

Institut für Formale Methoden der Informatik  
Universität Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Diplomarbeit Nr. 2557

# Über das Wachstum von Boyer-Moore-Automaten

Ralf Hantschel

**Studiengang:** Softwaretechnik  
**Prüfer:** Prof. Dr. Volker Claus  
**Betreuer:** Dipl. -Inf. Sascha Riexinger

**begonnen am:** 16. November 2006

**beendet am:** 18. Mai 2007

**CR-Klassifikation:** F.1.1, F.1.2, G.1.6, I.2.8, I.5.4, K.5.2



In dieser Arbeit wird untersucht, in welchem Ausmaß Boyer-Moore-Automaten, abhängig vom zu suchenden Suchwort wachsen. Von manchen wird ein exponentielles Wachstum vermutet, andere behaupten dagegen, dass das Wachstum nur polynomiell ist. Es wird für einfache Fälle (kleines Alphabet, kurzes Suchwort) das Suchwort ermittelt, für welches die Anzahl der Zustände des zugehörigen Boyer-Moore-Automaten maximal wird. Bei einem zweielementigen Alphabet ist dies bis zur Suchwortlänge 30 möglich, bei einem dreielementigen Alphabet bis zur Suchwortlänge 19.

Für längere Suchworte wird ein Genetischer Algorithmus entwickelt, der versucht die Suchworte zu finden, die sehr große Boyer-Moore-Automaten erzeugen.



# Inhaltsverzeichnis

---

<b>1</b>	<b>Einleitung</b>	<b>5</b>
1.1	Motivation . . . . .	5
1.2	Problemstellung . . . . .	5
1.3	Überblick . . . . .	6
1.4	Aufbau des Dokuments . . . . .	7
<b>2</b>	<b>Boyer-Moore-Algorithmus und Boyer-Moore-Automat</b>	<b>9</b>
2.1	Der klassische Boyer-Moore-Algorithmus . . . . .	9
2.2	Der Boyer-Moore-Automat . . . . .	13
2.3	Der erweiterte Boyer-Moore-Algorithmus . . . . .	28
2.4	Gierige Algorithmen zur Berechnung „großer“ Suchwörter . . . . .	30
<b>3</b>	<b>Untersuchung kurzer Wörter</b>	<b>35</b>
3.1	Prinzipielles Vorgehen . . . . .	35
3.2	Maximale Suchwörter für kurze Wortlängen . . . . .	36
3.3	Interessante Wortarten . . . . .	37
3.4	Das Wachstum der durchschnittlichen BMA-Größe . . . . .	38
<b>4</b>	<b>Evolutionäre Algorithmen</b>	<b>51</b>
4.1	Vorbild aus der Natur . . . . .	51
4.2	Der Genetische Algorithmus . . . . .	51
4.3	Begriffsdefinitionen . . . . .	53
4.4	Der Begriff der Population . . . . .	57
4.5	Die Initialisierung des Evolutionären Algorithmus . . . . .	57
4.6	Der Evolutionszyklus eines Evolutionären Algorithmus . . . . .	58
4.7	Die Komponenten des Evolutionszyklus . . . . .	59
4.8	Funktion von Mutation und Rekombination . . . . .	74
<b>5</b>	<b>Der Genetische Algorithmus zur Wortsuche</b>	<b>75</b>
5.1	Unterschiede zum allgemeinen Genetischen Algorithmus . . . . .	75
5.2	Mutationen . . . . .	79
5.3	Rekombinationen und Crossoveroperatoren . . . . .	83
5.4	Selektionen . . . . .	90
5.5	Bewertungen . . . . .	93
<b>6</b>	<b>Auswertungen</b>	<b>95</b>
6.1	Beurteilung der Mutationen . . . . .	96
6.2	Beurteilung der Rekombinationen . . . . .	105

6.3	Beurteilung der Selektionen . . . . .	112
6.4	Parametereinstellungen . . . . .	117
6.5	Wachstumsverhalten der Suchwörter . . . . .	118
6.6	Fazit . . . . .	120
<b>7</b>	<b>Ausblick</b>	<b>123</b>
<b>A</b>	<b>Entwurf des Genetischen Algorithmus zur Wortsuche</b>	<b>125</b>
A.1	Die verschiedenen Komponenten des Genetischen Algorithmus in der Implementierung . . . . .	125
A.2	Die abstrakten Klassen . . . . .	126
<b>B</b>	<b>CD</b>	<b>137</b>
	<b>Literaturverzeichnis</b>	<b>139</b>
	<b>Tabellenverzeichnis</b>	<b>141</b>
	<b>Abbildungsverzeichnis</b>	<b>143</b>

# Einleitung

---

## 1.1 Motivation

Die Suche nach einem Wort in einem größeren Text ist ein sehr häufiges Problem in der Informatik. In jedem guten Textverarbeitungsprogramm kann nach einem Wort im Text gesucht werden. Alle Suchseiten im Internet (Google, Alta-Vista usw.) benutzen eine Wortsuche in den Texten der Internetseiten, um gewünschte Seiten zu finden. Auch bei der Untersuchung von Genom-Sequenzen wird ein Wort (eine Gen-Sequenz) innerhalb eines sehr langen Textes (dem DNA/RNA-String) gesucht. Algorithmen, die diese Suche durchführen besitzen für gewöhnlich zwei Phasen: Die Phase der Verarbeitung des Suchwortes und die eigentliche Suchphase.

Der mit Boyer-Moore-Automaten erweiterte Algorithmus von Boyer-Moore ist in der Suchphase sehr schnell. Er erreicht die optimale Laufzeit.

Sucht man in mehreren Texten das gleiche Suchwort, ist der Aufwand zum Aufbau des Automaten vernachlässigbar, da die Laufzeit der Suchphase dominiert und somit den Gesamtaufwand bestimmt. Für die Überwachung von Datenströmen nach festen Suchwörtern muss der Automat nur ein einziges Mal erzeugt werden und steht anschließend für mehrere Suchen mit optimaler Laufzeit zur Verfügung. Spezielle Hardware für das Suchwort ermöglicht die Überwachung von großen Datenströmen effizient durchzuführen.

Diese Hardware muss den erzeugten Boyer-Moore-Automaten speichern, daher ist die Größe der Boyer-Moore-Automaten auch für die Hardwareimplementierungen wichtig, obwohl die Laufzeit der Erstellung des Automaten nicht ins Gewicht fällt.

Wegen diesen Aspekten, und dem theoretischen Interesse an diesem Automatentyp lohnt die Betrachtung der Größe der Boyer-Moore-Automaten für ein Suchwort.

## 1.2 Problemstellung

Die in [KJP77] vorgeschlagene Erweiterung des Boyer-Moore-Algorithmus verwendet einen endlichen Automaten, den Boyer-Moore-Automaten, um sich das bisher eingelesene Wort zu 'merken'. Durch diesen Automaten wird niemals ein Zeichen des Suchtextes doppelt gelesen. Der Algorithmus mit dem Boyer-Moore-Automat ist während der Suche aufwandsoptimal. Es muss aber für jedes Suchwort ein eigener Boyer-Moore-Automat aufgebaut werden. Alle bekannten Algorithmen zur Erzeugung von Boyer-Moore-Automaten haben einen Aufwand, der abhängig von der Größe des entstehenden Automaten ist. Um die Laufzeit der Erzeugung des Boyer-Moore-Automaten für ein Wort der Länge  $n$  abzuschätzen, muss bekannt sein, wie viele Zustände ein

Automat maximal für Suchwörter der Länge  $n$  haben kann. Die maximale Größe des Automaten für eine gegebene Suchwortlänge ist ein bisher offenes Problem. Derzeit kann nur durch das vollständige Ausrechnen aller möglichen Suchwörter sicher der Maximalwert für kurze Suchwörter bestimmt werden.

Ziel dieser Arbeit ist es, durch vollständige Suche für möglichst viele  $n$  gesichert das größte Suchwort eines 2-, 3-, 4-elementigen Alphabets zu finden und deren Wachstumsverhalten in Abhängigkeit zur Suchwortlänge zu untersuchen. Der Raum aller Suchwörter für diese  $n$  soll untersucht werden. Anschließend sollen mit einem Genetischen Algorithmus für größere  $n$  ( $\geq 30$ ) die Suchwörter gefunden werden, die die größten (oder möglichst große) Boyer-Moore-Automaten erzeugen. Mit diesen gefundenen Werten soll das Wachstum der maximalen Boyer-Moore-Automaten mit wachsender Wortlänge untersucht werden. Ziel ist es obere- und untere Grenzen anzugeben, die das Wachstum der größten Boyer-Moore-Automaten für eine gegebene Wortlänge begrenzen.

### 1.3 Überblick

Das Problem der Wortsuche ist ein gut untersuchtes und häufig betrachtetes Problem. Hier soll ein Suchwort der Länge  $m$  in einem Text der Länge  $n$  gefunden werden. Diverse Algorithmen sind entwickelt worden, um dieses Problem zu lösen. Die bekanntesten Algorithmen sind der naive Vergleichsalgorithmus, mit einem durchschnittlichen Aufwand von  $O(n \cdot m)$ , den in der Praxis häufig eingesetzten Algorithmus von Knuth-Morris-Pratt (Aufwand  $O(n)$ ) und der Algorithmus von Boyer und Moore. Dieser Algorithmus hat einen durchschnittlichen Aufwand für die Suche bei großem Alphabet  $\Sigma$  von  $O(n/m)$ .

Alle Algorithmen versuchen das gesuchte Pattern  $w$  im Suchtext  $t$  zu finden, indem sie das Pattern (bildlich gesprochen) über den Text legen und für jedes Zeichen des Patterns testen, ob es gleich dem Text ist. Diesen Sachverhalt ist in Abbildung 1.1 zu sehen. Sie unterscheiden sich nur in der Art, in der das Pattern über den Text geschoben wird.

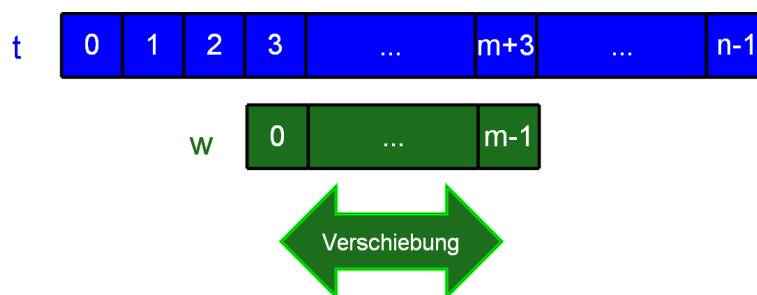


Abbildung 1.1: Prinzipieller Ablauf der Suche eines Wortes  $w$  im Text  $t$ .

In dieser Arbeit soll im Speziellen eine Erweiterung des klassischen Boyer-Moore-Algorithmus betrachtet werden, die von Knuth, Morris und Pratt vorgeschlagen wurde. Das Wachstum der Boyer-Moore-Automaten im Verhältnis zur Suchwortlänge ist hierbei von besonderem Interesse. Dies wurde schon für kurze Suchwörter 2-elementiger Alphabete durchgeführt (bis zur Länge 26). In dieser Arbeit werden diese Messungen nachvollzogen und zusätzliche Messungen für Suchwörter aus einem 3- und 4-elementigen Alphabet durchgeführt. Der Zusammenhang zwischen dem Wachstum der Automaten und der Größe des Alphabets ist Gegenstand der Untersuchungen.

Als Lösungsstrategie wird ein Genetischer Algorithmus verwendet, um für längere Suchwörter das Suchwort mit dem größten Automaten zu finden.

## 1.4 Aufbau des Dokuments

Das Dokument gliedert sich in 7 Kapitel und den Anhang.

**Das erste Kapitel** soll ein Überblick über das untersuchte Problem vermitteln.

**Das zweite Kapitel** beschreibt den klassische Boyer-Moore-Algorithmus zunächst informell und definiert ihn anschließend formal. Anschließend erfolgt eine Einführung in den Boyer-Moore-Automaten. Schließlich wird der erweiterte Boyer-Moore-Algorithmus, ein Boyer-Moore-Algorithmus mit einem Boyer-Moore-Automat, behandelt.

**Das dritte Kapitel** beginnt mit der Untersuchung von kurzen Suchwörtern. Für die Berechnung der Größe des Boyer-Moore-Automaten wird ein Algorithmus verwendet, der alle möglichen Suchwörter einer bestimmten Länge erzeugt und berechnet. Dieser Algorithmus wird beschrieben und die Ergebnisse der Messungen werden aufgeführt und analysiert.

**Das vierte Kapitel** beschäftigt sich mit dem prinzipiellen Aufbau eines Genetischen Algorithmus. In diesem Kapitel wird zunächst der Genetische Algorithmus gegenüber den anderen Evolutionären Algorithmen, wie z. B. der genetischen Programmierung, der evolutionären Programmierung und den Evolutionsstrategien abgegrenzt. Anschließend werden der Evolutionszyklus und die verschiedenen Bausteine des Genetischen Algorithmus beschrieben.

**Das fünfte Kapitel** behandelt den Genetischen Algorithmus zur Berechnung längerer Suchwörter. Das Augenmerk liegt nicht auf den Implementierungsaspekten des Algorithmus, sondern auf den prinzipiellen Varianten des allgemeinen Genetischen Algorithmus, die in den späteren Messungen verwendet werden.

**Das sechste Kapitel** stellt die Auswertungen der verschiedenen Testläufe des Genetischen Algorithmus vor und analysiert sie. Zunächst werden die Testläufe besprochen, die zur Einstellung der Parameter des Genetischen Algorithmus gemacht wurden. Anschließend werden die Ergebnisse für lange Suchwörter beschrieben, die mit diesen Einstellungen des Genetischen Algorithmus zur Wortsuche gewonnen wurden. In diesem Kapitel wird, auf der Grundlage der Messergebnisse für kurze Suchwörter, eine Aussage über das Wachstum der Automaten bei zunehmender Suchwortlänge gemacht.

**Das siebte Kapitel** zeigt mögliche weitere Arbeiten zu diesem Thema auf.

**Im Anhang** wird der Entwurf des Genetischen Algorithmus zur Wortsuche behandelt. Es werden die verschiedenen Klassen, die den Genetischen Algorithmus zur Wortsuche implementieren und deren Aufgabe sowie die Vererbungshierarchie behandelt.

Außerdem befindet sich hier die CD, auf der alle benötigten Daten für das Weiterführen der Software, sowie die Messergebnisse des Genetischen Algorithmus zur Wortsuche abgespeichert sind.



# Boyer-Moore-Algorithmus und Boyer-Moore-Automat

---

Der klassische Boyer-Moore-Algorithmus beinhaltet keinen Boyer-Moore-Automaten. Dieser wurde von Knuth in [KJP77] vorgeschlagen. Die Variante des Boyer-Moore-Algorithmus mit Boyer-Moore-Automat wird von nun an als erweiterter Boyer-Moore Algorithmus bezeichnet. In diesem Abschnitt wird zuerst der klassische Boyer-Moore-Algorithmus beschrieben, anschließend wird der Boyer-Moore Automat erklärt und der erweiterte Boyer-Moore-Algorithmus behandelt, der diesen Boyer-Moore-Automaten benutzt.

## 2.1 Der klassische Boyer-Moore-Algorithmus

### 2.1.1 Beschreibung des Algorithmus

Der Boyer-Moore-Algorithmus versucht ein Vorkommen des Wortes  $w$  in einem Text  $t$  zu finden. Dabei ist  $t$  im Allgemeinen viel länger als  $w$ . Der gesamte Ablauf des Algorithmus kann in zwei Phasen aufgeteilt werden. Die Suchphase und die Vorberechnungsphase. Zuerst werden in der Vorberechnungsphase die benötigten Datenstrukturen für das spezielle Suchwort  $w$  für die Suchphase erzeugt. Während der Vorberechnungsphase wird nur  $w$  verwendet, der Text in dem gesucht werden soll, wird nur bei der Suchphase verwendet. Wenn dasselbe Wort in mehreren Texten gesucht werden soll, muss die Vorberechnungsphase nur einmal durchlaufen werden. Anschließend kann die Suchphase mit diesen Datenstrukturen für alle Texte  $t$  verwendet werden.

### 2.1.2 Ablauf der Suchphase

Der Boyer-Moore-Algorithmus „legt“  $w$  am Ende von  $t$  an und beginnt von rechts nach links die Zeichen von  $w$  mit dem entsprechenden Zeichen von  $t$  zu vergleichen. Wenn alle Zeichen von  $w$  mit denen in  $t$  übereinstimmen, ist ein Vorkommen gefunden worden und der Algorithmus kann terminieren. Ansonsten findet der Algorithmus ein Zeichen, bei dem der Vergleich zwischen  $w$  und  $t$  einen Unterschied ergibt.

In diesem Fall kann  $w$  in  $t$  nicht mehr an der angelegten Stelle vorkommen und  $w$  muss nach links verschoben werden. Wie weit  $w$  verschoben werden kann, entscheidet der Algorithmus anhand von zwei Tabellen  $shift_1$  und  $shift_2$ . In diesen Tabellen sind die maximal möglichen Verschiebungen für  $w$  abgespeichert. Dabei betrachtet jede der Tabellen ein anderes Kriterium und gibt an, wie groß für dieses Kriterium die maximale Verschiebung für  $w$  sein darf:

**shift<sub>1</sub>** Hier steht für jedes Zeichen des Alphabets  $\Sigma$  an welcher Stelle das Zeichen in  $w$ , von rechts aus gezählt, zum ersten Mal vorkommt. Wenn in  $w$  ein Zeichen nicht vorkommt, so kann über das komplette  $w$  verschoben werden, so dass das gelesene Zeichen nun links neben  $w$  steht. Die Berechnung dieser Tabelle geschieht in der Vorbereitungsphase.

**shift<sub>2</sub>** In dieser Tabelle steht für jede Position in  $w$ , wie weit  $w$  verschoben werden kann, wenn an dieser Position ein Zeichen in  $t$  gelesen wird, dass nicht gleich dem Zeichen in  $w$  ist. Die Berechnung dieser Tabelle geschieht in der Vorbereitungsphase.

Nun werden die Werte aus den beiden Tabellen betrachtet, und der größere Wert für die Verschiebung angewendet. Nach der Verschiebung „vergisst“ der Algorithmus allerdings alle Zeichen, die er bisher eingelesen hat und beginnt das letzte Zeichen vom verschobenen  $w$  mit dem entsprechenden Zeichen von  $t$  zu vergleichen.

### 2.1.3 Ablauf der Vorberechnungsphase

In der Vorbereitungsphase werden die beiden Tabellen  $shift_1$  und  $shift_2$  berechnet. Diese beiden Tabellen müssen für jedes  $w$  neu berechnet werden, können aber, wenn sie für ein  $w$  berechnet wurden, für beliebige Texte  $t$  benutzt werden. Bevor die Berechnung der beiden Tabellen besprochen wird, nun zunächst einige Definitionen.

### 2.1.4 Formale Definitionen

Sei  $\Sigma$  das Alphabet und  $w, t \in \Sigma^*$  ( $w = w_0w_1\dots w_{m-1}, t = t_0t_1\dots t_{n-1}$ ). Die Vorkommen des Wortes  $w$  sollen im Text  $t$  gesucht werden. Dabei gilt, dass das Wort  $w$  im Text  $t$  ab der Stelle  $i$  vorkommt *gdw*

$$\forall k \in \{0..m-1\} : w_k = t_{i+k}$$

Der Boyer-Moore-Algorithmus  $BM$  soll nun ein Vorkommen von  $w$  finden. Zu diesem Zweck erhält  $BM$  als Eingaben  $w$  und  $t$ , sowie  $\Sigma$ .  $BM$  liefert als Ausgabe einen Wahrheitswert  $l$ , für den gilt:

$$l = BM(w, t, \Sigma) = true \Leftrightarrow \exists i \in \{0..m-1\} : w \text{ kommt in } t \text{ ab Stelle } i \text{ vor.}$$

Nun werden die beiden shift-Tabellen definiert.

#### shift<sub>1</sub>

Die Tabelle  $shift_1$  (siehe Abbildung 2.1) ist eine Abbildung

$$shift_1 : \Sigma \rightarrow \mathbb{N}_0^+$$

Sie bildet jedes Zeichen des Alphabets des Textes  $t$ , in dem  $w$  gesucht werden soll auf eine ganze positive Zahl  $\leq m$  oder null ab.

$$shift_1(a) = \min \{s | s = m \vee (0 \leq s < m \wedge w_{m-s} = a)\}$$

Wenn das Zeichen  $a$  in  $w$  vorkommt, wird in dieser Liste der Abstand von rechts bis zum ersten Vorkommen des Zeichens  $a$  in  $w$  eingetragen. Wenn das Zeichen  $a$  in  $w$  nicht vorkommt, so wird in die Liste  $m$  eingetragen.  $w$  kann somit komplett um seine eigene Länge verschoben werden.

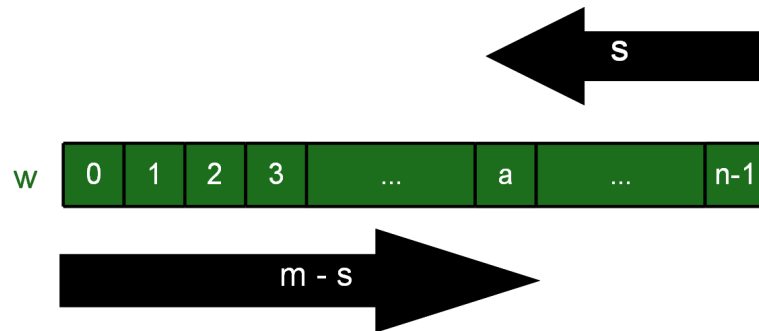


Abbildung 2.1: Die Tabelle  $shift_1$  speichert  $s$ , den Abstand des Zeichens  $a$  vom rechten Rand.

### shift<sub>2</sub>

Auch die Tabelle  $shift_2$  (siehe Abbildung 2.2) ist eine Abbildung

$$shift_2 : \{0, \dots, m - 1\} \rightarrow \mathbb{N}_0^+$$

Sie bildet jede Stelle des Suchwortes  $w$  auf eine ganze positive Zahl ab. Diese Zahl ist die minimale Verschiebung von  $w$  für die gilt, dass alle bisher aufgedeckten Zeichen (alle Zeichen rechts des aktuell gelesenen Zeichens in  $w$ ) auch an dieser Verschiebung in  $w$  vorkommen:

$$shift_2(j) = \min \{s + m - j \mid s \leq 1 \wedge ((s \leq 1 \vee w_{i-s} = w_i) \forall j < i \leq m)\}$$

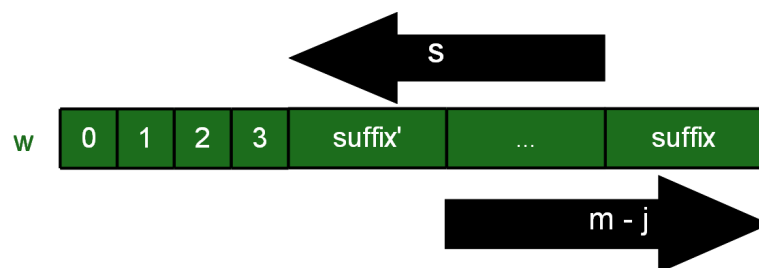


Abbildung 2.2: Die Tabelle  $shift_2$  speichert  $s + m - j$  für jede Stelle in  $w$ .

