Survival of the Fittest: Evolutionäre Algorithmen am Beispiel des Spiels Snake

Bachelorarbeit

Tom Pruggmayer
413399

5. Mai 2024

Betreuer: Prof. Dr. Benjamin Blankertz
Dr.-Ing. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Kurzfassung

Ein wichtiger Faktor für die erfolgreiche Anwendung eines Evolutionären Algorithmus ist die richtige Wahl der Simulationsvariablen. Diese Arbeit soll einen Beitrag leisten bestehende Erkenntnisse über die beiden Simualtionsvariablen Generationsgröße und Mutationsgröße zu bestätigen.
In dieser Arbeit wurde ein Evolutionärer Algorithmus entwickelt, der Bots zum Spielen des Spiels Snake trainiert. Inwiefern sich die resultierenden Ergebnisse beim Verändern der Simulationsvariablen Generationsgröße und Mutationsgröße verhalten, wurde hier untersucht. Um die Forschungsfrage zu lösen wurden eine Reihe von Simulationen durchgeführt, bei denen die Simulationsvariablen immer wieder angepasst wurden.
Wie zu erwarten konnte festgestellt werden, dass es für die Generationsgröße einen optimalen Wert gibt der zu der größten durchschnittlichen Spielstärke führt.
Außerdem konnte bestätigt werden, dass bei einer kleinen Generationsgröße das Finden einer guten Lösung stärker vom Zufall abhängig ist, während große Generationsgrößen zuverlässiger gute Lösungen finden.
Des weiteren zeigte sich, dass eine im Laufe der Zeit abnehmende Mutationsgröße bessere Ergebnisse liefert als jede andere untersuchte konstante Mutationsgröße ist.
Der untersuchte Evolutionäre Algorithmus hat für kleine Mutationsgrößen die besten Ergebnisse hervorgebracht während zu große Werte schnell zu immer schlechteren Ergebnissen führten.
Zusammenfassend kann man sagen, dass sich der entwickelte Evolutionäre Algorithmus wie erwartet verhalten hat und die vorher aufgestellten Hypothesen bestätigt werden konnten.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Snake . . . 1
1.2 Ziel der Arbeit . . . 2
1.3 Literatur . . . 3
1.4 Struktur der Arbeit . . . 3

2 Methoden **4**
2.1 Theoretische Grundlagen . . . 4
2.1.1 Künstliche Neuronale Netze (KNN) . . . 4
2.1.2 Evolutionäre Algorithmen . . . 9
2.2 Der Evolutionäre Algorithmus in dieser Arbeit . . . 15
2.2.1 Allgemeine Spielbedingungen . . . 15
2.2.2 Informationsübertragung an das KNN . . . 16
2.2.3 Simulationsvariablen . . . 18
2.2.4 Ablauf des Evolutionären Algorithmus . . . 19
2.2.5 Pseudocode . . . 22
2.3 Durchführung der Experimente . . . 25
2.3.1 Simulationen für verschiedene Generationsgrößen . . . 25
2.3.2 Simulationen für verschiedene Mutationsgrößen . . . 26

3 Ergebnisse **27**
3.1 Ergebnisse für verschiedene Generationsgrößen . . . 27
3.2 Ergebnisse für verschiedene Mutationsgrößen . . . 35

4 Diskussion **43**
4.1 Diskussion Experimente mit verschiedenen Generationsgrößen . . . 43
4.2 Diskussion Experimente mit verschiedenen Mutationsgrößen . . . 44

5 Fazit **45**

6 Anhang **46**
6.1 Git Repositories . . . 46

Abbildungsverzeichnis

1.1 Beispielsituation im Spiel Snake. Das rote Quadrat ist der Apfel und die Reihe von grünen Quadraten die Snake. Links oben in der Ecke sieht man einen aktuellen Score von 23 da bereits 23 Äpfel gefressen wurden. Die Schlange ist 24 Quadrate lang da sie am Anfang des Spiels mit der einem Quadrat startet und mit jedem gefressenen Apfel ein Quadrat länger wird. 2

2.1 Aufbau eines Künstlichen Neurons übernommen aus Wikipedia [5] 5
2.2 Heavyside Funktion mit $\theta = 0$, Grafik erstellt auf geogebra.org[4] 5
2.3 Logistische Funktion mit $\theta = 0$ und verschiedenen k (grün k = 0.2, blau k = 0.5 und rot k = 1, Grafik erstellt auf geogebra.org[4] 6
2.4 Tangens Hyperbolicus mit $\theta = 0$, Grafik erstellt auf geogebra.org[4] 6
2.5 Gleichrichter Funktion (auch Rectifier- oder ReLU-Funktion genannt) mit $\theta = 0$, Grafik erstellt auf geogebra.org[4] 7
2.6 Ein beispielhaftes Feed-Forward-Netz mit einer versteckten Verarbeitungsschicht und drei KN in der Ausgabeschicht. Die einzelnen KN werden mit N[i,j] bezeichnet, wobei i für die jeweilige Schicht und j für die Position in der Schicht steht, in der sich die KN befinden. Quelle: eigene Abbildung 8
2.7 Ablauf eines evolutionären Algorithmus. Quelle: eigene Abbildung 9
2.8 One-Point-Crossover mit Teilung der Elterngenome in der Mitte. Die Gene sind hier durch reelle Zahlen repräsentiert. Quelle: eigene Darstellung 12
2.9 Uniform Crossover. Die Gene sind hier durch reelle Zahlen repräsentiert. Quelle: eigene Darstellung 13
2.10 Real Crossover Method. Die Gene sind hier durch reelle Zahlen repräsentiert. Quelle: eigene Darstellung 13
2.11 Das linke Bild zeigt eine Snake die sich gerade Richtung Osten bewegt,das rechte Bild zeigt die neue Bewegungsrichtung nachdem der Output des KNN „Süden“ ergeben hat und die Snake den nächsten Schritt gemacht hat 15
2.12 Auf dem linken Bild sieht man eine Beispielsituation wie der Abstand vom Kopf der Snake zum Apfel gemessen wird und das Sichtfeld der Snake um den Kopf herum aussieht. In der rechten Tabelle ist die entsprechende Eingabe ins KNN für diese Spielsituation sichtbar. 17

2.13 Hier eine beispielhafte Veranschaulichung des Ablaufs der Reproduktion, mit $GG = 18$ und $BS = 34\%$. Aus den Werten für BS und GG folgt, dass 6 Bots für die Reproduktion ausgewählt werden, diese werden dann in zwei Gruppen von jeweils 3 Bots aufgeteilt, einmal die „weiblichen“ Bots und einmal die „männlichen“ Bots. Die Bilder I., II. und III. zeigen wie durch die Liste der „weiblichen“ Bots iteriert wird und die Rekombination mit jedem „männlichen“ Bot erfolgt. Wenn die entsprechenden Werte für diesen Fall in die Formel 2.8 eingesetzt werden, erhält man $\frac{18}{3 \cdot 3} = 2$, demzufolge werden aus jedem Elternpaar 2 neue Kind-Bots rekombiniert. In Bild III. ist die Reproduktion abgeschlossen und man sieht, dass eine neue Generation mit einer Größe von 18 Bots entstanden ist. 21

3.1 Simulation mit einer Generationsgröße von 250 Individuen 27
3.2 Simulation mit einer Generationsgröße von 500 Individuen 28
3.3 Simulation mit einer Generationsgröße von 750 Individuen 28
3.4 Simulation mit einer Generationsgröße von 1000 Individuen 29
3.5 Simulation mit einer Generationsgröße von 1500 Individuen 29
3.6 Simulation mit einer Generationsgröße von 2000 Individuen 30
3.7 Simulation mit einer Generationsgröße von 3000 Individuen 30
3.8 Simulation mit einer Generationsgröße von 4000 Individuen 31
3.9 Simulation mit einer Generationsgröße von 5000 Individuen 31
3.10 Simulation mit einer Generationsgröße von 10000 Individuen 32
3.11 Simulation mit einer Generationsgröße von 15000 Individuen 32
3.12 Durchschnittliche Verläufe aller simulierten Generationsgrößen, hier aufgeteilt auf zwei Diagramme für eine bessere Übersichtlichkeit 33
3.13 Mittelwerte aus den letzten Generationen aller Simulationsdurchläufe für alle simulierten Generationsgrößen 33
3.14 Varianzen der Mittelwerte aus den letzen Generationen für alle simulierten Generationsgrößen 34
3.15 Simulation mit 0% Mutationsgröße 35
3.16 Simulation mit 1.5% Mutationsgröße 35
3.17 Simulation mit 5% Mutationsgröße 36
3.18 Simulation mit 7.5% Mutationsgröße 36
3.19 Simulation mit 10% Mutationsgröße 37
3.20 Simulation mit 12.5% Mutationsgröße 37
3.21 Simulation mit 15% Mutationsgröße 38
3.22 Simulation mit 20% Mutationsgröße 38
3.23 Simulation mit 30% Mutationsgröße 39
3.24 Simulation mit 35% Mutationsgröße 39
3.25 Simulation mit 50% Mutationsgröße, zu beachten ist, dass die y-Achse in diesem Histogramm bis zu einer Anzhal von 400 und die x-Achse bis zu einer Länge von 30 reicht 40
3.26 Simulation mit im Verlauf der Zeit kleiner werdende Mutationsgröße von 15%-1.5% 40
3.27 Durchschnittliche Verläufe aller simulierten Mutationsgrößen, hier aufgeteilt auf zwei Diagramme für eine bessere Übersichtlichkeit 41

3.28 Mittelwerte aus den letzen Generationen aller Simulationsdurchläufe für alle simulierten Mutationsgrößen . . . 41
3.29 Varianzen der Mittelwerte aus den letzen Generationen für alle simulierten Mutationsgrößen . . . 42

Tabellenverzeichnis

2.1 Belegung der Neuronen an der Eingabeschicht. Die Werte x_{Snake}, x_{Apfel}, y_{Snake}, und y_{Apfel} sind die Koordinaten des Apfels und des Kopf der Snake auf dem Spielfeld . . 16
2.2 Simulationsvariablen für die Simulationen mit verschiedenen GG 25
2.3 Simulationsvariablen für die Simulationen mit verschiedenen MG 26
2.4 Simulationsvariablen für die Simulationen mit sich verändernder MG 26

1 Einleitung

Von den einfachsten Mikroben bis zum Menschen: die Natur hat bewiesen, dass sie durch die einfachen Gesetzmäßigkeiten der Evolution in der Lage ist Lebewesen zu erschaffen die sich auch an die widrigsten Bedingungen anpassen können. Um sich den Erfolg der biologischen Evolution auch für Problemlösungen in der Informatik zunutze zu machen, wurden die Evolutionären Algorithmen entwickelt.
Ein wichtiger Faktor für die erfolgreiche Anwendung eines Evolutionären Algorithmus ist die richtige Wahl der Simulationsvariablen. Diese Arbeit soll einen Beitrag leisten bestehende Erkenntnisse über die beiden Simualtionsvariablen Generationsgröße und Mutationsgröße zu bestätigen. Es soll untesucht werden ob eine veränderliche Mutationsgröße besser ist als eine Konstante. Außerdem soll geprüft werden ob es eine optimale Generationsgröße gibt und wie sich die Ergebnisse für davon abweichende Werte verhalten.
In dieser Arbeit wurde ein Evolutionärer Algorithmus entwickelt, der Bots zum Spielen des Spiels Snake trainiert. Inwiefern sich die resultierenden Ergebnisse beim Verändern der Simulationsvariablen Generationsgröße und Mutationsgröße verhalten, wurde hier untersucht.

1.1 Snake

Snake ist ein klassisches Computerspiel, bei dem ein Spieler eine Schlange steuert und mit dieser, Äpfel in einem quadratischen Spielfeld fressen kann. Die Schlange bewegt sich immer gleichförmig in eine Richtung und der Spieler kann durch betätigen der Pfeiltasten die Richtung der Schlange nach links oder rechts ändern. Mit jedem gefressenen Apfel wird die Schlange ein Stück länger. Wenn die Schlange mit ihrem Kopf den Rand des Spielfelds oder den eigenen Körper berührt, ist das Spiel vorbei. Ziel des Spiels ist es nun eine möglichst viele Äpfel zu fressen und somit eine möglichst lange Schlange zu erhalten. Ursprünglich geht das Spiel auf das 1976 erschienene „Blockade“ von Gremlin Industries zurück, bei dem zwei Spieler gegeneinander spielen und versuchen solange wie möglich zu Überleben. Daraus entwickelten sich viele verschiedenen Varianten und Klone. Nokia brachte 1997 auf dem Nokia 6110 erstmals das Spiel Snake für das Handy heraus, diese Version des Spielprinzips wird in dieser Arbeit betrachtet.[6]

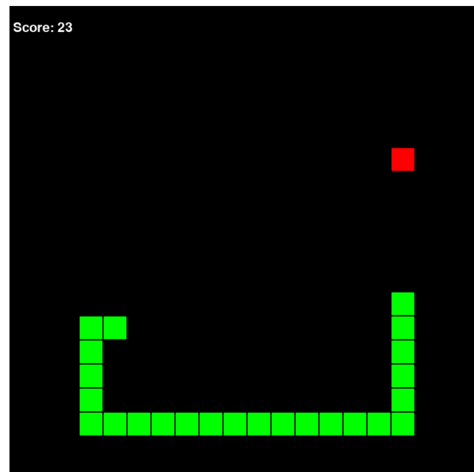


Abbildung 1.1: Beispielsituation im Spiel Snake. Das rote Quadrat ist der Apfel und die Reihe von grünen Quadraten die Snake. Links oben in der Ecke sieht man einen aktuellen Score von 23 da bereits 23 Äpfel gefressen wurden. Die Schlange ist 24 Quadrate lang da sie am Anfang des Spiels mit der einem Quadrat startet und mit jedem gefressenen Apfel ein Quadrat länger wird.

1.2 Ziel der Arbeit

Diese Arbeit beschäftigt sich mit dem Trainieren von Bots zum Spielen von Snake mithilfe eines Evolutionären Algorithmus (EA). Jeder Bot besitzt ein künstliches Neuronales Netzwerk (KNN). Diese KNN werden mit Daten, die über den Spielzustand Auskunft geben, versorgt und treffen dann mit diesen Informationen die Entscheidung in welche Richtung sich die Snake bewegen soll. Ziel der Arbeit ist es herauszufinden inwieweit sich die Leistung der trainierten Snakes verändert wenn die Simulationsvariablen verändert werden.
Es folgt eine Betrachtung wie sich unterschiedliche Mutationsgrößen (MG) auf das Ergebnis der Simulationen auswirken, dabei werden eine Reihe von konstanten MG und eine im Laufe der Zeit kleiner werdende MG miteinander verglichen. Wie schon in anderen Papern gezeigt wurde liefert eine veränderliche MG in den meisten Fällen bessere Ergebnisse als eine Konstante.[15] Ob dieser Zusammenhang auch für die hier betrachteten Simulationen gilt, soll geprüft werden.
Des weiteren werden unterschiedliche Generationsgrößen (GG) miteinander verglichen. Die Wahl einer angemessenen Generationsgröße ist wichtig um eine gute Lösung für das vorliegende Problem zu finden. Zu kleine Werte führen zu suboptimalen Ergebnissen, während zu große Werte unnötig Rechenkapazitäten benötigen und so in der gegebenen Zeit ebenfalls weniger gute Lösungen finden. [10] [9] Es soll getestet werden wo die optimale GG für den vorliegenden EA liegt und wie sich der EA bei großen und kleinen GG verhält.

1.3 Literatur

Der Entwurf eines EA der KNN trainiert um Snake zu spielen, wurde bereits von Piotr Białas in „Implementation of artificial intelligence in Snake game using genetic algorithm and neural networks" [8] untersucht. Die Implementation des EA in dieser Arbeit ist an den von Białas vorgestellten Vorgehensweisen inspiriert.
Der Einfluss von veränderlichen MG wurde unter anderem von Dirk Thierens in „Adaptive mutation rate control schemes in genetic algorithms" [15] untersucht. Seine Betrachtungen und die anderer Studien weisen darauf hin, dass eine veränderliche Mutationsgröße einer Konstanten überlegen ist. In Thierens Arbeit wurden verschiedene komplexere Methoden der Mutationsgrößenanpassung besprochen, die vorliegende Arbeit beschränkt sich aber nur auf die Betrachtung einer gleichmäßig, im Laufe der Zeit abnehmende MG.
Wie in dem Paper „Genetic Algorithms, Noise, and the Sizing of Populations" [9] von David E. Goldberg, Kalyanmoy Deb und James H. Clark unterstrichen wurde, ist die richtige Generationsgröße ein wichtiger Faktor um mit einem EA eine gute Lösung zu finden. Das Paper trifft zusammenfassend die Aussage, dass zu kleine Generationsgrößen stark vom Zufall abhängen ob sie eine gute Lösung für das Problem finden oder bei suboptimalen Lösungen stecken bleiben. Wohin gegen EA mit großen GG und genug Zeit, mit hoher Wahrscheinlichkeit eine gute Lösung finden. Mit Hinblick auf diese Erkenntnisse soll auch dieser EA mit verschiedenen GG untersucht werden.

