Gregor-Mendel-Gymnasium
Amberg

Abiturjahrgang
2021

S E M I N A R A R B E I T

Rahmenthema des Wissenschaftspropädeutischen Seminars:

Simulationen in Java

Leitfach: *Informatik*

Thema der Arbeit:

Genetische Algorithmen im zellulären Automaten

Verfasser/in:

Daniel Benner

Kursleiter/in:

StD Armin Eckert

Abgabetermin:
(2. Unterrichtstag im November)

10. November 2020

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
schriftliche Arbeit				x 3	
Abschlusspräsentation				x 1	
				Summe:	
				Gesamtleistung nach § 61 (7) GSO = Summe:2 (gerundet)	

Datum und Unterschrift der Kursleiterin bzw. des Kursleiters

Inhaltsverzeichnis

1. Einführung

Seit jeher dient die Natur als Vorbild für Erfindungen und Entwicklungen des Menschen. Angefangen bei Vögeln, deren Flügel die ersten Fluggeräte inspirierten, bis hin zu dem Elefantenrüssel, der die Entwicklung von Roboterarmen maßgeblich beeinflusste. Man spricht hierbei von Bionik, einem Wissenschaftsbereich, der versucht meist technische Probleme mithilfe von Vorbildern aus der Natur zu lösen.
Hier lässt sich auch das Thema dieser Arbeit einordnen: Die Evolutionstheorie und ihre Verwendung in Form von genetischen Algorithmen.

1.1 Biologischer Hintergrund

1859 veröffentlichte der britische Naturwissenschaftler Charles Darwin seine Theorie über die Evolution in seinem Buch „On the Origin of Species." Damit veränderte er das bis dahin gültige Weltbild, da seine Theorie bedeutete, dass der Mensch nicht schon immer so war wie er ist, sondern sich erst dahin entwickelte. Darwin hatte entdeckt, dass viele bekannte Pflanzen- und Tierarten voneinander abstammen oder gemeinsame Vorfahren haben und sich durch sogenannte *Genmutationen*, also Veränderungen in den Erbinformationen, weiterentwickeln. Da es sich hierbei um zufällige Veränderungen handelt, können diese sowohl positive als auch negative Auswirkungen haben und können somit für das Überleben, aber auch das Aussterben, des Individuums oder der ganzen Art verantwortlich sein. Durch diese sog. *Selektion* (natürliche Auslese) bleiben nur die Individuen bestehen, deren Mutationen sich positiv ausgewirkt haben, da diese sich wahrscheinlicher *fortpflanzen* und somit ihre Gene in die nächste Generation übertragen. Über einen langen Zeitraum bringt dieser Prozess eine Art immer näher an das ultimative Ziel: Die perfekte Anpassung an ihre Umgebung.

1.2 Ziel dieser Arbeit

Ziel dieser Arbeit soll es sein, die allgemeine Funktionsweise genetischer Algorithmen anschaulich darzulegen und an einem konkreten Beispiel zu verdeutlichen. Hierbei wird ein genetischer Algorithmus basierend auf bereits existierenden Prinzipien entworfen und bezüglich Grenzen, Möglichkeiten und Verhalten untersucht.

1.3 Aufbau und Funktionsweise eines genetischen Algorithmus

Genetische Algorithmen sind das bionische Äquivalent zu Darwins Evolutionstheorie. Sie verwenden die Grundprinzipien der Evolution: *Mutation*, *Selektion* und *Kreuzung*, um ein ultimatives Ziel zu erreichen. Dabei kann es sich um den besten Weg, die größte Menge oder einfach die beste Möglichkeit handeln. Allgemein eignen sich genetische Algorithmen

hervorragend um jegliche Optimierungsprobleme, wie mathematische Funktionen und deren Extremstellen (vgl. Shawn Keen: Genetische Algorithmen S.7), zu lösen.
Hierbei gehen diese Algorithmen in fünf Schritten vor, wobei die Schritte zwei bis fünf so lange wiederholt werden bis eine Abbruchbedingung (z.B. Anzahl an Generationen, keine Ergebnisveränderungen mehr zwischen den Generationen, Ziel erreicht, …) erfüllt ist:

1. *Initialisierung:* Meist zufälliges Erzeugen der ersten Generation sogenannter Lösungskandidaten
2. *Evaluation:* Jedem Kandidaten wird durch eine Fitnessfunktion eine Güte zugewiesen
3. *Selektion:* Auswahl bestimmter Individuen für Rekombination (basierend auf ihrer Güte)
4. *Rekombination*: Kreuzung der ausgewählten Kandidaten
5. *Mutation:* Zufällige Veränderung des Lösungsweges

Spricht man hierbei von der Fitnessfunktion, ist eine Methode gemeint, die den einzelnen Individuen, basierend auf ihrer Nähe zum Ziel, eine Güte zuweist. Sie wird genutzt, um positive und negative Mutationen voneinander zu trennen und somit die positive Entwicklung zu fördern.