### Die Konfigurationen

Eine Konfiguration des Algorithmus ist ein 4-Tupel  $(i, pos, w, t)$

$$i \in \{0..n - 1\}$$

$$pos \in \{0..m - 1\}$$

$i$  ist die Stelle an der der Algorithmus in  $t$  prüft.  $pos$  beschreibt die Position des Zeichens  $w_{pos}$  in  $w$ , das als nächstes mit dem Zeichen  $t_i$  verglichen wird.

Diesen Vorgang des Vergleichens und Abfragens des Zeichens von  $t$  wird 'aufdecken' genannt. Ein Ablauf des Algorithmus ist eine Folge von Konfigurationen.

Die Startkonfiguration  $k_0$  ist  $(m - 1, m - 1, w, t)$ .

Hat der Algorithmus ein Vorkommen von  $w$  in  $t$  gefunden, liefert er genau dann **true** zurück, wenn er eine Konfiguration  $(i, -1, w, t)$ ,  $i \in \{0..n - 1\}$  erreicht hat.

Eine Konfiguration  $k_e$  ist erreichbar *gdw* es einen Ablauf  $k_0 k_1 .. k_e$  gibt und

$$\forall j \in \{1..e\} \exists a \in \Sigma : k_j = \delta(k_{j-1}, a).$$

Wir definieren nun die Funktion  $\delta$  der möglichen Konfigurationsübergänge im Algorithmus. Für zwei Konfigurationen  $k_u = (i, pos, w, t)$  und  $k_v = (i', pos', w, t)$  und dem Zeichen  $a \in \Sigma \wedge a = t_i$  gilt  $k_u = \delta(k_v, t_i)$  *gdw*  $1 \vee 2$ :

1. Es gilt  $a = w_{pos} \wedge i' = i - 1 \wedge pos' = pos - 1$ . Es wird also im nächsten Schritt das Zeichen links vom aktuellen Zeichen verglichen. Wir nennen einen solchen Übergang einen **Treffer**.
2. Es gilt  $a \neq w_{pos} \wedge i' = i + \max(shift_1(a), shift_2(pos)) \wedge pos' = m - 1$ . Hier wird also  $w$  um die in den shift-Tabellen angegebene Anzahl von Zeichen verschoben. Deshalb nennen wir einen solchen Übergang eine **Verschiebung**.

Da beide Tabellen jeweils eine notwendige Bedingung für eine gültige Verschiebung beschreiben, wird der größere der beiden Werte für die Verschiebung angewendet. Insbesondere kann es bei der in  $shift_1$  definierten Verschiebung geschehen, dass die Verschiebung geringer ist, als der Abstand des verglichenen Zeichens vom rechten Rand des Suchwortes. Dadurch könnten Linksverschiebungen entstehen, die nicht erlaubt sind. Dies wird durch  $\max(shift_1(a), shift_2(pos))$  gelöst, da die Verschiebung in  $shift_2(pos)$  immer mindestens der Abstand zum rechten Rand des Suchwortes + 1 ist.

### 2.1.5 Beispiel

Im folgenden Beispiel soll in einem Text das Suchwort  $w = abba$  gefunden werden. Das Alphabet  $\Sigma = \{a, b\}$  und die beiden Abbildungen  $shift_1$  und  $shift_2$  sind folgendermaßen definiert:

$$shift_1(x) = \begin{cases} 1 & x = a \\ 2 & x = b \end{cases}$$

Da  $a$  an erster Stelle von rechts aus gesehen in  $w$  vorkommt und  $b$  an zweiter Stelle.

$$shift_2(j) = \begin{cases} 6 & j = 0 \\ 5 & j = 1 \\ 4 & j = 2 \\ 1 & j = 3 \end{cases}$$

Hier muss das bisher eingelesene Teilstück des Suchwortes  $w$  soweit verschoben werden, bis wieder alle Zeichen an einer Stelle im verschobenen Suchwort  $w'$  stehen, an denen sie vorkommen dürfen. Auf diese Verschiebung muss noch die Verschiebung innerhalb des Suchwortes aufaddiert werden, um wieder das letzte Zeichen des Suchwortes mit dem Text zu vergleichen (an dieser Stelle „vergisst“ der klassische Boyer-Moore-Algorithmus die Zeichen, die er bisher verglichen hatte). Z. B. für  $j = 0$  muss das Suchwort um drei Zeichen verschoben werden, damit das erste eingelesene  $a$  an die Position  $w'_0$  geschoben wird, an der ein  $a$  erlaubt ist. Auf diese Verschiebung wird die Verschiebung um drei aufaddiert, um wieder das letzte Zeichen des verschobenen Suchwortes  $w'$  mit dem Text vergleichen zu können. Die resultierende Verschiebung im Text  $t$  ist also sechs. Auf die gleiche Weise können die anderen Verschiebungen von  $shift_2$  berechnet werden.

Nachdem die beiden Abbildungen gegeben sind, wird die Suche auf dem Suchtext

$$t = \text{abbabaabbaba}$$

durchgeführt.

Zur Übersicht werden das Suchwort  $w$  und der Text  $t$  in Tabelle 2.1 nochmal tabellarisch angegeben:

	0	1	2	3	4	5	6	7	8	9	10	11	12
$w$ :	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>									
$t$ :	<i>a</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>

Tabelle 2.1: Das Suchwort  $w$  und der Text  $t$  des Beispiels.

Die Anfangskonfiguration für die Suche ist also:  $k_0 = (0, 3, \text{abba}, \text{abbabaabbaba})$ .

Wir geben nun die Folge der Konfigurationsübergänge an:

$$\begin{aligned}
 & (3, 3, \text{abba}, \text{abb**ba**aabbaba}) \xrightarrow{\max\{\text{shift}_1(a), \text{shift}_2(3)\}=1} (4, 3, \text{abba}, \text{abbabaabbaba}) \\
 & \quad (4, 3, \text{abba}, \text{abbabaabbaba}) \xrightarrow{w_3=t_4} (3, 2, \text{ab**ba**, abbabaabbaba}) \\
 & \quad (3, 2, \text{ab**ba**, abbabaabbaba}) \xrightarrow{w_2=t_3} (2, 1, \text{a**bb**a, abbabaabbaba}) \\
 & \quad (2, 1, \text{a**bb**a, abbabaabbaba}) \xrightarrow{w_1=t_2} (1, 0, \text{a**bb**a, abbabaabbaba}) \\
 & (1, 0, \text{a**bb**a, abbabaabbaba}) \xrightarrow{\max\{\text{shift}_1(a), \text{shift}_2(0)\}=6} (7, 3, \text{ab**ba**, abbabaabbaba}) \\
 & \quad (7, 3, \text{ab**ba**, abbabaabbaba}) \xrightarrow{w_3=t_7} (6, 2, \text{ab**ba**, abbabaabbaba}) \\
 & (6, 2, \text{ab**ba**, abbabaabbaba}) \xrightarrow{\max\{\text{shift}_1(a), \text{shift}_2(2)\}=4} (10, 3, \text{ab**ba**, abbabaabbaba}) \\
 & \quad (10, 3, \text{ab**ba**, abbabaabbaba}) \xrightarrow{w_3=t_{10}} (9, 2, \text{ab**ba**, abbabaabbaba}) \\
 & \quad (9, 2, \text{ab**ba**, abbabaabbaba}) \xrightarrow{w_2=t_9} (8, 1, \text{a**bb**a, abbabaabbaba}) \\
 & \quad (8, 1, \text{a**bb**a, abbabaabbaba}) \xrightarrow{w_1=t_8} (7, 0, \text{a**bb**a, abbabaabbaba}) \\
 & \quad (7, 0, \text{a**bb**a, abbabaabbaba}) \xrightarrow{w_0=t_7} (6, -1, \text{abba, abbabaabbaba})
 \end{aligned}$$

Für die letzte Konfiguration ist  $i = -1$ . Der Algorithmus gibt *true* zurück und zeigt an, dass er ein Vorkommen von  $w$  in  $t$  gefunden hat.

## 2.2 Der Boyer-Moore-Automat

Mit dem Boyer-Moore-Automat (BMA) kann man das 'Gedächtnis' des Algorithmus erweitern. Es werden jetzt bei jeder Verschiebung alle bisher gelesenen Zeichen beachtet, während beim klassischen Boyer-Moore-Algorithmus nur die Zeichen beachtet werden, die seit der letzten Verschiebung verglichen wurden. Ein Boyer-Moore-Automat wird also immer für ein spezielles Suchwort  $w$  konstruiert und ist auch nur für dieses gültig.

### 2.2.1 Definitionen

**Definition 1** Ein Boyer-Moore-Automat ist ein endlicher Automat mit Ausgabe, also ein 7-Tupel  $BMA = (Q, \Sigma, \Gamma, \delta, \omega, q_0, F)$ , wobei gilt:

$Q$  Ist die Menge der Zustände. Für jeden Zustand  $q \in Q$  gilt  $|q| = |w|$  und  $q = \{0, 1\}^*$ .

Eine 1 bedeutet, dass dieses Zeichen in  $w$  aufgedeckt wurde und 0, dass es noch nicht aufgedeckt wurde.

$\Sigma$  Dies ist das Eingabealphabet.

$\Gamma$  Dies ist das Ausgabealphabet  $\{0, 1, \dots, |w|\}$ .

$\delta : Q \times \Sigma \rightarrow Q$  Ist die Zustandsüberföhrungsfunktion.

$\omega : Q \times \Sigma \rightarrow \Gamma$ , Ist die Ausgabefunktion.

$q_0$  Ist der Startzustand.  $q_0 = (0, \dots, 0)$ . Es sind also noch keine Zeichen aufgedeckt.

$F$  Die Menge der Endzustände. Die Menge besteht nur aus  $q_e = (1, \dots, 1)$ . Bei diesem Zustand sind alle Zeichen aufgedeckt, also wurde das Suchwort gefunden.

Im Folgenden werden die Zustände nicht mehr als Vektoren angegeben, sondern die einzelnen Zeichen direkt ausgeschrieben (z. B.  $q_0 = 0^m$ ).

In der Aufzählung wurden  $\delta$  und  $\omega$  nicht genauer definiert. Dies wird nun nachgeholt. Dafür werden zunächst noch einige Begriffe benötigt.

Seien  $q' = q'_0 \dots q'_{m-1}$  und  $q = q_0 \dots q_{m-1}$  Zustände eines Boyer-Moore-Automaten.  $q'$  erfüllt die folgenden Eigenschaften in Bezug auf  $q$ :

$$\exists d \in \mathbb{N}_0 \forall i \in \{0, \dots, m-1\} : q'_i = \begin{cases} q_{i+d} & \text{Für } i < m-d \\ 0 & \text{Für } m > i \geq m-d \end{cases}$$

**Definition 2**  $q'$  ist ein zulässiger Zustand für  $q$  wenn gilt:

$$q'_i = 1 \rightarrow (q_{i+d} = 1 \wedge w_i = w_{i+d})$$

Hierbei wird  $d$  als eine zulässige Verschiebung des Zustands  $q$  bezeichnet.

**Definition 3** Die Funktion  $\text{dist}(q, q')$  liefert für einen Zustand  $q$  und einem Zustand  $q'$  das kleinste  $d$ , so dass gilt:

$$q'_i = \begin{cases} q_{i+d} & \text{Für } i < m-d \\ 0 & \text{Für } m > i \geq m-d \end{cases}$$

Sei  $k = \max \{i | q_i = 0\}$ . Ein Zustand  $q'$  ist ein **zulässiger Nachfolgezustand** von  $q$  beim lesen von  $a \in \Sigma$ , wenn gilt:

$$\exists d \in \mathbb{N}_0 \forall i \in \{0, \dots, m-1\} : q'_i = \begin{cases} q_{i+d} & \text{Für } i < m-d \\ 1 & \text{Für } i = k-d (\wedge k-d \geq 0) \\ 0 & \text{Für } m > i \geq m-d \end{cases}$$

und  $q'_i = 1 \rightarrow w_i = w_{i+d}$ .

$Z(z)$  sei die Menge der zulässigen Nachfolgezustände von  $z$  beim Lesen von  $a$ , so ist  $\delta(z, a) = y$ ,  $y = \min \{x \mid x \in Z \wedge \text{dist}(z, x)\}$ . Den Zustand  $y$ , nennt man den **gültigen Nachfolgezustand** von  $z$  beim Lesen von  $a$ .

Die Ausgabefunktion gibt nun die notwendige Verschiebung zwischen den beiden Zuständen  $z$  und  $y$ , beim Lesen von  $a$  zurück. Dies entspricht  $\omega(z, a) = \text{dist}(z, \delta(z, a))$ .

Nun definieren wir die Funktion.

**Definition 4** Die Funktion  $\text{pos}(q)$  liefert für einen Zustand den Index des Zeichens in  $w$ , das mit dem Zeichen in  $t$  verglichen werden muss.

$$\text{pos} : Q \rightarrow \mathbb{N}_0^+$$

zu  $\text{pos}(q) = \max \{i \mid q_i = 0\}$ . Diese Funktion liefert den Index des Zeichens zurück, das bei diesem Zustand als nächstes mit dem Text verglichen werden muss.

## 2.2.2 Eigenschaften der Zustände

Durch die Definition der Zustände des Boyer-Moore-Automaten können einige Eigenschaften der Zustände abgeleitet werden. Diese Eigenschaften werden in den nächsten Abschnitten behandelt.

### Triviale Schranken für die Anzahl der Zustände

Für die Anzahl der Zustände können Schranken direkt angegeben werden. So kann die Anzahl der Zustände in einem Boyer-Moore-Automaten niemals größer als  $2^m$  sein, wenn  $m$  die Länge des Zustandes ist. Andererseits muss es mindestens einen Pfad vom Startzustand zum Endzustand des Automaten geben. Dieser Pfad wird beim Einlesen des gesuchten Wortes vom Startzustand aus abgelaufen. Dieser Pfad enthält (inklusive dem Start- und dem Endzustand)  $m + 1$  Zustände. Also kann es auch keinen Boyer-Moore-Automaten mit weniger als  $m + 1$  Zuständen geben.

Sei nun  $|BMA|$  die Anzahl der Zustände eines Automaten, so gilt  $m + 1 \leq |BMA| \leq 2^m$ .

## 2.2.3 Konstruktion eines Boyer-Moore-Automaten

In diesem Abschnitt werden verschiedene Algorithmen angegeben, die zur Konstruktion des Boyer-Moore-Automaten für ein bestimmtes Suchwort  $w \in \Sigma^*$  verwendet werden können.

Allen diesen Algorithmen ist gemein, dass sie um für ein Suchwort  $w$  den Boyer-Moore-Automaten zu konstruieren vom Startzustand  $q_0$  aus für alle Zeichen  $a \in \Sigma$  die gültigen Nachfolgezustände für  $q_0$  beim Lesen von  $a$  erzeugen. Anschließend werden alle gültigen Nachfolgezustände der gültigen Nachfolgezustände von  $q_0$  erzeugt bis schließlich alle Zustände des Boyer-Moore-Automaten berechnet wurden.

### Naive Konstruktion

Der Algorithmus verwaltet verschiedene Mengen von Zuständen. Die Menge  $qBearbeitet$  beinhaltet alle Zustände, die schon in den Boyer-Moore-Automaten aufgenommen wurden und für die die Zustandsübergangsfunktion  $\delta$  und die Ausgabe (die Verschiebung)  $\omega$  schon für alle Zeichen bekannt ist. Die Menge  $qAkt$  ist die Menge der Zustände, für die diese beiden Funktionen in diesem Schritt des Algorithmus berechnet werden und die Menge  $qNext$  ist die Menge der Zustände  $q_n$ , für die gilt  $\exists q_u \in qAkt, a \in \Sigma : \delta(q_u, a) = q_n$ .  $qAkt$  enthält also die Zustände, welche direkt von einem aktuellen Zustand aus erreichbar sind. Die Funktionen  $\delta$  und  $\omega$  werden im Algorithmus als 3-Tupel  $\delta_m$  und  $\omega_m$  gespeichert:

$\delta_m$  Das Tupel besteht aus  $(q, a, q')$ .  $q$  ist der Zustand,  $a$  das Eingabezeichen für den Automaten und  $q'$  ist der gültige Nachfolgezustand für  $q$  beim Lesen von  $a$ .

$\omega_m$  Das Tupel besteht aus  $(q, a, d)$ .  $q$  ist der Zustand,  $a$  das Eingabezeichen für den Automaten und  $d$  ist die Verschiebung für den gültigen Nachfolgezustand  $z' = \delta(q, a)$ .

$\delta(q, a) = q' \Leftrightarrow (q, a, q') \in \delta_m$  und  $\omega(q, a) = d \Leftrightarrow (q, a, d) \in \omega_m$  definieren die beiden Funktionen. In jedem Durchlauf der *While*-Schleife wird die Menge  $qBearbeitet$  um die in diesem Schritt in  $qAkt$  enthaltenen Zustände erweitert. So wächst der Boyer-Moore-Automat immer weiter, bis alle vom Startzustand aus über Pfade erreichbaren Zustände in ihm enthalten sind. Dann ist die Menge  $qAkt$  leer, die Schleife wird verlassen und der Boyer-Moore-Automat zurückgegeben. Der Ablauf ist in Algorithmus 1 dargestellt.

### Algorithmus 1

**function** KONSTRUKTIONBOYERMOOREAUTOMAT ( $w:String$ ) **return** BMA **is**

$qAkt := \{q_0\};$

$qBearbeitet := \emptyset;$

$\delta := \emptyset;$

$\omega := \emptyset;$

**while**  $qAkt \neq \emptyset$  **loop**

$qNext = \emptyset;$

$\forall q \in qAkt$  **loop**

$\forall a \in \Sigma$  **loop**

$(q', d) = \text{GibGueltigenNachfolgezustand}(q, a);$

**if**  $q' \notin qBearbeitet$  **then**

$\delta_m := \delta_m + \{(q, a, q')\};$

$\omega_m := \omega_m + \{(q, a, d)\};$

$qNext := qNext + \{z'\};$

**end if;**

**end loop;**

$qBearbeitet := qBearbeitet + \{q\};$

**end loop;**

$qAkt := qNext;$

**end loop;**

$BMA\ bma := (qBearbeitet, \Sigma, \delta, \omega, q_0, \{1^m\});$

**return**  $bma;$

**end function;**

Es muss noch die Funktion *GibGueltigenNachfolgezustand* definiert werden. Sie gibt ein 2-Tupel  $(q', d)$  bestehend aus  $q'$ , dem gültigen Nachfolgezustand des übergebenen  $qs$  beim Lesen

von  $a$  und  $d$ , der Verschiebung zwischen  $q$  und  $q'$  zurück. Um den gültigen Nachfolgezustand zu einem Zustand  $q$  und dessen Verschiebung  $d$  zu finden, müsste der in diesem Zustand aufgedeckte Teil des Suchwortes  $w$  mit Verschiebungen, beginnend bei der Verschiebung 0, von sich selber verglichen werden. Für eine zulässige Verschiebung  $d$  von  $q$  muss gelten, dass  $\forall i \in \{0..m-1\} : q_i = 1 \rightarrow w_i = w_{i-d} \vee i-d < 0$ . Die gültige Verschiebung ist nun das minimale  $d$ , das eine zulässige Verschiebung ist. Dieses minimale  $d$ , für einen Zustand  $q$  bei einem bestimmten Suchwort  $w$  kann durch Algorithmus 2 ausgerechnet werden. Dabei ist  $pos = \max\{pos | q_{pos} = 0\}$  die Position der rechtensten 0 im Zustand  $q$ , also des Zeichens, das als nächstes aufgedeckt wird.

### Algorithmus 2

```

function GIBGUELTIGENNACHFOLGEZUSTAND ( $q:(0|1)^m, a:\Sigma$ ) return  $Q \times \mathbb{N}$  is
   $d = 0;$ 
  while  $d \leq m \wedge \text{groessereVerschiebung}$  loop
     $\text{groessereVerschiebung} := \text{false};$ 
    for  $i \in \{0..m-1\}$  loop
      if  $i = \text{pos}(q) - d$  then
        if  $w_i \neq a$  then
           $\text{groessereVerschiebung} := \text{true};$ 
        else
           $q'_i = 1;$ 
        end if;
      else
        if  $i + d < m$  then
          if  $(q_{i+d} = 1 \wedge (w_{i+d} \neq w_i))$  then
             $\text{groessereVerschiebung} := \text{true};$ 
          else
             $q'_i := q_{i+d};$ 
          end if;
        else
           $q'_i := 0;$ 
        end if;
      end if;
    end loop;
     $d := d + 1;$ 
  end loop;
  return  $(q', d - 1);$ 
end function;

```

In der *while*-Schleife werden, beginnend bei der kleinsten Verschiebung, alle möglichen Verschiebungen ausprobiert, deshalb wird am Ende der *while*-Schleife  $d$  inkrementiert. Die erste zulässige Verschiebung die gefunden wird, ist automatisch auch die gültige Verschiebung, da es keine kleinere zulässige Verschiebung gegeben hat. In der *for*-Schleife werden nun alle aufgedeckten Zeichen des Suchwortes  $w$  mit den verschobenen Zeichen verglichen und anhand dieser entschieden, ob die Verschiebung zulässig ist. Wenn die Verschiebung nicht zulässig ist, wird *groessereVerschiebung* auf *true* gesetzt, um anzuzeigen, dass erst die nächste Verschiebung zu einem gültigen Nachfolgezustand führen kann.

Die Laufzeit von *GibGueltigenNachfolgezustand* ist wegen der ineinander geschachtelten Schleifen in  $O(m^2)$ . Dadurch ist die Gesamtlaufzeit von *KonstruktionBoyerMooreAutomat* in  $O(|Q| \cdot \Sigma \cdot O(\text{GibGueltigenNachfolgezustand}))$ , also  $O(|Q| \cdot |\Sigma| \cdot m^2)$ .

### Beschleunigte Konstruktion

Der Algorithmus der naiven Konstruktion wird mit einigen Veränderungen übernommen. Es wird nur die Funktion *GibGueltigenNachfolgezustand* angepasst. Diese Änderungen (von [BYCG94]) beschleunigen die Laufzeit der Konstruktion auf  $O(|Q| \cdot \Sigma \cdot m)$ .