1.4 Struktur der Arbeit

Zunächst gibt es einen Einblick in die theoretischen Grundlagen die zum Verständnis dieser Arbeit nötig sind. Danach erfolgt eine detaillierte Beschreibung des evolutionären Algorithmus der zum Trainieren der KNN verwendet wurde. Im Anschluss werden die Ergebnisse vorgestellt und interpretiert. Am Ende wird ein Fazit gezogen und ein Ausblick auf die zukünftigen Entwicklungen zu diesem Thema gemacht.

2 Methoden

2.1 Theoretische Grundlagen

2.1.1 Künstliche Neuronale Netze (KNN)

Die folgenden Erläuterungen sind keine vollständige Erklärung aller Konzepte zu diesem Thema. Sie dienen dazu die nötigen Grundlagen zu liefern um die simplen künstlichen Neuronalen Netze, die in den Simulationen verwendet wurden, verstehen zu können. Künstliche Neuronale Netze sind lernfähige Netze die aus künstlichen Neuronen und den dazugehörigen gerichteten Verbindungen bestehen. Jede Verbindung besitzt ein sogenanntes Gewicht w, das einen einfachen Zahlenwert darstellt, der angibt wie stark zwei Neuronen miteinander verbunden sind. Die gerichteten Verbindungen zwischen den Neuronen dienen der Datenübertragung, wobei das Gewicht jeweils mit dem zu übertragenen Wert multipliziert wird. Im Gegensatz zu herkömmlichen Algorithmen, die immer, stur einen vorgegebenen Ablauf abarbeiten, finden KNN durch das Training selber einen Weg, um das Problem zu lösen (ohne, dass von außen sichtbar ist, wie die Probleme gelöst werden). KNN sind in der Lage nicht lineare Probleme zu lösen. [11] [2]

Künstliche Neuronen

Künstliche Neuronen (KN) sind einfache Recheneinheiten, die sich ihrer Struktur und Funktionsweise an den Neuronen aus der Natur orientieren, aus denen z.B. das menschliche Gehirn besteht und die dort für die Datenverarbeitung zuständig sind. Ein Neuron bekommt immer einen Eingabevektor x mit n Werten aus dem dann ein skalarer Wert berechnet wird. Die Berechnung, für ein Neuron mit dem Index j, läuft dabei so ab, dass zuerst jeder Wert x_i des Vektors mit dem dazugehörigen Gewicht w_{ij} multipliziert wird und alle diese Einzelergebnisse dann addiert werden. Diese erste Berechnung produziert die sogenannte Netzeingabe net_j. Zusammengefasst ist die Netzeingabe also definiert als:

$$net_j = \sum_{i=0}^{n} (w_{ij} \cdot x_i) \tag{2.1}$$

Im nächsten Schritt wird die net_j in eine Aktivierungsfunktion φ eingesetzt um, somit die Aktivierung o_j zu berechnen. Dabei wird die Aktivierungsfunktion vom Schwellwert θ_j beeinflusst, er gibt an, an welchem Punkt φ die größte Steigung besitzt. Dieses Verhalten der Aktivierung ist den biologischen Neuronen nachempfunden die ebenfalls eine Reizschwelle besitzen ab der sie angeregt werden. Die Aktivierung berechnet sich also wie folgt:

$$o_j = \varphi(net_j, \theta_j) \tag{2.2}$$

Der ausgegebene Wert für o_j kann nun an die nächsten Neuronen weitergeleitet werden. [2]

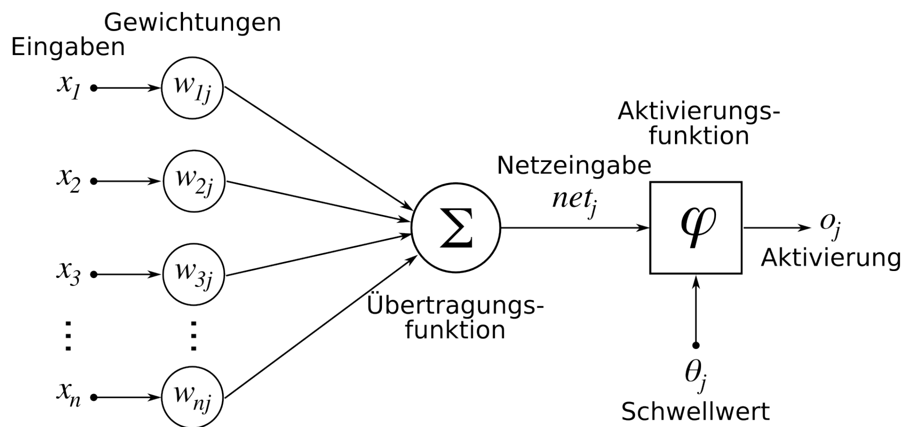


Abbildung 2.1: Aufbau eines Künstlichen Neurons übernommen aus Wikipedia [5]

Aktivierungsfunktionen

Für die Wahl der Aktivierungsfunktion gibt es viele verschiedene Möglichkeiten. Die am häufigsten verwendeten sind die Heavyside-Funktion, die Logistik-Funktion, der Tangens Hyperbolicus und die Gleichrichter Funktion (auch ReLU oder Rectifier Funktion genannt). Welche Aktivierungsfunktion für ein KNN verwendet wird hängt stark, von dem zu lösenden Problem ab. Es empfiehlt sich verschiedene Varianten in Simulationen zu testen und miteinander zu vergleichen. Im Allgemeinen kann man sagen, dass Funktionen mit linearen Verläufen nur für einfachere KNN mit einer versteckten Verarbeitungsschicht geeignet sind, da eine Komposition linearer Funktionen wieder eine lineare Funktion ergibt. [1][2]
Im Folgenden sind die Heavyside-Funktion, die Logistik-Funktion, der Tangens Hyperbolicus und die Gleichrichter Funktion nochmal dargestellt.

Heavyside Funktion [1]

$$\varphi(x, \theta) = \begin{cases} 0 & \text{für } x < \theta \\ 1 & \text{für } x \geq \theta \end{cases}$$

Abbildung 2.2: Heavyside Funktion mit $\theta = 0$,
Grafik erstellt auf geogebra.org[4]

Logistische Funktion [1]

$$\varphi(x, \theta) = \frac{1}{1 + e^{-\frac{x-\theta}{k}}} \tag{2.3}$$

Die Logistische Funktion besitzt eine zusätzliche Variable k, mit der die Steigung um den Schwellwert herum verändert werden kann (siehe Grafik).

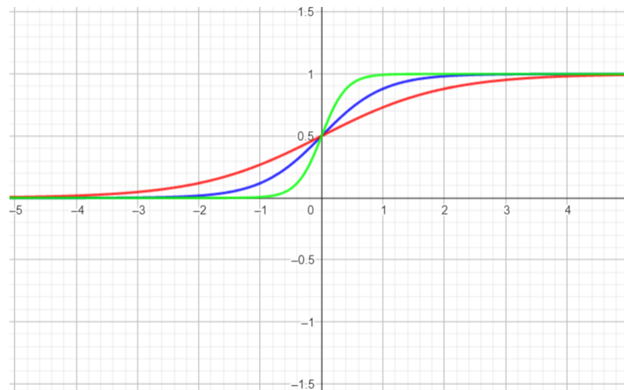


Abbildung 2.3: Logistische Funktion mit $\theta = 0$ und verschiedenen k (grün k = 0.2, blau k = 0.5 und rot k = 1,
Grafik erstellt auf geogebra.org[4]

Tangens Hyperbolicus [1]

$$\varphi(x, \theta) = tanh(x - \theta) \tag{2.4}$$

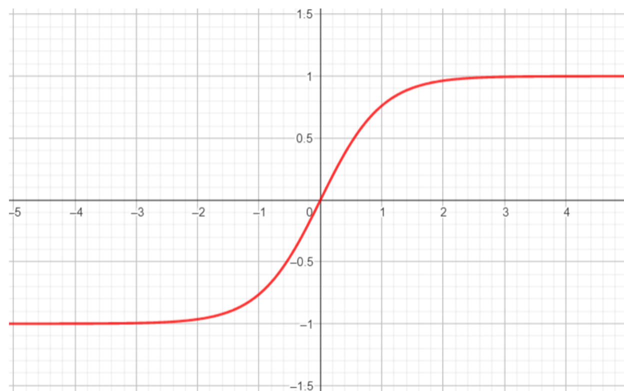


Abbildung 2.4: Tangens Hyperbolicus mit $\theta = 0$,
Grafik erstellt auf geogebra.org[4]

Gleichrichter Funktion [1]

$$\varphi(x,\theta) = \begin{cases} 0 & \text{für } x < \theta_j \\ x & \text{für } x \geq \theta_j \end{cases}$$

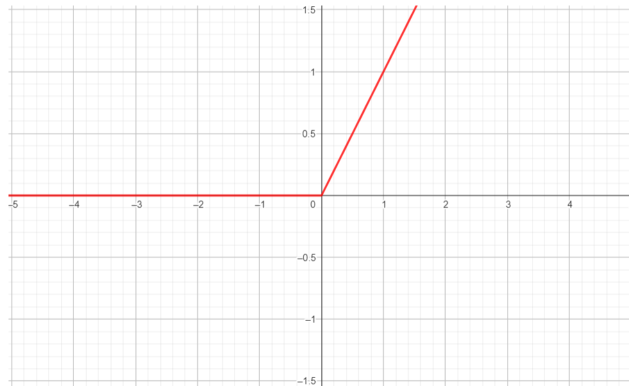


Abbildung 2.5: Gleichrichter Funktion (auch Rectifier- oder ReLU-Funktion genannt) mit $\theta = 0$, Grafik erstellt auf geogebra.org[4]

Aufbau künstlicher Neuronaler Netze

Eine häufige Art der KNN, die auch für die Simulationen in dieser Arbeit verwendet wurden, sind die so genannten Feed-Forward-Netze. Diese Netze bestehen aus mehreren Schichten von künstlichen Neuronen, dabei gibt es immer eine Eingabeschicht (engl. *Input Layer*), eine oder mehrere versteckte Verarbeitungsschichten (engl. *Hidden Layer*) und eine Ausgabeschicht (engl. *Output Layer*). Ein Beispiel für ein Feed-Forward-Netz ist in der folgenden Abbildung 2.6 sichtbar. Dieses Netz besitzt eine Eingabeschicht mit n KN, eine versteckte Verarbeitungsschicht mit m KN und eine Ausgabeschicht mit drei KN. Wie in der Abbildung zu sehen sind die künstlichen Neuronen bei Feed-Forward-Netzen immer nur mit den direkt benachbarten Schichten verbunden und geben ihre Output Werte gerichtet von links nach rechts weiter. Es gibt also kein Verbindungen zwischen KN in der selben Schicht oder Rückkopplungen in die vorherige Schicht. [2]

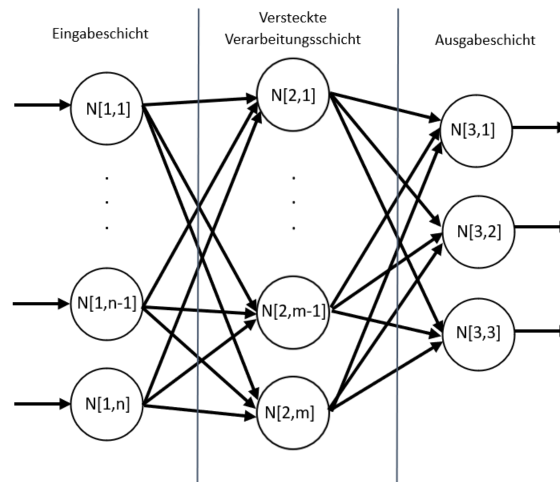


Abbildung 2.6: Ein beispielhaftes Feed-Forward-Netz mit einer versteckten Verarbeitungsschicht und drei KN in der Ausgabeschicht. Die einzelnen KN werden mit N[i,j] bezeichnet, wobei i für die jeweilige Schicht und j für die Position in der Schicht steht, in der sich die KN befinden. Quelle: eigene Abbildung

Trainieren

Künstliche Neuronale Netze zu trainieren ist ein sehr umfangreiches Thema, zu dem es viele Herangehensweisen gibt. Allgemein gilt, dass die Bestandteile des Netzes stückweise verändert werden bis das Netz die gewünschten Ausgaben liefert. Gängige Maßnahmen zum Trainieren sind: die Werte der Gewichte anpassen, die Schwellenwerte der Aktivierungsfunktionen anpassen, löschen oder hinzufügen von Verbindungen etc. Es ist aber möglich nur durch die Veränderung der Gewichte bereits sehr gute Lernergebnisse zu erreichen. Die häufigsten Lernverfahren sind: überwachtes Lernen (engl. *Supervised Learning*), unüberwachtes Lernen (engl. *Unsupervised Learning*) und bestärkendes Lernen (engl. *Reinforcement Learning*). [2]
Auf diese Lernverfahren wird in dieser Arbeit nicht weiter eingegangen da die KNN der Bots mit einem Evolutionären Algorithmus trainiert werden, dazu mehr im nächsten Kapitel.

2.1.2 Evolutionäre Algorithmen

Evolutionäre Algorithmen sind eine Klasse von Optimierungsverfahren. Ihre Funktionsweise ist der natürliche Evolution nachempfunden. EA besitzen mannigfaltige Anwendungsmöglichkeiten, z.B. in der Industrie, in der Finanzwelt oder in der Robotik, um nur einige Beispiele zu nennen. Im Folgenden wird die Funktionsweise eines EA zum Trainieren von Bots zum Spielen eines Spiels erklärt. Jeder Bot hat ein KNN, dass die Entscheidungen während des Spiels trifft. Nach dem Generieren der ersten Generation gliedert sich der Algorithmus in die folgenden fünf Phasen: Spielphase, Bewertung der Fitness, Selektion, Reproduktion und Mutation. Der Ablauf der einzelnen Phasen ist schematisch in Abbildung 2.7 zu sehen. [8]

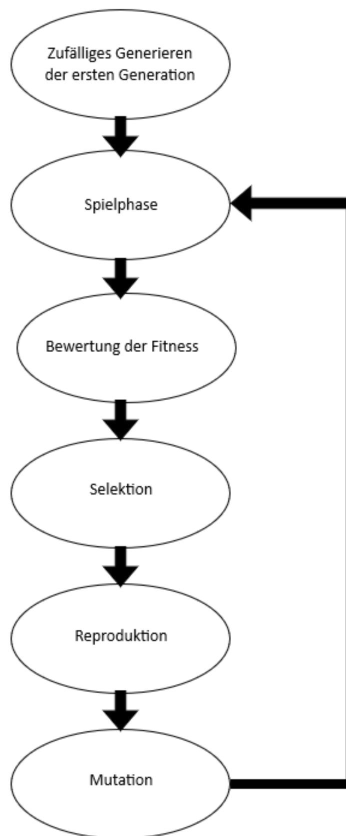


Abbildung 2.7: Ablauf eines evolutionären Algorithmus. Quelle: eigene Abbildung

Zufälliges Generieren der ersten Generation

Der erste Schritt besteht darin, die initiale Generation zu generieren. Das Ziel dieser Phase ist es eine anfängliche Generation von Individuen mit gleichverteilten, zufälligen Genen zu bilden, auf die dann die Optimierung durch die Mittel der Evolutionären Algorithmen angewendet werden kann. Die Gene bilden hierbei die Gewichte des KNN. Dafür werden so viele KNN mit zufälligen Gewichten generiert, wie im Vorfeld als Generationsgröße (GG) definiert wurden. Jedes Individuum wird durch eines der KNN repräsentiert.[8] [17]

Spielphase

Jedes Individuum in der aktuellen Generation spielt das Spiel entweder bis eine festgelegte Zeit abgelaufen ist oder bis es das Spiel verloren hat. [8]

Bewerten der Fitness

Wenn alle Individuen einer Generation das Spiel gespielt haben, bekommen jedes Individuum, auf Grundlage einer Fitness-Funktion, eine Bewertung zugewiesen. Die Fitness-Funktion misst anhand einer oder mehrerer Parameter wie gut das jeweilige Individuum das Spiel gespielt hat. Mögliche Parameter können z.B. die erreichte Punktzahl, die Spielzeit etc. sein. Die Beschaffenheit der Fitness-Funktion ist stark von der Aufgabe abhängig die die Bots lösen sollen. Um eine Fitnessfunktion zu finden muss darüber nachgedacht werden welches Verhalten der Bots wahrscheinlich zum gewünschten Ziel führt. Oft ist es sinnvoll nicht nur beim erreichen des Ziels einen hohen Fitnesswert zu vergeben, sondern auch schon Teilerfolge zu belohnen. Es ist vorteilhaft verschiedene Varianten auszuprobieren, immer unter dem Gesichtspunkt welche Funktion das beste Ergebnis liefert. Es ist außerdem möglich die Fitness-Funktion im Laufe der Simulation anzupassen, um so zu versuchen bestimmte Eigenschaften der Bots stärker zu Gewichten. [12] [8]

Selektion

Nachdem jedes Individuum einen Fitness-Wert erhalten hat, geht es nun in die Selektions-Phase. In diesem Schritt geht es darum die Individuen auszuwählen die sich im nächsten Schritt reproduzieren und ihre Gene an die nächste Generation weitergeben. Hier gibt es viele verschiedene Möglichkeiten die Individuen auszuwählen, Beispiele sind:

Proportionate Selection [3]:
Bei diesem Verfahren werden Individuen zufällig ausgewählt die Wahrscheinlichkeit für ein einzelnes Individuum ausgewählt zu werden wird durch die folgende Formel berechnet:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{2.5}$$

f_i : Fitnesswert eines Individuums
n : Anzahl der Individuen in der Generation

Tournament Selection [7]:
Dabei werden aus k zufällig ausgewählten Individuen einer Generation, die Besten, mit einer bestimmten Wahrscheinlichkeit P, ausgewählt. Die Wahrscheinlichkeit P wird berechnet durch:

$$P = p \cdot (1 - p)^i \tag{2.6}$$

p : Grundwahrscheinlichkeit
i : Fitness-Rang des jeweiligen Individuums, das beste Individuum hat i=0 das zweitbeste i=1 und so weiter (i=[0...k-1])

Die richtige Belegung der Parameter p und k wird experimentell durch Variation der Werte ermittelt.