Die folgenden, für diese Arbeit relevanten Fachbegriffe werden in „Evolutionäre Algorithmen: Genetische Algorithmen – Strategien und Optimierungsverfahren – Beispielanwendungen“ von Ingrid Gerdes, Frank Klawonn und Rudolf Kruse (vgl. S. 34) und in „Genetische Algorithmen“ von Bianca Selzam (vgl. S.2ff) erklärt.:

Generation	Die Population während einer Iteration der evolutionären Schleife (Schritte 2-5)
Population	Menge aller Individuen
Start-Population	Population der ersten Generation aus der Initialisierung
Individuum / Chromosom	Lösungskandidat, beinhaltet DNA
DNA	Gesamtheit aller Gene Biologie: speichert Erbinformationen
Gen	Speichert eine Stelle des Lösungswegs
Allel	Ausprägung eines Gens Verändert durch Mutation & Rekombination

2. In dieser Arbeit verwendetes Beispiel

Genetische Algorithmen stellen immer nur ein Prinzip dar, das verändert und an das jeweilige Problem angepasst werden muss, um funktionieren zu können.
Während Jan van Brügge sich mit der Simulation eines Fechtkampfes befasste (vgl. Jan van Brügge: Simulation von genetischen Algorithmen: Zufall am Beispiel von Fechtern), war meine erste Idee für ein Optimierungsproblem die Simulation der Evolution im Tierreich, wie sie in Darwins Evolutionstheorie beschrieben wird. Nach genaueren Überlegungen fiel der Fokus jedoch auf ein simpleres Problem, das die Funktionsweise genetischer Algorithmen aber verständlicher aufzeigen kann. Hierbei soll ein Weg von einem Startpunkt zu einem Endpunkt in einer zweidimensionalen Matrix gefunden und in einer sinnvollen Umgebung dargestellt werden.
Bei der Wahl der richtigen Umgebung für die Visualisierung und Umsetzung stehen zahlreiche Möglichkeiten zur Verfügung. Es sollte sich aber um eine grafische Darstellung handeln, die genau zeigt, was passiert und somit auswertbare, eindeutige Bilder liefert. Die Basis hierfür schafft ein zweidimensionaler, zellulärer Automat. Dieser stellt ein Gitter zur Verfügung, in welchem jedes Feld einzeln und simpel angesprochen werden kann.

3. Die Simulation

Das in den folgenden Abschnitten beschriebene und erklärte Programm wurde ausschließlich von mir verfasst und vollständig in Java geschrieben. Lediglich die Klasse „Kaestchen“, welche für den zellulären Automaten zuständig ist, wurde von Herrn Armin Eckert zur Verfügung gestellt und wird nicht bezüglich ihrer Funktionsweise erklärt.

3.1 Erste Idee der Umsetzung

Wie bereits erwähnt, handelte es sich bei meinen ersten Überlegungen um eine Simulation der im Tierreich stattfindenden Evolution. Hierbei sollten mehrere Gruppen an Individuen im Wettstreit um Nahrung gegeneinander antreten. Um zu überleben, müsste jedes Individuum pro Generation einmal Nahrung auf einer zweidimensionalen Fläche finden und zum Bau seiner Gruppe zurückkehren. Das Erreichen dieses Ziels würde immer abhängen von der Auslegung bestimmter Attribute, wie beispielsweise Geschwindigkeit oder Reichweite des Einzelnen. Mithilfe eines genetischen Algorithmus wären nach jeder Generation diejenigen, die innerhalb einer Gruppe überlebt haben, gekreuzt worden und in der nächsten Generation hätte die Gruppe aus Nachkommen mit den Werten der besten Individuen aus der letzten Generation bestanden. Durch Mutation wären zufällige Veränderungen vorgenommen worden und über einen längeren Zeitraum hätte sich untersuchen lassen, wie sich die Attribute entwickeln und welche Gruppe sich durchsetzt.

Obwohl diese Simulation sicher zu interessanten Ergebnissen geführt hätte, war sie viel zu komplex und hatte keinen wirklichen Sinn, beziehungsweise keine praktische Anwendungsmöglichkeit. Es hätte sich lediglich um eine aufwendige Spielerei gehandelt.

3.2 Zweite Version: Pfadfindung als Ziel

In der Theorie sollte ein genetischer Algorithmus mit dem Ziel, einen Weg zwischen zwei Koordinaten zu finden, wie folgt funktionieren:
In einer zweidimensionalen Matrix werden ein Start- und ein Zielpunkt festgelegt und etwaige Hindernisse platziert. Die Initialisierung wird ausgeführt, d.h. die Start-Population wird erstellt und für jedes Individuum wird eine zufällige DNA generiert. Diese beinhaltet den, aus einzelnen Schritten in eine Richtung bestehenden, Weg des Individuums. Daraufhin wird das erste Mal der Weg und die daraus resultierende Position jedes Einzelnen berechnet und visualisiert. Ausgehend von der letzten Position wird die Fitness, also die Entfernung zum Ziel, erfasst, durch Selektion der Fitteste bestimmt und durch Kreuzung dessen mit den anderen Individuen eine neue Population erschaffen. Der Weg jedes Einzelnen der neuen Population besteht zu einem gewissen Prozentsatz aus dem Weg des Fittesten der letzten Generation und wird nun mutiert. Dabei wird jeder Stelle in der DNA jedes Individuums mit einer festgelegten Wahrscheinlichkeit ein neuer Wert zugewiesen. Es ist eine neue Generation entstanden, die wieder visualisiert werden muss und deren bestes Individuum (das Individuum mit der geringsten Distanz zum Ziel) berechnet werden muss.
Diese Schleife wird so lange ausgeführt, bis eines der Individuen sich auf der Position des Ziels befindet und somit ein Weg gefunden wurde.

3.2.1 Probleme und Lösungen

3.2.1.1 Aufbau der DNA

Für genetische Algorithmen ist die DNA als Binärvektor definiert, das heißt die Gene einer DNA können nur binäre Formen annehmen, also ‚1' oder ‚0'. Da hier aber in einer zweidimensionalen Ebene navigiert wird, und somit ein Schritt in vier mögliche Richtungen erfolgen kann, ist ein Binärsystem nicht ausreichend und es muss mit einem Quarternärsystem gearbeitet werden. So kann ein Gen beispielsweise Zahlen von Null bis Drei beinhalten, oder Zeichen wie ‚L' für links, ‚O' für oben, usw. um die vier Richtungen zu repräsentieren. Hierfür muss von der klassischen Programmierung genetischer Algorithmen abgewichen werden.