Während der Konstruktion wird für jeden Zustand eine Liste aller zulässigen Verschiebungen mitgespeichert. Wenn nun ein neues Zeichen gelesen wird, muss nur noch für jede zulässige Verschiebung getestet werden, ob das Zeichen an der Stelle  $pos - d$  das gleiche wie das gelesene Zeichen ist. Da nur  $O(m)$  Verschiebungen möglich sind und ein Vergleich in  $O(1)$  möglich ist, resultiert die Laufzeit von *GibGueltigenNachfolgezustand* mit  $O(m)$ . Um diese Liste für jeden Zustand zu speichern, muss *KonstruktionBoyerMooreAutomat* minimal geändert werden. Dies ist in Algorithmus 3 dargestellt.

#### Algorithmus 3

```

function KONSTRUKTIONBOYERMOOREAUTOMAT (w:String) return BMA is
  qAkt := {(q0, 1m)};
  qBearbeitet := ∅;
  δ := ∅;
  ω := ∅;
  while qAkt ≠ ∅ loop
    qNext = ∅;
    ∀ (q, liste) ∈ qAkt loop
      ∀ a ∈ Σ loop
        (q', liste', d) = GibGueltigenNachfolgezustand(q, liste, a);
        if q' ∉ qBearbeitet then
          δm := δm + {(q, a, q')};
          ωm := ωm + {(q, a, d)};
          qNext := qNext + {(q', liste')};
        end if;
      end loop;
    qBearbeitet := qBearbeitet + {q};
  end loop;
  qAkt := qNext;
end loop;
  BMA bma := (qBearbeitet, Σ, δ, ω, q0, {1m});
  return bma;
end function;

```

*qAkt* ist nun nicht mehr eine Menge von Zuständen, sondern eine Menge von 2-Tupeln  $\in Q \times \{0|1\}^m$ . Auch die Signatur von *GibGueltigenNachfolgezustand* hat sich geändert. Die Funktion übernimmt nun auch *liste*. In *liste* sind alle zulässigen Verschiebungen für den aktuellen Zustand *q* gespeichert (von 0 bis *m*). In *liste'* werden alle zulässigen Verschiebungen für den Zustand *q'* zurückgegeben. Die Definition von *GibGueltigenNachfolgezustand* ist in Algorithmus 4 angegeben.

**Algorithmus 4**

```

function GIBGUELTIGENNACHFOLGEZUSTAND ( $q:\{0|1\}^m, liste:\{0|1\}^m, a:\Sigma$ ) return  $Q \times \{0|1\}^m \times \mathbb{N}$  is
   $gueltigeVerschiebung := 0;$ 
   $weiter := true;$  ▷ Gueltige Verschiebung finden.
  while  $gueltigeVerschiebung \leq m \wedge weiter$  loop
    if  $liste_i = 1$  then
      if  $pos(q) - gueltigeVerschiebung < 0 \vee w_{pos(q)-gueltigeVerschiebung} = a$  then
         $weiter := false;$ 
      end if;
    end if;
     $gueltigeVerschiebung := gueltigeVerschiebung + 1;$ 
  end loop; ▷ 1.Fall:
  for  $d \in \{0, \dots, pos - gueltigeVerschiebung\}$  loop
    if  $liste_{d+gueltigeVerschiebung} = 1 \wedge w_{pos(q)-gueltigeVerschiebung-d} = a$  then
       $liste'_d := 1;$ 
    else
       $liste'_d := 0;$ 
    end if;
  end loop; ▷ 2.Fall:
  for  $d \in \{(pos(q) + 1) - gueltigeVerschiebung, \dots, m - gueltigeVerschiebung\}$  loop
    if  $d \geq 0$  then
      if  $liste_{d+gueltigeVerschiebung} = 1$  then
         $liste'_d := 1;$ 
      else
         $liste'_d := 0;$ 
      end if;
    end if;
  end loop; ▷ 3.Fall:
  for  $d \in \{m - gueltigeVerschiebung, \dots, m\}$  loop
     $liste'_d := 1;$ 
  end loop;
   $q' := berechneVerschobenenZustand(q, gueltigeVerschiebung);$ 
  return  $(q', liste', gueltigeVerschiebung);$ 
end function;

```

Um nun eine zulässige Verschiebung für das Zeichen  $a$  zu finden, muss nur für alle zulässigen Verschiebungen (die in  $liste$  gespeichert sind) überprüft werden, ob das Zeichen  $a$  an eine Stelle verschoben wird, an der auch im Suchwort  $w$  das Zeichen  $a$  steht. Die kleinste zulässige Verschiebung ist die gültige Verschiebung. Es muss also für eine gültige Verschiebung gelten, sie ist eine zulässige Verschiebung ( $liste_{gueltigeVerschiebung} = 1$ ) und  $w_{pos-gueltigeVerschiebung} = a$  und sie wird als erste zulässige Verschiebung für das Zeichen  $a$  in einer aufsteigenden Suche gefunden. Damit ist die gültige Verschiebung gefunden und auch der gültige Nachfolgezustand kann leicht berechnet werden. Die Hauptarbeit ist nun, die Liste der zulässigen Verschiebungen für den gültigen Nachfolgezustand anzupassen. Wenn nun  $gueltigeVerschiebung$  die gültige Verschiebung von  $q$  zum gültigen Nachfolgezustand  $q'$  für das Zeichen  $a$  ist, gilt folgendes:

$$\forall d \in \mathbb{N} : list'_d = 1 \Rightarrow list_{d+gueltigeVerschiebung} = 1$$

Allerdings ist die Rückrichtung offensichtlich nicht gültig

$$\forall d \in \mathbb{N} : list_{d+gueltigeVerschiebung} = 1 \not\Rightarrow list'_d = 1$$

da in  $q'$  das Zeichen  $a$  aufgedeckt wurde. Alle Verschiebungen  $list_{d+gueltigeVerschiebung} = 1$  für die  $pos(q) - gueltigeVerschiebung - d \geq 0 \wedge w_{pos(q)-gueltigeVerschiebung-d} \neq a$  bewirken, dass  $list_{i+gueltigeVerschiebung} = 0$  ist. Für eine zulässige Verschiebung für den Zustand  $q'$  kann trotzdem  $liste$  verwendet werden. Hierbei müssen 3 Fälle unterschieden werden:

1. Die Verschiebung  $d$  belässt das gerade eingelesene Zeichen im Zustand ( $pos(q) - gueltigeVerschiebung \geq d$ )  
 $\Rightarrow list'_d = 1 \Leftrightarrow list_{gueltigeVerschiebung+d} = 1 \wedge w_{pos(q)-gueltigeVerschiebung-d} = a$ .
2. Die Verschiebung  $d$  verschiebt das gerade eingelesene Zeichen über den Anfang des Zustandes hinaus ( $pos(q) - gueltigeVerschiebung < d$ ). Es kann jedoch sein, dass die Verschiebung im alten Zustand  $q$  nicht zulässig war ( $d \leq m - gueltigeVerschiebung$ ). Hier gilt  $list'_d = list_{d+gueltigeVerschiebung}$ .
3. Alle größeren Verschiebungen sind zulässig, da sie alle bisher eingelesenen Zeichen aus dem Zustand schieben ( $m \geq d > m - gueltigeVerschiebung$ ).

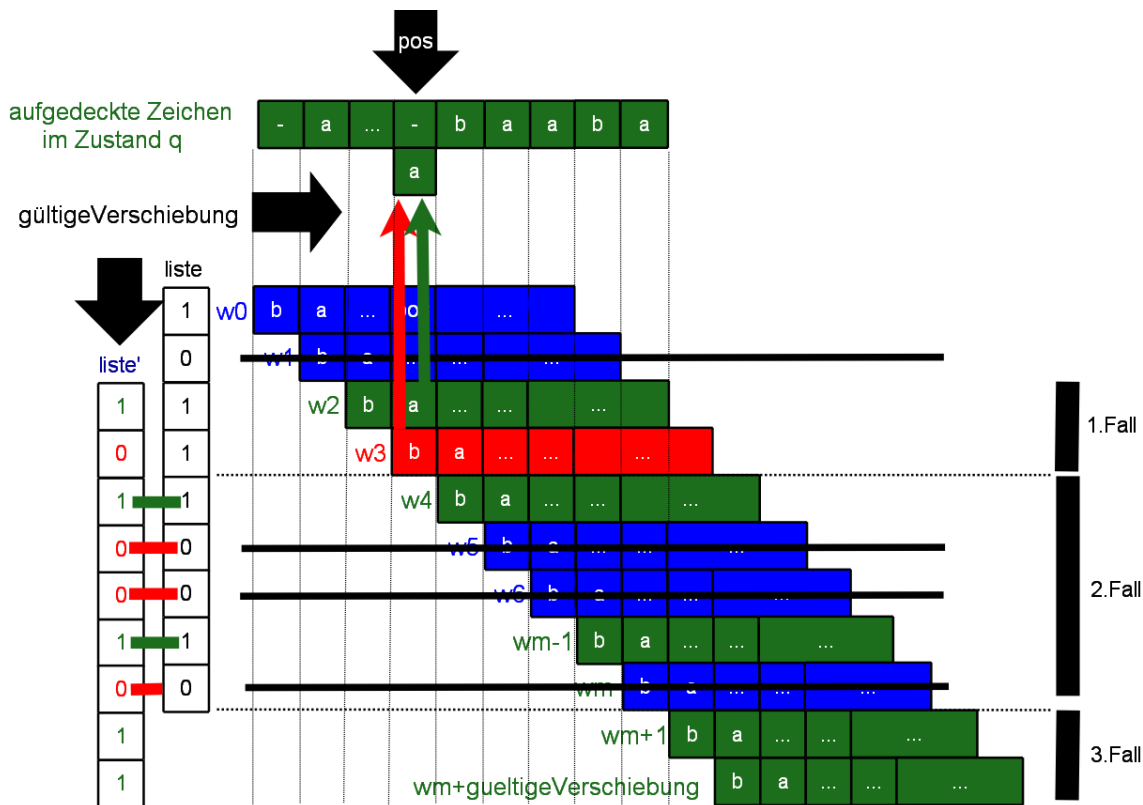


Abbildung 2.3: Berechnung von  $liste'$  aus  $liste$  bei angenommenen gelesenen Zeichen  $a$  im Zustand  $q$

In Abbildung 2.3 wird die Berechnung der neuen zulässigen Verschiebungen für den Zustand  $q'$  veranschaulicht. Gegeben ist der Zustand  $q$  und das Suchwort  $w$ . Die aufgedeckten Zeichen in  $q$  entsprechen den Zeichen in  $w$ , wenn der Zustand an dieser Stelle 1 ist, also:

$$AufgedeckteZeichen_i = \begin{cases} w_i & q_i = 1 \\ - & q_i = 0 \end{cases}$$

Die gültige Verschiebung für  $q$  beim Lesen von  $a$  wurde schon berechnet (und beträgt im Beispiel 2). Im Bild sieht man nun für jede zulässige Verschiebung von  $q$  das verschobene  $w$  ( $w_1, w_2, \dots$ ), die nicht zulässigen Verschiebungen in  $q$  sind durchgestrichen. Nun wird die Liste der zulässigen Verschiebungen für  $q'$ , also  $liste'$ , auch um die gültige Verschiebung versetzt angelegt. Zulässige  $w$ 's in  $q'$  sind grün gekennzeichnet. Wenn sie in  $q$  zulässig waren, nun aber nicht mehr zulässig sind, sind sie rot gekennzeichnet. Die drei verschiedenen Fälle können im Bild erkannt werden.

Beim ersten Fall ist nach der Verschiebung das an  $pos$  gelesene Zeichen noch im verschobenen  $w$ . Es muss überprüft werden, ob das gelesene Zeichen an dieser Position stehen darf, aber auch nur, wenn die Verschiebung vorher zulässig war ( $liste_{d-gueltigeVerschiebung} = 1$ ).

Im zweiten Fall wird durch die Verschiebung das an  $pos$  gelesene Zeichen über den Anfang des verschobenen  $w$ s hinausgeschoben. Hier ist nur noch wichtig, ob die Verschiebung  $d - gueltigeVerschiebung$  für  $q$  zulässig war ( $liste_{d-gueltigeVerschiebung} = 1$ ).

Im dritten Fall wird durch die Verschiebung das gesamte verschobene  $w$  über die bisherigen aufgedeckten Zeichen hinaus verschoben. Da der Folgezustand hier nur aus nicht aufgedeckten Zeichen besteht ( $q''_j = 0 \forall j$ ), sind alle diese Verschiebungen im Zustand  $q'$  zulässig.

Da bei jedem der drei Fälle maximal ein Vergleich eines Zeichens nötig wird ( $O(1)$ ) und  $d \in \{0, \dots, m\}$  gilt, ist die Laufzeit der Berechnung des gültigen Nachfolgezustands mit  $liste'$  in  $O(m)$ . Womit sich die Gesamtlaufzeit der Konstruktion des BMA in  $O(|Q| \cdot |\Sigma| \cdot m)$  ergibt.

### Rekursive Konstruktion

Ein anderer Algorithmus, um den Boyer-Moore-Automaten für ein Suchwort  $w = aw', a \in \Sigma$  zu generieren, verwendet den Boyer-Moore-Automaten für das Suchwort  $w'$ . Hierbei muss für jeden Zustandsübergang  $\delta(q_{w'}, a) = q''_{w'}$  im Automaten von  $w'$  ( $A_{w'}$ ) untersucht werden, ob der Folgezustand  $q'_w = (0|1)q''_{w'}$  ein zulässiger Folgezustand von  $q_w = 0q_{w'}$  ist. Außerdem müssen für den Endzustand von  $A_{w'}$  die Folgezustände in  $A_w$  berechnet werden.

Die Verschiebungen in  $A_w$  sind alle größer gleich den Verschiebungen von  $A_{w'}$ , da das hinzugefügte Zeichen  $a$  nur zulässige Zustände zu unzulässigen Zuständen verändern kann. Dies geschieht  $gdw w_0 \neq w_{\omega_w(q_w, a)}$ , also durch die Verschiebung ein Zeichen auf die erste Stelle des Zustandes geschoben wird, das nicht dort stehen darf. Im Extremfall wird also der Folgezustand zu einem unzulässigen Zustand in  $A_w$  und  $\delta_w(q_w, a) \neq \delta_{w'}(q_{w'}, a)$ , damit gilt dann  $\omega_w(q_w, a) > \omega_{w'}(q_{w'}, a)$ . In diesen Fällen muss der Folgezustand neu errechnet werden. Nun können zwei Fälle unterschieden werden:

$$q'_w = \delta_w(q_w, a) = xq''_{w'}, x \in \{0, 1\} \begin{cases} q''_{w'} \in Q_{w'} \wedge x = 0 & \text{Es entsteht kein neuer Zustand in } A_w \\ x = 0 \vee q''_{w'} \notin Q_{w'} & \text{Es entsteht ein neuer Zustand in } A_w \end{cases}$$

Im ersten Fall kann die Anzahl der Zustände  $Q_w$  kleiner werden als die Anzahl der Zustände  $Q_{w'}$ . Während durch den zweiten Fall Zustände erzeugt werden, die noch nicht erreichbar waren und somit hieraus  $Q_w$  mehr Zustände haben kann als  $Q_{w'}$ . Für die Anzahl der Zustände  $Q_w$  kann also keine Aussage gemacht werden, ob  $|Q_w| > |Q_{w'}|$  oder  $|Q_w| < |Q_{w'}|$  gilt. Die Laufzeit dieses Algorithmus wird dadurch bestimmt, wieviele Zustände entstehen, die keine Entsprechung im BMA für  $w'$  haben. Für all diese Zustände müssen mit dem naiven Verfahren die Nachfolgezustände bestimmt werden, da es keine vorbereitete Liste der gültigen Verschiebungen gibt. Schlimmstenfalls ist Laufzeit also gleich der des naiven Algorithmus.

### Eine notwendige Bedingung für erreichbare Zustände

Aus der Konstruktion des Boyer-Moore-Automaten kann man eine Forderung an alle Zustände ableiten, die vom Startzustand aus erreichbar sind:

- Wenn ein Block aus 1'en in einem Zustand nicht mit dem letzten Zeichen des Zustandes aufhört, so ist er durch eine Verschiebung vom Suffix des Zustandes an die aktuelle Position geschoben worden. In diesem Fall muss er sich mit dem ersten Zeichen vom damals aktuellen Zeichen des Suchwortes  $w$  unterscheiden haben. Nach dieser Verschiebung können sich nie wieder neue 1'er Blöcke aus diesem Block entwickeln. Er kann sich nur mit einem rechts von ihm liegenden 1'er Block zu einem größeren Block vereinigen.

Für jeden Block von 1'en muss somit gelten, dass er sich an genau einer Stelle von einem gleichlangen Suffix von  $w$  unterscheiden muss. In Abbildung 2.4 ist ein Beispielsuchwort  $w$  mit seinen Suffixen gegeben. Für dieses Suchwort sind nach der notwendigen Bedingung nun z. B. die Zustände in Abbildung 2.5 möglich.

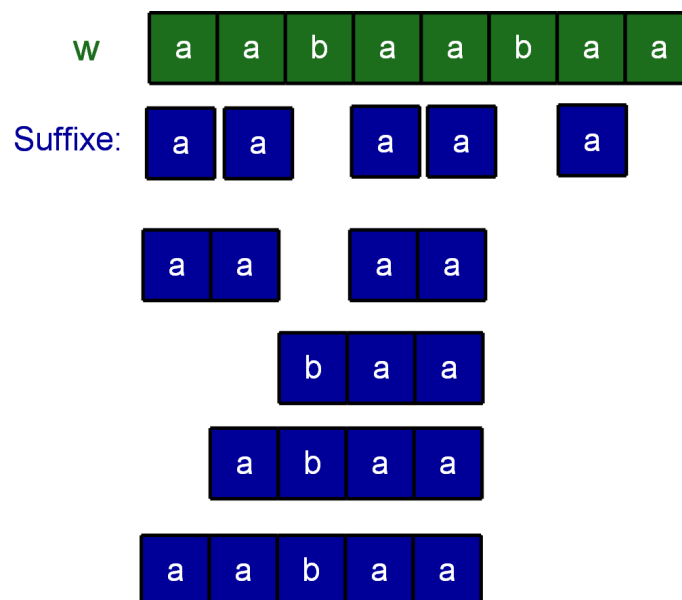


Abbildung 2.4: Die Suffixe für ein gegebenes  $w$ .



Abbildung 2.5: Einige Zustände, die die notwendige Bedingung für das Suchwort  $w$  erfüllen.

### 2.2.4 Beispiel

Als Beispiel soll in diesem Abschnitt der Boyer-Moore-Automat für das Suchwort  $w = abb$  konstruiert werden. Es wird die naive Konstruktion verwendet, da sie am anschaulichsten ist. Zunächst besteht die Menge  $qAkt$  nur aus dem Startzustand 000 (siehe 2.6). Dieser wird in

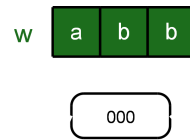


Abbildung 2.6: BMA mit Startzustand.

der *while*-Schleife bearbeitet. Zuerst wird für das Eingabezeichen  $a$  der Nachfolgezustand mit *GibGültigenNachfolgezustand* gesucht. Es wird die Verschiebung um zwei Zeichen und der gültige Nachfolgezustand  $q' = 100$  für das Zeichen  $a$  gefunden und dem Automaten hinzugefügt. Dies ist in Abbildung 2.7 zu sehen. Der Zustand 100 wird der Menge  $qAkt$  hinzugefügt.

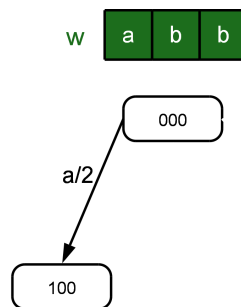


Abbildung 2.7: BMA nach hinzufügen von 100.

Anschließend wird der gültige Nachfolgezustand von 000 beim Lesen von  $b$  gesucht. Hier ist keine Verschiebung nötig und es wird der Zustand 001 gefunden. Auch dieser Zustand wird dem Boyer-Moore-Automaten hinzugefügt und in der Menge  $qAkt$  abgelegt (Abbildung 2.8). Nun sind alle Nachfolgezustände von 000 berechnet. Es wird ein Zustand aus der Menge  $qAkt$

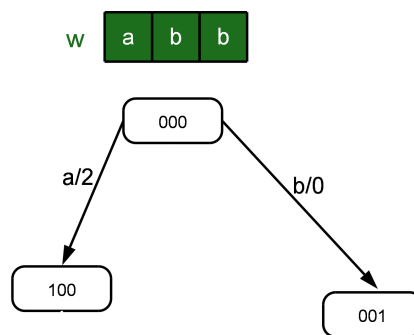
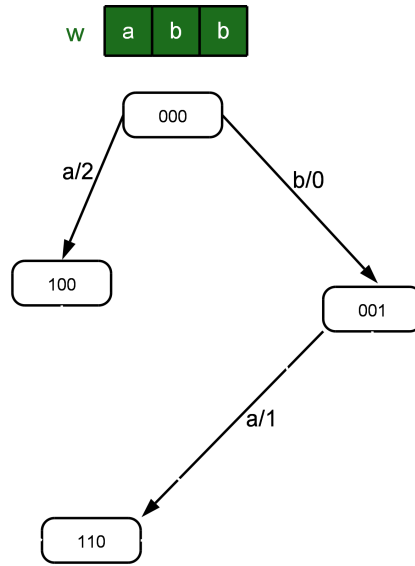


Abbildung 2.8: BMA nach dem 000 komplett verarbeitet wurde.

entnommen. Es ist nicht wichtig, ob nun 100 oder 001 zurückgegeben wird. In diesem Beispiel wird ein „Keller-verhalten“ angenommen und es wird als nächstes der Zustand 001 bearbeitet. Für das Zeichen  $a$  wird der Nachfolgezustand 110 gefunden, da hier  $ab$  im Suchwort  $w$  gefunden werden muss (siehe Abbildung 2.9). Hierzu muss um eins verschoben werden, damit das gelesene  $a$  erlaubt ist. Beim Zeichen  $b$  ist wieder keine Verschiebung nötig um den Zustand 011 zu finden. In Abbildung 2.10 ist der bis jetzt aufgebaute Boyer-Moore-Automat zu sehen.

Abbildung 2.9: BMA nach dem Bearbeiten von  $a$  im Zustand 001.