Truncation Selection [16]:
Hierbei werden alle Individuen der Generation nach ihren Fitness-Werten sortiert und dann werden die besten x% ausgewählt. Die passende Größe von x sollte ebenfalls durch ausprobieren verschiedener Werte ermittelt werden.
Welche Art der Selektion die Richtige für ein gegebenes Problem ist muss durch Experimente herausgefunden werden. [16] [3] [7]

Reproduktion

In der Phase der Reproduktion (engl. *Crossover*) pflanzen sich die vorher selektierten Individuen fort und produzieren so die neue Generation von Bots. Hierbei gibt es ebenfalls viele verschiedene Herangehensweisen, die je Problemstellung einen hohen Einfluss auf das Ergebnis haben können. Ein wichtiger Faktor für der Wahl der Art der Reproduktions Methode ist die Art der Problemrepräsentation. Die Gene der Individuen können entweder durch einzelne Bits, reelle Zahlen oder Integer repräsentiert werden. Im Allgemeinen werden immer zwei Bots ausgewählt, die dann ihre Gene rekombinieren. Dabei sind grundsätzlich zwei Fragen zu klären: erstens wie die Bots ausgewählt werden und wie die Rekombination der Gene erfolgt. Im Folgenden werden gängige Arten der Reproduktion vorgestellt.[8] [17]

One-Point Crossover:
Hierbei werden aus den zuvor selektierten Individuen zwei zufällig ausgewählt (hier als Vater- und Mutter-Bot bezeichnet). Danach werden alle ihre Gene als lange Listen betrachtet die dann in an einem bestimmten Punkt in zwei kleineren Listen geteilt werden. Aus jeweils einer Teilliste von Mutter- und Vater-Bot wird dann das Genom eines neuen Individuums gebildet (hier der Kind-Bot). Der Punkt an dem die Listen getrennt werden, kann entweder fest sein oder zufällig gewählt werden. Diese Methode ist sowohl für Genrepräsentationen mit Bits, reelen Zahlen und mit Integern möglich. Der Prozess ist in der Abbildung 2.8 veranschaulicht. [17]

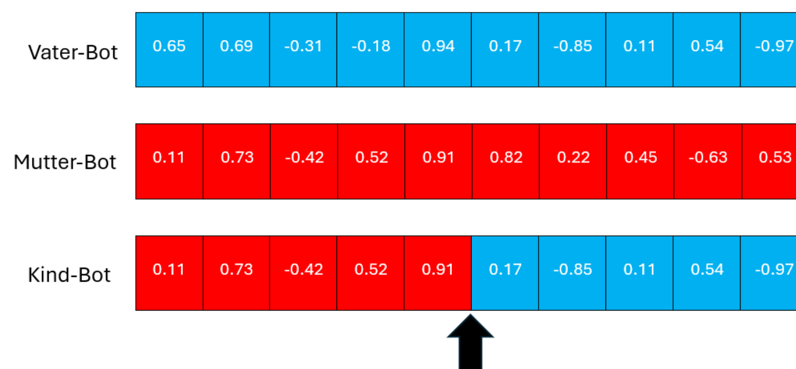


Abbildung 2.8: One-Point-Crossover mit Teilung der Elterngenome in der Mitte. Die Gene sind hier durch reelle Zahlen repräsentiert. Quelle: eigene Darstellung

Uniform Crossover:
Eine weitere verbreitete Form des Crossovers ist das Uniform Crossover. Hierbei werden wieder zwei Bots als Eltern für den neuen Bot ausgewählt. Im Anschluss werden alle ihre Gene als lange Listen betrachtet und durch beide Listen wird parallel iteriert. Beim Iterieren wird an jeder Position zufällig das jeweilige Gen des Vaters oder das der Mutter ausgewählt und für den Kind-Bot verwendet. Diese Form des Crossovers gilt als eine der besten Varianten und sorgt für eine gleichmäßige Mischung der Gene der Eltern.[17]

Vater-Bot	0.65	0.69	-0.31	-0.18	0.94	0.17	-0.85	0.11	0.54	-0.97
Mutter-Bot	0.11	0.73	-0.42	0.52	0.91	0.82	0.22	0.45	-0.63	0.53
Kind-Bot	0.65	0.73	-0.42	-0.18	0.91	0.17	-0.85	0.11	-0.63	0.53

Abbildung 2.9: Uniform Crossover. Die Gene sind hier durch reelle Zahlen repräsentiert. Quelle: eigene Darstellung

Real Crossover:
Wenn die Gene durch reelle Zahlen repräsentiert werden, kann auch die *Real Crossover* Methode verwendet werden. Jedes neue Gen des Kind-Bots wird dann aus dem Durchschnitt der beiden Gene der Eltern (an dieser Position) berechnet. Der Ablauf ist in Abbildung 2.10 nochmal verdeutlicht. [17] [8]

Vater-Bot	0.65	0.69	-0.31	-0.18	0.94	0.17	-0.85	0.11	0.54	-0.97
Mutter-Bot	0.11	0.73	-0.42	0.52	0.91	0.82	0.22	0.45	-0.63	0.53
Kind-Bot	0.38	0.71	-0.365	0.17	0.925	0.495	-0.315	0.28	-0.045	-0.22

Abbildung 2.10: Real Crossover Method. Die Gene sind hier durch reelle Zahlen repräsentiert. Quelle: eigene Darstellung

Mutation

Im Anschluss an die Reproduktionsphase, durchläuft die neu kreierte Generation nun die Mutationsphase. In dieser Phase wird nun ein bestimmter Prozentsatz an Individuen der neuen Generation ausgewählt und deren Erbgut anschließend mutiert. Das Mutieren erfolgt so, dass ein gewisser Anteil der Gewichte des KNN eines Bots zufällig verändert werden. Das kann so aussehen, dass diese Gewichte komplett durch eine Zufallszahl ersetzt werden oder aber mit einem gewissen zufälligen Wert addiert werden. Diese zufällige Veränderung der Gene ist notwendig, um den Suchraum nach guten Lösungen absuchen zu können. Die Mutationsrate darf jedoch auch nicht zu groß sein, da die Bots sonst nicht in der Lage sind dazuzulernen (da zu viele Gewichte nach jedem Duchlauf der Mutation durch Zufallszahlen ersetzt werden). Es kann ratsam sein, den Anteil an mutierten Bots und den Anteil an den Gewichten die mutiert werden im Laufe der Simulation zu verändern. Ähnlich wie bei der Selektion ist es auch hier hilfreich zu Beginn eher viele Bots zu mutieren und eine höhere Mutationsrate zu haben und beide Werte zum Ende hin kleiner werden zu lassen. Die hohe Anzhal an zufälligen Veränderungen ist zum Start hilfreich um alle hinreichend guten Lösungskanidaten ausfindig zu machen. Später können durch weniger Mutation bereits gefundene, gute Lösungen weiter verfeinert werden, ohne dabei die Fortschritte durch zu viel Zufälligkeit zu verlieren. [16] [15][8]

2.2 Der Evolutionäre Algorithmus in dieser Arbeit

Nachdem im vorherigen Abschnitt die allgemeine Funktionsweise von EA gezeigt wurden, wird im Folgenden der für diese Arbeit entwickelte EA erklärt.

2.2.1 Allgemeine Spielbedingungen

Das Spiel findet auf einem 20x20 Feldern großen Spielfeld statt. Die Snake hat am Anfang die Länge eins und startet auf einem zufälligen Feld innerhalb eines 10x10 Feldes um die Mitte des Spielfeldes herum. Es wird immer ein Apfel an einer zufälligen Stelle platziert. Wenn die Snake den Apfel gefressen hat, wird ein Neuer zufällig platziert. Die Snake bewegt sich durchgängig und kann vom Bot nach „Norden“, „Süden“, „Osten“ oder „Westen“ gesteuert werden (Abhängig vom Output des KNN). Abhängig davon, in welche Richtung sich die Snake aktuell bewegt, biegt sie dann nach links oder rechts ab oder behält ihre Richtung bei.
Wenn sich zum Beispiel die Snake gerade nach „Osten“ bewegt und das KNN entscheidet, dass die neue Bewegungsrichtung „Süden“ ist wird die Snake nach rechts abbiegen (Siehe Abbildung 2.11). Wenn die Entscheidung wiederum auf „Westen“ fällt behält die Snake ihre Richtung bei, da es nicht möglich ist sich direkt entgegengesetzt zur aktuellen Bewegungsrichtung zu drehen.

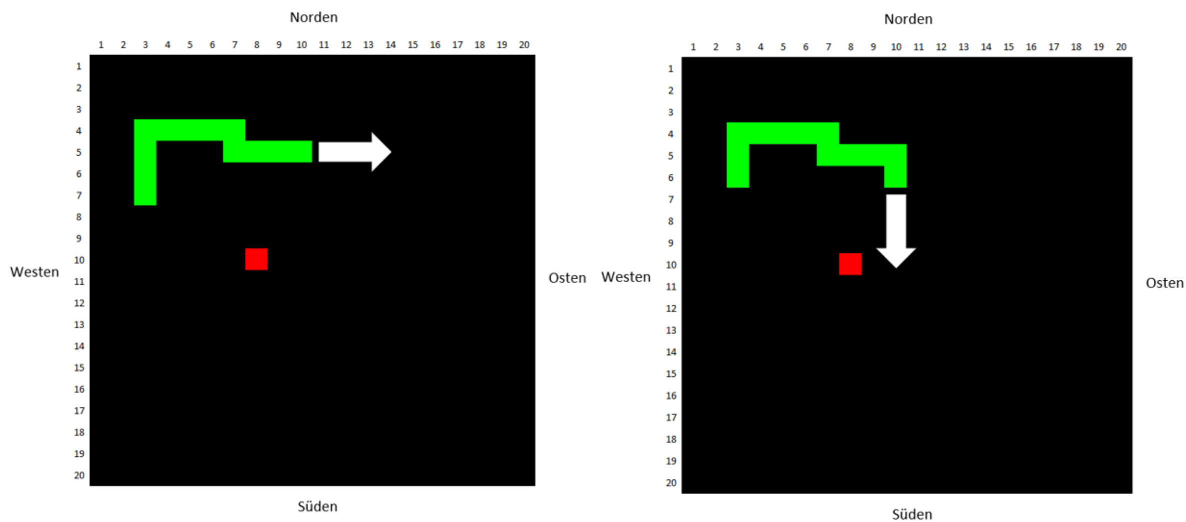


Abbildung 2.11: Das linke Bild zeigt eine Snake die sich gerade Richtung Osten bewegt,das rechte Bild zeigt die neue Bewegungsrichtung nachdem der Output des KNN „Süden“ ergeben hat und die Snake den nächsten Schritt gemacht hat

Um die Zeit adäquat messen zu können, wird das ganze Geschehen in einzelne Zeitschritte aufgeteilt. Ein Zeitschritt ist das Weiterbewegen der Snake um ein Feld nach vorne, links oder rechts. Um dem Spiel eine Zeitbegrenzung zu geben, bekommt die Snake mit jedem gefressenen Apfel 150 Zeitschritten dazu. Wenn die Snake alle Zeitschritte verbraucht hat, ist das Spiel vorbei. Das Spiel endet außerdem, wie gewöhnlich, wenn der Kopf der Snake die Spielfeldgrenze oder den eigenen Körper berührt.

2.2.2 Informationsübertragung an das KNN

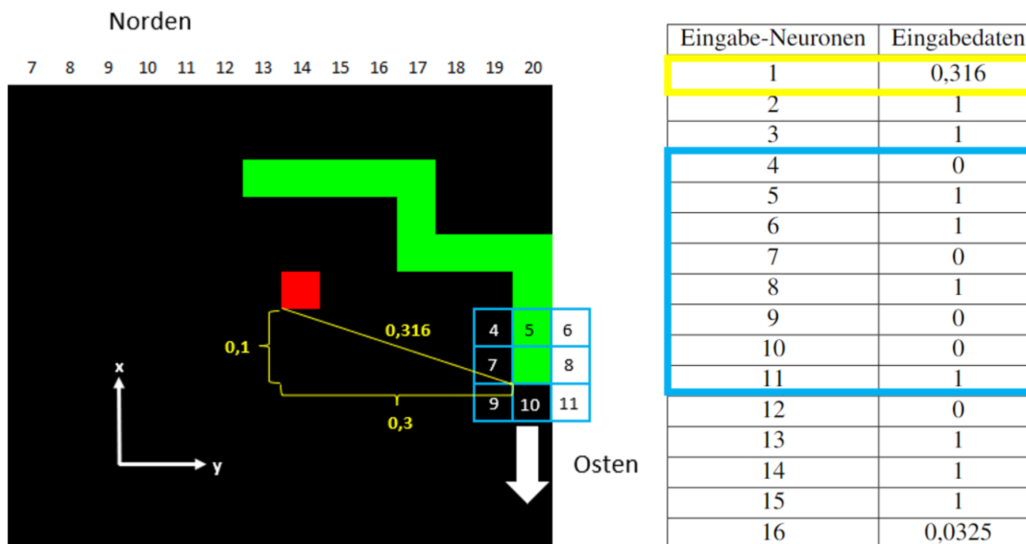
Jede Snake wird durch die Ausgaben eines KNN gesteuert, in das einmal pro Zeitschritt Daten über die aktuelle Spielsituation gegeben werden. Welche Daten das KNN bekommt ist in Tabelle 2.1 beschrieben.

Eingabe-Neuronen	Eingabedaten
1	Abstand vom Kopf der Snake zum Apfel, wird berechnet mit: $\sqrt{(\frac{x_{Snake}-x_{Apfel}}{Fenstergröße})^2 + (\frac{y_{Snake}-y_{Apfel}}{Fenstergröße})^2}$
2	1 wenn $x_{Snake} - x_{Apfel} > 0$, sonst 0
3	1 wenn $y_{Snake} - y_{Apfel} > 0$, sonst 0
4	Das Qudrat in der 1. Reihe und 1. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
5	Das Qudrat in der 1. Reihe und 2. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
6	Das Qudrat in der 1. Reihe und 3. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
7	Das Qudrat in der 2. Reihe und 1. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
8	Das Qudrat in der 2. Reihe und 3. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
9	Das Qudrat in der 3. Reihe und 1. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
10	Das Qudrat in der 3. Reihe und 2. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
11	Das Qudrat in der 3. Reihe und 3. Spalte im 3x3 Feld um den snake-Kopf, 0 wenn dort freies Feld ist, 1 wenn dort der Körper der Snake oder die Wand ist
12	1 wenn sich die Snake im nächsten Zeitschritt nach Norden bewegen kann, sonst 0
13	1 wenn sich die Snake im nächsten Zeitschritt nach Süden bewegen kann, sonst 0
14	1 wenn sich die Snake im nächsten Zeitschritt nach Westen bewegen kann, sonst 0
15	1 wenn sich die Snake im nächsten Zeitschritt nach Osten bewegen kann, sonst 0
16	Anzahl der Qudrate die die Snake lang ist, im Verhältnis zur gesamten Anzahl an Quadraten des Spielfeld, wird berechnet mit: $\frac{Quadrate_{Snake}}{Quadrate_{Spielfeld}}$

Tabelle 2.1: Belegung der Neuronen an der Eingabeschicht. Die Werte x_{Snake}, x_{Apfel}, y_{Snake}, und y_{Apfel} sind die Koordinaten des Apfels und des Kopf der Snake auf dem Spielfeld

Die Daten die den Eingabe-Neuronen erhalten wurden intuitiv ausgewählt. Das Ziel war es eine möglichst gute Beschreibung der Spielsituation zu liefern mit möglichst wenig Eingabe-Neuronen, um die Rechenzeit zu verringern. Die Eingabewerte 1 bis 3 sollen dem Bot zeigen wo sich der Apfel befindet. Die Werte 4 bis 11 stellen ein Sichtfenster für die direkt angrenzenden Felder um den Kopf der Snake herum dar. Dieses Sichtfenster soll es dem Bot ermöglichen einen freien Weg zu finden, ohne die Spielfeldgrenze oder den eigenen Körper zu berühren. Die Werte 12 bis 15 geben Auskunft darüber,

in welche Richtung sich die Snake aktuell bewegt und demzufolge auch, welche Bewegungsrichtungen als nächstes möglich sind. Der Wert für das Neuron 16 wurde einbezogen um eine Information darüber zu liefern, welche Länge die Snake aktuell besitzt, so dass der Bot die Möglichkeit hat sein Spielverhalten daran anpassen zu können. Die Eingabewerte wurden intuitiv normalisiert da Studien gezeigt haben, dass Normalisierung für das Trainieren von KNN mittels Backpropagation zu einer schnelleren Lernrate und einer niedrigeren Fehlerrate führen können. [14]



Eingabe-Neuronen	Eingabedaten
1	0,316
2	1
3	1
4	0
5	1
6	1
7	0
8	1
9	0
10	0
11	1
12	0
13	1
14	1
15	1
16	0,0325

Abbildung 2.12: Auf dem linken Bild sieht man eine Beispielsituation wie der Abstand vom Kopf der Snake zum Apfel gemessen wird und das Sichtfeld der Snake um den Kopf herum aussieht. In der rechten Tabelle ist die entsprechende Eingabe ins KNN für diese Spielsituation sichtbar.

