beiden Algorithmen aussehen. Der Genetische Algorithmus (mit einer Mutationsrate von 0,1 und einer Population von 10) erzielt eine Geschwindigkeit von 35,3 Sekunden und eine Performance von $96,5\%$, der Simulated Annealing (mit einer Abkühlungsrate von 0,8) erreicht hier $93,6\%$ bei einer Geschwindigkeit von 34,1 Sekunden. Wie zu sehen ist, erreicht der Simulated Annealing Algorithmus nicht ganz die Genauigkeit des Genetischen Algorithmus, er ist dafür jedoch ein wenig schneller.

5.2 Praktikabilität

Nun stellt sich die Frage: Wie praktikabel sind beide Algorithmen?

Tabelle 5.1: Benötigte Zeit für die Umsetzung der Klasse des Farbkugelproblems.

Klasse mit allgemeinen Methoden	4 Minuten
Zielfunktion	5 Minuten
GA Methoden	10 Minuten
SA Methoden	8 Minuten
GA Gesamt	19 Minuten
SA Gesamt	17 Minuten

Tabelle 5.2: Benötigte Zeit für die Umsetzung der Klasse für die Armeezusammenstellung. Für das Erstellen der Armeen wird für dieses Problem ein eigener Algorithmus benötigt, welcher hier hinzugezählt wird, denn das würde normalerweise der Designer machen in dem er ausrechnet, wie viele Einheiten er von einem Typ mit den Ressourcen erstellen kann.

Klasse mit allgemeinen Methoden	5 Minuten
Armeeerstellung	22 Minuten
Zielfunktion	1 Minuten
GA Methoden	8 Minuten
SA Methoden	7 Minuten
GA Gesamt	36 Minuten
SA Gesamt	35 Minuten

Dazu sollte auch die Zeit zur Erstellung der Klassen berücksichtigt werden. Denn wenn ein Problem händisch schneller gelöst wird, als die Umsetzung für die Algorithmen, dann ist der Einsatz des Tools sinnlos. In den Tabellen 5.1, 5.2 und 5.3 wird die benötigte Zeit für einzelne Teile der Umsetzung gezeigt. Der Punkt „Klasse mit allgemeinen Methoden“, zeigt die benötigte Zeit, um die getPerformance(), getAttributes() und toString() Methoden sowie die Membervariablen einzubauen. Wobei hier zu beachten ist, wenn beide Algorithmen verwendet werden sollen, für den zweiten nur mehr die Zeit für die jeweiligen Methoden hinzugezählt werden muss. Eine einmalige Einarbeitungszeit für das Lesen der Dokumentation und das Zurechtfinden im Tool wird auf 5-10 Minuten geschätzt, dementsprechend ist dies für die Zeiten in den Tabellen bei Erstbenutzung hinzuzuzählen.

In Tabelle 5.4 findet sich eine Übersicht, welcher Algorithmus in der Umsetzung schneller ist, bei der Anwendung schneller ist und bei der An-

Tabelle 5.3: Benötigte Zeit für die Umsetzung der Klasse für die Landkartengenerierung.

Klasse mit allgemeinen Methoden	5 Minuten
Zielfunktion	5 Minuten
GA Methoden	9 Minuten
SA Methoden	6 Minuten
GA Gesamt	19 Minuten
SA Gesamt	16 Minuten

Tabelle 5.4: Hier die Übersicht, welcher Algorithmus schneller Umgesetzt werden kann, welcher im besten Fall schneller und genauer ist. Bei dem Armeeproblem sollte die Fußnote im Text beachtet werden.

	Schneller (Umsetzung)	**Schneller (Anwendung)**	**Genauer**
Farbkugeln	SA (um 2 Minuten)	GA (um 0,32 ms)	GA (um $1,13\%$)
Armee	SA (um 1 Minute)	GA (um 86,6 ms)	GA (um $1,9\%$)
Landkarten	SA (um 3 Minuten)	SA (um 1,2 Sekunden)	GA (um $2,9\%$)

wendung genauere Ergebnisse liefert[1].