Als nächstes wird nun der zuletzt in die Menge  $qAkt$  eingefügte Zustand 011 weiterverarbeitet. Für das Eingabezeichen  $a$  muss die Verschiebung der Länge  $m$  zum Startzustand verwendet werden. Der Startzustand ist schon verarbeitet worden und wird deshalb nicht in die Menge  $qAkt$  eingefügt.

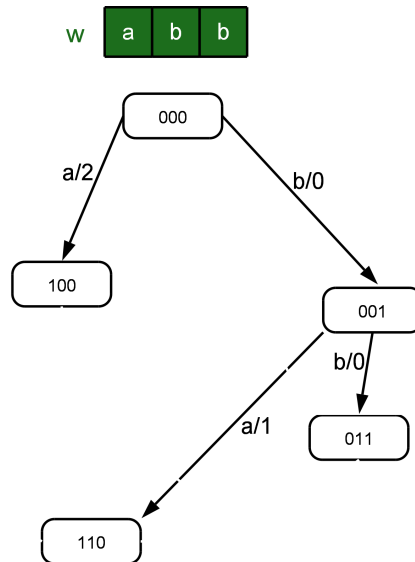


Abbildung 2.10: BMA nach Verarbeitung von 001.

Für das Eingabezeichen  $b$  ist keine Verschiebung nötig und der Nachfolgezustand ist der Endzustand 111, auch dieser wird per Definition nicht in die Menge  $qAkt$  aufgenommen (in Abbildung 2.11 zu sehen).

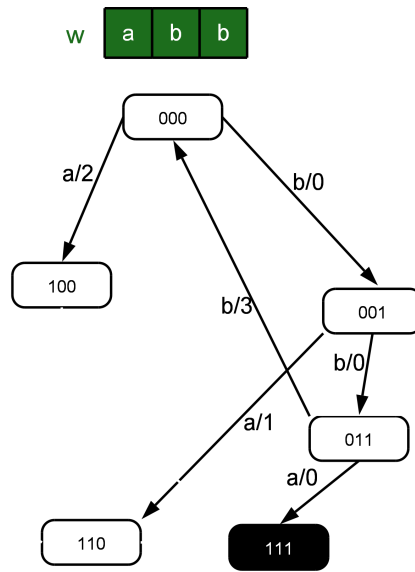


Abbildung 2.11: Der BMA nach Verarbeitung von 011.

Nun wird der Zustand 110 bearbeitet. Für das Zeichen  $a$  wird hier der Nachfolgezustand 100 und für  $b$  der Endzustand 111 gefunden. Der Zustand 100 ist schon in der Menge  $qAkt$  enthalten, wird aber trotzdem nochmal hinzugefügt. Dies ist nur für die Verarbeitungsreihenfolge in diesem Beispiel wichtig. Danach ist der Boyer-Moore-Automat wie in Abbildung 2.12 zu sehen aufgebaut.

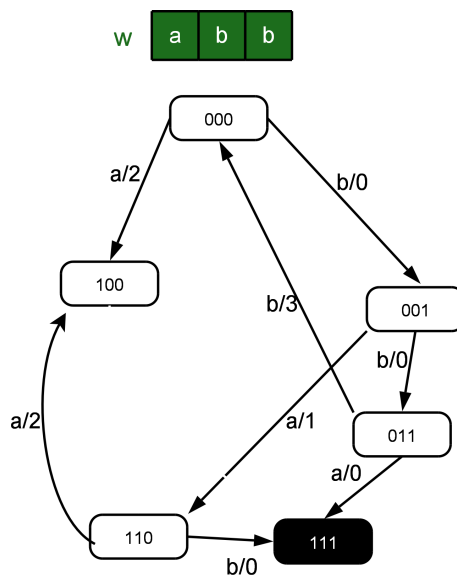


Abbildung 2.12: Der BMA nach Verarbeitung von 110.

Für den Zustand 100 wird zuerst der Nachfolgezustand beim Lesen von  $a$  berechnet. Dies ist der Zustand 100 selber. Der Zustand zählt schon als bearbeitet und wird deshalb nicht in  $qAkt$

aufgenommen. Für das Zeichen  $b$  wird der gültige Nachfolgezustand 101 berechnet und in die Menge  $qAkt$  eingefügt (siehe Abbildung 2.13).

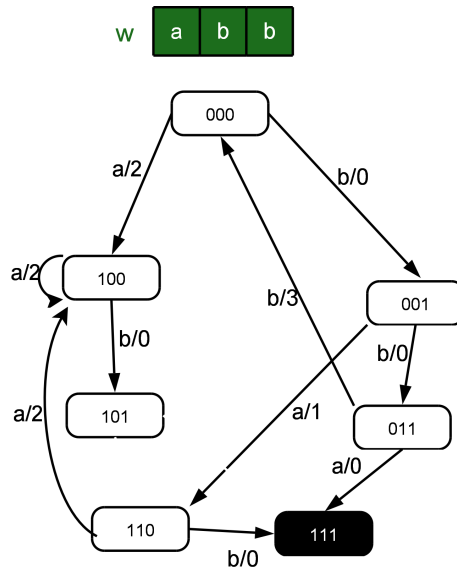


Abbildung 2.13: Der BMA nach Verarbeitung von 100.

Als letzter Zustand wird der Zustand 101 bearbeitet. Die gültigen Nachfolgezustände sind hier für  $a$  der Zustand 110 und für  $b$  der Zustand 111. 110 wurde schon bearbeitet und 111 ist der Endzustand. Es werden also keine Zustände in die Menge  $qAkt$  aufgenommen und die Menge bleibt leer, die *while*-Schleife wird verlassen und die Konstruktion des Boyer-Moore-Automaten ist abgeschlossen. Der komplette Boyer-Moore-Automat ist in Abbildung 2.14 zu sehen.

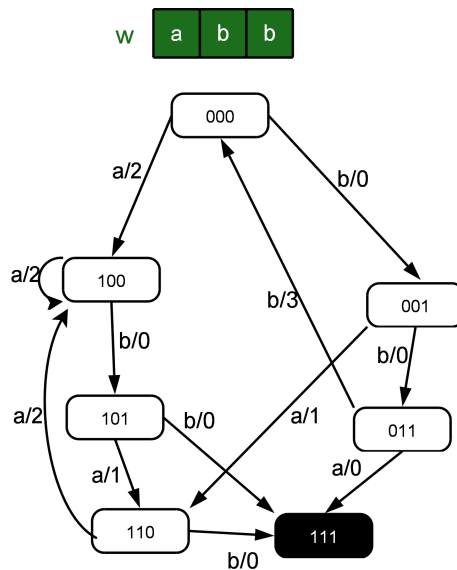


Abbildung 2.14: Der BMA nachdem die Konstruktion abgeschlossen wurde.

## 2.3 Der erweiterte Boyer-Moore-Algorithmus

### 2.3.1 Beschreibung des Algorithmus

Der prinzipielle Ablauf des erweiterten Boyer-Moore-Algorithmus ist gleich dem des klassischen Boyer-Moore-Algorithmus. Der erweiterte Boyer-Moore-Algorithmus benutzt einen Boyer-Moore-Automaten, um sich die bisher aufgedeckten Zeichen von  $w$  zu merken. Der Boyer-Moore-Automat ersetzt auch die Tabellen `shift1` und `shift2`. In der Suchphase wird statt diesen Tabellen der Boyer-Moore-Automat benutzt. In der Vorberechnungsphase muss statt der Tabellen der Boyer-Moore-Automat berechnet werden.

### 2.3.2 Die Konfiguration des erweiterten Boyer-Moore-Algorithmus

Ähnlich dem klassischen Boyer-Moore-Algorithmus kann hier eine Konfiguration des Algorithmus als ein 4-Tupel  $(i, q, w, t)$   $i \in \{0..n - 1\}$  definiert werden.  $i$  ist die Stelle, an der der Algorithmus in  $t$  prüft.  $q$  ist der Zustand, in dem sich der Boyer-Moore-Automat befindet und  $pos(q)$  beschreibt die Position des Zeichens  $w_{pos}$  in  $w$ , das als nächstes mit dem Zeichen  $t_i$  verglichen wird. Diesen Vorgang des Vergleichens und Abfragens des Zeichens von  $t$  wird auch hier 'aufdecken' genannt. Ein Ablauf des Algorithmus ist eine Folge von Konfigurationen. Die Startkonfiguration  $k_0$  ist  $(m - 1, q_0, w, t)$ . Hat der Algorithmus ein Vorkommen von  $w$  in  $t$  gefunden, liefert er genau dann `true` zurück, wenn er eine Konfiguration  $(i, q_e, w, t)$ ,  $q_e \in F$  erreicht hat. Eine Konfiguration  $k_e$  ist genau dann erreichbar wenn es einen Ablauf  $k_0 k_1 .. k_e$  und  $\forall j \in \{1..e\} \exists a \in \Sigma : k_j = \delta(k_{j-1}, a)$  gibt.

Die Funktion  $\delta$  der möglichen Konfigurationsübergängen im Algorithmus wird nun definiert. Für zwei Konfigurationen  $k_u = (i, q, w, t)$  und  $k_v = (i', q', w, t)$  und dem Zeichen  $a \in \Sigma \wedge a = t_i$  gilt,  $k_u = \delta(k_v, t_i)$  gdw  $q' = \delta(q, a) \wedge i' = i + \sigma(q, a) + pos(q') - pos(q)$ . Hierbei ist  $\sigma(q, a)$  die Verschiebung des Suchwortes  $w$  am Text  $t$  und der Term  $pos(q') - pos(q)$  die Verschiebung innerhalb des Suchwortes  $w$ , um wieder das rechteste nicht aufgedeckte Zeichen zu vergleichen.

### 2.3.3 Ablauf der Suchphase

Zu Beginn ist kein Zeichen von  $w$  aufgedeckt und der aktuelle Zustand ist  $z = 0^m$ . Der Zustand wird nun anschaulich ab der Stelle  $i$  über  $t$  gelegt. Eine 0 im Zustand bedeutet, dass das Zeichen noch nicht aufgedeckt ist. Eine 1 zeigt an, dass das Zeichen aufgedeckt ist und mit dem entsprechenden Zeichen von  $w$  übereinstimmt. Das Zeichen, das der letzten 0 des Zustandes entspricht, wird nun gelesen. Der Boyer-Moore-Automat wechselt seinen Zustand gemäß der Zustandsübergangsfunktion  $\delta(z, a)$  für das eingelesene Zeichen  $a$ . Gleichzeitig wird eine Verschiebung gemäß  $\omega(z, a)$  ausgeführt. Wenn eine Verschiebung ausgeführt wurde, wird der Zustand ab Zeichen  $i + \omega(z, a)$  über den Text  $t$  gelegt. Das gesuchte Wort  $w$  ist gefunden, sobald der Zustand  $1^m$  erreicht wird.

### 2.3.4 Ablauf der Vorberechnungsphase

In der Vorberechnungsphase müssen alle, vom Startzustand aus erreichbaren Zustände berechnet werden, damit später in der Suchphase direkt der neue Zustand und die Verschiebung angegeben werden kann. Hierfür kann der Algorithmus aus Abschnitt 2.2.3 verwendet werden.

### 2.3.5 Beispiel

Als Beispiel soll nun mit dem erweiterten Boyer-Moore-Algorithmus das Vorkommen von  $w = abba$  im Suchtext  $t = abbabaabbaba$  gefunden werden. Der Boyer-Moore-Automat für das Suchwort  $w$  ist in Abbildung 2.15 dargestellt.

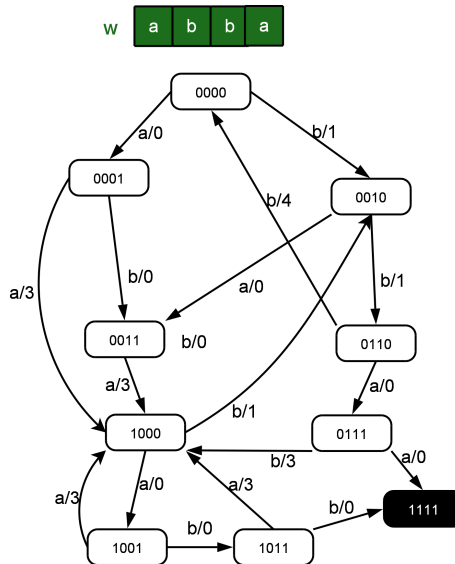


Abbildung 2.15: Der Beispiel-BMA

Dieser Automat kann mit einem Verfahren aus dem vorherigen Abschnitt erzeugt werden. Die Suchphase benutzt den Automat, um die Verschiebungen und die neuen Zustände zu berechnen.

In der Suchphase starten wir wieder mit der Konfiguration

$$k_0 = (3, (0, 0, 0, 0), abba, abbabaabbaba)$$

Der Ablauf der Berechnung ist folgender:

$$\begin{aligned} (3, (0, 0, 0, 0), abba, abbabaabbaba) &\xrightarrow{\omega((0,0,0,0),b)+pos(q')-pos(q)=1} (4, (0, 0, 1, 0), abba, abbabaabbaba) \\ (4, (0, 0, 1, 0), abba, abbabaabbaba) &\xrightarrow{\omega((0,0,1,0),a)+pos(q')-pos(q)=-2} (2, (0, 0, 1, 1), abba, abbabaabbaba) \\ (2, (0, 0, 1, 1), abba, abbabaabbaba) &\xrightarrow{\omega((0,0,1,1),b)+pos(q')-pos(q)=-1} (1, (0, 1, 1, 1), abba, abbabaabbaba) \\ (1, (0, 1, 1, 1), abba, abbabaabbaba) &\xrightarrow{\omega((0,1,1,1),b)+pos(q')-pos(q)=6} (7, (1, 0, 0, 0), abba, abbabaabbaba) \\ (7, (1, 0, 0, 0), abba, abbabaabbaba) &\xrightarrow{\omega((1,0,0,0),a)+pos(q')-pos(q)=-1} (6, (1, 0, 0, 1), abba, abbabaabbaba) \\ (6, (1, 0, 0, 1), abba, abbabaabbaba) &\xrightarrow{\omega((1,0,0,1),a)+pos(q')-pos(q)=4} (10, (1, 0, 0, 0), abba, abbabaabbaba) \\ (10, (1, 0, 0, 0), abba, abbabaabbaba) &\xrightarrow{\omega((1,0,0,0),a)+pos(q')-pos(q)=-1} (9, (1, 0, 0, 1), abba, abbabaabbaba) \\ (9, (1, 0, 0, 1), abba, abbabaabbaba) &\xrightarrow{\omega((1,0,0,1),b)+pos(q')-pos(q)=-1} (8, (1, 0, 1, 1), abba, abbabaabbaba) \end{aligned}$$

$(8, (1, 0, 1, 1), abba, abbabaabbaba) \xrightarrow{\omega((1,0,1,1),b)+\text{pos}(q')-\text{pos}(q)=-1} (7, (1, 1, 1, 1), abba, abbabaabbaba)$

Der letzte Zustand ist ein Endzustand ( $\in F$ ) und damit terminiert der Algorithmus. Im Vergleich zum klassischen Beispiel werden weniger Schritte benötigt, um das Vorkommen zu finden, da beim erweiterten Boyer-Moore-Algorithmus keine Zeichen des Textes doppelt gelesen werden.

## 2.4 Gierige Algorithmen zur Berechnung „großer“ Suchwörter

In diesem Abschnitt werden einige gierige Algorithmen beschrieben, mit deren Hilfe Suchwörter gefunden werden sollen, die relativ große Boyer-Moore-Automaten bei der Konstruktion erzeugen. Die Verfahren wurden für das 2- und 3-elementige Alphabet  $A_2 = \{a, b\}$ ,  $A_3 = \{a, b, c\}$  durchgeführt. Für einige dieser Algorithmen kann eine geschlossene Formel der Größe der gefundenen Boyer-Moore-Automaten angegeben werden. Diese Formeln können als Verbesserung der unteren Schranke der Abschätzung aus Abschnitt 2.2.2 benutzt werden. Andererseits ist es nicht möglich für diese Suchwörter sicherzustellen, dass sie die optimalen Suchwörter für eine bestimmte Suchwortlänge sind, so dass die obere Schranke der Abschätzung unberührt bleibt. Tatsächlich wächst die Differenz der Größen der BMA's mit der Suchwortlänge an.

Alle untersuchten gierigen Verfahren erzeugen ein Suchwort  $w$  einer bestimmten Länge aus dem besten gefundenen Suchwort  $w'$  der Länge  $|w'| + 1 = |w|$ , außer es soll das beste Suchwort der Länge 1 gefunden werden. In diesem Fall wird das Suchwort  $w = a$  zurückgegeben.

### 2.4.1 Sukzessives Anhängen des letzten Zeichens

Bei diesem Verfahren wird das beste Suchwort  $|w'| = n - 1$  erweitert durch  $w = \max \{w'x | x \in A_i, i \in \{2, 3\}\}$ .  $A_i$  bezeichnet hier das  $i$  elementige Alphabet. Das Verfahren kann definiert werden durch folgende Funktion:

$$V_l(n) = \begin{cases} \max \{V_l(n-1) \circ x, x \in A_i\} & n > 1 \\ a & n = 1 \end{cases}$$

Dieser Algorithmus findet für  $|w| = n$  immer das Suchwort  $w = a^n$ . Diese Aussage kann durch folgenden Induktionsbeweis gezeigt werden:

*IA :*

Für die Länge 1 wird das Suchwort  $w = a$  berechnet.  $|w| = 1$ ,  $a = a^1$ .

*IV :*

$$\exists n : V_l(n) = a^n$$

.

*IB :*

$$\forall n \in \mathbb{N} : V_l(n) = a^n \Rightarrow V_l(n+1) = a^{n+1}$$

*ISchritt :*

Wir nehmen an:  $n \geq 1$ , daraus folgt:  $V_l(n+1) = V_l(n) \circ x \Rightarrow^{IV} V_l(n+1) = a^n \circ x$

$$BMA(a^n \circ x) = \begin{cases} \frac{1}{2}(n+1)^2 + \frac{1}{2}(n+1) + 1 & x = a \\ 2(n+1) & x \neq a \end{cases}$$

Da  $\frac{1}{2}(1+1)^2 + \frac{1}{2}(1+1) + 1 = 2 + 1 + 1 = 4 \geq 2(1+1)$  und die erste Lösung bevorzugt wird gilt immer  $V_l(n+1) = a^{n+1}$ .

Für die Abschätzung der Anzahl der Zustände für ein optimales Suchwort  $w_{opt}$  mit der Länge  $n$  gilt:

$$\frac{1}{2}(n+1)^2 + \frac{1}{2}(n+1) + 1 \leq |BMA(w_{opt})| \leq 2^n$$

### 2.4.2 Sukzessives Anhängen des ersten Zeichens

Bei diesem Verfahren wird das beste Suchwort  $|w'| = n - 1$  erweitert durch  $w = \max \{xw' | x \in A_i, i \in \{2, 3\}\}$ . Das Verfahren kann definiert werden durch eine rekursiv definierte Funktion  $V_e : \mathbb{N} \rightarrow \Sigma^n$ .

$$V_e(n) = \begin{cases} \max \{x \circ V_l(n-1), x \in A_i\} & n > 1 \\ a & n = 1 \end{cases}$$

Bei diesem Verfahren kann ein ähnlicher Induktionsbeweis durchgeführt werden, wie bei dem vorherigen Verfahren, allerdings sind hier die Boyer-Moore-Automaten der Suchwörter der beiden Fälle  $x = a$  und  $x \neq a$  gleich groß, so dass hier nur noch über die Bevorzugung des ersten Zeichen des Alphabets ( $a$ ) argumentiert werden könnte. Wenn zufällig eine der beiden Möglichkeiten gewählt wird, so ist dies kein Unterschied mehr zum absolut zufälligen Ziehen eines Suchwortes aus dem Raum der möglichen Suchwörter. Aus diesem Verfahren können also keine Schlüsse gezogen werden.

### 2.4.3 Sukzessives Anhängen an einer beliebigen Stelle im Wort

Dieses Verfahren hat den höchsten Freiheitsgrad der hier vorgestellten gierigen Verfahren. Hier wird das neue Zeichen an einer beliebigen Stelle eingefügt und das entstehende Suchwort bewertet. Wenn alle Möglichkeiten bewertet wurden, wird das Suchwort in den nächsten Schritt übernommen das die beste Bewertung hatte. Insbesondere wird auch an erster und letzter Stelle das neue Zeichen angefügt und das entstehende Suchwort bewertet, so dass alle Lösungen die die vorherigen Verfahren finden können auch gefunden werden. Formal kann dieser Algorithmus auch durch eine rekursive Funktion  $V_b : \mathbb{N} \rightarrow \Sigma^n$  beschrieben werden:

$$V_b(n) = \begin{cases} \max \{V_l(n-1) \circ x, x \in A_i\} & n > 1 \\ a & n = 1 \end{cases}$$

Für die Ergebnisse dieses Verfahrens konnte keine geschlossene Formel gefunden werden. Es wurden Testläufe bis zur Suchwortlänge 49 durchgeführt. Die genauen Ergebnisse sind in der Tabelle 2.2 und der graphische Verlauf in Abbildung 2.16 zu sehen.