In Abbildung 2.12 ist eine Beispielsituation in einem Spiel von Snake auf einem Ausschnitt des 20x20 Spielfeldes. Der Kopf der Snake befindet sich am rechten Rand des Spielfeldes, die Blau markierten Quadrate um den Kopf herum stellen das Sichtfeld der Snake dar. Die Eingabe in das KNN für diesen Fall ist in der rechten Tabelle eingetragen.
Die erste, gelb markierte Zeile enthält den Abstand vom Kopf der Snake zum Apfel (genaue Berechnung siehe Tabelle 2.1). Die Zeile 2 enthält den Wert 1 weil sich der Apfel weiter nördlich als der Kopf der Snake befindet. Ebenso erhält das Eingabe-Neuron 3 eine 1 da sich der Apfel weiter westlich als der Kopf der Snake befindet (für die Berechnung siehe Tabelle 2.1).
Die Eingabe-Neuronen 4 bis 11 erhalten entweder 1 oder 0 als Eingabe, je nach dem ob sich die Snake in die umliegenden Felder bewegen kann oder nicht. Das Eingabe-Neuron 12 bekommt eine 0 da sich die Snake momentan nach Süden bewegt und deswegen im nächsten Schritt nicht ihre Richtung auf Norden ändern kann. Dementsprechend bekommen die Eingabe-Neuronen von 13 bis 15 die Werte 1 da sich die Snake im nächsten Zeitschritt in diese Richtungen bewegen kann. Die Zeile 16 enthält den Wert $0,0325$ da die Snake 13 Felder lang ist und das gesamte Spielfeld 400 Felder groß ist ($\frac{13}{400} = 0,0325$, siehe auch Tabelle 2.1)

2.2.3 Simulationsvariablen

Im Nachfolgend werden alle Simulationsvariablen erklärt die das Verhalten des entwickelten EA bestimmen. In den weiteren Erklärungen auf den nächsten Seiten werden meistens die entsprechenden Abkürzungen der Variablen verwendet.

Generationsgröße (GG):

Die Generationsgröße definiert wie viele Individuen jede Generation enthält. Sie wird am Anfang einmal festgelegt und kann während der Simulation nicht mehr verändert werden.

Bestsize (BS):

Die Bestsize legt fest wieviel Prozent der Bots einer Generation für die Produktion der nächsten Generation ausgewählt werden. Dabei werden immer die x% Individuen mit den höchsten Fitnesswerten gewält. Der Wert kann während der Simulation, durch vorher festgelegt Start- und Endwerte, angepasst werden.

Mutationsgröße (MG):

Die Mutationsgröße definiert wie viel Prozent an Individuen einer Generation, nach dem Crossover, für die Mutation ausgewählt werden. Die zu mutierenden Individuen werden dabei zufällig aus der Menge der neuen Individuen ausgewählt. Diese Variable kann während des Verlaufs der Simulation angepasst werden (wie das gemacht wird wird im Abschnitt „Durchführung der Experimente“ erklärt).

Mutationsfaktor (MF):

Der Mutationsfaktor legt fest wie viele Gewichte in den KNN, der für die Mutation ausgewählten Individuen, mutiert werden. Die Gewichte die in den KNN mutiert werden, werden zufällig ausgewählt. Auch dieser Wert kann während der Simulation, durch vorher festgelegte Start- und Endwerte, angepasst werden.

Timesteps (TS):

Die Variable Timesteps definiert wieviele zusätzliche Zeitschritte jedes Individuum beim Fressen eines Apfels erhält. Mit jedem Schritt den die Snakes beim Spielen machen wird ein Zeitschritt verbraucht. Wenn die Snake kein Zeitschritte mehr hat ist das Spiel beendet. Diese Variable wurde fest auf 150 für alle Simulationen festgelegt.

2.2.4 Ablauf des Evolutionären Algorithmus

Zufälliges Generieren der ersten Generation

Für die erste Generation werden so viele KNN erstellt wie vorher in GG definiert worden sind. Es handelt sich dabei immer um 16x11x4 KNN. Das bedeutet, dass sich in der Eingabeschicht 16, in der versteckten Verarbeitungsschicht 11 und in der Ausgabeschicht 4 künstlichen Neuronen befinden. Die 4 KN in der Ausgabeschicht stehen jeweils für die Bewegungsrichtungen „Norden", „Süden", „Osten" und „Westen". Das KN das nach der Eingabe der Daten in die Eingabeschicht den höchsten Wert ausgibt bestimmt die Bewegungsrichtung im nächsten Zeitschritt. Die Gewichte werden mit gleichverteilten, reellen Zahlen zwischen -1 und 1 initialisiert. Als Aktivierungsfunktion wird die ReLU-Funktion verwendet, dessen Schwellwert fest bei $\theta = 0$ liegt. Das Trainieren der KNN erfolgt später ausschließlich über die Veränderung der Gewichte.

Spielphase

In der Spielphase startet jeder Bot mit $TS = 150$ Zeitschritten auf seinem Zeitkonto. Mit jedem gefressenen Apfel, bekommt der Bot weitere TS Zeitschritte. Das Spiel endet für den Bot wenn das Zeitkonto leer ist. Das Spiel endet außerdem, wie gewöhnlich, wenn der Kopf der Snake die Spielfeldgrenze oder den eigenen Körper berührt. Das Spielen läuft so ab, dass die Snake bei jedem Zeitschritt die Informationen über den aktuellen Spielstand in sein KNN gespeist bekommt (siehe Abschnitt „Informationsübertragung an das KNN"). Das KNN berechnet daraus die neue Bewegungsrichtung (also „Norden", „Süden", „Osten" oder „Westen") und diese wird dann von der Snake auf dem Spielfeld umgesetzt.

Bewerten der Fitness

Zum Bewerten der Fitness wurde die folgende Fitnessfunktion F verwendet:

$$F = l^2 + \frac{\tau}{TS} \tag{2.7}$$

l : Länge die die Snake erreicht hat
τ : Zeitschritte die die Snake gespielt hat
TS : Die Simulationsvariable TS hat den Wert 150

Für die Wahl der Fitnessfunktion gibt es viele Möglichkeiten und sie kann, je nach dem was das gewünschte Endresultat ist, verändert werden. In diesem Fall wurde eine Funktion gewählt die sowohl die erreichte Länge l der Snake als auch die Zeit, die die Snake gespielt hat, τ mit einkalkuliert. Der Gedanke dahinter ist, dass es am Anfang wichtig ist, dass die Bots erst einmal lernen nicht gegen die Spielfeldgrenze zu laufen, da sie dann sofort verlieren würden. Zusätzlich wird l^2 addiert um das eigentliche Ziel, die Maximierung der Länge, ebenfalls zu belohnen. Dadurch, dass l quadriert wird fällt die erreichte Länge am Anfang bei kleinen Längen noch nicht so stark ins Gewicht im Verhältnis zur Division aus τ und TS. Bei großen Längen ist die Spielzeit dann wiederum vernachlässigbar und hat fast keinen Einfluss mehr da l^2 sehr stark wächst.

Selektion

Die Selektion gestaltet sich so, dass die gesamte Generation nach Ihren Fitness-Werten sortiert wird. Im Anschluss werden davon die Besten $x\%$ ausgewählt (x lag in den durchgeführten Simulationen immer bei 10%). Danach wird die Menge der Auserwählten noch einmal gemischt um im nächsten Schritt, bei der Reproduktion, eine bessere Durchmischung der Erbanlagen zu gewährleisten.

Reproduktion

Im Anschluss an die Selektion erfolgt nun die Reproduktion. Dafür wird im ersten Schritt die Menge der vorher selektierten Bots in zwei gleich große Gruppen aufgeteilt, „weibliche" und „männliche" Bots. Im zweiten Schritt werden beide Gruppen nach den Fitnesswerten sortiert, so dass die besten Bots der beiden Gruppen die meisten Nachkommen produzieren. Im dritten Schritt wird jeweils ein Bot aus der Menge der „weiblichen" Bots ausgewählt und dann mit allen „männlichen" Bots rekombiniert. Dabei produziert jedes Paar mehrere Nachkommen, so dass die neue Generation vollständig mit neuen Bots aufgefüllt werden kann. Die Menge der produzierten Nachkommen pro Elternpaar (P) wird mit folgender Formel berechnet:

$$P = \frac{GG}{\#F \cdot \#W} \tag{2.8}$$

GG : Generationsgröße
$\#M$: Anzahl der „männlichen" Bots
$\#W$: Anzahl der „weiblichen" Bots

Durch die Formel wird garantiert, dass immer mindestens genau so viele Bots produziert werden, dass die neue Generation wieder auf eine Gesamtgröße von GG kommt.
Der ganze Prozess wird so lange wiederholt bis jeder „weibliche" Bot mit jedem „männlichen" Bot rekombiniert wurde. Wenn mehr Individuen produziert wurden, als in GG definiert ist, werden die überschüssigen aussortiert so, dass die neue Generation wieder genau gleich groß ist wie die Vorherige. Bei der Rekombination wird die *Uniform Crossover* Methode verwendet (Erklärung siehe „*Uniform Crossover*" im Abschnitt 2.1.2).
In Abbildung 2.13 ist der Ablauf der Reproduktion nochmal an einem kleinen Beispiel erklärt.

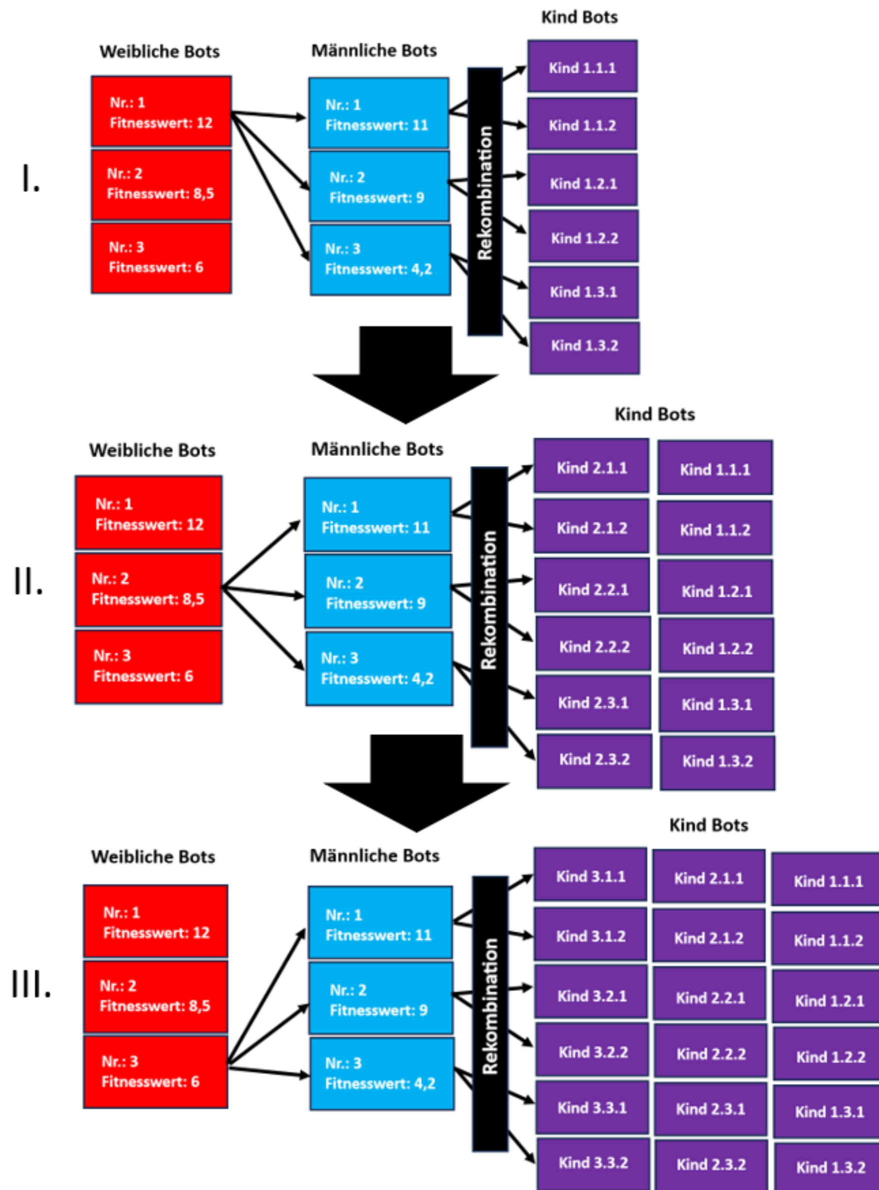


Abbildung 2.13: Hier eine beispielhafte Veranschaulichung des Ablaufs der Reproduktion, mit $GG = 18$ und $BS = 34\%$. Aus den Werten für BS und GG folgt, dass 6 Bots für die Reproduktion ausgewählt werden, diese werden dann in zwei Gruppen von jeweils 3 Bots aufgeteilt, einmal die „weiblichen“ Bots und einmal die „männlichen“ Bots. Die Bilder I., II. und III. zeigen wie durch die Liste der „weiblichen“ Bots iteriert wird und die Rekombination mit jedem „männlichen“ Bot erfolgt. Wenn die entsprechenden Werte für diesen Fall in die Formel 2.8 eingesetzt werden, erhält man $\frac{18}{3 \cdot 3} = 2$, demzufolge werden aus jedem Elternpaar 2 neue Kind-Bots rekombiniert. In Bild III. ist die Reproduktion abgeschlossen und man sieht, dass eine neue Generation mit einer Größe von 18 Bots entstanden ist.

Mutation

Im letzten Schritt durchläuft die neu gebildete Generation die Mutation. Hier werden prozentual so viele Bots für die Mutation ausgewählt wie in der Simulationsvariable MG definiert ist. Von den ausgewählten Individuen werden $MF\%$ der Gewichte ihrer KNN mutiert. Wenn ein Gewicht mutiert werden soll, wird sein ursprünglicher Wert durch eine gleichverteilte, reelle Zufallszahl zwischen -1 und 1 ersetzt.
Nachdem auch die Mutation beendet wurde, kommt die neue Generation wieder in die Spielphase und der Prozess beginnt von neuem. Der Algorithmus wird so lange wiederholt bis die am Anfang definierte Simulationszeit zu Ende ist.

2.2.5 Pseudocode

In den folgenden Abbildungen wird der Pseudocode für den Programmcode des EA dargestellt und kurz erklärt.

Hauptalgorithmus

In Listing 2.1 ist der Hauptalgorithmus des EA dargestellt. In den Zeilen 1 bis 5 sind nochmal die Simulationsvariablen eingetragen, sie müssen vor dem Start der Simulation mit Werten initialisiert werden. Sie werden in den folgenden Listings als globale Variablen betrachtet.
In Zeile 7 wird die Variable „snakes“ mit der ersten Generation von zufällig generierten KNN initialisiert. Ab Zeile 8 wird dann die Simulation so lange ausgeführt bis die Simulationszeit zu Ende ist. Der Algorithmus ruft die Funktionen „evalFitness()“ und „newGeneration()“ auf, diese werden in den nächsten Abschnitten gezeigt.