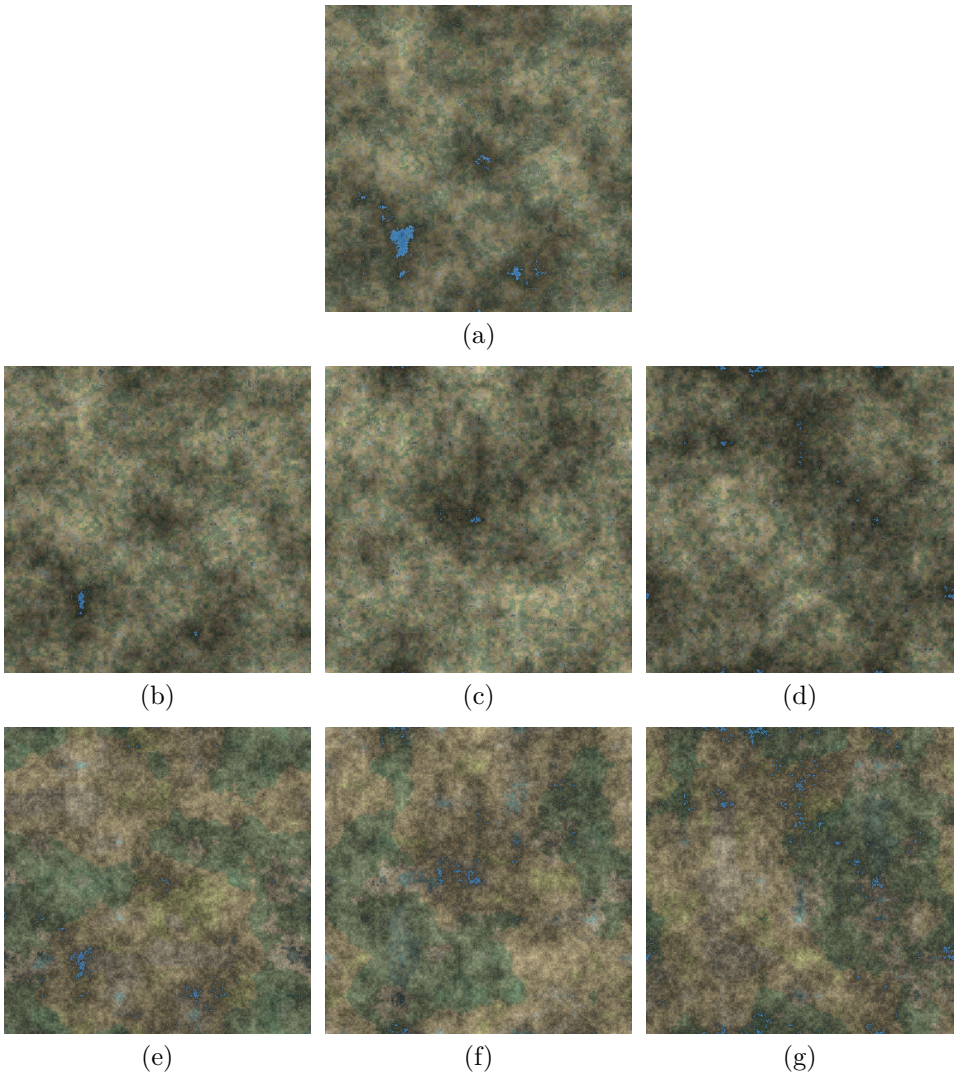
Bei dem Testproblem der Farbkugeln, das dazu diente die Implementierung auf Fehler zu überprüfen, kann ein Designer die gewünschten Farben natürlich sofort einstellen. Hier wird sich der Einsatz des Tools aufgrund der hohen Umsetzungskosten nicht auszahlen.

Für die Armeezusammensetzung sieht es anders aus. Die Einstellungen waren eine feindliche Armee von 10 Bogenschützen, 5 Rittern und 20 Speerträger sowie Ressourcen von 350 Holz, 200 Stahl, 350 Leder. Bei einem manuellen Versuch wurde für einen Vorschlag mit 5 Rittern, 20 Bogenschützen und wie viel auch immer Speerträgern sich noch ausgehen (13) eine Zeit von 4 Minuten und 8 Sekunden benötigt und sie erzielte eine Performance von $73,2\%$. Ein darauffolgender Versuch mit 15 Bogenschützen, 10 Ritter und wie viel Speerträger sich ausgehen (20) wurden 2 Minuten und 43 Sekunden

[1] Wie in Abschnitt 5.1.2 erklärt, liegt hier der enorme Geschwindigkeitsvorteil des GA an der Begrenzung auf 10 Generationen, während der SA keine Beschränkung hatte. Dafür hätte der SA mehr Zeit gehabt hat, ein besseres Ergebnis zu finden.

benötigt, welcher eine Performance von 86, 1% erzielte. Die Performance ist gut, was aber nicht gewusst werden kann und deshalb wird vermutlich noch mehr Ausprobiert werden, bis dies erkannt wird. Der gesamte Versuch hat insgesamt 6 Minuten und 51 Sekunden gedauert. Dies bedeutet, dass sich das Tool ab sechs Berechnungen auszahlt. Dies wäre der Fall, wenn es mehr als sechs Level gibt oder sich während der Entwicklung Eigenschaften der Einheiten (Kosten, Stärke, ...) sechs mal ändern.