3.2.1.2 Auswählen der passenden Mechanismen für Selektion, Kreuzung und Mutation

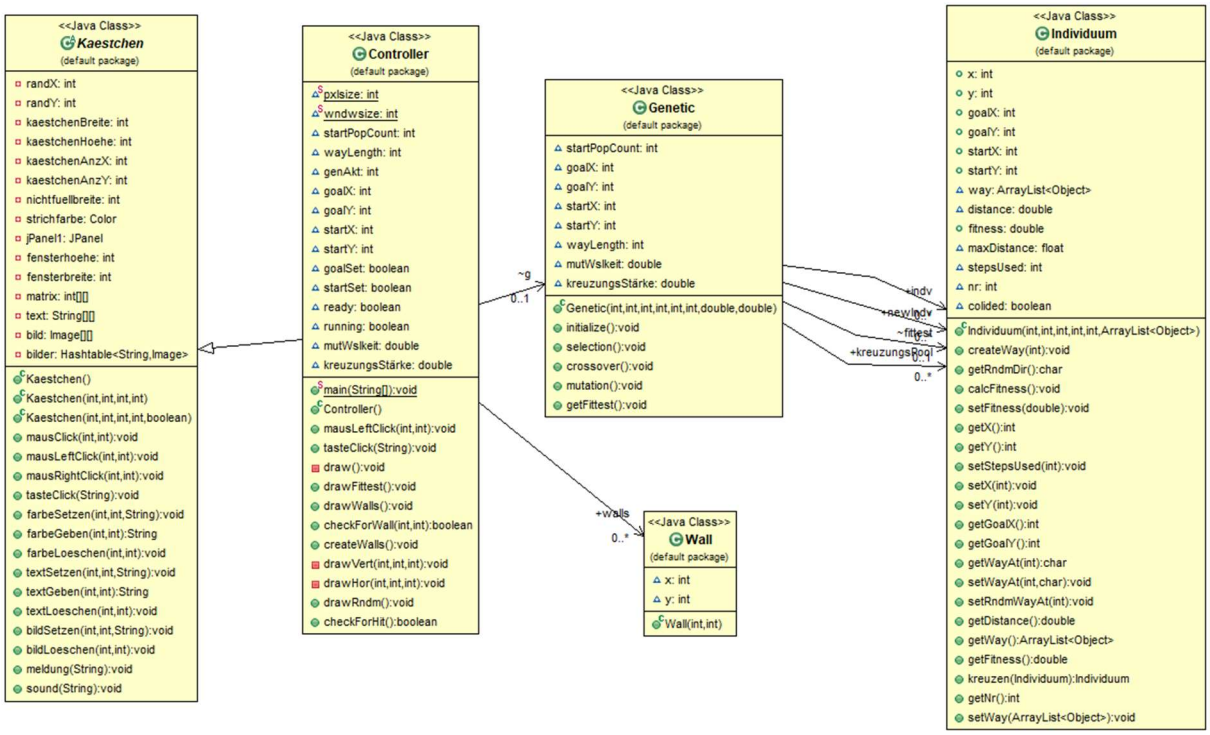
Die grundsätzliche Methode der Selektion kann variieren und ist von dem vorliegenden Optimierungsproblem abhängig. Trotzdem gibt es einige bewährte Strategien.
Die wohl einfachste ist die sogenannte Bestenselektion. Hierbei wird die Liste an Individuen nach der Fitness sortiert und der/ die Beste(n) für die Kreuzung verwendet.
Bessere Ergebnisse bekommt man jedoch meist mit der Turnierselektion (auch Wettkampfselektion genannt). Hierfür werden zufällige Individuen der Population entnommen und miteinander verglichen. Dadurch bekommen auch die Zweit- oder Drittbesten eine Chance, da es nicht garantiert ist, dass der Beste einer Generation überhaupt für den Vergleich ausgewählt wird.
Es existieren noch weitere bekannte Varianten wie die Rouletteselektion oder die Rangselektion, wobei jedem Individuum ein, seiner Fitness entsprechend, großer Bereich in einem Roulettekessel zugewiesen wird, oder alle Individuen in Ränge, basierend auf ihrer Fitness, eingeteilt werden.
Für mein ausgewähltes Optimierungsproblem reicht die Bestenselektion basierend auf der Distanz zum Ziel vollkommen aus. Dadurch lässt sich aber nicht der schnellste Weg vom Start zum Ziel finden, sondern nur ein möglicher Weg. Würde man den besten Weg suchen, müsste man in die Selektion nicht nur die Distanz zum Ziel einberechnen, sondern auch die Anzahl der Schritte, die benötigt werden, um zur aktuellen Position zu gelangen. Mir war es jedoch nicht möglich eine mathematische Formel zu finden, die einen eindeutigen Fitness-Wert ausgibt. Das liegt daran, dass sowohl Distanz als auch Schrittzahl besser bewertet werden, je niedriger sie sind.

Grundsätzliches Ziel der Kreuzung ist es, gute Eigenschaften auf die neue Generation zu übertragen. Oft wird dafür nur mit einer gewissen Wahrscheinlichkeit ein Gen des neuen Individuums mit dem Gen an derselben Position des Besten ausgetauscht. Da aber in meinem Fall der ganze Weg relevant ist und nicht einzelne Gene eines Individuums eindeutig als gut oder schlecht definiert werden können, wird der Anfang des Weges des Besten der vorherigen Generation auf die nächste übernommen, um diese sozusagen grob in die richtige Richtung zu lenken.