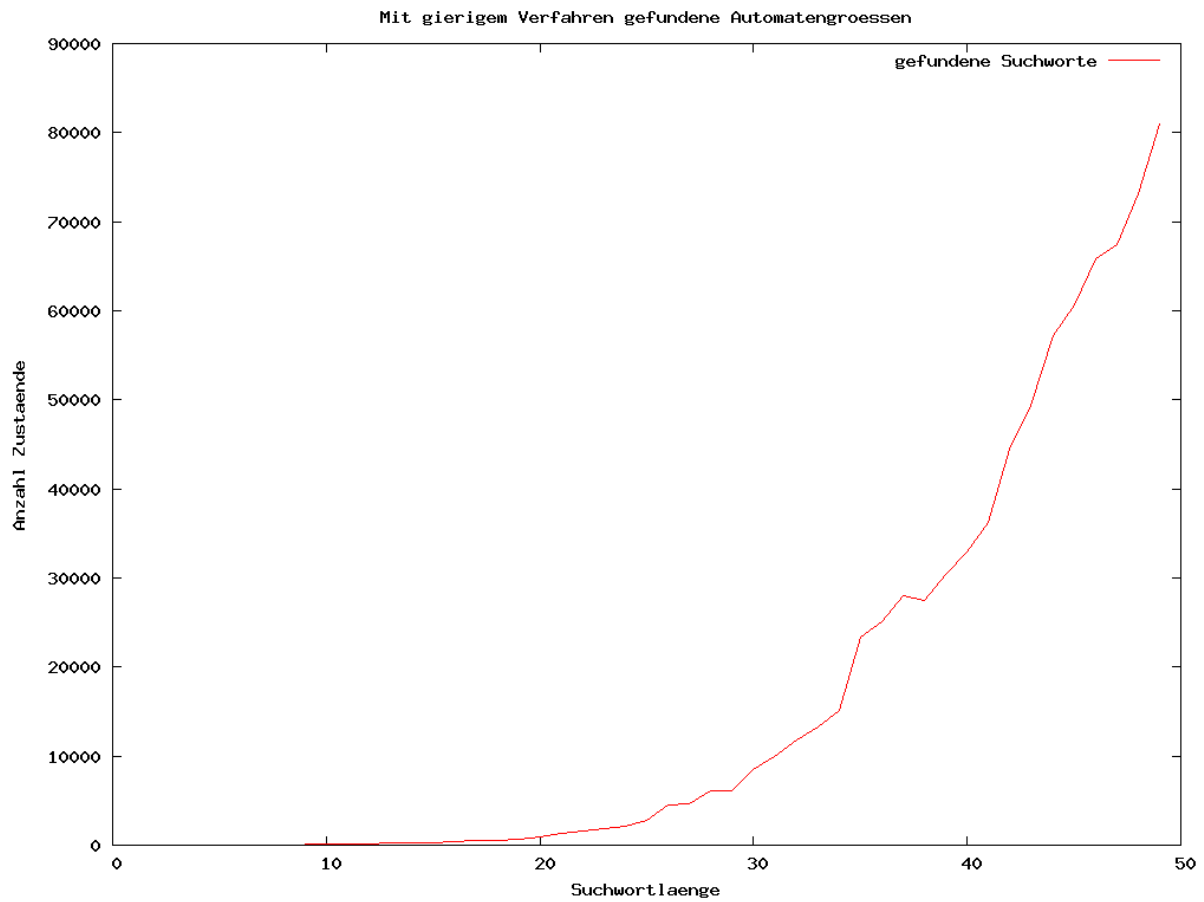


Abbildung 2.16: Größe der Boyer-Moore-Automaten für Suchwörter, die durch sukzessives Anhängen an einer beliebigen Stelle im Wort erzeugt wurden.

---

Länge	Größe des BMA für das gierig gefundene Suchwort
9	75
10	95
11	121
12	165
13	206
14	256
15	322
16	385
17	465
18	543
19	684
20	993
21	1281
22	1533
23	1826
24	2103
25	2843
26	4529
27	4604
28	6064
29	6079
30	8553
31	9910
32	11750
33	13234
34	15113
35	23365
36	25077
37	28011
38	27531
39	30387
40	32928
41	36287
42	44502
43	49321
44	57187
45	60602
46	65788
47	67482
48	73188
49	80957

---

Tabelle 2.2: Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet  $\Sigma = \{a, b\}$



# Untersuchung kurzer Wörter

---

Um den Aufbau des Boyer-Moore-Automaten besser verstehen zu können, wurden bevor der Genetische Algorithmus entworfen wurde, Untersuchungen auf Suchwörtern kurzer Länge (bis Länge 30) durchgeführt. Es war das Ziel zunächst den optimierten Algorithmus für den Aufbau des Boyer-Moore-Automaten zu testen und dann mit den erzielten Ergebnissen (die gefundenen Suchwörter mit den größten Boyer-Moore-Automaten) Rückschlüsse über den Aufbau dieser Suchwörter zu ziehen.

## 3.1 Prinzipielles Vorgehen

Es wurde eine vollständige Suche auf dem Suchraum ausgeführt. Es wurden alle möglichen Suchwörter einer bestimmten Länge erzeugt und dann deren Boyer-Moore-Automat berechnet. Bei der Erzeugung des Boyer-Moore-Automaten wurde der verbesserte Algorithmus von [BYCG94] benutzt.

### 3.1.1 Die Tabelle der Zustandszahlen

Der Algorithmus erzeugt alle möglichen Suchwörter für ein Alphabet, dessen Zeichenanzahl im Programm eingestellt werden kann. Diese Suchwörter werden nacheinander mit dem beschleunigten Verfahren nach [BYCG94] bewertet. Das Suchwort, das den Boyer-Moore-Automaten mit den meisten Zuständen erzeugt wird gespeichert.

Der Algorithmus muss alle  $|\Sigma|^n$  Suchwörter, die im Alphabet  $\Sigma$  möglich sind bewerten. Dadurch kann die Laufzeit des Algorithmus nicht besser als  $O(|\Sigma|^n)$  sein (die Laufzeit der einzelnen Bewertungen noch nicht einbezogen). Durch diese Einschränkung sind aus Laufzeitgründen nur Bewertungen bis zur Suchwortlänge 30 möglich. Das Suchwort, dessen Boyer-Moore-Automat die meisten Zustände hatte, wird zusammen mit der Anzahl der Zustände in eine Datei ausgeschrieben.

Die Zustandszahl jedes Boyer-Moore-Automaten, der erzeugt wurde wird in eine Tabelle eingetragen. Diese Tabelle ist nach den Zustandszahlen der Automaten in Zehnerschritten aufgeteilt. In der Tabelle wird abgespeichert wie häufig Boyer-Moore-Automaten einer bestimmten Zustandsanzahl während der Berechnung vorkamen. Ein Suchwort dessen Boyer-Moore-Automat 345 Zustände hatte erhöht in der Tabelle den Zähler für 340-349 Zustände. Nachdem alle Suchwörter einer bestimmten Länge erzeugt wurden sind in dieser Tabelle alle Boyer-Moore-Automaten mit ihrer Zustandszahl gezählt.

Mit dieser Tabelle wurden die Durchschnittszahlen in diesem Kapitel errechnet. Auch die Verteilungsdiagramme wurden aus diesen Tabellen generiert. Die Dateien mit den vollständigen Tabellen finden sich auf der CD in Anhang B.

## 3.2 Maximale Suchwörter für kurze Wortlängen

In diesem Abschnitt werden die Ergebnisse der vollständigen Suche dargestellt. Die Ergebnisse werden nach der Größe des Alphabets  $\Sigma$  getrennt präsentiert. Das Hauptaugenmerk liegt dabei auf der Analyse des Wachstumsverhaltens der größten Boyer-Moore-Automaten für ein 2-elementiges Alphabet. Anschließend wird das Wachstumsverhalten der 2-,3- und 4-elementigen Alphabete einander gegenübergestellt. Wir bezeichnen nun ein Suchwort  $w$  als „maximales“ Suchwort für eine bestimmte Länge  $l$ , wenn es kein Suchwort  $w'$  mit  $|w'| = l$  gibt mit  $|BMA(w')| > |BMA(w)|$ .

### 3.2.1 2-elementiges Alphabet

In der Tabelle 3.2.1 sind die maximalen Suchwörter eines zweielementigen Alphabets  $\Sigma = \{a, b\}$  bis zur Länge 30 aufgelistet, die durch die vollständige Suche berechnet wurden. Man kann sehen das die meisten Suchwörter einen bestimmten Aufbau haben. Dieser Aufbau wird in 3.3.2 näher beschrieben. Es gibt auch einige Fälle, in denen das maximale Suchwort für eine bestimmte Länge  $t$  aus dem maximalen Suchwort der Länge  $t - 1$  besteht, an das dann ein Zeichen angehängt wurde. Dies ist z. B. für  $t = 20$  der Fall. Hier wird aus *bbababbbbbbbbababa* durch hinzufügen von  $b$  dann *bbbababbbbbbbbababa*, dies ist das maximale Suchwort für die Länge 20.

An den Differenzen in Tabelle 3.2.1 erkennt man, das sich das Wachstum der Boyer-Moore-Automaten nicht gleichmäßig verhält. Auch durch das Bilden der Differenzen dieser Differenzen kann keine Regelmäßigkeit festgestellt werden (dies hätte auf eine polynomielle Funktion hingedeutet). Es gibt immer wieder Unregelmäßigkeiten, also einen Wechsel zwischen positiven und negativen Werten. Mit dem Verfahren der kleinsten Fehlerquadrate wurden sowohl eine polynomielle sowie eine exponentielle Näherungsfunktion berechnet. Als polynomielle Näherung für die Messwerte wurde die Funktion

$$f(x) = 0.63049x^4 - 36.352x^3 + 772.64x^2 - 6896.3x + 21369$$

gefunden. Als exponentielle Näherung wurde die Funktion

$$e(x) = 5.22666 \cdot 2.56333^{0.31862x} + 264.42463$$

gefunden. Wie man an dem Diagramm 3.2.1 sieht nähern beide Funktionen die Messwerte relativ gut an, liefern allerdings bei kurzen Suchwörtern erheblich zu große Werte zurück. Im Abschnitt 6 werden längere Suchwörter mit einem Genetischen Algorithmus untersucht. In diesem Abschnitt werden die beiden Funktionen benutzt, um zu entscheiden, ob sich das Wachstum der Boyer-Moore-Automaten exponentiell oder polynomiell zu der Länge der Suchwörter verhält.

### 3.2.2 3-elementiges Alphabet

In der Tabelle 3.2.2 sind die maximalen Suchwörter eines dreielementigen Alphabets  $\Sigma = \{a, b, c\}$  bis zur Länge 19 aufgelistet, die durch die vollständige Suche berechnet wurden. Die Anzahl der Zustände ist etwas höher als für das 2-elementige Alphabet. Auffallend ist, dass das dritte Zeichen des Alphabets  $c$  immer nur einmal in diesen Suchwörtern vorkommt, und dann immer als letztes Zeichen des Suchworts. An dieser Stelle sorgt es dafür, dass es für alle Zustände potentiell die Verschiebung um 1 geben kann. Diese Verschiebung ist genau dann möglich, wenn alle anderen aufgedeckten Zeichen bei dieser Verschiebung auch noch gültig sind.

### 3.2.3 4-elementiges Alphabet

In der Tabelle 3.2.3 sind die maximalen Suchwörter eines vierelementigen Alphabets  $\Sigma = \{a, b, c, d\}$  bis zur Länge 16 aufgelistet, die durch die vollständige Suche berechnet wurden. Die Anzahl der Zustände ist nochmal etwas höher als für das 3-elementige Alphabet. Wie man an den maximalen Suchwörtern in der Tabelle sehen kann, werden wieder hauptsächlich zwei Buchstaben des Alphabets benutzt, um das Suchwort aufzubauen. Die restlichen Zeichen kommen entweder nicht oder selten im Suchwort vor.

Das maximale Suchwort der Länge 4 besteht nur aus den Buchstaben  $a$  und  $b$ . Das maximale Suchwort der Länge 15 ist das Gleiche wie für das 3-elementige Alphabet und benutzt das 3. Zeichen auch nur, um die Verschiebung um 1 bei allen Zuständen potentiell möglich zu machen.

## 3.3 Interessante Wortarten

Im folgenden sollen einige spezielle Klassen von Suchwörtern näher untersucht werden.

### 3.3.1 Einstufen-Palindrome

Die Einstufen-Palindrome sind Worte der Form:

$$ab^n ab^n a$$

Also Wörter, die mit einem  $a$  beginnen, gefolgt von einem Block mit  $b$ 's, dann einem  $a$ , wieder gefolgt von einem Block von  $b$ 's, der gleichlang ist wie der erste. Abschließend steht ein  $a$  am Ende des Wortes.

Wie man an den gemessenen Zustandszahlen der Boyer-Moore-Automaten für die Einstufen-Palindrome sehen kann, wächst der Boyer-Moore-Automat für diese Klasse an Worten sehr regelmäßig und relativ langsam, wenn man in Betracht zieht, dass jede Erhöhung von  $n$  um 1 eine Verlängerung des Wortes um 2 bewirkt.

Aus den Messwerten kann man leicht folgende Formel für das Wachstum des Boyer-Moore-Automaten in Abhängigkeit zu  $n$  herleiten. Diese Formel ist:

$$|BMA(ab^n ab^n a)| = 2x^2 + 7x + 7$$

### 3.3.2 Updated-Palindrome

Wie in [BBYDS96] beschrieben gibt es eine Klasse von Suchwörtern, die große Automaten für ein 2-elementiges Alphabet erzeugen. Die Menge dieser Suchwörter ist:

$$(b^+a)^+ \cup a(b^+a)^+$$

Also Blöcke von  $b$ 's getrennt von einzelnen  $a$ 's. Im Vergleich zu den Einstufen-Palindromen muss die Anzahl der  $b$ 's in diesen Blöcken nicht gleich sein und es kann mehr als 2  $b$ -Blöcke geben. Wie in 3.2.1 zu sehen ist sind sehr viele maximale Suchwörter in dieser Klasse enthalten. Wir nennen Wörter dieser Klasse `Updated Palindrome`.

Auch für die maximalen Suchwörter für das 3-elementige Alphabet scheint es eine ähnliche Klasse von Suchwörtern zu geben. Die maximalen Wörter bestehen aus Blöcken von  $b$ 's getrennt mit  $a$ 's. Das letzte Zeichen des Suchworts ist dann das  $c$ . Auch für längere Suchwörter scheint dies, wie in Abschnitt 6.5.2 zu sehen, ein guter Aufbau des Suchwortes zu sein.

## 3.4 Das Wachstum der durchschnittlichen BMA-Größe

Während der Berechnung des Suchwortes, das den größten Boyer-Moore-Automaten erzeugt, werden alle möglichen Suchwörter einer bestimmten Länge erzeugt und die Boyer-Moore-Automaten für diese Suchwörter berechnet. Diese Informationen über die Zustandszahlen der Boyer-Moore-Automaten für alle Suchwörter wird in der Tabelle der Zustandszahlen abgespeichert (siehe 3.1.1). Mit diesen Daten können nun auch Analysen über das Wachstum eines durchschnittlichen Suchwortes  $w$  gemacht werden.

### 3.4.1 Aufbau des Suchraums

#### Häufigkeit von Boyer-Moore-Automaten bestimmter Größe

Beispielhaft sind in Abbildung 3.2 die Häufigkeiten der Boyer-Moore-Automaten für die Suchwortlänge  $|w| = 30$  in Abhängigkeit zu ihrer Größe aufgetragen. Man erkennt, dass es eindeutig ein Maximum der Häufigkeiten gibt. Dieses Maximum nennen wir das dominierende Maximum. Die Häufigkeiten von kleineren oder größeren Boyer-Moore-Automaten nimmt, um so mehr man sich von dieser dominierenden Maximum entfernt ab. Dabei fällt auf, dass der Abfall der Häufigkeiten zu größeren BMA's geringer ausfällt als zu kleineren BMA's. Die Häufigkeitsverteilung der BMA's ist nicht symmetrisch um das dominierende Maximum der Häufigkeiten, sondern zieht sich in Richtung der größeren Boyer-Moore-Automaten.

Je mehr man sich vom dominierenden Maximum der Häufigkeiten der BMA's entfernt, desto kleiner werden die Häufigkeiten in den gemessenen Bereichen der Zustandszahlen, nähern sich aber nur sehr langsam der 0 an. Erst wenn die Größe des maximalen Boyer-Moore-Automaten für diesen Suchraum erreicht ist werden keine größeren BMA's in den Bereichen mehr gefunden.

Der Effekt des Verzerrens scheint mit wachsender Suchwortlänge gleich zu bleiben. Die Kurve der Häufigkeiten für die Suchwortlänge  $|w| = 20$  in Abbildung 3.3 zeigt ein ähnlichen Verlauf. Hier liegt das dominierende Maximum bei etwa 500 und bei ca 1000 nähert sich die Kurve der Häufigkeiten der 0 an. Bei der Suchwortlänge  $|w| = 30$  waren die entsprechenden Werte 2000 und 4000. Diese Werte sind also im gleichen Verhältnis zueinander geblieben. Allerdings ist der Boyer-Moore-Automat des maximalen Suchwortes für  $|w| = 20$  mit 2323 Zuständen nur ca. 4.6

mal größer als das dominierende Maximum, während der entsprechende BMA bei  $|w| = 30$  mit 40331 Zuständen um ca. 20 mal größer ist als das dominierende Maximum.

Die Größe des BMA vom maximalen Suchwort scheint sich mit wachsender Länge des Suchwortes immer weiter vom dominierenden Maximum zu entfernen.

### Veränderung der Zustandszahl bei der Veränderung eines Zeichens

Aus der Häufigkeitsverteilung der Boyer-Moore-Automaten aus dem vorherigen Abschnitt wird nicht ersichtlich, wo im Suchraum die Boyer-Moore-Automaten bestimmter Größe angeordnet sind. Der Suchraum ist  $n$ -dimensional, deshalb ist eine Darstellung der Verteilungen der Boyer-Moore-Automaten im Suchraum nicht zweidimensional möglich.

Da eine Darstellung der Fitnesslandschaft nicht möglich ist, wurde eine Untersuchung durchgeführt, wie stark sich die Zustandszahl eines Boyer-Moore-Automaten ändert, wenn man ein Zeichen seines Suchwortes ändert. Die Untersuchungen wurden für ein 2-elementiges Alphabet durchgeführt, hier kann das Zeichen das geändert werden soll nur in das andere Zeichen des Alphabets umgewandelt werden.

Es wurden zwei Werte für die Suchwortlängen bis  $|w| = 20$  berechnet:

- maximale Differenz
- durchschnittliche Differenz

Für alle Suchwörter  $w$  der gewünschten Länge wurden für alle möglichen Änderungen eines Zeichens das entstehende Suchwort  $w'$  berechnet. Dann wurde die Differenz der Zustandszahlen der beiden Boyer-Moore-Automaten  $BMA(w)$  und  $BMA(w')$  berechnet.

Die maximale Differenz ist also die größte Änderung der Größe der Boyer-Moore-Automaten von zwei Suchwörtern  $w, w'$  die sich nur in einem Zeichen unterscheiden. Der Durchschnitt ist entsprechend die durchschnittliche Änderung der BMA-Größe über alle möglichen Paare von Suchwörtern  $w, w'$ , die eine bestimmte Länge haben.

In Abbildung 3.4 sieht man das Wachstumsverhalten der maximalen und der durchschnittlichen Differenz bei steigender Suchwortlänge, bis zur Suchwortlänge  $|w| = 15$ .

### 3.4.2 Wachstumsverhalten

In Tabelle 3.4.2 sind die durchschnittlichen Größen der Boyer-Moore-Automaten für ein zweielementiges Alphabet über alle Suchwörter einer bestimmten Länge aufgelistet. Das Wachstumsverhalten dieser durchschnittlichen Größe ist sehr viel langsamer als das Wachstum des maximalen Boyer-Moore-Automaten.

Die Differenzen der Messwerte deuten darauf hin, dass das Wachstumsverhalten mit einem Polynom 3. Grades nachgebildet werden kann. Durch die Methode der kleinsten Fehlerquadrate wurde die Funktion

$$f(x) = 0.11079n^3 - 1.59819n^2 + 14.85133n - 24.51809$$

ermittelt. Diese Funktion liegt bei den Wortlängen kleiner 4 nicht nahe an den richtigen Werten. Bei größeren Wortlängen ist die Näherung der Funktion jedoch nicht weit von den gemessenen

Werten entfernt. Das Wachstum der durchschnittlichen Größe der Boyer-Moore-Automaten scheint also in  $O(n^3)$  zu liegen.

Das Wachstumsverhalten der durchschnittlichen Größe der Boyer-Moore-Automaten für 3- und 4-elementige Alphabete wird in Abbildung 3.5 dem Wachstumsverhalten des 2-elementigen Alphabets gegenübergestellt. Man sieht, dass die Größe der durchschnittlichen Boyer-Moore-Automaten der 3- und 4-elementigen Alphabete langsamer wächst als das des 2-elementigen Alphabets. Dieses Verhalten ist damit erklärbar, dass sich Teilstücke eines Suchwortes aus einem 2-elementigen Alphabet mit höherer Wahrscheinlichkeit ähneln, da es hier nicht so viele Variationen wie bei mehrelementigen Alphabeten gibt. Durch diese Selbstähnlichkeiten werden kleinere Verschiebungen möglich und der Boyer-Moore-Automat kann mehr Zustände enthalten (siehe notwendige Bedingung in 2.2.3).

Das Wachstum der Suchwörter von 3- und 4-elementigen Alphabeten liegt demnach auch in  $O(n^3)$ .