Für die Landkartengenerierung wurde auch ein manueller Versuch unternommen. Es soll versucht werden, Einstellungen zu finden, um Karten zu generieren, die einer bestimmten Karte ähnlich sehen. Es werden für die selben Einstellungen drei Karten generiert, um zu sehen, ob konstanter gleiche Karten erzeugt werden können. Der manuelle Versuch hat 4 Minuten und 2 Sekunden gedauert. Der Vergleich beider ist in Abbildung 5.7 zu sehen. In Abbildung 5.8 ist ein Beispiel zu sehen, wie aus 10 Startkarten eine gute berechnet wird. Hier zahlt sich das automatisierte Vorgehen aus, wenn mehr als 5 verschiedene Generatoren gefunden werden müssen und es muss beachtet werden, dass das automatische Vergleichen der Karten nicht perfekt ist, was aber zu erwarten war.

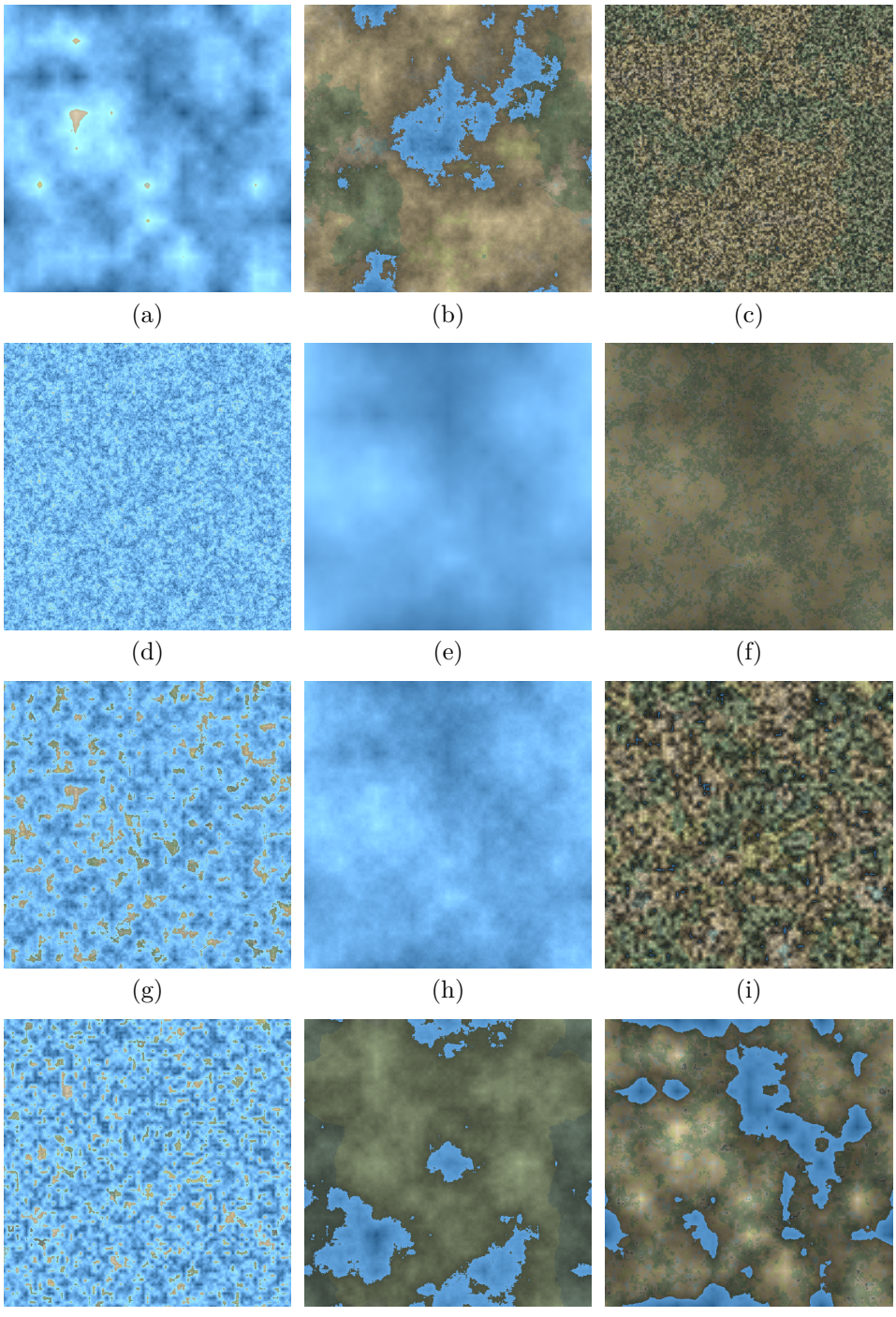


(a)

(b) (c) (d)

(e) (f) (g)

Abbildung 5.7: Die Abbildung zeigt, den Vergleich von manuell gesuchten und automatisiert gefundenen Karten zur originalen (a). Die manuellen haben folgende Performancewerte (b) 0,99, (c) 0,93 und (d) 0,83, die automatisch gesuchten diese (e) 0,97, (f) 0,96 und (g) 0,93. Wenn die durchschnittliche Performance betrachtet wird, ergeben sich für die manuell gesuchten 92% während die automatischen 95% erreichen. Obwohl also die manuellen der gesuchten Karte ähnlicher schauen, haben sie eine niedrigere Performance. Dies liegt an der Definition der Zielfunktion. Unser Bildvergleich wird nur grob durchgeführt, und deshalb ist das Ergebnis des Vergleichs nicht optimal.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Abbildung 5.8: Die Karten (a) - (j) zeigen die Startpopulation. Die Karte (k) ist jene, für welche ähnliche Kandidaten gesucht werden und (l) ist die beste, die gefunden wurde. Die Performance der Karten ist wie folgt (a) 0,78, (b) 0,92, (c) 0,57, (d) 0,56, (e) 0,78, (f) 0,88, (g) 0,73, (h) 0,77, (i) 0,91, (j) 0,69 und (k) 1 sowie (l) 0,96.

Kapitel 6

Fazit

6.1 Kritische Betrachtung