Diese grobe Richtung kann jederzeit durch Mutation wieder zunichte gemacht werden. Die Mutation kann aber auch dafür sorgen, dass das Individuum noch näher an sein Ziel kommt. Hierfür gibt es keine genormten, bewährten Strategien. Da Mutationen rein zufällige Abweichungen sind, reicht es aus, die Gene einer DNA mit einer festgelegten Chance abzuändern und so für die gewollte Diversität zu sorgen.

3.2.2 Aufbau des Quellcodes

Das Projekt besteht aus insgesamt fünf Klassen: Der Haupt-Klasse Controller, Genetic, Individuum, Wall und Kaestchen.



Controller beherbergt die main()-Funktion, welche den Konstruktor Controller() (siehe M1) aufruft. Dieser wiederum ruft den Konstruktor der übergeordneten Klasse Kaestchen auf, um das Fenster, welches die Vorgänge visualisiert, zu erstellen und zeichnet darauf vorher festgelegte Wände der Klasse Wall sowie die temporären Start- und Zielpunkte ein. Ab diesem Zeitpunkt wartet das Programm auf einen Input in Form eines Tastendrucks, um die Simulation zu starten. Optional lassen sich davor noch benutzerdefinierte Start- und Zielpunkte setzen. Wird das Startsignal gegeben, erstellt sich die aufgerufene Methode ein Objekt der Klasse Genetic, ruft dessen initialize()-Funktion auf und sperrt den Input, sodass die Simulation nicht mehrmals gleichzeitig gestartet werden kann. Im Anschluss läuft innerhalb eines Timers (siehe M2) der Java-internen Klasse Timer der genetische Algorithmus so lange ab, bis eins der Individuen das Ziel erreicht hat. Hierbei wird die Anzahl an benötigten Iterationen gezählt, sowie in jeder Generation einmal selection(), crossover() und mutation() des Objekts g der Klasse Genetics aufgerufen und die neue Generation, bzw. die Wege ihrer Individuen

```
public Controller() {
        super(pxlsize, pxlsize, wndwsize *2, wndwsize, true);
        createWalls();
        drawWalls();
        farbeSetzen(startX, startY, "blau");
        farbeSetzen(goalX, goalY, "cyan");
}
```
M1

```
public void run() {
        genAkt++;
        g.selection();
        g.crossover();
        g.mutation();
        draw();
        System.out.println("akt. Generation: " + genAkt);
        if(checkForHit() == true) {
                drawFittest();
                cancel();
        }
        drawFittest();
}
```
M2

gezeichnet. Für das Zeichnen ist die Funktion draw() zuständig. Zuerst setzt sie jedes Feld, auf dem sich nicht eine Wand, das Ziel, oder der Start befindet auf „weiß“ und löscht somit alle eingezeichneten Wege der letzten Generation. Daraufhin berechnet sie für jedes Individuum die Position nach jedem Schritt und färbt das Kästchen and dieser Stelle rot (siehe M3). Dadurch übernimmt draw() gleich das Berechnen der neuen Position nach Mutation und Kreuzung und setzt anschließend die Position des Individuums gleich der neuen, berechneten. Befindet sich ein Individuum aber auf der gleichen Position wie eine Wand, ist es „gefangen“ und sein Pfad wird nicht weiter berechnet. Außerdem existiert noch die Funktion drawFittest(), die die gleichen Schritte für den Fittesten durchführt und seinen Weg grün färbt. Dadurch entsteht ein interessantes Bild, welches pro Generation den Weg des Besten umgeben von einer „Wolke“ der weniger erfolgreichen Individuen darstellt.

```
for(int i = 0; i < startPopCount; i++) {
        int tempSteps = 0;
        int tempX = startX, tempY = startY;
        for(int j = 0; j < wayLength; j++) {
                if(((tempX != goalX) || (tempY != goalY))
                && checkForWall(tempX, tempY) == false) {
                        switch(g.indv.get(i).getWayAt(j)) {
                        case 'L':
                                tempX++;
                                break;
                        case 'O':
                                tempY--;
                                break;
                        case 'R':
                                tempX--;
                                break;
                        case 'U':
                                tempY++;
                                break;
                        }
                        tempSteps++;
                }
                farbeSetzen(tempX, tempY, "rot");
        }
        g.indv.get(i).setStepsUsed(tempSteps);
        g.indv.get(i).setX(tempX);
        g.indv.get(i).setY(tempY);
}
drawWalls();
farbeSetzen(startX, startY, "blau");
farbeSetzen(goalX, goalY, "cyan");
```
M3

```
public void initialize() {
        for(int i = 0; i < startPopCount; i++) {
                indv.add(new Individuum(startX, startY,
                goalX, goalY, i, new ArrayList<Object>()));
                indv.get(i).createWay(wayLength);
        }
        fittest = indv.get(0);
}
```
M4

Die Klasse Genetic beinhaltet alle, für den genetischen Algorithmus wichtigen Funktionen und Listen. Sie speichert alle Individuen, sowie den Fittesten und kriegt von der Klasse Controller alle wichtigen Daten, wie Startposition, Zielposition, Anzahl an Individuen, deren Weg-Länge, Kreuzungsstärke und Mutationswahrscheinlichkeit mit. Wird die Funktion initialize() der Klasse aufgerufen, füllt die Funktion die Liste der Individuen mit neuen Objekten und weist jedem Individuum der Liste einen neuen, zufälligen Weg zu (siehe M4). Die Funktion selection() lässt jedes Individuum seine Fitness berechnen und bestimmt mithilfe der Funktion getFittest() den Fittesten. Wie bereits erwähnt, findet hier die Bestenselektion Verwendung (siehe M5). Die crossover() – Funktion übernimmt für jedes Individuum einen Teil des Weges des Besten, beginnend beim Anfang bis zu kreuzungsStärke multipliziert mit der Länge der Liste (siehe M6). Die letzte und simpelste Funktion mutation() geht jedes Gen jedes Individuums durch und ändert dieses zufällig ab, wenn eine Zufallszahl niedriger als die vorher festgelegte Mutationswahrscheinlichkeit ist.

```
public void getFittest() {
        fittest = indv.get(0);
        for(int i = 0; i < startPopCount; i++) {
                if(indv.get(i).getFitness() < fittest.fitness) {
                        fittest = indv.get(i);
                }
        }
}
```
M5