Listing 2.1: Hauptalgorithmus des EA

```
1  GG : Number of snakes per Generation
2  BS : Percentage of snakes choosen for reproduction
3  MF : Percentage of weights to be mutated
4  MG : Percentage of snakes to be mutated
5  TS : Gained time steps per eaten apple
6
7  snakes = initializeFirstGeneration()
8  while time left do:
9      ergebnisse_generation := (empty list)
10     for snake in snakes do
11         snake_player := (emtpty list)
12         snake_player.append(tile at a random position)
13         apple := tile at a random position
14         t := TS
15         steps := 0
16         while t >= 0 do
17             t = t - 1
18             steps = steps + 1
19             move := getMoveFromNN(snake)
20             snake_player = makeMove(snake_player,move)
21             if snake_player collides with border of board then:
22                 exit loop
```

```
23              if snake_player collides with own body then:
24                  exit loop
25              if snake_player collides with apple then:
26                  snake_player.append(tile)
27                  t = t + TS
28          fitness := evalFitness(snake_player, steps)
29          ergebnisse_generation.append((fitness, snake))
30      snakes = newGeneration(snakes)
```

Bewertung der Fitness

In Listing 2.2 ist die Funktion zum Bewerten der Fitness der Bots zu sehen. Die Funktion berechnet dabei die Formel 2.7, Die im Abschnitt davor definiert wurde.

Listing 2.2: Code zum berechnen der Fitnessfunktion

```
1 function evalFitness(steps : integer;
2                     snake_player : list) : float
3     fitness_value = snake_player.length^2 + steps/TS
4     return fitness_value
```

Neue Generation generieren

In Listing 2.3 ist die Funktion zum Generieren einer neuen Generation dargestellt. Sie enthält die Phasen der Selektion und der Reproduktion. Innerhalb von „newGeneration()“ wird die Funktion „mutateGeneration()“ aufgerufen die die neue Generation mutiert.

Listing 2.3: Code zum Generieren einer neuen Generation

```
1  function newGeneration(snakes : list) : list
2      snakes.sortByFitness()
3      snakes = snakes[(1-BS)*snakes.length : snakes.length]
4      snakes.shuffle()
5      femaleSnakes := [0.5*snakes.length : snakes.length]
6      maleSnakes := [0 : 0.5*snakes.length]
7      femaleSnakes.sortByFitness()
8      maleSnakes.sortByFitness()
9      newSnakes = (empty list)
10     for f_snake in femaleSnakes do
11         for m_snake in maleSnakes do
12             while not enough snakes do
13                 newSnake = crossover(m_snake, f_snake)
14                 newSnakes.append(newSnake)
15     newSnakes = mutateGeneration(newSnakes)
16     return newSnakes
```

Mutation

In Listing 2.4 ist die Funktion zum Mutieren einer Generation dargestellt.

Listing 2.4: Code zum Mutieren einer Generation

```
function mutateGeneration(newSnakes : list) : list
    newSnakes.shuffle()
    to_be_mutated = := [0 : snakes.length*MG]
    normal := [snakes.length*MG : snakes.length]
    mutated = (empty list)
    for obj in to_be_mutated do
        mutated.append(mutate(obj,MF))
    mutatedSnakes := concat(normal, mutated)
    mutatedSnakes.shuffle()
    return mutatedSnakes
```

2.3 Durchführung der Experimente

Wie bereits in der Einleitung erwähnt, wurden in dieser Arbeit zwei Simulationsvariablen für die Durchführung der Experimente besonders betrachtet: die MG und die GG. Dafür wurden Simulationen mit verschiedenen Werte für GG und MG durchgeführt. Verändert wurden immer nur diese beiden Variablen während die restlichen in jeder Simulation gleich blieben.
Jeglicher Code wurde in Python (Version: 3.10.6) geschrieben.
Alle Simulationen wurden auf dem „hydra cluster " der TU Berlin mit der job partition cpu-5h, ausgeführt (für weiteren Informationen, siehe Link im Anhang) .
Jede Simulationsvariante wurde 400 Mal mit den gleichen Variablen ausgeführt um später die Endresultate aus allen Ergebnissen der einzelnen Durchläufe mitteln zu können. Jeder Durchlauf wurde immer für 3 Stunden simuliert. Nach jedem Simulationsdurchlauf wird eine .pkl-Datei gespeichert die jede einzelne Länge für alle Snakes in allen Generationen enthält. Zusätzlich werden in dieser Datei auch die Zeitpunkte gespeichert in denen jede Generation von der Spielphase in die Reproduktionsphase über geht.

2.3.1 Simulationen für verschiedene Generationsgrößen

In Tabelle 2.2 sind die Belegungen der Simulationsvariablen für die Simulationsdurchläufe mit verschiedenen GG aufgelistet. Die Werte in den eckigen Klammern sind alle GG die einzeln simuliert wurden, also wurden insgesamt 11 verschiedene GG simuliert. Die Werte für GG wurden so gewählt um einen möglichst guten Überblick über den Einfluss dieser Variable liefern zu können.

Simulationsvariablen	Verwendete Werte
GG	[250, 500, 750, 1000, 1500, 2000, 3000, 4000, 5000, 10000, 15000]
BS	10%
MG	15%
MF	15%
TS	150

Tabelle 2.2: Simulationsvariablen für die Simulationen mit verschiedenen GG

2.3.2 Simulationen für verschiedene Mutationsgrößen

In Tabelle 2.3 sind die Belegungen der Simulationsvariablen für die Simulationsdurchläufe mit verschiedenen MG aufgelistet. Die Werte in den eckigen Klammern sind alle MG die einzeln simuliert wurden, also wurden ebenfalls insgesamt 11 verschiedene MG simuliert. Die Werte für MG wurden so gewählt um einen möglichst guten Überblick über den Einfluss dieser Variable liefern zu können.

Simulationsvariablen	Verwendete Werte
GG	1000
BS	10%
MG	[0%, 1.5%, 5%, 7.5%, 10%, 12.5%, 15%, 20%, 30%, 35%, 50%]
MF	15%
TS	150

Tabelle 2.3: Simulationsvariablen für die Simulationen mit verschiedenen MG

Außerdem wurde noch eine Simulation mit sich im Laufe der Zeit anpassende $MG(t)$ durchgeführt. Dabei wurde $MG(t)$ durch die Formel 2.9 im Laufe der Simulation vor dem Mutieren jeder neuen Generation neu berechnet:

$$MG(t) = \frac{MG_{Ende} - MG_{Start}}{T} \cdot t + MG_{Start} \tag{2.9}$$

MG_{Start} : Startwert der Mutationsgröße
MG_{Ende} : Endwert der Mutationsgröße
t : Aktuelle Zeit
T : Gesamte Simulationszeit

Die Belegung der Simulationsvariablen sah dabei folgendermaßen aus:

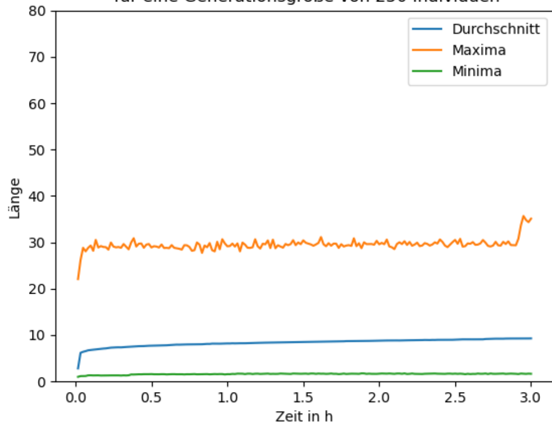
Simulationsvariablen	Verwendete Werte
GG	1000
BS	10%
MG_{Start}	15%
MG_{Ende}	1.5%
MF	15%
TS	150

Tabelle 2.4: Simulationsvariablen für die Simulationen mit sich verändernder MG

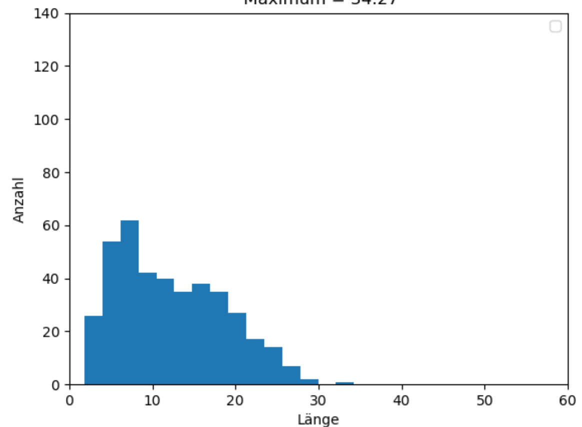
3 Ergebnisse

3.1 Ergebnisse für verschiedene Generationsgrößen

Im Folgenden werden die Ergebnisse der Experimente beschrieben. Zum Plotten der Diagramme mit dem Titel „Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit“ wurden zuerst aus allen Simulationsdurchläufen für jede Generation die durchschnittliche Länge berechnet. Diese wurden dann, abhängig von der Zeit wann die jeweilige Generationen von der Spielphase in die Reproduktionsphase übergegangen sind, in einzelne Zeitintervalle sortiert. Danach wurden dann von den Werten in den Zeitintervallen der Durchschnitt berechnet und diese Ergebnisse wurden dann geplottet. Die Ergebnisse wurden auf diese Weise dargestellt, da die verschiedenen Simulationsdurchläufe in der vorgegebenen Zeit unterschiedlich viele Generationen durchlaufen haben.
Für die Verläufe von Maxima und Minima wurden dann noch die jeweiligen maximalen und minimalen Werte aus den Zeitslots geplottet. Für die Diagramme unter den Titeln „Histogramm über die durchschnittlichen Längen“ wurde von allen letzten Generationen jedes Simulationsdurchlaufs die durchschnittlichen Längen berechnet und als Histogramme dargestellt. Da also für jede Simulation 400 Simulationsdurchläufe gemacht wurden enthält jedes Histogramm 400 Werte.

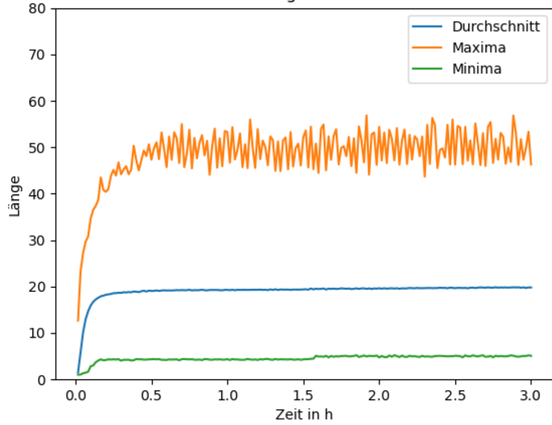


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

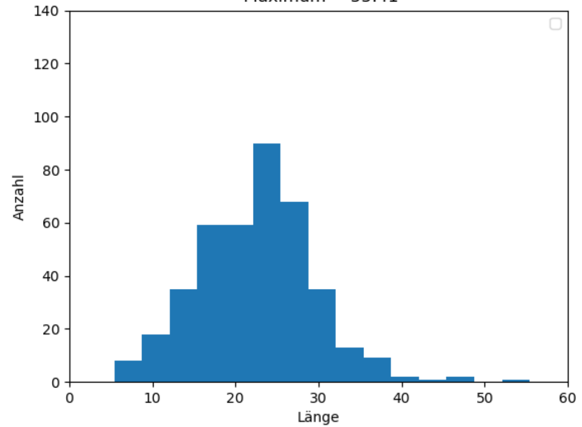


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.1: Simulation mit einer Generationsgröße von 250 Individuen

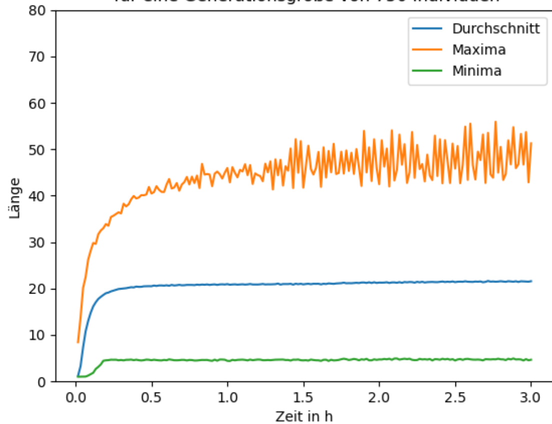


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

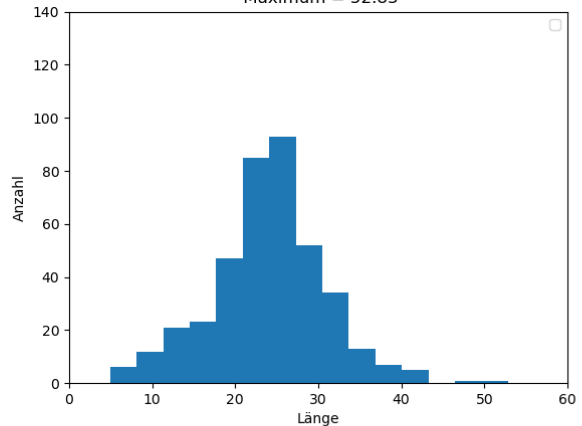


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.2: Simulation mit einer Generationsgröße von 500 Individuen

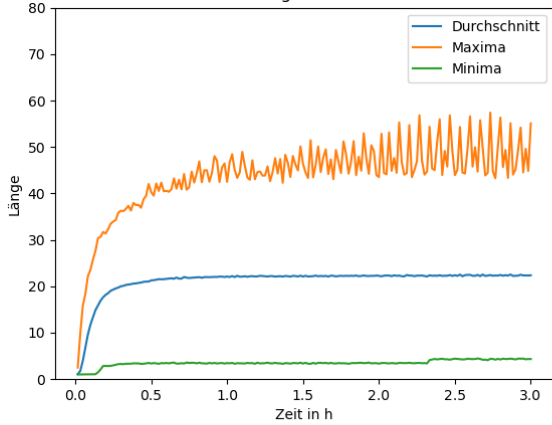


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

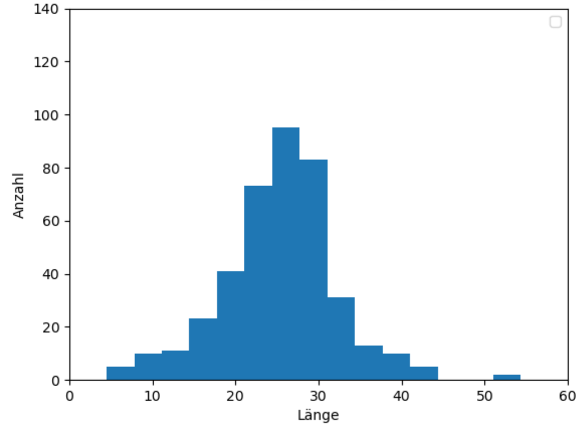


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.3: Simulation mit einer Generationsgröße von 750 Individuen

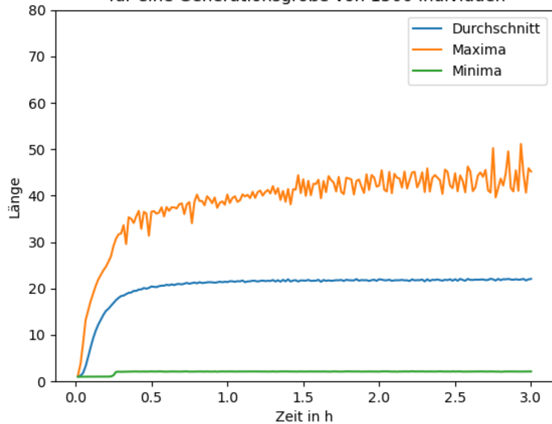


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

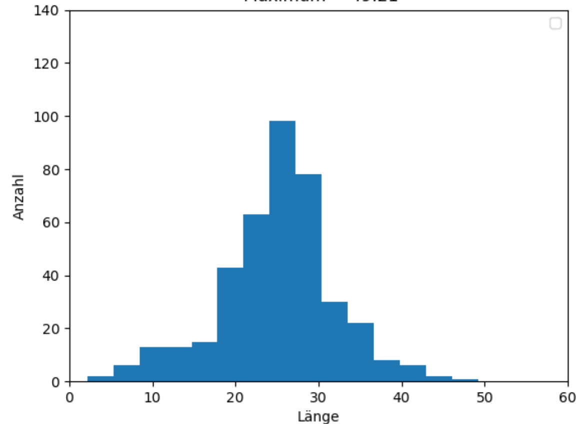


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.4: Simulation mit einer Generationsgröße von 1000 Individuen