Es hat sich herausgestellt, dass alle gestellten Probleme mit den beiden Algorithmen gelöst werden konnten und großteils sogar sehr genaue Ergebnisse finden. Weiters zeigt sich, dass beide Algorithmen beinahe gleich gut geeignet sind um die gezeigten Probleme zu lösen. Der Genetische Algorithmus findet ein wenig genauere Ergebnisse und ist bei zwei von drei Problemen schneller. Er benötigt jedoch eine gering längere Implementationszeit. Somit kommt es auf den Benutzer an, was für ihn wichtiger ist. Auch wird festgehalten, dass diese Algorithmen nur so gut sind, wie ihre Zielfunktion es zulässt. Bei dem Kartenproblem hat sich die Berechnung für die Zielfunktion als nicht optimal herausgestellt, trotzdem lieferte sie annehmbare Ergebnisse.

Es zeigt sich auch, dass die Algorithmen in annehmbarer Zeit Ergebnisse liefern. Jedoch hängt ein Großteil der Berechnungszeit von der Zielfunktion ab. Für aufwändigere Probleme könnte dies zu viel längeren Berechnungen führen, die eventuell nicht mehr hinnehmbar sind.

Bei der Implementierung wurde auf Threads verzichtet, was dem Genetischen Algorithmus einen weiteren Performance-Schub verleihen könnte, wenn die Individuen zugleich berechnet werden können. Für das Kartenproblem hat sich jedoch gezeigt, dass es unklug sein könnte, dies zu tun, denn bei einer großen Population wird auch mehr RAM benötigt und dieser ist begrenzt. Auch würde eine automatische Erkennung von Konvergenz dem Algorithmus helfen.

6.2 Weiterer Ausblick

Interessant wären ein Vergleich mit den anderen zwei, hier nicht umgesetzten Algorithmen, dem Hit-and-Run Algorithmus und dem Partikelschwarm-Algorithmus.

Die hier gezeigten Probleme untersuchen Bereiche des Balancing und der Content-Erstellung, diese haben aber noch mehr Herausforderungen, die in vielen Spielen ähnlich sind. Abgesehen davon gibt es auch noch andere Gebiete in der Spieleentwicklung, für die so ein Tool auch nützlich sein könnte, wie beispielsweise das Testen des Spieles auf verschiedener Hardware, um gute Grafikeinstellungen für die jeweiligen Systeme zu finden.