Aus der Klasse Wall werden lediglich Objekte erstellt, die ihre Position beinhalten.

```
public void crossover() {
        for(int i = 0; i < startPopCount; i++) {
                Individuum temp = indv.get(i);
                ArrayList<Object> tempWay = indv.get(i).getWay();
                for(int j = 0; j <  indv.get(i).getWay().size() * kreuzungsStärke; j++) {
                        tempWay.set(j, fittest.getWayAt(j));
                }
                indv.set(i, new Individuum(temp.getX(),temp.getY(), temp.getGoalX(),
                temp.getGoalY(), temp.getNr(), tempWay));
        }
}
```
M6

Die Klasse Individuum dagegen speichert ihre Position, die des Ziels und ihren Weg, sowie Distanz zum Ziel und Fitness. Die createWay()-Funktion füllt die Weg-Liste (die DNA) mit zufälligen Werten ‚L', ‚O', ‚R' oder ‚U', welche Schritte in eine gewisse Richtung symbolisieren. calcFitness() dagegen bestimmt die Distanz zwischen der aktuellen Position und dem Ziel mithilfe des Satzes des Pythagoras: $d = \sqrt{(x - xZiel)^2 + (y - yZiel)^2}$ und setzt die Fitness gleich der Distanz (siehe M7).

```
public void createWay(int wayLength){
        stepsUsed = 0;
        for(int i = 0; i < wayLength; i++) {
                way.add(getRndmDir());
                stepsUsed++;
        }
}

public void calcFitness() {
        distance = Math.sqrt(((x)-(goalX))*((x)-(goalX)) + ((y)-(goalY))*((y)-(goalY)));
        fitness = distance;
}
```
M7

4. Ergebnisse

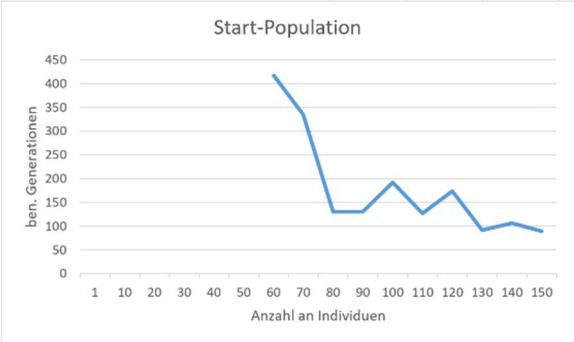
4.1 Untersuchungen und Feststellungen

4.1.1 Auswirkungen der entscheidenden Variablen

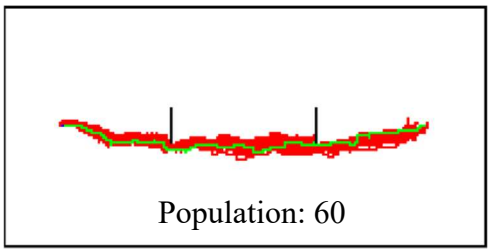
Grundsätzlich bietet das Programm, abgesehen von den Positionen des Start- und des Zielpunktes, vier Variablen, die maßgeblich das Ergebnis der Simulation beeinflussen. Dazu gehört die Start-Population, die Weglänge, die Mutationswahrscheinlichkeit und die Kreuzungsstärke. Im folgenden Abschnitt soll genauer untersucht werden, welche Auswirkungen die verschiedenen Variablen auf das Ergebnis und insbesondere auf die Geschwindigkeit, mit der das Ergebnis erlangt wird, haben. Ausgegangen wird hierbei von einer Fläche mit zwei parallelen Mauern, auf der der genetische Algorithmus einen Weg vom Startpunkt zu dem, 100 Felder entfernten, Zielpunkt finden soll. Bei allen Tests haben die Variablen, die nicht verändert beziehungsweise untersucht werden, die gleichen Werte. Ausgegangen wird von einer Startpopulation bestehend aus 100 Individuen, einer Weglänge von 200 Feldern, einer Mutationswahrscheinlichkeit von 0,03 und einer Kreuzungsstärke von 0,99.

Die genaue Anzahl an benötigten Generationen hängt sehr stark vom Zufall ab, befindet sich aber immer in einem gewissen Bereich. Bei den gemessenen Ergebnissen handelt es sich um Durchschnittswerte.

4.1.1.1 Start-Population



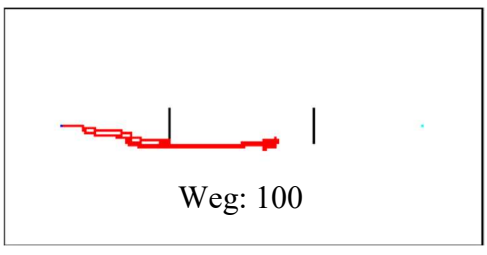
Begonnen wird mit der Start-Population. Diese stellt die Anzahl an Individuen der ersten Generation dar und sollte logischerweise die Effizienz des Algorithmus mit jedem einzelnen weiteren Individuum steigern. Tatsächlich ist der Algorithmus aber erst ab 60 Individuen in der Lage einen Pfad herzustellen. Mit ca. 400 Generationen braucht er dafür auch verhältnismäßig lange, jedoch halbiert sich die Dauer bereits bei 80 Individuen. Ab hier ist die Verbesserung überraschenderweise nur noch gering. Während sich bei 80-100 Individuen Ergebnisse von ca. 150 Generationen ergeben, sind es bei einer Population von 130-150 ca. 100 benötigte Generationen. Erst bei sehr großen Mengen an Individuen lässt sich wieder eine relevante Verbesserung in der Anzahl der Generationen gegenüber 150 Individuen erkennen, allerdings wird die Simulation so langsam, dass kein zeitlicher Vorteil erzielt werden kann.