Länge	maximales Suchwort	Zustandszahl	$\delta$ gerade Wortl.	$\delta$ ungerade Wortl.
1	b	2		
2	bb	4		
3	bbb	7		5
4	babb	13	9	
5	babbb	21		14
6	babbbb	31	18	
7	babbbbb	43		22
8	bababbbb	58	27	
9	baaabaabb	84		41
10	bbababbbaa	107	49	
11	baababaaabb	156		72
12	baabaaaababb	197	90	
13	baababaaababb	282		126
14	babbbbbabbbaba	362	165	
15	baababaaabaaabb	518		236
16	baaabababaaababb	635	273	
17	baaaaababaaabab	968		450
18	babbbbbabbbabbbaba	1230	595	
19	bbababbbbbbbbbbbaba	1949		981
20	bbbababbbbbbbbbbbaba	2323	1093	
21	abbbabbbbbabbbbbabbba	3461		1512
22	babbbabbbbbabbbbbabbba	4052	1729	
23	bbabbbbbabbbbbabbbbbba	8440		4979
24	bbbabbbbbabbbbbabbbbbba	9826	5774	
25	ababbbbbabbbbbbbbbbbaba	10505		2065
26	bababbbbbabbbbbbbbbbbaba	12270	2444	
27	abbabbbbbabbbbbbbabbba	16259		5754
28	babbbabbbbbabbbbbabbba	18338	6068	
29	ababbbbbabbbbbbbbbbbaba	35104		
30	bababbbbbabbbbbbbbbbbaba	40331		

Tabelle 3.1: Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet  $\Sigma = \{a, b\}$

Länge der Suchwörter	Differenz gerade Wortlänge[dg]	Differenzen dg[dg']	Differenzen dg'[dg'']
1			
2			
3			
4	9		
5		9	
6	18		0
7		9	
8	27		13
9		22	
10	49		19
11		41	
12	90		34
13		75	
14	165		33
15		108	
16	273		214
17		322	
18	595		176
19		498	
20	1093		138
21		636	
22	1729		3409
23		4045	
24	5774		-7375
25		-3330	
26	2444		6954
27		3624	
28	6068		

Tabelle 3.2: Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet  $\Sigma = \{a, b\}$

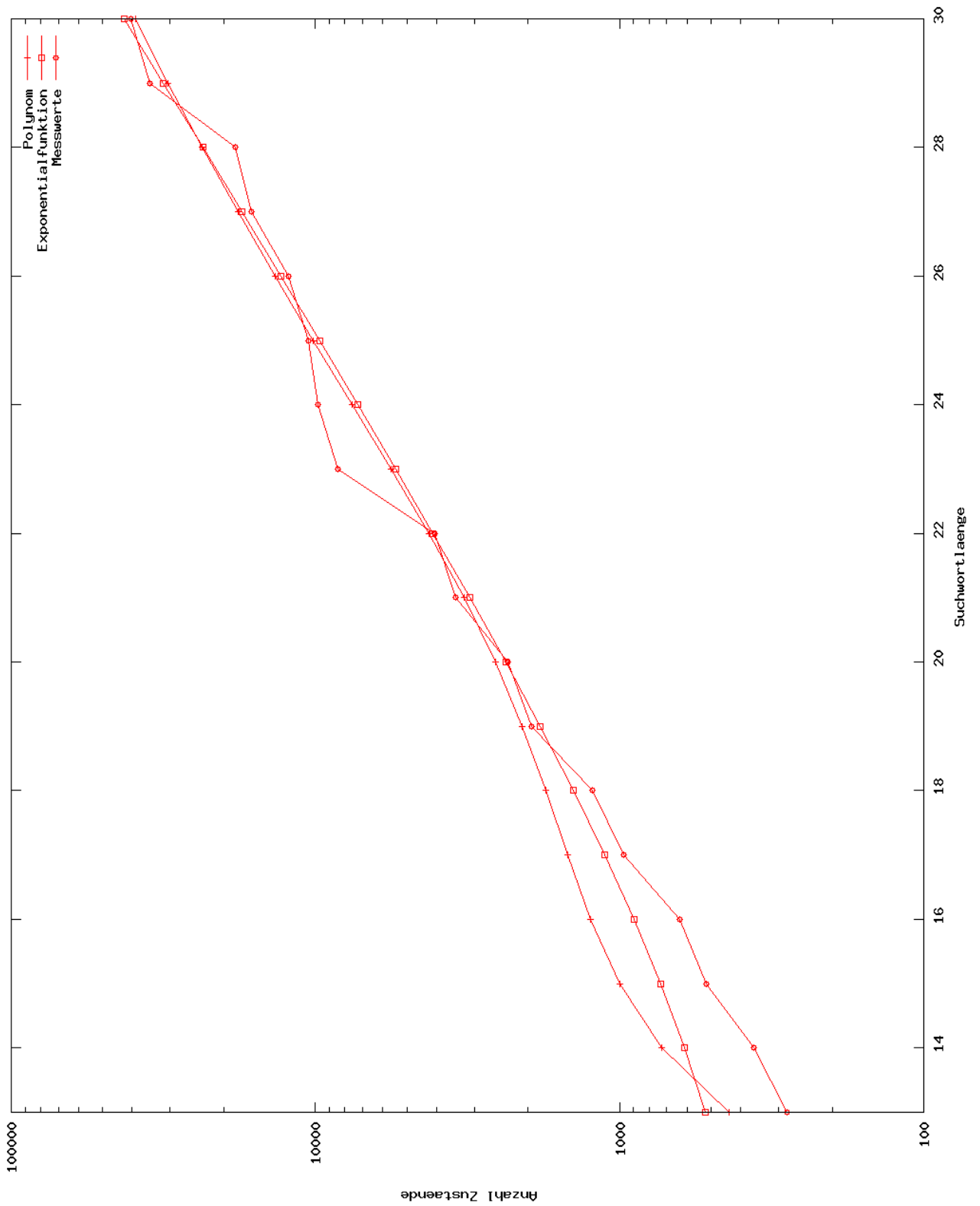


Abbildung 3.1: Vergleich zwischen den durch vollständige Suche gefundenen größten Boyer-Moore-Automaten und den Funktionswerten der Näherungsfunktionen

Länge	maximales Suchwort	Anzahl der Zustände
1	a	2
2	ba	4
3	abc	7
4	babb	13
5	babbc	22
6	babbbc	34
7	bbabbbc	51
8	bbababbc	70
9	bbbabbbc	94
10	bbababbbc	132
11	babbbaabbc	187
12	baabbbbabbc	253
13	baaababaaaac	371
14	babbbababbbc	536
15	bbabbbbababbbc	712
16	bababbbabbbbbb	847
17	ababbbbbbbbbbbabc	1548
18	bababbbbbbbbbbbabc	1946
19	bbabbbbbabbbabbbc	2423

Tabelle 3.3: Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet  $\Sigma = \{a, b, c\}$

Länge	maximales Suchwort	Anzahl der Zustände
1	a	2
2	ba	4
3	abc	7
4	babb	13
5	babbc	22
6	babbbc	34
7	bbabbbc	51
8	bbababbc	70
9	bbabcbbbd	100
10	babbcabbd	138
11	babbbaabbc	187
12	babcbbbcbbd	261
13	abbbababbbbc	371
14	babbbababbbbc	536
15	bbabbbbababbbc	712
16	bcabbbbbbbbabd	852

Tabelle 3.4: Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet  $\Sigma = \{a, b, c, d\}$

n	Anzahl der Zustände
1	16
2	29
3	46
4	67
5	92
6	121
7	154
8	191
9	232
10	277
11	326
12	379
13	436
14	497
15	562
16	631

Tabelle 3.5: Wachstum des Boyer-Moore-Automaten für Einstufen-Palindrome  $ab^n ab^n a$  mit wachsendem  $n$

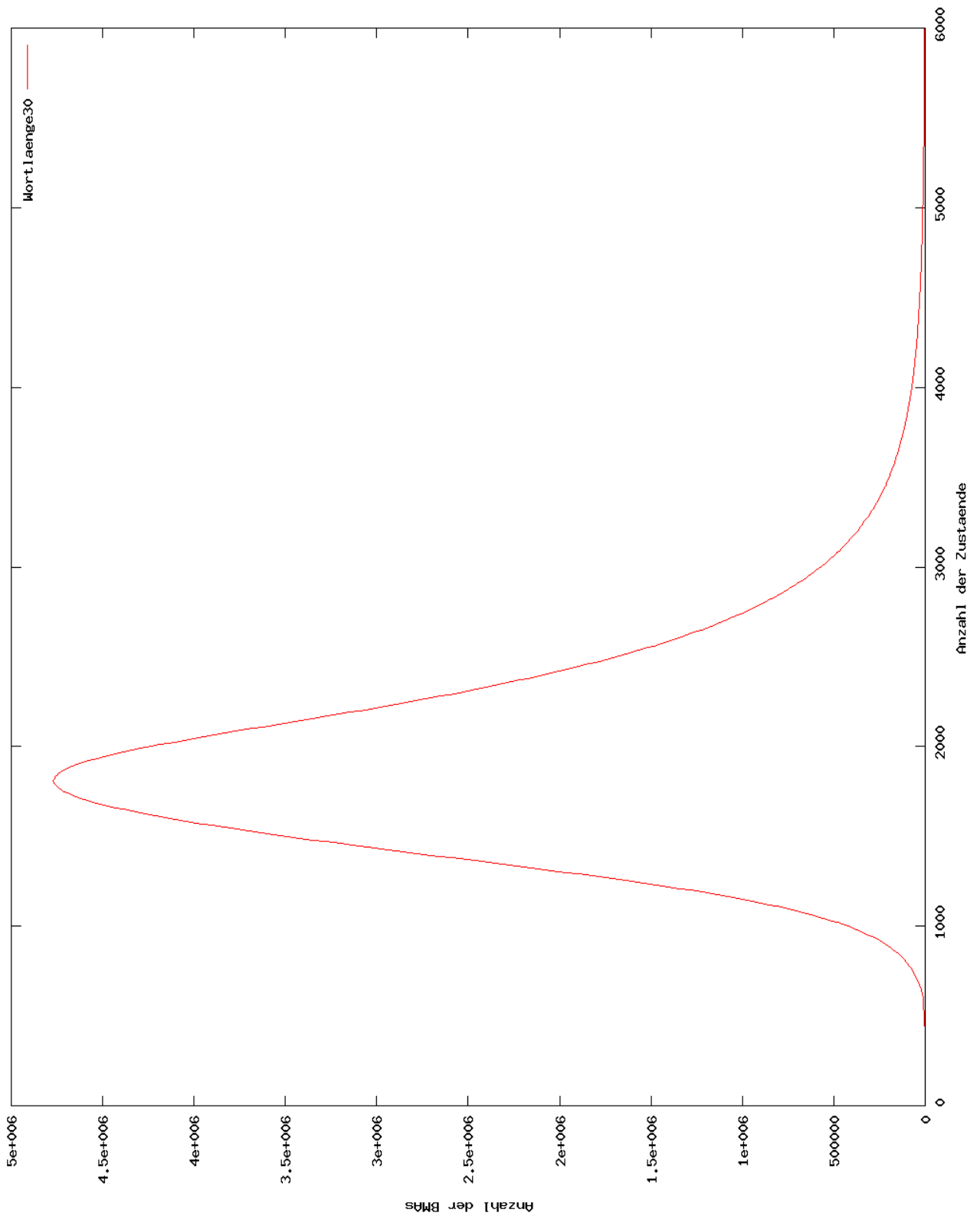


Abbildung 3.2: Häufigkeiten von Boyer-Moore-Automaten einer bestimmten Größe für Suchwörter der Länge 30.

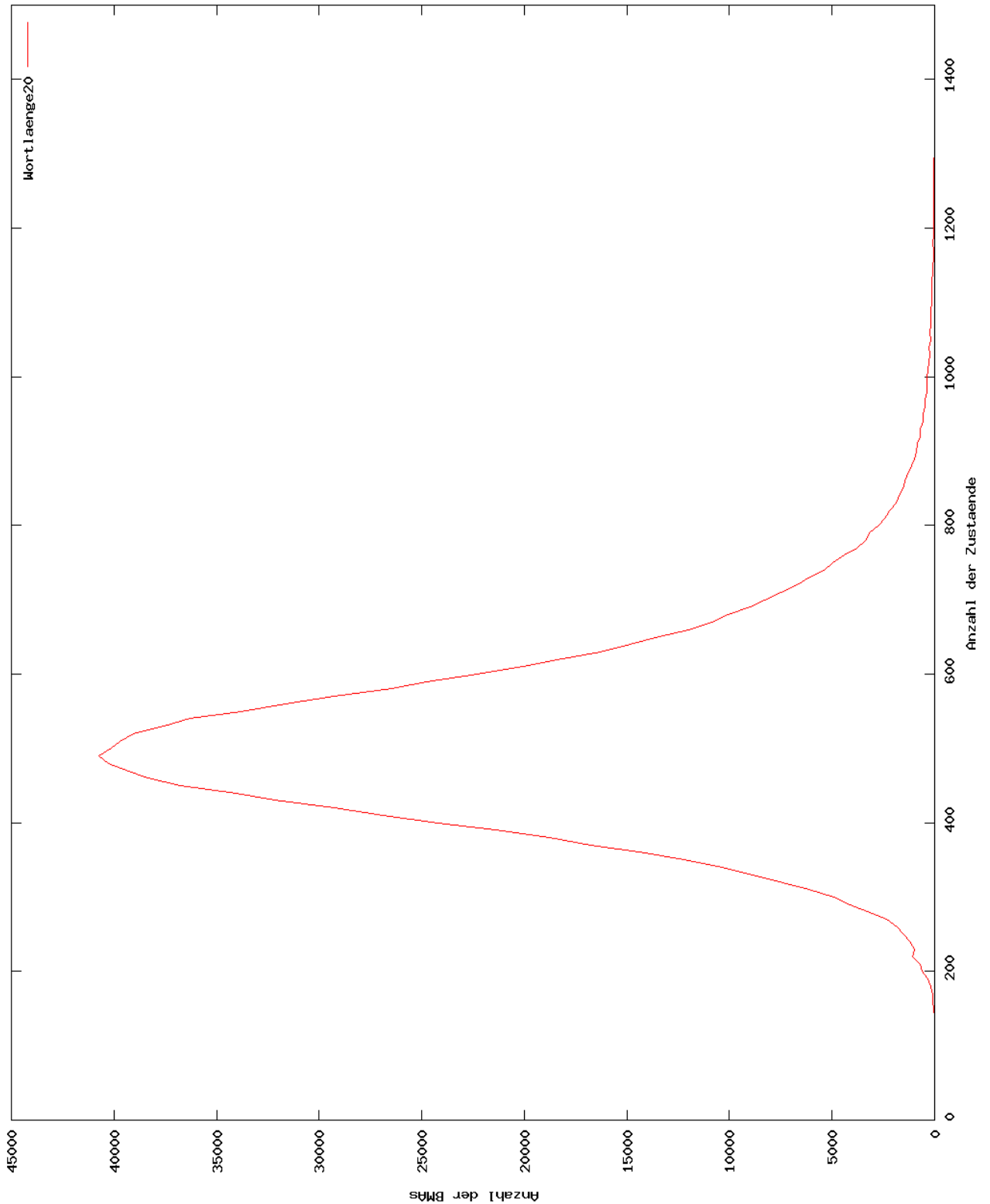


Abbildung 3.3: Häufigkeiten von Boyer-Moore-Automaten einer bestimmten Größe für Suchwörter der Länge 20.

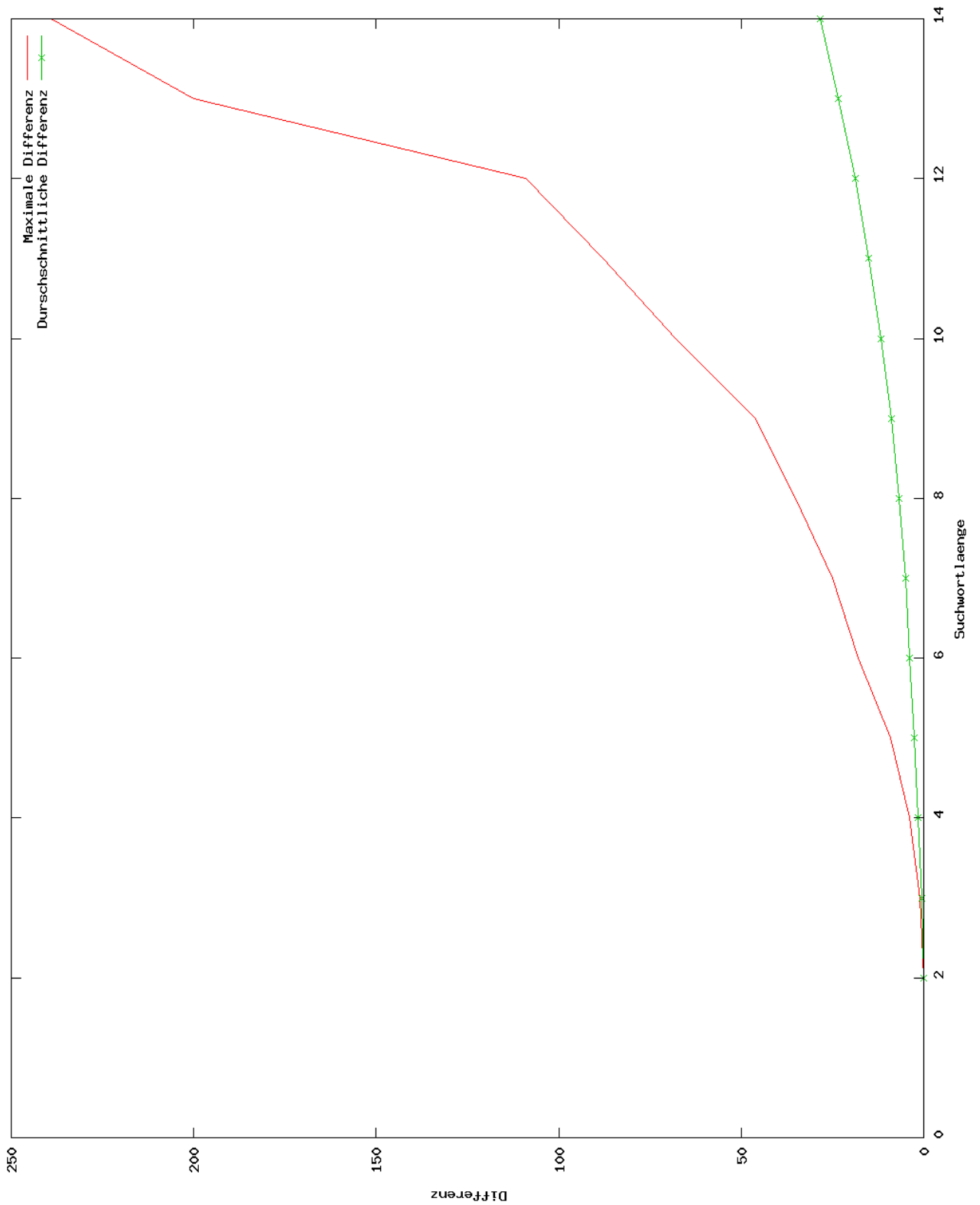


Abbildung 3.4: Maximale und durchschnittliche Änderung der Größe der BMA's für Suchwörter verschiedener Längen.

Suchwortlänge	Anzahl der Zustände
1	4.0
2	4.0
3	4.0
4	12.75
5	15.25
6	23.6875
7	31.8125
8	43.375
9	57.3203125
10	74.21484375
11	94.966796875
12	118.4267578125
13	147.80126953125
14	181.42.67578125
15	220.64794921875
16	265.95037841796875
17	318.13543701171875
18	377.19488525390625
19	445.50943756103516
20	521.6186180114746
21	608.3913173675537
22	705.4145612716675
23	814.5398726463318
24	934.7582378387451
25	1071.0711696147919
26	1219.8056166768074
27	1385.8713295459747
28	1568.2016017436981
29	1769.9538305178285
30	1989.4040850512683

Tabelle 3.6: Wachstum der durchschnittlichen Zustandszahl aus allen Boyer-Moore-Automaten für ein Suchwort  $w$  bestimmter Länge.

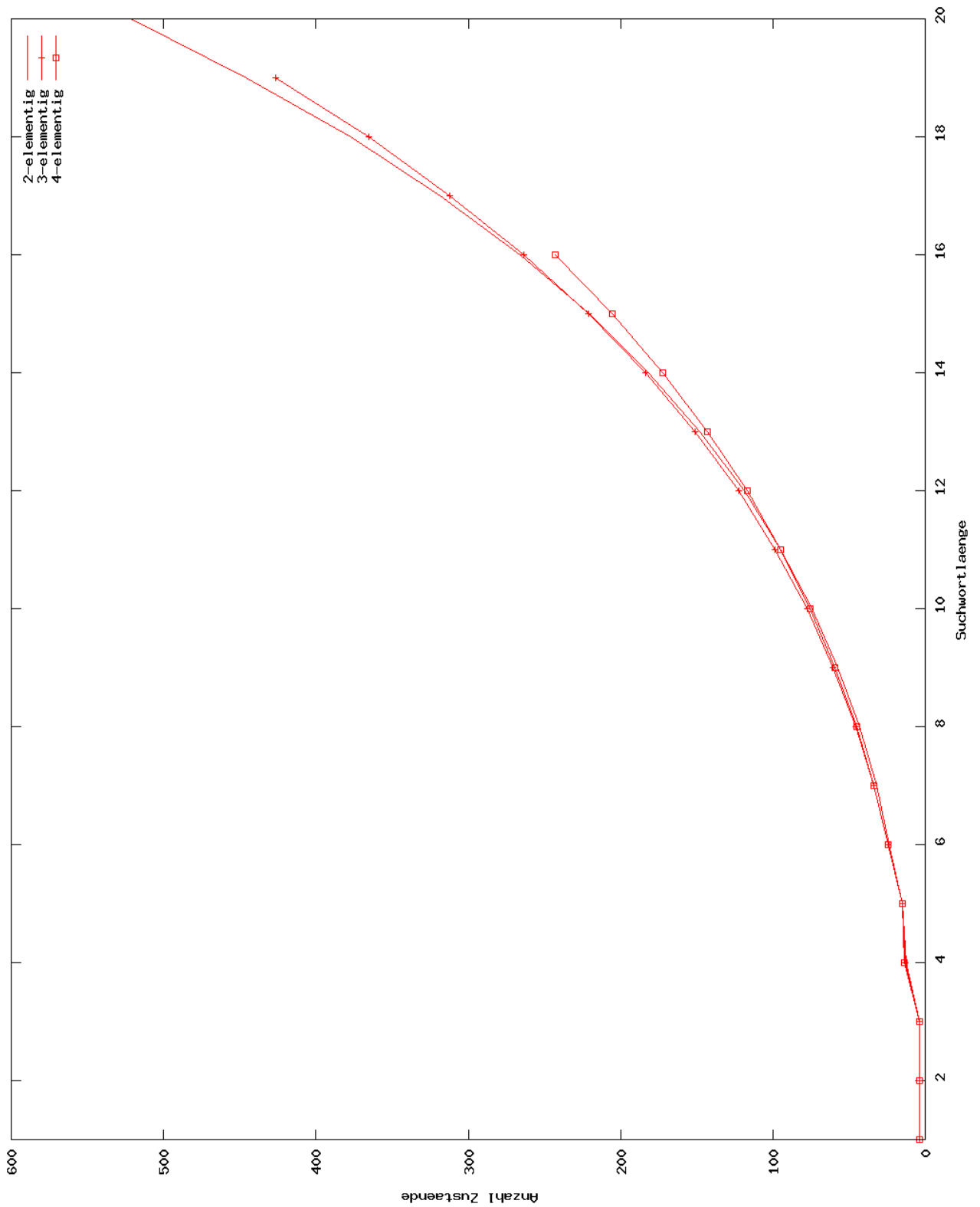


Abbildung 3.5: Das Wachstumsverhalten der durchschnittlichen BMA-Größe für verschiedene große Alphabete.

# Evolutionäre Algorithmen

---

In diesem Kapitel wird der grundlegende Aufbau eines Evolutionären Algorithmus beschrieben. Zunächst werden die Vorbilder aus der Natur betrachtet um die Verwendung von Evolutionären Algorithmen zu motivieren. Anschließend werden die Begriffe der Evolution beschrieben. Danach werden die groben Unterschiede zwischen der Evolution in der Biologie und den Evolutionären Algorithmen beschrieben. Dann wird der Überbegriff des Evolutionären Algorithmus genauer behandelt, um danach den Genetischen Algorithmus genau gegenüber den anderen Evolutionären Algorithmen abgrenzen zu können. Dabei wird im besonderen auf den Evolutionskreis und auf die verschiedenen Komponenten des Evolutionskreises eingegangen. Dieses Kapitel stützt sich in weiten Teilen auf [Wei02a] sowie [Cla06] und in dem Abschnitt über die Vorbilder aus der Natur zusätzlich auf [LM03].