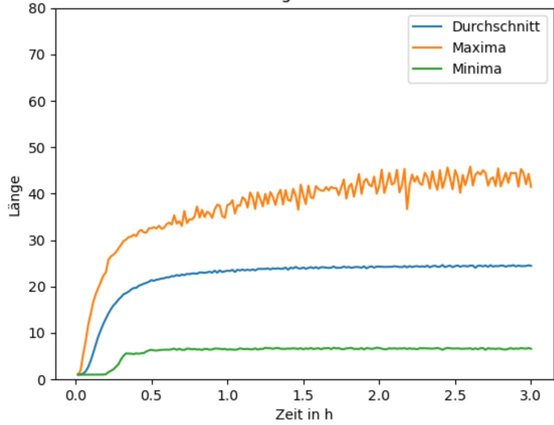


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

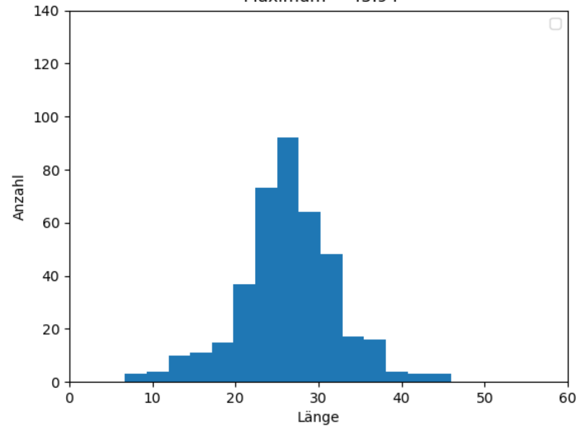


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.5: Simulation mit einer Generationsgröße von 1500 Individuen

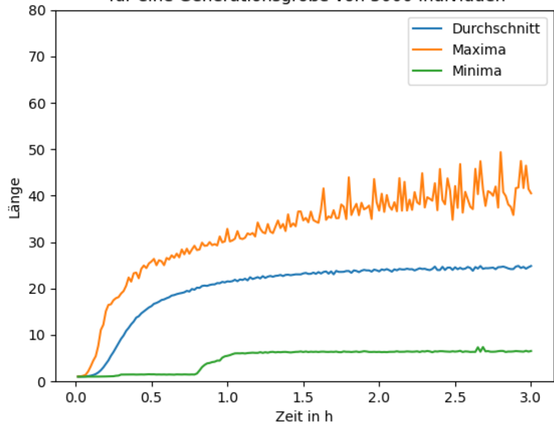


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

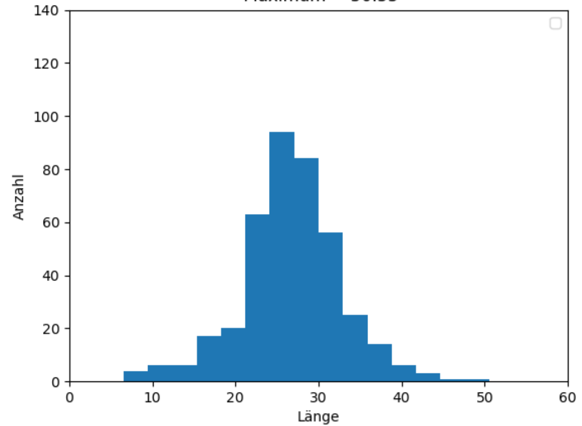


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.6: Simulation mit einer Generationsgröße von 2000 Individuen

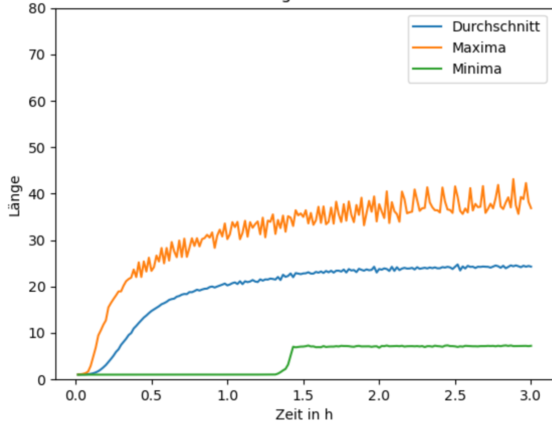


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

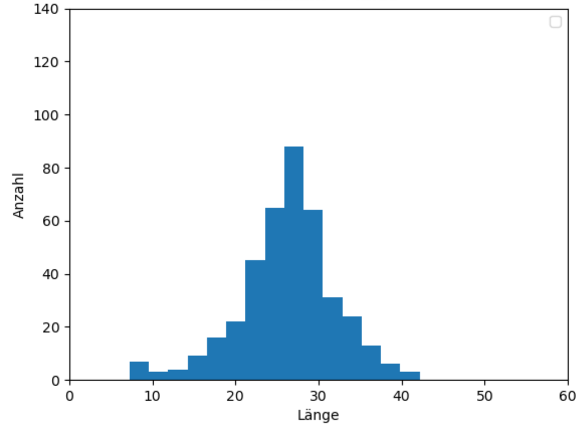


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.7: Simulation mit einer Generationsgröße von 3000 Individuen

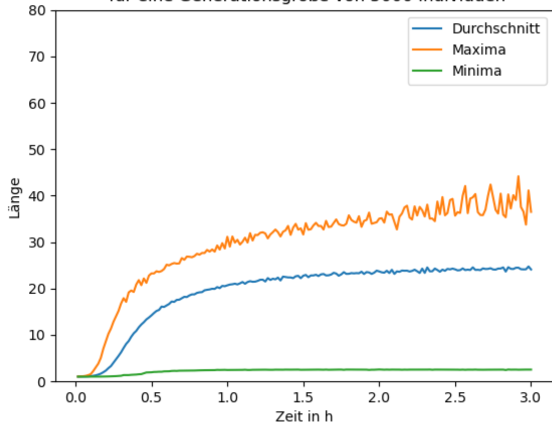


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

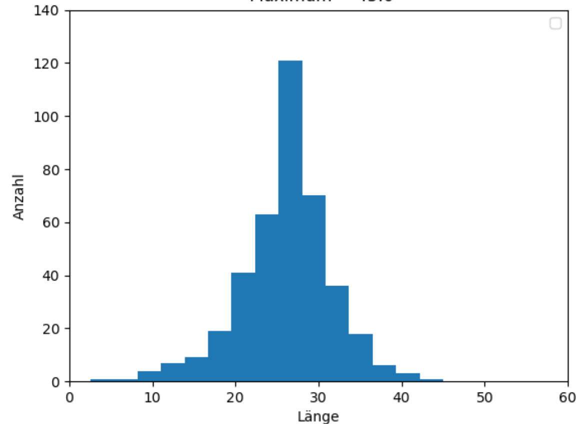


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.8: Simulation mit einer Generationsgröße von 4000 Individuen

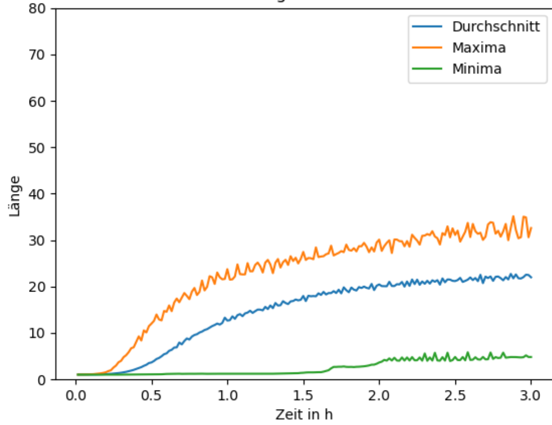


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

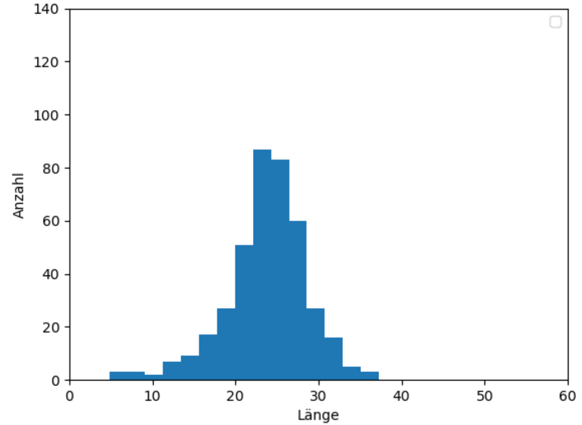


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.9: Simulation mit einer Generationsgröße von 5000 Individuen

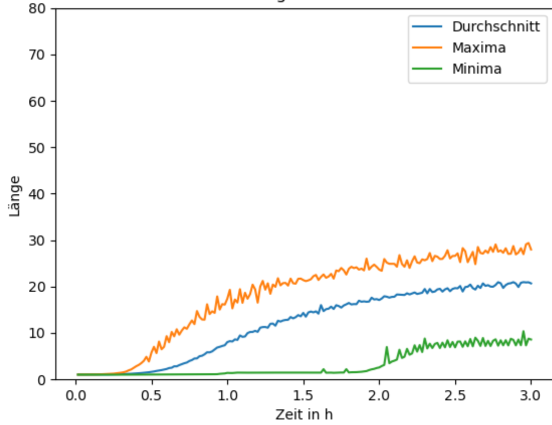


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

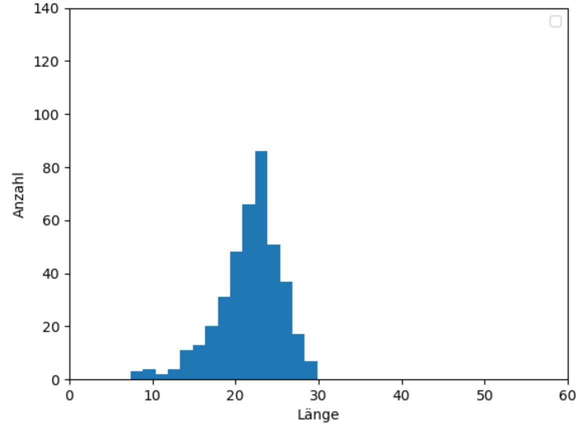


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.10: Simulation mit einer Generationsgröße von 10000 Individuen

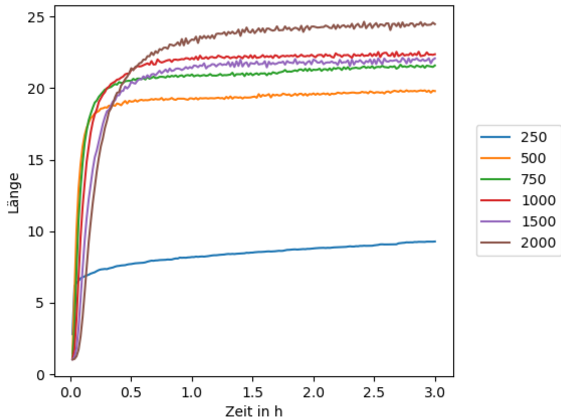


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

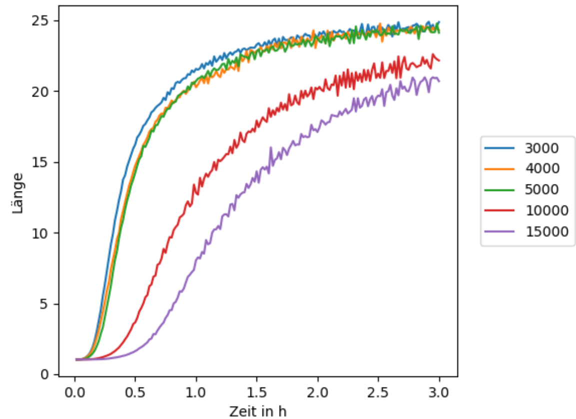


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.11: Simulation mit einer Generationsgröße von 15000 Individuen



(a) Verlauf der durchschnittliche Längen über die Zeit



(b) Verlauf der durchschnittliche Längen über die Zeit

Abbildung 3.12: Durchschnittliche Verläufe aller simulierten Generationsgrößen, hier aufgeteilt auf zwei Diagramme für eine bessere Übersichtlichkeit

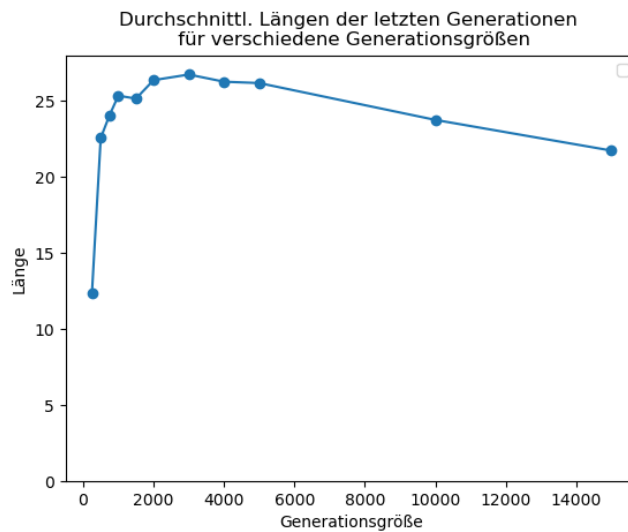


Abbildung 3.13: Mittelwerte aus den letzten Generationen aller Simulationsdurchläufe für alle simulierten Generationsgrößen

In Abbildung 3.13 wurden die durchschnittlichen Ergebnisse der letzten Generation aller simulierten GG eingezeichnet. Erkennbar ist, dass der Graph erst schnell ansteigt, dann bei einer GG von 3000 sein Maximum erreicht und dann wieder langsam, in einem linear anmutenden Verlauf, abfällt.

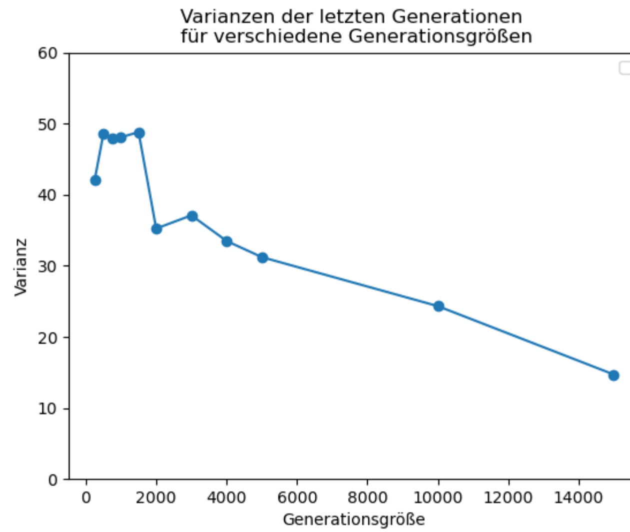
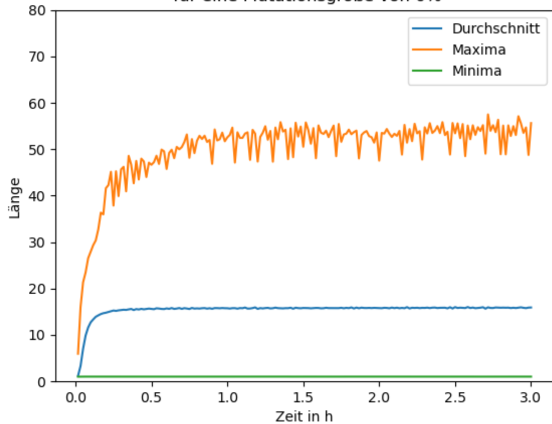


Abbildung 3.14: Varianzen der Mittelwerte aus den letzen Generationen für alle simulierten Generationsgrößen

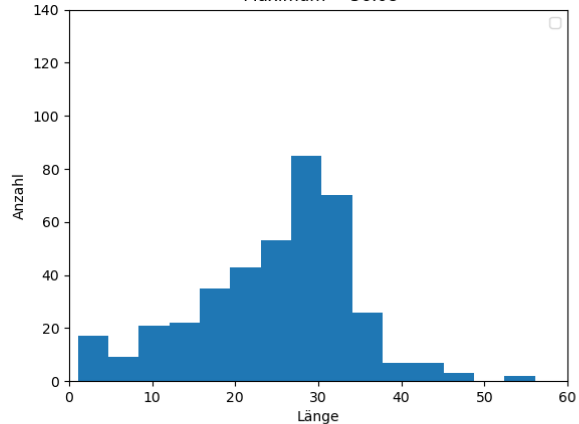
In der Abbildung 3.14 wurden alle Varianzen der obigen Histogramme nochmal vereint dargestellt. Ersichtlich ist, dass die Varianz mit zunehmender GG, im Mittel, kleiner wird.

3.2 Ergebnisse für verschiedene Mutationsgrößen

Die folgenden Diagramme zeigen die Ergebnisse für die Simulationen mit verschiedenen Mutationsgrößen. Die Diagramme wurden auf die gleiche Weise wie im vorherigen Abschnitt erstellt.