Trotzdem wird das Ergebnis immer konstanter und es werden weniger Generationen benötigt, je größer die Population wird. Während zu Beginn noch Schwankungen von 100-200 Generationen auftreten können, sind Ergebnisse bereits bei 150 Individuen sehr nah beieinander. Somit scheint ein Wert von 150 für die Start-Population bereits ausreichend, ein höherer Wert kann jedoch gewählt werden, sofern der PC genug Leistung zur Verfügung stellt.

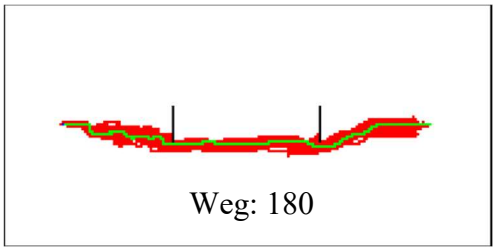
4.1.1.2 Weglänge

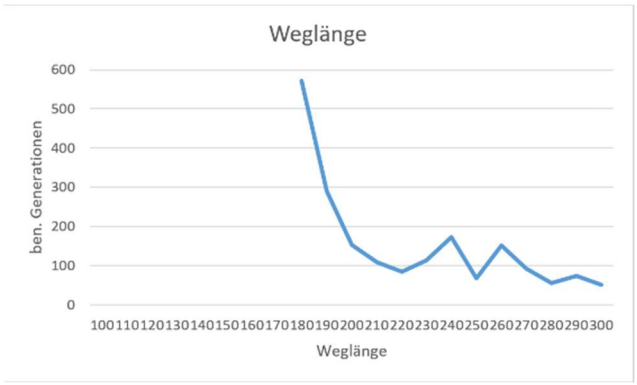
Bei der Weglänge kann natürlich nicht mit einem Wert begonnen werden, der geringer ist als der direkte Abstand zwischen Start und Ziel, daher beginnt die Testreihe hier bei einem Weg der Länge 100. Bei einem Wert, der der direkten Entfernung zwischen Start und Ziel



entspricht, ist es dem Algorithmus allerdings nicht möglich, sein Ziel zu erreichen. Das erste Ergebnis bekommt man bei einer Weglänge von ca. 180, wobei auffällt, wie nah das Ergebnis an dem vermutlich schnellstmöglichen Weg liegt. In den darauffolgenden Messungen lässt diese Eigenschaft immer mehr nach,

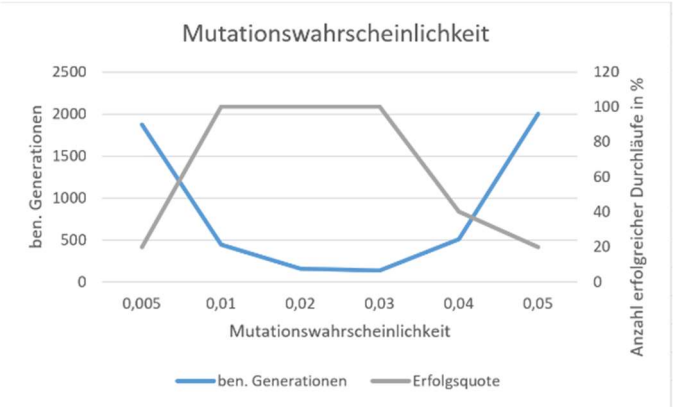
während die Geschwindigkeit zunimmt. Hierbei ergibt sich ein ähnliches Bild wie bei der Population, da die Messungen gerade im Bereich ab 270 um einiges konstanter werden. Zu Beginn findet wieder eine sehr deutliche Verbesserung statt, bei der die Anzahl an benötigten Generationen zwischen 180 und 210 um fast 500 sinkt. Interessanterweise scheinen die Ergebnisse im Bereich 220 – 280 sehr inkonsistent zu sein, wobei die starken Abweichungen sich um zufallsbedingte Ausreißer handeln, da für diese Werte auch Ergebnisse von ca. 100 Generationen erzielt werden können. Im Gegensatz zur Startpopulation benötigt die Weglänge deutlich weniger Performance, weshalb hier auch Werte um 300 oder sogar noch höher gewählt werden können, allerdings auf Kosten der Genauigkeit des Weges. Außerdem muss beachtet werden, dass diese Ergebnisse für eine direkte Distanz zwischen Start und Ziel von 100 erreicht wurden. Ändert sich dies, verschieben sich auch die berechneten Werte. Das heißt, dass die Weglänge, im Gegensatz zur Population, immer unter Beachtung der platzierten Hindernisse und der direkten Entfernung zwischen Start und Ziel gewählt werden sollte. Bewährt hat sich hierfür, die direkte Entfernung mit mindestens 2,5 zu multiplizieren, um stabile Ergebnisse zu erhalten und die Hindernisse und Imperfektionen im Weg auszugleichen.