## 4.1 Vorbild aus der Natur

Die Natur hat es seit dem Bestehen der Erde geschafft eine Vielzahl von Lebewesen zu erschaffen. Dabei ist eine riesige Bandbreite an verschiedenen Konzepten und Strategien entstanden, die diese Lebewesen verwenden, um ihren Fortbestand zu sichern. Nicht nur die Verhaltensweisen der Lebewesen und die Lebewesen selber sind sehr komplex, es haben sich auch sehr große Kreisläufe und Abhängigkeiten eingestellt, die nicht zu durchschauen sind, und von der die einzelnen Lebewesen nicht unmittelbar Vorteile haben. Während der Entwicklungsgeschichte der Erde und des Lebens auf ihr gab es häufig radikale Änderungen der Umwelt, auf die sich das Leben auf der Erde einstellen musste. So war der Sauerstoff, der für viele Lebewesen heute auf der Erde unmittelbar zum Überleben nötig ist, für die ersten Lebewesen auf der Erde hochgiftig. Das Leben auf der Erde passt sich den Umweltbedingungen an und verändert diese sogar. Ohne Leben würde es z. B. keinen freien Sauerstoff auf der Erde geben.

Nun stellt sich die Frage, welche Mechanismen diese Entwicklung ermöglichen, die wir Evolution nennen. Wenn diese in der Natur erkannt sind könnte man sie auch zur Lösung von Problemen in der Informatik einsetzen. Diesen Schritt versuchen die Evolutionären Algorithmen zu leisten.

## 4.2 Der Genetische Algorithmus

Die Menge der Evolutionären Algorithmen lässt sich in viele Teilgebiete aufgliedern. Jedes Teilgebiet behandelt einen bestimmten Aspekt der Evolutionären Algorithmen anders, verwendet also z. B. andere Operatoren oder kodiert die Individuen anders. Trotz dieser Unterschiede handelt es sich aber bei allen Algorithmen dieser Teilgebiete um Evolutionäre Algorithmen.

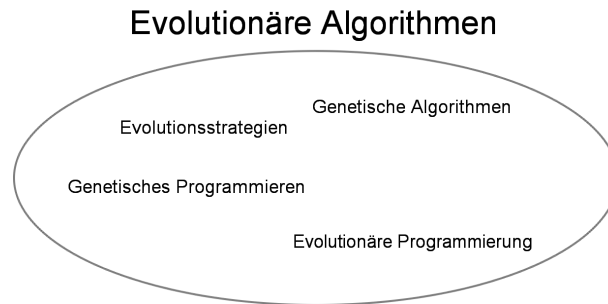


Abbildung 4.1: Unterdisziplinen der Evolutionären Algorithmen.

Einen Überblick über die verschiedenen Teilgebiete zeigt Abbildung 4.1. Historisch gesehen ist die Lösungsstrategie „Evolutionärer Algorithmus“ aus den verschiedenen Teilgebieten entstanden. Zuerst gab es die verschiedenen Teilgebiete der Evolutionären Algorithmen, die mit verschiedenen Strategien und verschiedenen Ansätzen arbeiteten und erst nach und nach setzte sich die Ansicht durch, dass diese verschiedenen Teilgebiete zu einem größeren und abstrakteren Lösungsansatz gehörten, den Evolutionären Algorithmen.

Die Beschreibungen und Definitionen sind im Weiteren für Evolutionäre Algorithmen allgemein gehalten. Da aber in dieser Arbeit ein Genetischer Algorithmus implementiert wird, wird der Fokus auf die Aspekte gelegt, die für Genetische Algorithmen relevant sind. Andere Bereiche der Evolutionären Algorithmen werden nur teilweise behandelt.

Um die Genetischen Algorithmen von den Evolutionären Algorithmen abzugrenzen werden nun einige Besonderheiten der Genetischen Algorithmen angesprochen.

Bei Genetischen Algorithmen werden die Individuen als künstliche „Chromosomen“ aufgefasst und bestehen aus einer Kette von  $n$  Bits. Dieser Strang von Bits lässt sich aufteilen in sogenannte Segmente. Jedes Segment korrespondiert zu einer Variablen im betrachteten Optimierungsproblem ([Nis97d]). Diese Aufteilung des Genoms eines Individuums in eine sequentielle Darstellung von Bits ist eine Besonderheit der Genetischen Algorithmen. Mittlerweile gibt es neuere Varianten der Genetischen Algorithmen die den Genotyp der Individuen nicht mehr binär kodieren, sondern „natürlicher“ aus dem Phänotyp abgeleitet abspeichern.

Ein weiteres Kennzeichen eines Genetischen Algorithmus sind die Rekombinationsoperatoren, die die Bits für die Kindindividuen komponentenweise aus den Elternindividuen übernehmen. Dieses Verhalten ist aus den crossing-over Effekten in der Biologie bekannt, deshalb werden diese Rekombinationen auch Crossover-Operationen genannt.

Ein zusätzliches Merkmal eines Genetischen Algorithmus ist, dass während des Evolutionszyklus genau  $\lambda = \mu$  Kinder erzeugt werden. Diese werden direkt als die Population ( $pop_{t+1}$ ) für den nächsten Zyklus verwendet.

Außerdem wird im Evolutionszyklus nur eine Selektion durchgeführt, die man je nach Anschauung als Umweltselektion oder als Elternselektion betrachten kann.

Diese Beschränkungen gegenüber dem Evolutionären Algorithmus werden aber bei den neueren Varianten des Genetischen Algorithmus aufgeweicht, so dass „hybride Algorithmen“ entstehen, die nicht mehr klar einem der Teilgebiete der Evolutionären Algorithmen zugeweiht werden können.

### 4.3 Begriffsdefinitionen

In diesem Kapitel werden die Begriffe, die später in diesem und in den folgenden Kapiteln verwendet werden näher beschrieben. In dieser Arbeit ist es wichtig die verschiedenen Begriffe klar zu definieren. Um so mehr, da die Begriffe in der biologischen Evolution und in dem Gebiet der Evolutionären Algorithmen unterschiedlich definiert sind. In diesem Abschnitt werden die Definitionen, wie sie bei Evolutionären Algorithmen gebräuchlich sind beschrieben. Zur Abgrenzung zu den biologischen Definitionen werden die Unterschiede bei den Begriffen angesprochen und verdeutlicht. Viele Begriffe wurden aus [Wei02a] übernommen. Die anderen Quellen werden bei ihrer Verwendung angegeben.

Evolutionäre Algorithmen sollen meistens ein Optimierungsproblem lösen. Nach [Wei02a] ist ein Optimierungsproblem folgendermaßen definiert.

**Definition 5** *Ein Optimierungsproblem ist ein 3-Tupel*

$$(\Omega, f, \prec)$$

mit

$\Omega$  Der Suchraum.

$f: \Omega \rightarrow \mathbb{R}$  ist die Bewertungsfunktion, die jeder Lösung des Suchraums  $\Omega$  einen Güterwert zuweist.

$\prec \in \{<, >\}$  eine Vergleichsrelation auf dem Suchraum  $\Omega$ .

Die Menge der globalen Optima  $X \subseteq \Omega$  ist

$$X = \{x \in \Omega \mid \forall_{x' \in \Omega} f(x) \succeq f(x')\}$$

Nun kann man folgende Operationen auf einer beliebigen Menge von Güterwerten  $X$  anwenden

**bester X** =  $x : x \in X \wedge \forall_{y \in X} x \succeq y$

**schlechtester X** =  $x : x \in X \wedge \forall_{y \in X} x \preceq y$

Der Evolutionäre Algorithmus soll nun für ein definiertes Optimierungsproblem „bester X“ finden.

Im Weiteren werden nun die Begriffe kurz beschrieben. Wichtige Begriffe werden anschließend noch eingehender besprochen. Für genauere Informationen über die restlichen Begriffe sollen die Literaturhinweise dienen:

**Initialisierung** Im Gebiet der Evolutionären Algorithmen versteht man unter der Initialisierung die Erzeugung der ersten Population, auf die die Mechanismen des Evolutionszyklus angewendet werden ( $pop_0$ ). Für den weiteren Ablauf des Evolutionären Algorithmus kann es von entscheidender Bedeutung sein, dass die Initialisierungs-Population „gut“ war (siehe 4.5).

**Gen** In der Biologie sind mehrere Genbegriffe bekannt ([Hen98d]). Im Allgemeinen wird unter dem Begriff ein abgegrenzter Teil des Genotyps verstanden, dessen Änderung eine begrenzte Auswirkung auf den Phänotyp des Individuums haben kann. Bei den Evolutionären Algorithmen ist ein Gen die kleinste Einheit des Individuums, das Informationen für die Vererbung trägt und durch die Mutation einzeln geändert werden kann.

**Allel** In der Biologie bezeichnet man ein Allel als eine konkret für ein Gen vorliegende „Besetzung“ des Gens. Die Allele gehen durch Mutation auseinander hervor ([Sey98b]). Bei Evolutionären Algorithmen übernimmt man diese Definition. Ein Allel ist eine konkrete Besetzung des Gens mit Werten in einem Individuum. Allerdings ist ein Allel hier normalerweise keine RNA-Folge, sondern ein Wert der Datenstruktur, in der das Gen kodiert ist.

**Güte/Fitness** In der Biologie ist die Güte bzw. die Fitness eines Individuums darüber definiert, wieviele Nachkommen es erzeugt (siehe [SCS05a]). Bei dieser Definition ist die Fitness für einen Genotyp definiert. Bei den Evolutionären Algorithmen wird die Güte/Fitness eines Individuums als der Wert, den die Bewertungsfunktion  $f$  für den Phänotyp des Individuums zurückliefert definiert. Mit der Funktion  $F : \mathcal{G} \rightarrow \mathbb{R}$  kann die Fitness auch direkt vom Genotyp aus berechnet werden.

**Diversität** In der Biologie wird dieser Begriff im Gebiet der Makroevolution verwendet, um das Entstehen neuer Arten zu erklären (siehe [SCS05c]). Auch bei Evolutionären Algorithmen ist dieser Begriff mehrfach belegt. Siehe hierzu Populations-Diversität und Gen-Diversität.

**Populations-Diversität** Unter diesem Begriff wird bei den Evolutionären Algorithmen die Verteilung der Individuen einer Population in einem bestimmten Raum verstanden. Man nimmt hier entweder die durch den Genotyp der Individuen definierte Verteilung im Raum aller möglichen Genotypen, oder die durch den Phänotyp des Individuums definierte Verteilung der Individuen im Raum aller möglichen Phänotypen. Der Abstand zwischen den Individuen muss dann mit einer Abstandsfunktion gemessen werden. Hier kann die Nachbarschaftsfunktion (bzw. die n-fache-Nachbarschaft) angewendet werden.

**Gen-Diversität** Die Gen-Diversität  $h$  wird in der Biologie für ein Gen angegeben. Sie gibt an, wie hoch die Wahrscheinlichkeit ist, dass bei der zufälligen Auswahl von zwei Individuen aus der Population genau zwei ausgewählt werden, deren Allele für dieses Gen unterschiedlich sind. Wenn  $x_i$  die Anteile der unterschiedlichen Allele  $x_i$  der Population sind, so ist  $h = 1 - \sum_i x_i^2$  (siehe [SCS05c]). Bei den Evolutionären Algorithmen übernehmen wir diese Definition.

**Nachbarschaft** Die Nachbarschaft eines Genotyps eines Individuums ist für die Evolutionären Algorithmen über eine Nachbarschaftsrelation definiert. Diese Relation  $\mathcal{N} : \mathcal{G} \times \mathcal{G}$  enthält normalerweise genau dann zwei Individuen  $(a, b) \in \mathcal{N}$ , wenn  $b$  durch eine Mutation von  $a$  entstehen kann ( $m(a) = b$ ). Bei einer n-fachen-Nachbarschaftsrelation gilt entsprechend, dass  $b$  aus  $a$  durch n-faches mutieren entstehen kann.

**Genotyp** Der Genotyp stellt die Gesamtheit der erblichen Eigenschaften dar, die von einem Individuum im evolutionären Prozess vererbt werden können. Ein Evolutionärer Algorithmus vererbt nur den Genotyp von Individuen. Diese Definition ist auch in der Biologie gebräuchlich (siehe [Hen98a]). Die Menge der verschiedenen möglichen Genotypen bezeichnen wir als  $\mathcal{G}$ .

**Phänotyp** Der Phänotyp bezeichnet das Erscheinungsbild eines Individuums ([Hen98a]). Bei Evolutionären Algorithmen entspricht der Phänotyp meist dem Raum der Lösungen für das zugehörige Optimierungsproblem  $\Omega$ . Er kann auch eine Teilmenge von  $\Omega$  sein. Der Phänotyp wird bei Evolutionären Algorithmen aus dem Genotyp des Individuums berechnet. Aus dem Phänotyp wird über die Bewertungsfunktion  $f$  die Güte des Individuums berechnet.

**Population** In der Biologie gibt es mehrere Definitionen dieses Begriffs ([Hen98b],[Sey98a]), in der allgemeinsten ist eine Population eine Gemeinschaft von Individuen, die ein geographisch zusammenhängendes Areal bewohnen, dieselbe Reproduktionsweise besitzen, denselben erblichen Schwankungen und den gleichen Selektionswirkungen unterliegen (nach

[Sey98a]). Bei Evolutionären Algorithmen betrachtet man eine Population als eine Menge von Individuen.

**Evolutionszyklus** Unter diesem Begriff versteht man einen Ablauf im Evolutionären Algorithmus in dem von einer Generation zur nächsten Generation gewechselt wird. Der Evolutionszyklus ist eine Abbildung von einer Population  $pop_t$  auf eine Population  $pop_{t+1}: pop_t \rightarrow pop_{t+1}$ . Dabei wird ein Durchlauf durch den Evolutionszyklus dem Ablauf einer Generation gleichgesetzt. Der Index  $t$  in den Bezeichnungen der Populationen dient dazu die Population eindeutig identifizieren zu können. Gebräuchlich ist, diesen Zähler von 0 an von Generation zu Generation zu inkrementieren. So ist z. B.  $pop_0$  die erste Population, also die Initialisierung des Evolutionären Algorithmus (siehe 4.5). Der Evolutionszyklus wird in einem Evolutionären Algorithmus durch die Abarbeitung folgender Schritte erfüllt: Elternselektion, Rekombination, Mutation, Umweltselektion. Zu Beginn eines Durchlaufs durch den Evolutionszyklus wird eine Abbruchbedingung überprüft. Ist diese Abbruchbedingung erfüllt wird der Durchlauf abgebrochen und die aktuelle Population als Ergebnis der Berechnung des Evolutionären Algorithmus zurückgeliefert (näheres siehe 4.6). Obwohl in der Biologie im Prinzip auch dieser Evolutionszyklus (allerdings ohne Abbruchbedingung) verwendet wird ist dieser Begriff dort nicht gebräuchlich.

**Komma-Strategie** Bei dieser Strategie werden im Evolutionszyklus für  $pop_{t+1}$  nur Kindindividuen übernommen. Individuen aus  $pop_t$  werden nicht in  $pop_{t+1}$  übernommen.

**Plus-Strategie** Bei der Plus-Strategie können Kinder und Elternindividuen im Evolutionszyklus in die nächste Generation übernommen werden. Es werden Individuen aus der Vereinigung von  $pop_t$  und den erzeugten Kindern für  $pop_{t+1}$  ausgewählt.

**Selektionsdruck** Der Selektionsdruck wird als die Übernahmezeit definiert. Die Übernahmezeit ist eine Anzahl von Generationen. Eine Generation entsteht aus der vorherigen Generation, indem nur die Selektion angewendet wird. Dadurch, dass die Selektion im Normalfall bestimmte Individuen häufiger selektiert als andere und keine Mutation ausgeführt wird, die die Individuen verändern könnte entsteht irgendwann eine Generation, die nur noch aus Individuen besteht die den gleichen Genotyp haben. Die Anzahl der Generationen, bis zu dieser Generation ist die Übernahmezeit (siehe [Nis97a]). In der Biologie wird dieser Begriff in einer ähnlichen Bedeutung benutzt. Allerdings wird hier keine genaue Definition gegeben. Mit diesem Begriff soll die „Härte“ der Selektion ausgedrückt werden (siehe [May02]).

**Abbruchbedingung** Die Abbruchbedingung  $a : pop_t \times Z \rightarrow \{false, true\}$  entscheidet für eine Population  $pop_t$  und weiteren Zusatzinformationen  $Z$ , ob der evolutionäre Algorithmus weiter durchgeführt werden soll, oder ob die Qualität der Ergebnisse bereits ausreicht, und die Berechnung beendet werden kann. Wenn die Funktion  $true$  zurückliefert wird die Berechnung abgebrochen und die aktuelle Population  $pop_t$  ist das Ergebnis des Evolutionären Algorithmus. Wenn  $a(pop_t, Z) = false$  wird der Evolutionszyklus von neuem durchgeführt. In der Evolution in der Biologie existiert keine Abbruchbedingung. Dieser Begriff ist dort unbekannt.

**Individuum** In der Biologie wird ein Individuum oft als Synonym zu dem Begriff des Organismus gebraucht. Bei den Evolutionären Algorithmen wird ein Individuum als eine Datenstruktur verstanden, die zumindest den Genotyp umfasst. Häufig wird die Güte oder Fitness des Genotyps (des Phänotyps) in dieser Datenstruktur mitgespeichert. In diesem Fall kann man ein Individuum als ein 2-Tupel  $A = (G, F)$  auffassen.  $A.G$  ist der Genotyp des Individuums und  $A.F$  die Güte. In manchen Fällen werden weitere Zusatzinformationen  $A.S$  für die Evolutionären Operatoren oder für andere Zwecke im Individuum abgelegt. Die Güte kann nicht direkt aus dem Genotyp des Individuums berechnet werden. Um sie zu berechnen

wird aus dem Genotyp mit der Funktion  $dec$  der Phänotyp des Individuums berechnet. Anschließend kann mit der Bewertungsfunktion  $f$  die Güte berechnet werden.

**Bewertungsfunktion** Dieser Begriff wird in der Biologie nicht verwendet. Bei den Evolutionären Algorithmen berechnet die Bewertungsfunktion  $f : \Omega \rightarrow \mathbb{R}$  aus dem Phänotyp des Individuums die Güte. Die Bewertungsfunktion ist durch das Optimierungsproblem vorgegeben. Nach dieser Güte soll optimiert werden. Es soll also ein Individuum gefunden werden, dessen Güte optimal ist („bester  $x$ “).

**Mutation** In der Biologie versteht man unter einer Mutation die spontane oder induzierte Änderung des Erbgefüges (siehe [Sey98c]). Bei den Evolutionären Algorithmen verändert die Mutation den Genotyp eines Individuums  $a$ , indem es ihn auf einen in der Nachbarschaft  $\mathcal{N}$  liegenden Genotyp  $b$  abbildet. Es gilt also  $(a, b) \in \Updownarrow$ .

**Rekombination** In der Biologie versteht man unter der Rekombination den Austausch von Chromosomenbereichen zwischen homologen Chromosomen bei der Meiose ([Hen98c]). Bei den Evolutionären Algorithmen berechnet die Rekombination  $R^{r,s} : \mathcal{G}^r \rightarrow \mathcal{G}^s$  aus  $r$  Elternindividuen  $s$  Kindindividuen.

**Dekodierungsfunktion** Die Dekodierungsfunktion  $dec : G \rightarrow \Omega$  berechnet aus dem Genotyp  $G$  eines Individuums den Phänotyp  $P \in \Omega$  des Individuums. Mit diesem Phänotyp kann die Güte des Individuums mit der Bewertungsfunktion  $f$  berechnet werden. In der Biologie wird dieser Begriff nicht verwendet.

**Rekombinationswahrscheinlichkeit** In der Biologie wird mit der Rekombinationswahrscheinlichkeit die Wahrscheinlichkeit ausgedrückt, dass bei einer Fortpflanzung eine Rekombination auftritt ([Hen98c]). Im Bereich der Evolutionären Algorithmen wird mit der Rekombinationswahrscheinlichkeit  $p_x$  die Wahrscheinlichkeit bezeichnet, mit der entschieden wird, ob die Rekombination durchgeführt wird, oder einfach eines oder mehrere der Elternindividuen direkt in die Kindpopulation übernommen werden.

**Mutationswahrscheinlichkeit** In der Biologie wird hiermit die Wahrscheinlichkeit einer Mutation angegeben. Bei Evolutionären Algorithmen wird mit der Mutationswahrscheinlichkeit  $p_m$  die Wahrscheinlichkeit bezeichnet, mit der für ein Gen eines Individuums entschieden wird, ob dieses Gen des Individuums mutiert wird.

**Elternselektion** Die Elternselektion wählt aus der Elternpopulation  $E$  eine bestimmte Anzahl von selektierten Individuen aus ( $s$  Stück). Formal ist die Abbildung also:

$$S_E^{r,s} : (\Omega)^r \rightarrow (\Omega)^s$$

Diese  $s$  selektierten Individuen werden für die Rekombination verwendet. Die Selektion kann dabei deterministisch, nicht-deterministisch, mit Duplikaten oder duplikatfrei ablaufen. In der Biologie kommt der Begriff der sexuellen-Selektion diesem Begriff am nächsten ([SCS05b]).

**Umweltselektion** Die Umweltselektion wählt, wie die Elternselektion, aus einer Population von Individuen eine bestimmte Anzahl von Individuen aus:

$$S_U^{\lambda,\mu} : (\Omega)^\lambda \Rightarrow (\Omega)^\mu$$

Diese  $\mu$  selektierten Individuen werden zur Population der nächsten Generation. In der Biologie ist der Begriff der natürlichen Selektion diesem ähnlich.

## 4.4 Der Begriff der Population

Evolutionäre Algorithmen verwalten normalerweise Individuen die in sogenannten Populationen zusammengefasst sind. Eine Population wird meistens als eine Menge von Individuen definiert, wie es auch in der Biologie der Fall ist. Da aber unter einem Individuum bei Evolutionären Algorithmen normalerweise nur dessen Genotyp und Fitness verstanden wird, ist es nicht möglich z. B. ein-Eiige Zwillinge in einer Menge zu verwalten, da diese beiden Individuen einen identischen Genotyp und eine identische Fitness haben. Als Lösung für dieses Problem bieten sich mehrere Möglichkeiten an, unter anderem:

- Multimengen
- Tupel
- erweitertes Individuum

Durch die Verwendung von Multimengen in Populationen können mehrere identischen Individuen in die Population eingefügt werden. Die Individuen werden mit ihren Vielfachheiten in der Multimenge eingetragen.

Durch die Verwendung von Tupeln ist es auch möglich ein Individuum mehrmals in eine Population einzutragen. Ein Tupel impliziert aber auch eine Ordnung über die in ihr eingetragenen Individuen. Diese Ordnung ist bei einer Population normalerweise nicht erwünscht. Trotzdem wird diese Schreibweise aus Einfachheitsgründen in der Literatur verwendet (siehe [Wei02b]).