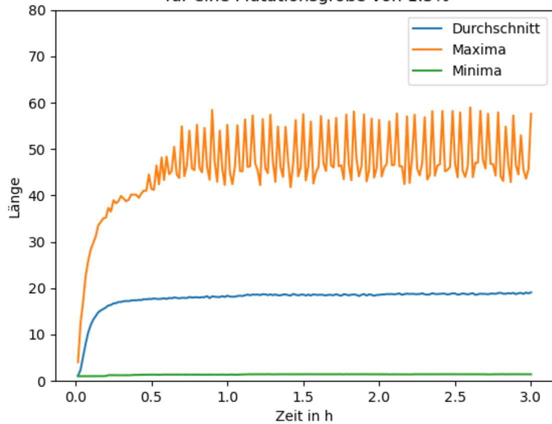


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

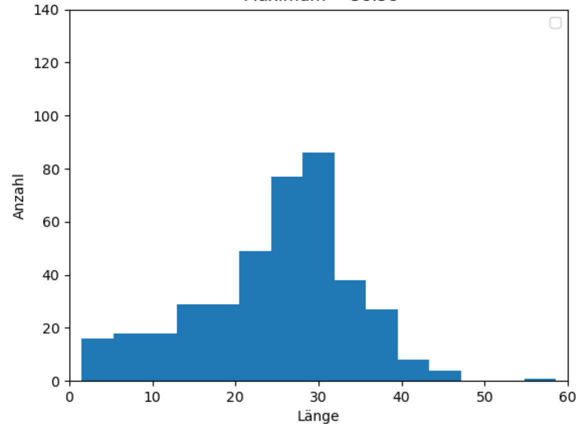


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.15: Simulation mit 0% Mutationsgröße

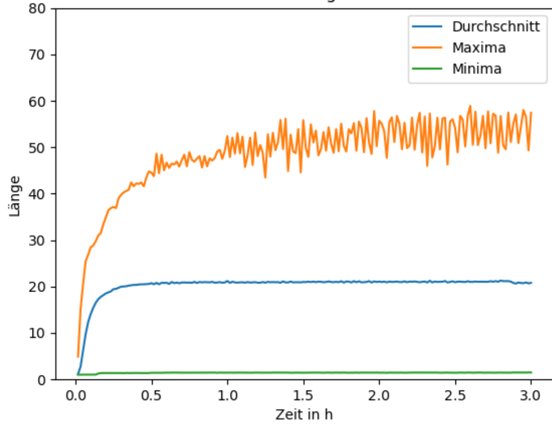


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

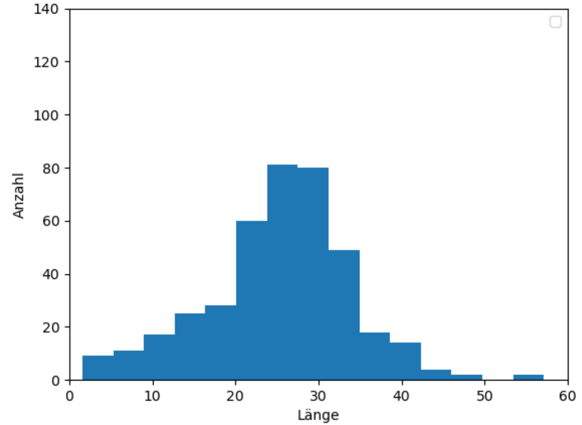


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.16: Simulation mit 1.5% Mutationsgröße

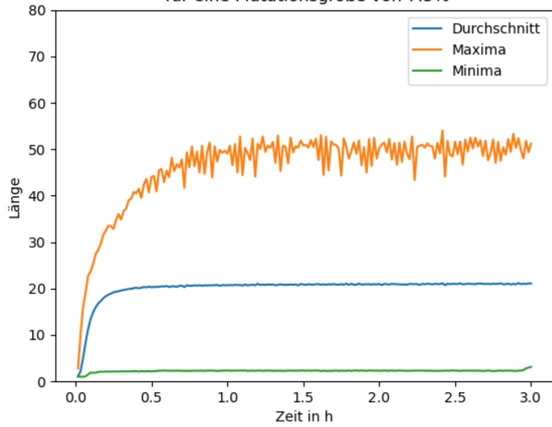


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

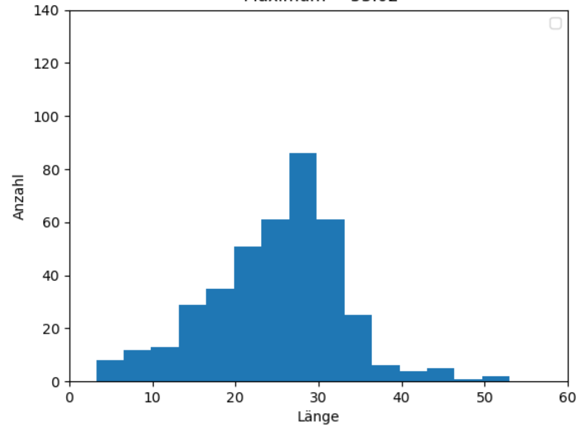


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.17: Simulation mit 5% Mutationsgröße

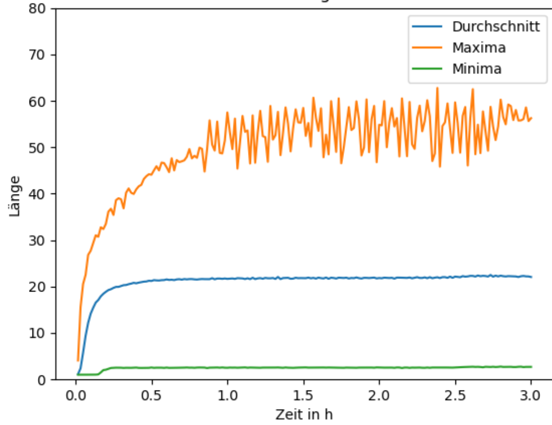


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

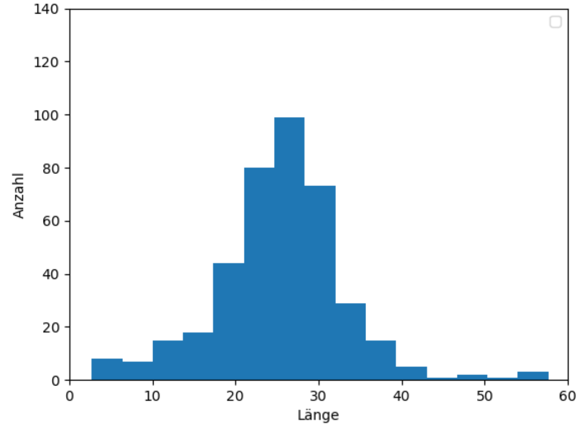


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.18: Simulation mit 7.5% Mutationsgröße

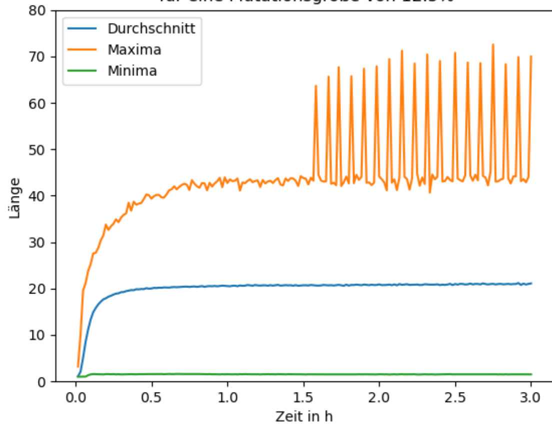


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

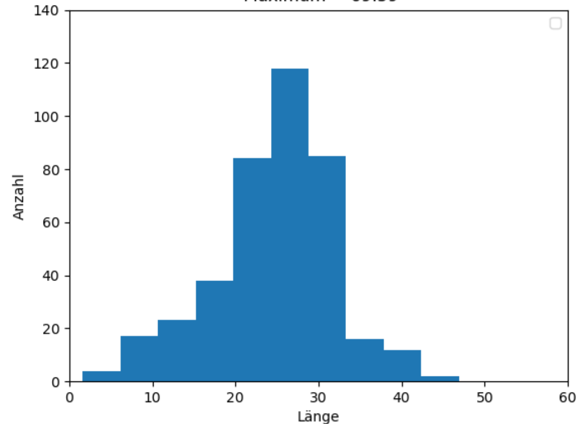


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.19: Simulation mit 10% Mutationsgröße

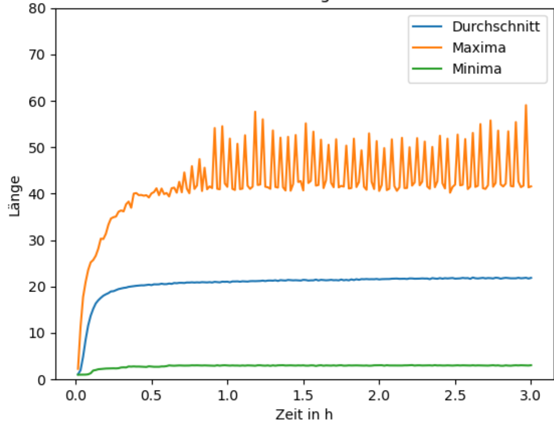


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

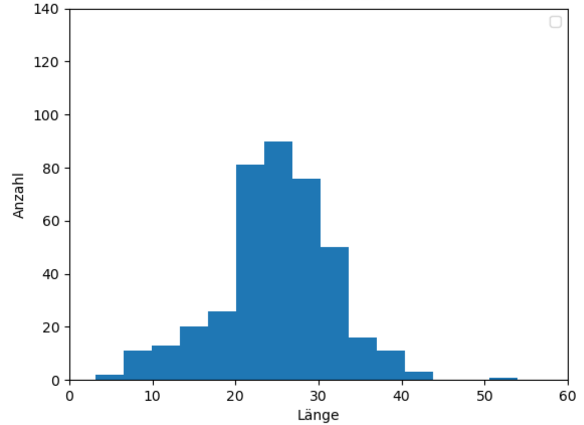


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.20: Simulation mit 12.5% Mutationsgröße

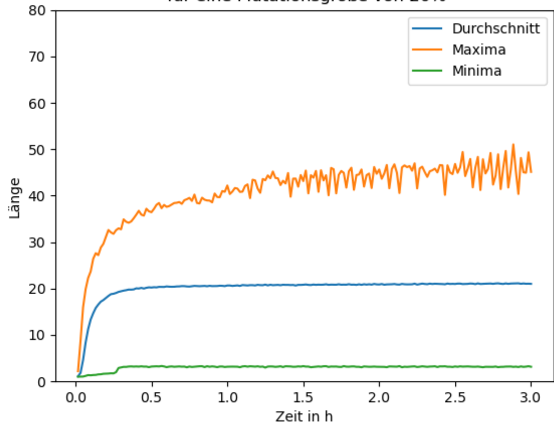


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

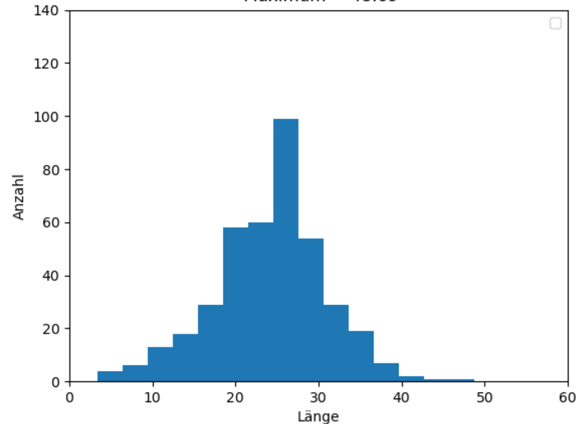


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.21: Simulation mit 15% Mutationsgröße

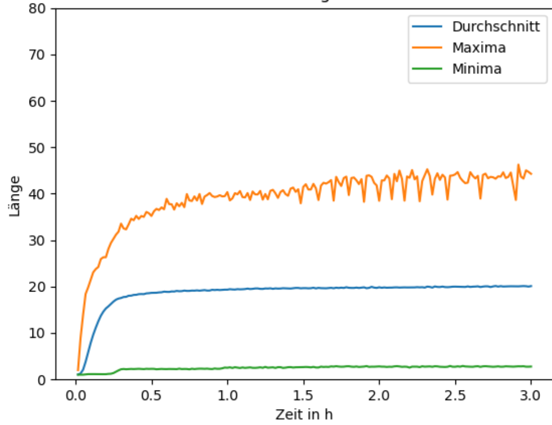


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

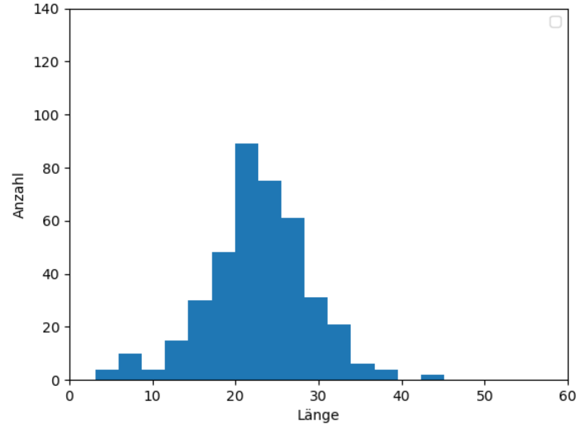


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.22: Simulation mit 20% Mutationsgröße

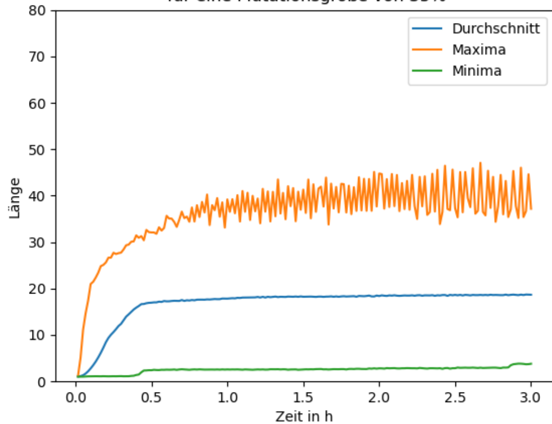


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

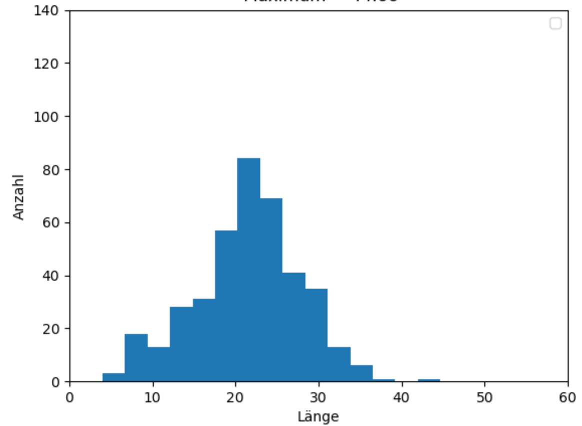


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.23: Simulation mit 30% Mutationsgröße

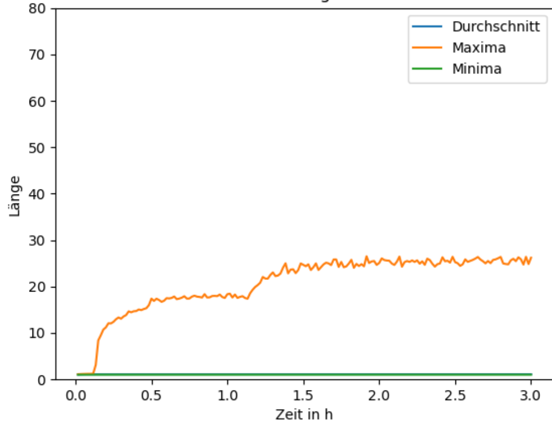


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

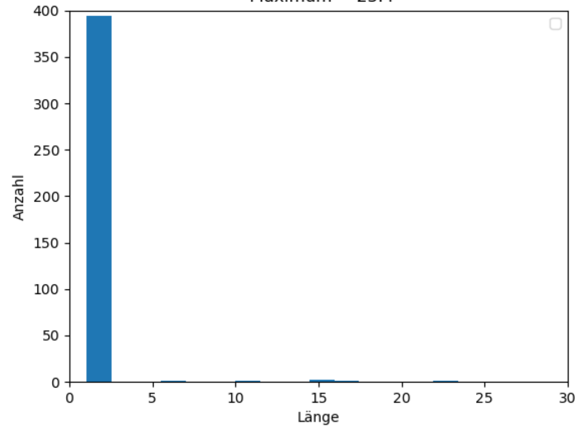


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.24: Simulation mit 35% Mutationsgröße

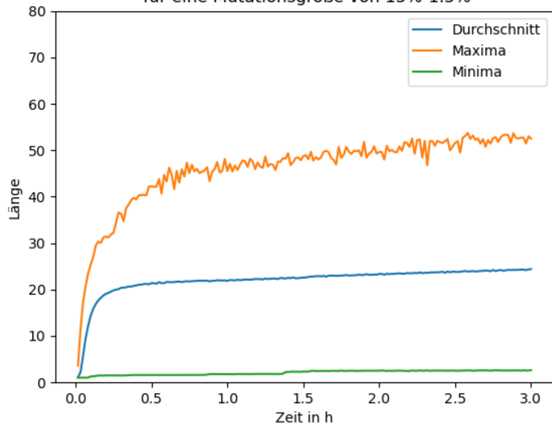


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

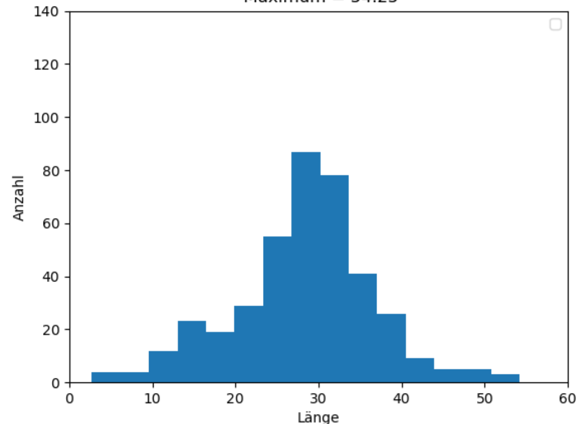


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.25: Simulation mit 50% Mutationsgröße, zu beachten ist, dass die y-Achse in diesem Histogramm bis zu einer Anzhal von 400 und die x-Achse bis zu einer Länge von 30 reicht