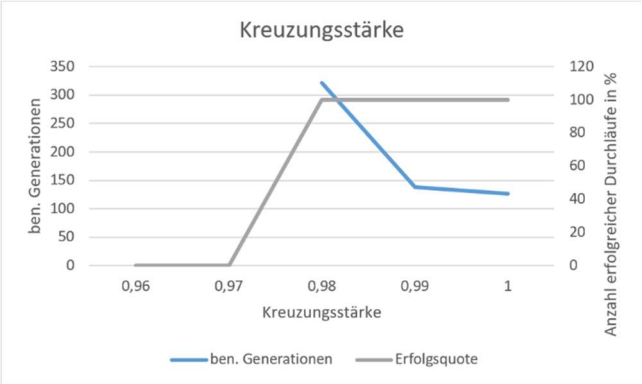
4.1.1.3 Mutationswahrscheinlichkeit

Die Mutationswahrscheinlichkeit gibt an, mit welcher Wahrscheinlichkeit ein Gen eines Individuums mutiert, also seine gespeicherte Bewegung abändert. Sie sorgt für Diversität und ist dafür verantwortlich, dass der Algorithmus einen neuen Weg um ein Hindernis finden kann. Hierbei ist sie sehr präzise zu wählen, da eine zu hohe oder zu niedrige Wahrscheinlichkeit den Algorithmus stark ausbremsen kann. Dementsprechend wurden Werte von 0,005 bis 0,05 getestet, wobei ein interessantes Bild entstanden ist. Unter 0,005 und über 0,05 war es dem Algorithmus nicht möglich einen Pfad herzustellen, währenddessen bei 0,02 und 0,03 die geringste Anzahl an benötigten Generationen mit fünf von fünf erfolgreichen Durchläufen aufgefunden wurde. Für die geringe Erfolgsquote bei 0,005 sorgt die

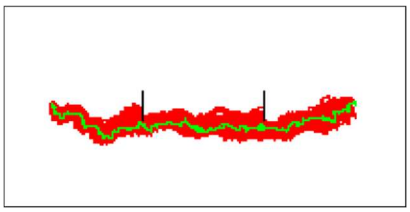


Mutationswahrscheinlichkeit selbst, da einfach nicht genug Diversität entsteht, um die Hindernisse zu überqueren. Ab 0,04 hingegen ergibt sich ein anderes Problem. Der Pfad wird aufgrund der hohen Mutationsrate so uneben und besitzt so viele „Umwege“, dass die Weglänge nicht mehr ausreicht. Somit ist eine Erhöhung der Mutationsrate nur bis zu einem gewissen Grad ohne Erhöhung der Weglänge machbar.

4.1.1.4 Kreuzungsstärke



Der Gegenspieler zur Mutationsrate ist die Kreuzungsstärke. Sie gibt an, wie groß der Bereich des Weges sein soll, der vom Besten der letzten Generation übernommen wird. Im Gegensatz zur Mutationsrate sorgt sie für eine gewisse Ähnlichkeit der Individuen, aber ihr Verringern hat dieselben negativen Auswirkungen wie das Vergrößern der Mutationsrate. Das Vergrößern der Kreuzungsstärke scheint bei diesem Beispiel keine Nachteile zu haben, weil selbst der Maximalwert von eins eine Verbesserung gegenüber 0,99 darstellt. Allgemein sollte die Kreuzungsstärke so groß wie möglich gewählt werden, da sie die Geschwindigkeit der Simulation deutlich erhöht. Falls ein zu großes Hindernis zwischen Start und Ziel liegt, kann es sein, dass ein ähnlicher Effekt wie bei zu geringer Mutationsrate auftritt und der Algorithmus keinen Weg an dem Hindernis vorbei findet. Dann ist es empfehlenswert, die Mutationsrate ein wenig zu erhöhen und dementsprechen auch die Weglänge und eventuell die Populationsgröße anzupassen.



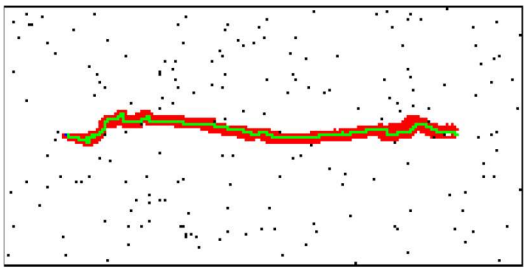
Wendet man nun für alle Variablen die günstigsten Einstellungen an, ist es möglich in nur 22 Generationen einen Weg zu finden. Verwendet wurde hierbei eine Population von 200, eine Weglänge von 400, eine Mutationswahrscheinlichkeit von 0,03 und eine Kreuzungsstärke von eins.

4.1.2 Verhaltensweisen des Algorithmus bei zufällig generierten Hindernissen

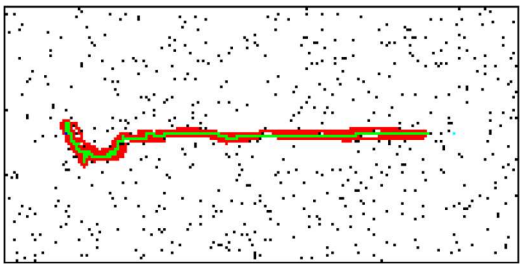
Um die Grenzen und Möglichkeiten meines Algorithmus noch weiter zu erforschen, wurde ein weiteres Szenario implementiert, bei welchem zufällige Hindernisse generiert werden. Hierbei wird ebenfalls versucht, die besten Ergebnisse basierend auf Erfahrungen und Entdeckungen der vorhergehenden Untersuchungen zu erlangen.