Quellenverzeichnis

Literatur

[1] M. Affenzeller. *Genetic Algorithms and Genetic Programming. Modern Concepts and Practical Applications*. 6. Aufl. Boca Raton, Fla.: Chapman & Hall / CRC, 2009 (siehe S. 13).

[2] Wilhelm Burger und Mark Burge. *Digitale Bildverarbeitung. Eine Einführung mit Java und ImageJ*. 2. Aufl. Heidelberg: Springer-Verlag, 2006 (siehe S. 27).

[3] John C. Clarke, S. Carlsson und Andrew Zisserman. „Detecting and Tracking Linear Features Efficiently“. In: *Proceedings of the British Machine Vision Conference*. University of Edinburgh, UK, 1996, S. 1–10 (siehe S. 27).

[4] Andrew R. Conn, Katya Scheinberg und Luis N. Vicente. *Introduction to Derivative-Free Optimization*. Philadelphia, PA, USA: Society for Industrial und Applied Mathematics, 2009 (siehe S. 7).

[5] P. E. Gill, W. Murray und M. H. Wright. *Practical Optimization*. London: Academic Press, 1981 (siehe S. 7).

[6] S. Kirkpatrick, C. D. Gelatt und M. P. Vecchi. „Optimization by Simulated Annealing“. *Science* 220.4598 (1983), S. 671–680 (siehe S. 10).

[7] Nikolaos V. Rios Luis Miguel and Sahinidis. „Derivative-free Optimization: A Review of Algorithms and Comparison of Software Implementations“. *Journal of Global Optimization* 56.3 (2013), S. 1247–1293 (siehe S. 7, 9, 10, 13).

Online-Quellen

[8] *Bild der Oktaven mit Persistenz*. URL: https://web.archive.org/web/20160310084426/http://freespace.virgin.net/hugo.elias/models/m_perlin.htm (siehe S. 27).

[9] *Das Bild des Totally Accurate Battle Simulators aus der Steamstore-page*. URL: http://cdn.akamai.steamstatic.com/steam/apps/508440/ss_e9ad0f96a1796f0129e0aab17cf55665ceb05b15.1920x1080.jpg?t=1472135601 (siehe S. 20).

[10] *Das Bild einer Karte von AoE2 HD*. URL: https://i.imgur.com/BB1Qo.jpg (siehe S. 25).

[11] *Das Bild einer Karte von vktj für Civilization 4*. URL: http://forums.civfanatics.com/showthread.php?t=461262 (siehe S. 25).

[12] *Devmag Value Noise Erstellung*. URL: http://devmag.org.za/2009/04/25/perlin-noise/ (besucht am 21. 03. 2016) (siehe S. 25).

[13] *Gaslampgames Biom Erstellung*. URL: https://www.gaslampgames.com/2016/03/23/whats-that-red-stuff/ (besucht am 23. 03. 2016) (siehe S. 28).

[14] *jMonkeyEngine*. URL: http://jmonkeyengine.org/ (siehe S. 15).

[15] *LibGDX Framework*. URL: https://libgdx.badlogicgames.com/index.html (siehe S. 15).

[16] *Light Weight Java Game Library*. URL: https://www.lwjgl.org/ (siehe S. 15).

[17] *TOIBEIndex*. URL: http://www.tiobe.com/tiobe-index/ (siehe S. 15).

[18] *TOIBEIndex Erklärung*. URL: http://www.tiobe.com/tiobe-index/programming-languages-definition/ (siehe S. 15).

[19] *Wikipedia Glatt (Funktion)*. URL: https://de.wikipedia.org/w/index.php?title=Glatte_Funktion&oldid=157192730 (besucht am 19. 11. 2016) (siehe S. 6).

[20] *Wikipedia Gradient Descent*. URL: https://en.wikipedia.org/w/index.php?title=Gradient_descent&oldid=694577758 (besucht am 07. 01. 2016) (siehe S. 6, 7).

[21] *Wikipedia HillClimbing*. URL: https://en.wikipedia.org/w/index.php?title=Hill_climbing&oldid=735241161 (besucht am 24. 08. 2016) (siehe S. 8).

[22] *Wikipedia Konvex (Funktion)*. URL: https://en.wikipedia.org/w/index.php?title=Convex_function&oldid=748525733 (besucht am 08. 11. 2016) (siehe S. 6).