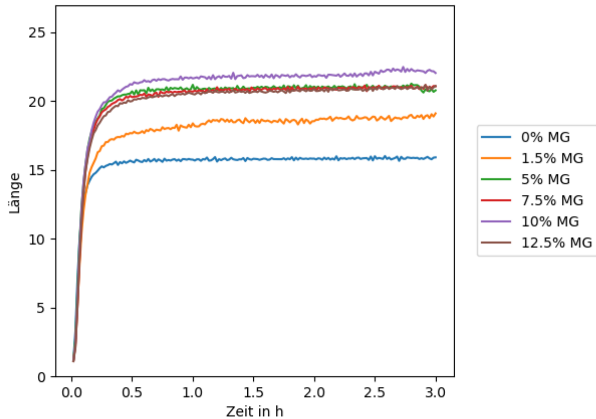


(a) Verlauf der durchschnittlichen, maximalen und minimalen Längen über die Zeit

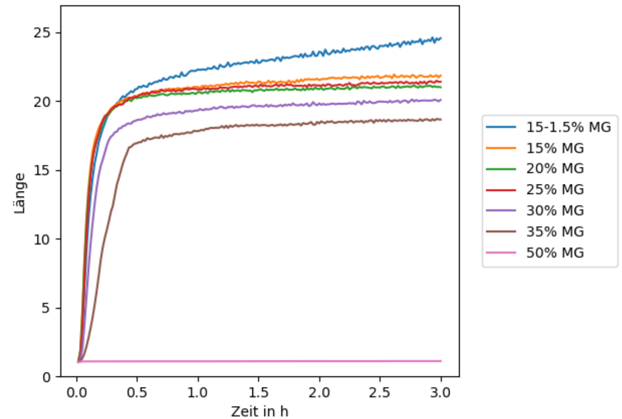


(b) Histogramm über die durchschnittlichen Längen

Abbildung 3.26: Simulation mit im Verlauf der Zeit kleiner werdende Mutationsgröße von 15%-1.5%



(a) Verlauf der durchschnittl. Längen über die Zeit



(b) Verlauf der durchschnittl. Längen über die Zeit

Abbildung 3.27: Durchschnittliche Verläufe aller simulierten Mutationsgrößen, hier aufgeteilt auf zwei Diagramme für eine bessere Übersichtlichkeit

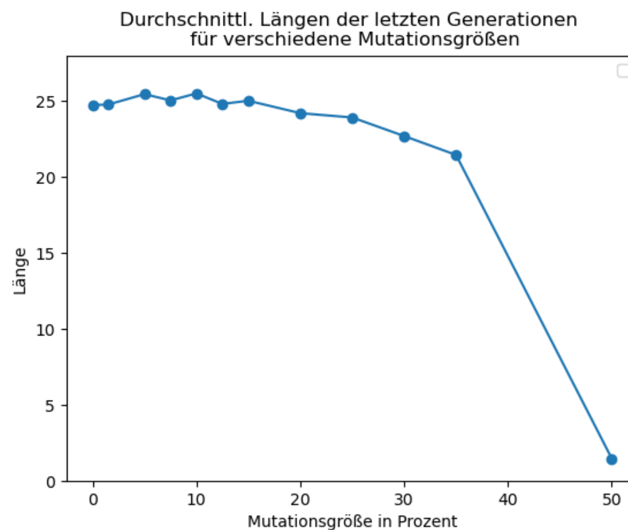


Abbildung 3.28: Mittelwerte aus den letzen Generationen aller Simulationsdurchläufe für alle simulierten Mutationsgrößen

In Abbildung 3.28 wurden die durchschnittlichen Ergebnisse der letzten Generation aller simulierten MG eingezeichnet. Es ist ersichtlich, dass für kleine MG der Graph um die Länge 25 stagniert und dann mit höheren MG immer schneller abfällt.

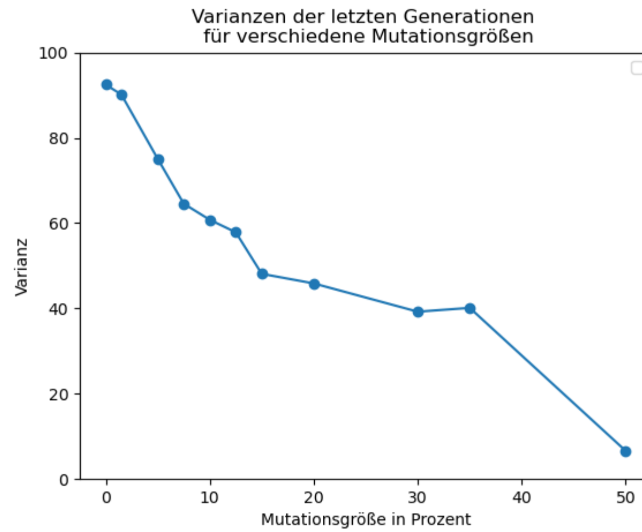


Abbildung 3.29: Varianzen der Mittelwerte aus den letzen Generationen für alle simulierten Mutationsgrößen

In der Abbildung 3.29 wurden alle Varianzen der obigen Histogramme nochmal vereint dargestellt. Ersichtlich ist, dass die Varianz mit zunehmender MG kleiner wird.

4 Diskussion

In diesem Teil werden die Ergebnisse diskutiert, dabei werden die Ergebnisse aus den Experimenten mit der GG und den Experimenten mit der MG getrennt betrachtet.

4.1 Diskussion Experimente mit verschiedenen Generationsgrößen

In dem Abschnitt „Ergebnisse für verschiedene Generationsgrößen“ hat sich gezeigt, dass die durchschnittlichen Längen der Snakes mit steigender Generationsgröße erst auch größer wird, jedoch ab einem bestimmten Punkt dann wieder kleiner (siehe Abbildung 3.13). Dies deckt sich mit der Aussage aus „A parameter-less genetic algorithm “, wonach Simulationen mit kleineren Generationsgrößen bei schlechteren Lösungen bleiben während größere Generationsgrößen zu viel Rechenkapazität verbrauchen und deswegen langsamer gute Lösungen finden. [10]
Die optimale GG für die betrachteten Fälle liegt bei 3000, dieser Wert lieferte die höchste durchschnittliche Länge.
Des weiteren hat sich gezeigt, dass umso kleiner die Generationsgrößen sind, der Verlauf der durchschnittlichen Länge über die Zeit schneller ein gewisses Plateau erreicht und dann stagniert. Während in den Diagrammen der größeren Generationsgrößen (siehe z.B. Abbildung 3.10a) durchschnittlichen Längen noch bis zum Ende hin größer werden. Wenn also nur begrenzte Rechenkapazitäten zur Verfügung stehen und eine schnelle aber weniger gute Lösung für ein Problem gesucht wird, empfiehlt es sich mit kleineren Generationsgröße zu arbeiten.

Wie zu erwarten wurde die Varianz mit steigender GG tendenziell immer kleiner. Wie in dem Paper „Genetic Algorithms, Noise, and the Sizing of Populations “ bereits herausgestellt wurde, sind kleine Generationsgrößen stärker vom Zufall abhängig ob sie eine guten Lösung für das vorliegende Problem finden, während große GG diesbezüglich zuverlässiger sind. Das führt vermutlich zu kleineren Varianzen bei höheren GG. Zwar wurde die durchschnittliche Leistung der Simulationen mit ansteigender GG langsam wieder schlechter, jedoch ist davon auszugehen, dass mit einer höheren Simulationszeit gleich gute oder bessere Ergebnisse hätten erzielt werden können als die Simulationen mit mittlerer GG. Dieser Zusammenhang wurde in „Genetic Algorithms,Noise, and the Sizing of Populations “ ebenfalls unterstrichen. [9]
Ob die größeren Generationsgrößen für den vorliegenden EA noch bessere Lösungen bei einer längeren Simulationszeit finden, könnte ein Thema für weitere Betrachtungen des Problems in der Zukunft sein. Um die Rechenzeit und den Ressourcenverbauch im Rahmen zu halten wurde aber hier nur eine Simulationszeit von 3h betrachtet.

4.2 Diskussion Experimente mit verschiedenen Mutationsgrößen

Interessanterweise hat sich gezeigt, dass mit größer werdender Mutationsgröße die Varianz der Mittelwerte der letzten Generationen abnimmt (siehe Abbildung 3.29). Dieser Zusammenhang ist wohl dadurch begründbar, dass bei höheren MG die durchschnittliche Leistung auch immer schlechter wird und somit die Spanne der erreichten Ergebnisse kleiner wird. Dieses Verhalten zeigt sich am stärksten in der Abbildung 3.25b. In diesem Histogramm sieht man, dass sich für eine MG von 50% die erreichten durchschnittlichen Längen fast alle zwischen 1 und 2 liegen. Somit ist die Varianz der Ergebnisse sehr klein.
Außerdem ist erkenntlich geworden, dass für die Simulationen mit kleinen Mutationsgrößen, bis einschließlich 0, die erreichten durchschnittlichen Längen relativ gleichbleibend auf einem Niveau stagnieren und erst bei großen Werten dann schnell schlechter werden. Ein Grund dafür, dass auch bei kleinen Mutationsgrößen gleich gute Ergebnisse erreicht werden könnte daran liegen, dass die verwendete Generationsgröße für diese Simulationen von 1000 Individuen groß genug ist um genug genetische Vielfalt zu bieten, um trotzdem eine gute Lösung zu finden. Wie in dem Paper „Optimal Population Size and the Genetic Algorithm “ betont wurde ist die Wahrscheinlichkeit bei höheren Generationsgrößen gute Lösungskanidaten zu finden höher (da offensichtlich mehr Kombinationen an Genen in der initialen Generation zur Verfügung stehen). [13] Wie sich der Verlauf dieses Diagramms bei anderen Generationsgrößen verhält könnte eine interessantes Thema für zukünftige Betrachtungen dieses Themas sein.
Überraschend war, wie stark die durchschnittlichen Ergebnisse bei einer MG von 50 % einbrachen. Das Histogramm in Abbildung 3.25b zeigt, dass sich fast alle der 400 Werte zwischen den Längen eins und zwei befinden. Dieses Ergebnis zeigt, dass bei einer zu starken Mutation, die Generationen im Laufe der Simulation fast nicht dazulernen können, da mit jeder neuen Generation wieder die Hälfte aller Gene zufällig ersetzt werden.

Wie in Abbildungen 3.27 und 3.26 ersichtlich ist, hat die Simulation mit einer über die Zeit kleiner werdende MG ein besseres durchschnittliches Ergebnis erreicht als alle untersuchten Simulationen mit konstanter MG. Diese Ergebnis bestätigt die Aussage aus „Adaptive mutation rate control schemes in genetic algorithms “ [15] von Dirk Thierens wo nach eine veränderliche MG einer Konstanten vorzuziehen ist.
Welchen Einfluss andere Methoden der Mutationsgrößenanpassung auf das Ergebnis haben könnte eine spannende Forschungsfrage für Betrachtungen des Themas in der Zukunft sein.

5 Fazit

Ziel dieser Arbeit war es einen Evolutionären Algorithmus zu kreieren der in der Lage ist Bots zum Spielen des Spiels Snake zu trainieren. Außerdem wurde der Einfluss der Veränderung der beiden Simulationsvariablen Generationsgröße und Mutationsgröße, auf die Endergebnisse der Simulationen, getestet. Vorhergehende Betrachtungen haben gezeigt, dass eine veränderliche MG einer Konstanten überlegen ist [15], diese Thematik sollte auch für den vorliegenden Fall getestet werden. Des weiteren ist bekannt, dass die Wahl der Generationsgröße ein wichtiger Faktor für das Finden einer guten Lösung darstellt [10]. Wo die optimale Generationsgröße für den betrachteten EA liegt, wurde in dieser Arbeit untersucht.

Es wurden folgende Ergebnisse gefunden:
1) Für die GG gibt es einen optimalen Punkt, sie darf weder zu groß noch zu klein sein. In dem betrachteten Simulationen brachte eine GG von 3000 das beste Ergebnis hervor.
2) Größere GG führen zu kleineren Varianzen der Mittelwerte der letzten Generationen.
3) Umso kleiner die GG ist umso schneller stagniert das durchschnittliche Ergebnis des EA.
4) Kleine MG haben die besten Ergebnisse geliefert, größere MG führten schnell zu immer schlechteren Ergebnissen.
5) Größere MG führen zu kleineren Varianzen der Mittelwerte der letzten Generationen.
6) Eine mit der Zeit kleiner werdende MG lieferte bessere Ergebnisse als alle Simulationen mit stagnierenden MG.

Es hat sich also gezeigt, dass bestehende Erkenntnisse über die Simulationsvariablen Generationsgröße und Mutationsgröße für den vorliegenden EA bestätigt werden konnten.
Zu erwähnen ist, dass die Beziehungen zwischen den unterschiedlichen Simulationsvariablen nicht geprüft wurden da für die verschiedenen Simulationen immer nur eine Variable verändert wurde. Um eine optimale Lösung für den gegebenen EA finden zu können müsste jedoch jede mögliche Kombination an Simulationsvariablen getestet werden. Diese Überprüfung ist an dieser Stelle aber aus Zeitgründen nicht möglich und könnte eine interessante Betrachtung für zukünftige Arbeiten zu diesem Thema sein.

6 Anhang

6.1 Git Repositories

- Das GitHub Repository mit dem Code für diese Arbeit befindet sich unter dem Link:
  https://github.com/tompruggmayer/EA-for-Snake.git
- Link zum hydra cluster:
  https://git.tu-berlin.de/ml-group/hydra/documentation

Literaturverzeichnis

[1] Aktivierungsfunktionen. https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron#Aktivierungsfunktionen. Accessed: 28.03.2024.

[2] Ein kleiner Überblick über neuronale netze. https://www.dkriesel.com/science/neural_networks. Accessed: 11.12.23.

[3] Fitness proportionate selection. https://en.wikipedia.org/wiki/Fitness_proportionate_selection#cite_ref-1. Accessed: 04.03.2024.

[4] Geogebra. https://www.geogebra.org/calculator. Accessed: 28.03.2024.

[5] Kn bild. https://commons.wikimedia.org/w/index.php?curid=224561. Accessed: 28.03.2024.

[6] Snake (video game genre). https://en.wikipedia.org/wiki/Snake_(video_game_genre)#:~:text=It%20originated%20in%20the%201976,eaten%E2%80%94often%20apples%20or%20eggs. Accessed: 01.03.2024.

[7] Tournament selection. https://en.wikipedia.org/wiki/Tournament_selection. Accessed: 04.03.24.

[8] Piotr Białas. Implementation of artificial intelligence in snake game using genetic algorithm and neural networks.

[9] David E Goldberg, Kalyanmoy Deb, and James H Clark. Genetic algorithms, noise, and the sizing of populations. *Complex systems*, 6:333–362, 1991.

[10] Georges R Harik, Fernando G Lobo, et al. A parameter-less genetic algorithm. In *GECCO*, volume 99, pages 258–267, 1999.

[11] SC Nayak, Bijan B Misra, and Himansu Sekhar Behera. Impact of data normalization on stock index forecasting. *International Journal of Computer Information Systems and Industrial Management Applications*, 6:13–13, 2014.

[12] Vassilios Petridis, Spyros Kazarlis, and Anastasios Bakirtzis. Varying fitness functions in genetic algorithm constrained optimization: the cutting stock and unit commitment problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 28(5):629–640, 1998.

[13] STANLEY GOTSHALL BART Rylander and Bart Gotshall. Optimal population size and the genetic algorithm. *Population*, 100(400):900, 2002.

[14] J. Sola and J. Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468, 1997.

[15] D. Thierens. Adaptive mutation rate control schemes in genetic algorithms. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, volume 1, pages 980–985 vol.1, 2002.

[16] Dirk Thierens and David Goldberg. Convergence models of genetic algorithm selection schemes. In *International conference on parallel problem solving from nature*, pages 119–129. Springer, 1994.

[17] Farah Ayiesya Zainuddin, Md Fahmi Abd Samad, and Durian Tunggal. A review of crossover methods and problem representation of genetic algorithm in recent engineering applications. *International Journal of Advanced Science and Technology*, 29(6s):759–769, 2020.