Die Implementierung zufällig generierter Hindernisse stellt insofern kein Problem dar, da lediglich für jedes Feld der Matrix ein Zufallswert bestimmt wird und, sofern dieser Wert kleiner als eine vorher festgelegte Wahrscheinlichkeit ist, eine Mauer an dieser Position hinzugefügt wird (siehe M8). Da die Hindernisse größtenteils aus nur einem Feld bestehen und somit der Weg jeweils nur gering angepasst werden muss, sollte die Mutationswahrscheinlichkeit sehr niedrig gewählt werden. Die Kreuzungsstärke hingegen sollte eins entsprechen, da möglichst akkurat durch das Labyrinth navigiert werden muss. Daraus lässt sich bereits schließen, dass ein Durchgang dieser Simulation länger dauern wird, dem kann man aber durch Erhöhen der Start-Population entgegenwirken. Nicht zu empfehlen ist, die Weglänge zu hoch zu wählen, da dies die gleichen negativen Effekte wie eine zu hohe Mutationswahrscheinlichkeit zur Folge hat. Hier sollte sich langsam an eine ausreichende Länge herangetastet werden. Wendet man diese Überlegungen bei einer Chance pro Feld, dass eine Wand entsteht, von 0,01 an, scheint diese Taktik gut zu funktionieren. Verdoppelt man nun aber genannte Chance auf 0,02, erhöht sich die Anzahl an benötigten Generationen von durchschnittlich 150 auf 800. Dieses Muster setzt sich fort und bereits bei einer Wahrscheinlichkeit von 3% scheint es dem Algorithmus unmöglich innerhalb mehrerer tausend Generationen das Ziel zu erreichen. Abhilfe schafft ein Erhöhen der Population und der Weglänge, jedoch verlängert sich dadurch die benötigte Zeit ebenfalls drastisch. Abschließend lässt sich zu diesem Experiment sagen, dass es möglich ist einen Weg bei einer Wahrscheinlichkeit pro Feld von 0.04 oder sogar noch höher zu finden, vorausgesetzt man hat genug Zeit und Perfomance zur Verfügung.

```
public void drawRndm() {
                for(int i = 0; i < wndwsize * 2; i++) {
                        for(int j = 0; j < wndwsize; j++) {
                                if(Math.random() < rndmWallWsl)
                                walls.add(new Wall(i, j));
                        }
                }
        }
```
M8



Chance: 0,02



Chance: 0,03

5. Zukunft genetischer Algorithmen

Ebenso wie andere Formen evolutionärer Algorithmen findet der genetische Algorithmus bereits heute viele Anwendungs- und Einsatzmöglichkeiten. Er repräsentiert die Evolution realer Lebewesen am besten und stellt somit eine geeignete Grundlage dar, Optimierungsprobleme zu lösen, die zuvor zu aufwendig oder sogar unmöglich zu lösen schienen. Basierend auf diesem Ziel der Optimierung, lassen sich durch ihre Hilfe Extremstellen jeglicher mathematischer Gleichungen ermitteln. In der Entwicklung künstlicher Intelligenzen bieten diese Algorithmen die Möglichkeit, eine KI effizient zu trainieren, oder neuronale Netze zu optimieren. Programme wie meins können ein neuronales Netz trainieren, das basierend auf vielen unterschiedlichen Wegen den besten und schnellsten aussucht. Dieses könnte Anwendung finden in Staubsaugrobotern, die einen Weg von einem Zimmer in ein anderes finden müssen, oder in Mährobotern, die möglichst effizient die Wiese mähen sollen.
Derart viele und unterschiedliche Einsatzmöglichkeiten in wachsenden Forschungsgebieten und Industrien wie der Automatisierung garantieren die stetige Weiterentwicklung und Verwendung genetischer Algorithmen in der Zukunft.

6. Quellenverzeichnis

- Autor: Shawn Keen
  Titel: Genetische Algorithmen
  Zu finden auf beigelegtem digitalem Medium oder unter:
  https://www.informatik.uni-ulm.de/ni/Lehre/SS04/ProsemSC/ausarbeitungen/Keen.pdf

- Autor: Jan van Brügge
  Titel: Simulation von genetischen Algorithmen: Zufall am Beispiel von Fechtern
  Zu finden auf beigelegtem digitalem Medium oder unter:
  https://www.hans-riegel-fachpreise.com/fileadmin/hans-riegel-fachpreise/Module/ausgezeichnete-arbeiten/hans-riegel-fachpreise-seminararbeit-vwa-2015-van-bruegge.pdf

- Autoren: Ingrid Gerdes, Frank Klawonn, Rudolf Kruse
  Titel: Evolutionäre Algorithmen: Genetische Algorithmen – Strategien und Optimierungsverfahren – Beispielanwendungen
  ISBN: 978-3-322-86839-8

- Autor: Bianca Selzam
  Titel: Genetische Algorithmen: Projektgruppe 431 – Metaheuristiken
  Zu finden auf beigelegtem digitalem Medium oder unter:
  https://ls11-www.cs.tu-dortmund.de/lehre/SoSe03/PG431/Ausarbeitungen/GA_Selzam.pdf

Internetseiten:

- Zu Bionik:
  - Vögel als Vorbild:
    https://www.planet-wissen.de/technik/luftfahrt/fliegen_nach_dem_vorbild_der_natur/index.html
  - Elefantenrüssel:
    https://www.ingenieur.de/technik/fachbereiche/robotik/roboterarme-so-biegsam-wie-elefantenruessel/
  - Bionik:
    https://www.biokon.de/bionik/was-ist-bionik/
- Zu genetischen Algorithmen:
  - Evolutionstheorie:
    https://www.planet-wissen.de/natur/forschung/evolutionsforschung/pwiesurvivalofthefittestdiehauptthesenderevolutionstheorie100.html
  - Grundsätzliches zu genetischen Algorithmen & Zusammenhang mit KI:
    https://jaai.de/genetische-algorithmen-1496/

Stand 04.11.20: Alle Quellen sind unter genannten Adressen aufrufbar

"Ich habe diese Seminararbeit ohne fremde Hilfe angefertigt und nur die im Quellenverzeichnis angeführten Quellen und Hilfsmittel benützt."

_____

Ort Datum Unterschrift