Eine weitere Möglichkeit das Problem zu lösen, ist die Individuen um eine Zusatzinformation zu erweitern, die sie eindeutig macht. Dann gibt es keine zwei Individuen, die identisch sind, obwohl ihr Genotyp und ihre Fitness gleich sind. In diesem Fall kann eine Menge verwendet werden, um die Individuen einer Population zu verwalten. Wie konkret dafür gesorgt wird, dass die Zusatzinformation eindeutig ist, muss im Einzelfall entschieden werden.

Im Weiteren wird die Population behandelt, als ob sie eine Menge wäre, da durch die Erweiterung der Individuen dieses Verhalten immer erzeugt werden kann. Allgemein können in einer Menge von Individuen mehrere Individuen vorkommen, die den gleichen Genotyp haben.

## 4.5 Die Initialisierung des Evolutionären Algorithmus

Ob ein Evolutionärer Algorithmus ein optimales Individuum (**best**er  $x$ ) finden kann hängt stark von der Initialisierung der ersten Population ab, mit der das erste Mal der Evolutionszyklus durchlaufen wird. Diese Population wird von der Initialisierungsroutine des Evolutionären Algorithmus erzeugt. Wir nennen diese Population von nun an Initialisierungspopulation und bezeichnen sie mit  $pop_0$ .

Damit **best**er  $x$  mit dem Evolutionären Algorithmus gefunden werden kann, muss die Initialisierungsroutine dafür sorgen, dass  $pop_0$  bestimmte Qualitätsmerkmale erfüllt, die von anderen Parametern des Evolutionären Algorithmus abhängen (das Optimierungsproblem, Art der Selektionen, Rekombinationen, Mutationen, Selektionswahrscheinlichkeit usw...).

Für gewöhnlich sollte die Initialisierungspopulation aus Individuen bestehen, die gleichmäßig im Raum der möglichen Genotypen  $\mathcal{G}$  verteilt sind. Durch diese Verteilung sorgt man dafür, dass der Raum, auf dem der Evolutionäre Algorithmus nach Lösungen suchen wird nicht künstlich eingeschränkt wird. Dies wäre der Fall, wenn die Individuen der Initialisierungspopulation nur aus einem kleinen Teil des Genotypraums  $\mathcal{G}$  ausgewählt werden. Diese Auswahl führt dazu,

dass normalerweise nur noch die Mutation dafür sorgen kann, dass dieser Unterraum von  $\mathcal{G}$  im Evolutionären Algorithmus wieder verlassen wird. Wenn **best**  $x$  ausserhalb des Unterraums liegt, so findet der Evolutionäre Algorithmus dieses Individuum nicht, oder erst nach sehr langer Laufzeit.

Wenn allerdings Wissen über die Struktur der Fitness der Individuen im Genotypraum vorhanden ist, so kann es sein, dass doch nur ein kleiner Teil des Genotypraums für die Auswahl der Individuen herangezogen werden muss, nämlich der Teil in dem **best**  $x$  erwartet wird. In diesem Fall kann eine solche Auswahl den Evolutionären Algorithmus stark beschleunigen.

Das Extrembeispiel dieser Einschränkung von  $\mathcal{G}$  zur Erzeugung von  $pop_0$  ist die Auswahl nur eines Genotyps für alle Individuen in  $pop_0$ . Durch die Rekombination von zwei Individuen erhält man normalerweise wieder den gleichen Genotyp, so dass für die Suche nach **best**  $x$  nur die Mutation benutzt werden kann. Wenn allerdings das Wissen über die Struktur des Suchraums so gut ist, das  $\mathcal{G}$  auf nur einen Genotyp eingeschränkt werden kann, für den erwartet wird, dass er **best**  $x$  ist, so hat dies den Evolutionären Algorithmus stark beschleunigt (die Initialisierungspopulation besteht nur aus **best**  $x$ ). Natürlich bräuchte man bei einem derart umfassenden Wissen über die Struktur des Suchraums keinen Evolutionären Algorithmus mehr, um **best**  $x$  zu finden.

## 4.6 Der Evolutionszyklus eines Evolutionären Algorithmus

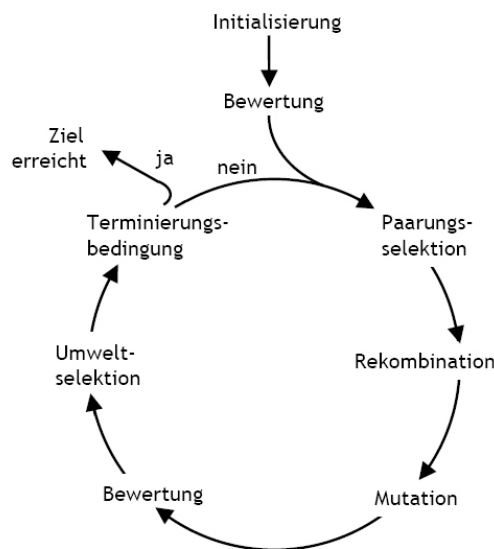


Abbildung 4.2: Der Evolutionszyklus (aus [Wei02b]).

In diesem Abschnitt werden einige Begriffe aus der obigen Begriffsdefinition in Zusammenhang gebracht und deren Funktionsweise erläutert.

Der Evolutionszyklus ist ein Grundgerüst, das der natürlichen Evolution nachempfunden ist. Viele Schritte der natürlichen Evolution sind in diesem Modell stark vereinfacht oder komplett weggelassen, so dass man nicht die volle Komplexität der natürlichen Evolution simulieren kann. Die einzelnen Schritte der Evolution, wie z. B. Elternselektion oder Rekombination werden streng

sequentiell abgearbeitet, die Generationen überlappen sich also nicht, ganz im Gegensatz zur natürlichen Evolution. Auch die Größe der Population  $\mu = |pop_t|$  und die Anzahl der Kinder  $\lambda$  ist normalerweise festgelegt und ändert sich von Generation zu Generation im Evolutionären Algorithmus nicht.

Vor dem Beginn des Evolutionszyklus wird durch eine Initialisierung eine erste Population erzeugt (siehe 4.5). Mit dieser Population (hier  $pop_0$  genannt) wird der Evolutionszyklus gestartet. Die Individuen werden bewertet und anschließend der Elternselektion unterzogen. Bei der Elternselektion werden so viele Individuen aus der Population der Eltern ausgewählt, dass bei der anschließenden Rekombination  $\lambda$  viele Kindindividuen entstehen. Je nachdem wie die Elternselektion durchgeführt wird und wie stark hierbei die Fitness der Elternindividuen berücksichtigt wird, wird über die Elternselektion der Selektionsdruck im Evolutionären Algorithmus beeinflusst. Für jedes erzeugte Kindindividuum wird nun mit der Wahrscheinlichkeit  $p_x$  die Rekombination ausgeführt. Mit der Wahrscheinlichkeit  $1 - p_x$  werden die Elternindividuen direkt als Kinder übernommen. Nach welchen Kriterien dies geschieht muss für den konkreten Evolutionären Algorithmus einzeln entschieden werden.

Wenn die Rekombination ausgeführt wird, so wird die Funktion  $R^{r,s} = \kappa_i$  ausgeführt. Die Rekombination erhält  $r$  Elternindividuen und erzeugt daraus  $s$  Kindindividuen. Nach der Durchführung aller Rekombinationen wurden alle Kinder erzeugt. Es gilt  $\bigcup \kappa_i = \lambda$ . Dann ist die Kinderpopulation dieser (der  $t$ .) Generation:  $pop_t^{Kinder} = \bigcup \kappa_i$ .

Jedes einzelne Kindindividuum wird nun mit der Wahrscheinlichkeit  $p_m$  mit der Funktion  $M$  mutiert. Mit der Wahrscheinlichkeit  $1 - p_m$  wird der Individuum nicht verändert. Anschließend wird die Güte der Kindindividuen mit der Bewertungsfunktion  $f$  bestimmt. Durch die Umweltselektion  $S_U^{\lambda,\mu}(pop_t^{Kinder}) = pop_{t+1}$  wird bei der sogenannten „Komma“-Strategie die Elternpopulation der nächsten Generation erzeugt. Es kann auch die „Plus“-Strategie verwendet werden, dann wird mit  $S_U^{\lambda+\mu,\mu}(pop_t^{Kinder} \cup par_t) = pop_{t+1}$ . Bei der Komma-Strategie werden nur Kinder übernommen, während bei der Plus-Strategie auch Eltern übernommen werden können. Bei der Plus-Strategie kann ein Individuum mehrere Generationen lang leben, während es bei der Komma-Strategie immer nach einer Generation „stirbt“.

Die Umweltselektion ist eine weitere Möglichkeit im Evolutionszyklus den Selektionsdruck einzustellen. Die Elternpopulation der nächsten Generation wird nun zu Beginn des nächsten Durchlaufs des Evolutionszyklus von der Abbruchbedingung überprüft und der Evolutionszyklus beginnt gegebenenfalls von neuem.

## 4.7 Die Komponenten des Evolutionszyklus

In diesem Abschnitt wird näher auf die Wirkungsweise und die Funktion einzelner Komponenten des Evolutionskreises eingegangen. Für eine kurze Beschreibung des gesamten Evolutionszyklus und deren Komponenten siehe 4.6 und 4.3.

### 4.7.1 Die Selektionen im Evolutionszyklus

Die Elternselektion und die Umweltselektion sind beides Mechanismen, die Individuen auswählen. Bei der Elternselektion werden die Elternteile selektiert, die Kinder erzeugen sollen, während bei der Umweltselektion entschieden wird, welche Kinder in die nächste Generation übernommen werden. Die Verfahren, nach denen selektiert wird sind ähnlich. Deshalb werden in diesem Abschnitt die

Gemeinsamkeiten der beiden Selektionen beschrieben. In den beiden Unterabschnitten wird dann auf die speziellen Eigenheiten der beiden Selektionen eingegangen.

Selektionen können auf verschiedene Arten durchgeführt werden:

deterministisch	probabilistisch
mit duplikaten	duplikatenfrei
allgemein	problembezogen

Eine allgemeine Selektion  $S^{r,s} : (\Omega)^r \rightarrow (\Omega)^s$  selektiert aus einer Menge von  $r$  Individuen eine Menge von  $s$  Individuen. Wenn nun eine Menge von Individuen gegeben ist  $M_r \in (I)^r$  gilt für jedes Individuum aus  $M_s = S^{r,s}(M_r)$  das jedes Element in dieser Menge auch in der ursprünglichen Menge enthalten war  $\forall i \in M_s : i \in M_r$ . Es werden also keine Individuen neu erzeugt, sondern nur Individuen aus der zu selektierenden Menge  $M_r$  ausgewählt. Die Elemente in der Menge  $M_r$  sind nach einer bestimmten Ordnung durchnummeriert, so das mit  $\rho_i$  auf das  $i$ . Element nach dieser Ordnung referenziert werden kann.

Im Besonderen ist es möglich, dass ein Individuum aus  $M_r$  durch eine Selektion mehrmals ausgewählt wird. Eine solche Selektion nennt man eine Selektion mit Duplikaten. In  $M_s$  kann aber ein Individuum nicht mehrmals vorkommen, so dass bei Selektionen mit Duplikaten entweder neue Individuen erzeugt werden müssen (mit eindeutiger Zusatzinformation), oder  $M_s$  zu einer Multimenge erweitert werden muss, um die Vielfachheiten der selektierten Individuen abzuspeichern. Dieses Problem wird in 4.4 schon beschrieben).

Wenn Duplikate in  $M_s$  ausgeschlossen werden, so kann  $M_s$  als Menge verwaltet werden. In diesem Fall nennt man die Selektion duplikatfrei.

Um Probleme bei der Formulierung zu vermeiden und die Beschreibungen allgemein zu halten arbeiten Selektionen auf Populationen. Sie erhalten eine Population mit  $r$  Individuen aus der sie  $s$  Individuen selektieren sollen. Je nach Variante der Selektion versteht man unter der Population dann eine Menge oder eine Multimenge.

Eine Selektion kann deterministisch oder probabilistisch durchgeführt werden. Bei einer deterministischen Selektion wird durch einen deterministischen Algorithmus die Selektion durchgeführt. Wenn also zwei mal die Selektion mit dem selben  $M_r$  aufgerufen wird, so wird auch zwei mal die gleiche Population  $M_s$  der selektierten Individuen zurückgegeben. Bei einer probabilistischen Selektion werden die Individuen mit einer bestimmten, meist durch die Fitness bestimmten Wahrscheinlichkeit ausgewählt. Die deterministische Selektion von Individuen zerstört die Diversität der Population und beschneidet das Fitnesspektrum sehr stark, wie in Abbildung 4.3 zu sehen. Dagegen wird bei einer probabilistischen Selektion auch eine Verbesserung der Durchschnittsfitness der Individuen in der entstandenen Menge erreicht, allerdings ohne diesen harten Schnitt im Fitnesspektrum. Die deterministischen Selektionen scheinen aus dieser Betrachtung nicht vielversprechend.

Im Folgenden werden nun einige gebräuchlichen allgemeinen Selektionen beschrieben. Die Selektionen sind probabilistisch, jedoch mit oder ohne Duplikaten möglich. Es werden die Versionen mit Duplikaten besprochen. Diese Beschreibung soll das Prinzip der jeweiligen Selektion verdeutlichen, jedoch nicht auf Implementierungsdetails eingehen, diese werden in Kapitel A angesprochen.

Bei den Beschreibungen wird immer davon ausgegangen, dass die Selektion  $S^{r,s}$ , die Zufallsvariable  $X_s(\omega)$  und des Ereignis  $\omega =$  Selektion eines Individuums folgendermaßen definiert sind:

$$X_s(\omega) = \begin{cases} 1 & \text{Es wird } \rho_1 \text{ ausgewählt} \\ 2 & \text{Es wird } \rho_2 \text{ ausgewählt} \\ \dots & \\ r & \text{Es wird } \rho_r \text{ ausgewählt} \end{cases}$$

### Zufallsbasierte Selektion

Bei der zufallsbasierten Selektion  $S^{r,s}$  ist die Wahrscheinlichkeit, dass das Ereignis  $i$  bei einer Realisierung auftritt nun  $Pr(X_s = i) = \frac{1}{r} \forall i \in \{1, \dots, r\}$ . Jedes Individuum hat also die gleiche Wahrscheinlichkeit bei einer Realisierung  $x_i$  selektiert zu werden. Für die Selektion aller  $s$  Individuen werden nun die Realisierungen  $x_1, \dots, x_s$  gemessen und entsprechend die Menge der selektierten Individuen erzeugt.

Diese Selektion ist zwar eine allgemeine Selektion, kann also den Fitnesswert der Individuen für die Selektion benutzen, nutzt ihn aber nicht. Die Selektion betrachtet überhaupt keine Eigenschaft der Individuen, deshalb geht von ihr auch kein Selektionsdruck aus. Die durchschnittliche erwartete Fitness der selektierten Individuen ist genauso hoch wie die durchschnittliche Fitness der Individuen vor der Selektion, wenn  $r$  und  $s$  groß genug sind, um das Gesetz der großen Zahlen anzuwenden zu können.

### Fitnessproportionale Selektion

Bei der fitnessproportionalen Selektion  $S^{r,s}$  wird für jedes Individuum  $i$  die Fitness  $f(i)$  berechnet.

Die Wahrscheinlichkeit, dass bei einer Realisierung  $x$  das  $i$ . Individuum selektiert wird  $Pr(X_s = i)$  ist nun abhängig von der Fitness des  $i$ . Individuums:

$$Pr(X_s = i) = \frac{f(i)}{\sum_{\forall i} f(i)}$$

Damit ist die Wahrscheinlichkeit, dass ein Individuum selektiert wird direkt abhängig von der Fitness dieses Individuums im Verhältnis zur Gesamtfitness der Population, aus der es ausgewählt werden soll (Die Summe der Fitnesswerte aller Individuen in der Population). Nun können wieder die Realisierungen  $x_1, \dots, x_s$  ausgewertet werden, um die selektierte Population zu erzeugen.

Bei dieser Selektion ergibt sich das Problem, dass die Bestrafung bzw. die Bevorzugung der Individuen abhängig ist von der durchschnittlichen Fitness eines Individuums in der Population. Wenn die Population sehr homogen ist, und sich die Fitnesswerte der Individuen nur minimal unterscheiden, so degeneriert die fitnessproportionale Selektion zu einer Zufallsselektion, da die Wahrscheinlichkeiten zur Auswahl eines Individuums  $Pr(X_s = i) \rightarrow \frac{1}{r}$  je homogener die Population wird.

Andererseits ist bei einer sehr inhomogenen Population mit nur einem Individuum, das eine sehr gute Fitness erreicht die Wahrscheinlichkeit sehr groß, dass dieses „super“ Individuum sehr häufig selektiert wird. Dadurch besteht die Gefahr, dass die Population von diesem Individuum übernommen wird (die Population besteht nur noch aus Individuen mit dem gleichen Genotyp).

Diese beiden Phänomene sind in der Abbildung 4.4 und 4.5 zu erkennen. Der Degenerierung zur Zufallsselektion kann dadurch entgegengewirkt werden, dass eine Verschiebung  $v(x) : \mathbb{R} \rightarrow$

$\mathbb{R} = x - v$  der Fitnesswerte abhängig von der Population definiert wird. Dabei sollte für das am schlechtesten bewertete Individuum der Population  $i_{min}$  gelten, das  $v(f(i_{min})) > 0$ . Nun wird

$$Pr(X_s = i) = \frac{v(f(i))}{\sum_{\forall i} v(f(i))}$$

Graphisch ist dieser Vorgang in Abbildung 4.6 zu sehen. Das Koordinatensystem der Bewertung wird einfach verschoben, wodurch die bestehenden Unterschiede der Fitness der Individuen prozentual größer werden und stärker in die Selektionswahrscheinlichkeit einfließen. Durch dieses Vorgehen wird aber nicht das Problem der Übernahme der Population durch ein „super“ Individuum behoben.

Die rangbasierte Selektion behebt die beiden genannten Probleme der fitnessproportionalen Selektion.

### Rangbasierte Selektion

Bei der rangbasierten Selektion wird für jedes Individuum  $i$  der Fitnesswert des Individuums  $f(i)$  bestimmt.

Die Wahrscheinlichkeit,  $Pr(X_s = i)$  ist nun wie bei der fitnessproportionalen Selektion abhängig von der Fitness des Individuums. Allerdings wird hier nicht die Fitness des Individuums direkt herangezogen, sondern bei dieser Selektion ist nur der Rang des Individuums in der Population wichtig. Es zählt also nur, an der wievielten Stelle das Individuum einsortiert werden würde, wenn die Individuen der Population sortiert werden würden. Wenn zwei Individuen die gleiche Fitness haben, so ist die Reihenfolge zwischen diesen beiden Individuen beliebig.

Die Wahrscheinlichkeit, dass ein Individuum selektiert wird, wird anhand des Rangs entschieden. Dabei kann ein linearer Anstieg der Wahrscheinlichkeit, oder ein beliebig anderer Zusammenhang zwischen dem Rang und der Selektionswahrscheinlichkeit angenommen werden (z. B. quadratischer Anstieg der Selektionswahrscheinlichkeit siehe 4.7). Dies verallgemeinert die aus der fitnessproportionalen Selektion bekannte Funktion  $v : \mathbb{R} \rightarrow \mathbb{R}$  zu einer allgemeinen Abbildung. Sie ist nicht mehr auf eine einfache Verschiebung beschränkt. Es muss allerdings für die Funktion  $v$  immer gelten, das

$$\sum_{\forall i \in pop_{sel}} v(f(i)) = 1$$

Die Funktion  $rang : \mathcal{G} \rightarrow \{0, \dots, n-1\}$  ordnet den Individuen ihren Rang zu. Eine lineare Bewertungsfunktion für ein Individuum unter Berücksichtigung seines Ranges kann dann folgendermaßen aussehen (abweichend von der Literatur wurde hier eine allgemeine Geradengleichung verwendet):

$$f(rang(i)) = a \cdot rang(i) + k$$

wobei  $a$  und  $k$  Konstanten und  $i$  den Genotyp des Individuums darstellen. Im weiteren Verlauf dieser Selektion wird mit  $j$  der Rang des Individuums bezeichnet. Wenn nun die Individuen den Rang 0 bis  $n-1$  haben, und das beste Individuum den Rang 0 hat, so ergibt sich aus der Bedingung

$$\begin{aligned} \sum_{\forall i \in pop_{sel}} v(f(i)) = 1 &\Rightarrow \sum_{j=0}^{n-1} (w(0) - (a \cdot j)) = n \cdot w(0) - \sum_{j=0}^{n-1} a \cdot j \\ &= n \cdot w(0) - a \sum_{j=0}^{n-1} j = n \cdot w(0) - a \frac{n(n-1)}{2} \stackrel{!}{=} 1 \end{aligned}$$

$$\Rightarrow a = \frac{2(n \cdot w(0)) - 2}{n(n-1)}$$

Somit kann die Steigung  $a$  ausgehend von der Wahrscheinlichkeit  $w(0)$  des besten Individuums errechnet werden, oder umgekehrt bei einer gewünschten Steigerung der Wahrscheinlichkeit zwischen zwei Rängen die Bewertung des besten Individuums  $w(0)$  durch

$$w(0) = \frac{2 + n \cdot a(b-1)}{2n}$$

Durch eine ähnliche Rechnung ergeben sich für eine quadratische Gleichung

$$f(j) = w(0) - aj^2 + bj$$

ergibt sich für die Parameter:

$$a = \frac{n \cdot w(0) - b\left(\frac{n(n-1)}{2}\right) - 1}{\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n}$$

und

$$b = \frac{2(n \cdot w(0) - a\left(\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n\right)) - 2}{n(n-1)}$$

Durch die Wahl zweier Parameter ergibt sich der Dritte automatisch. Durch die Wahl der Parameter kann man bestimmen, ob der quadratische Anteil, oder der lineare Anteil mehr Gewicht in der Funktion erhalten soll. Außerdem kann die Selektionswahrscheinlichkeit des besten Individuums und damit der Unterschied zwischen der Selektionswahrscheinlichkeit des besten und des schlechtesten Individuums eingestellt werden. Wenn  $a = 0$  gesetzt wird, so ergibt sich in Übereinstimmung zur Geradengleichung  $b = \frac{2(n \cdot w(0)) - 2}{n(n-1)}$ , da die Geradengleichung nur ein Spezialfall ist.

Auf ähnliche Weise können auch andere Funktionen für die Selektionswahrscheinlichkeit des Individuums des  $i$ . Ranges benutzt werden.

Bei dieser Selektion müssen  $O(r \log r)$  Vergleiche durchgeführt werden, bis die Individuen nach ihrem Rang sortiert sind. Anschließend können die Individuen ausgewählt werden. Dann kann in einem Schritt aus der gezogenen Zufallszahl das selektierte Individuum bestimmt werden.

### q-Turnierselektion

Bei der  $q$ -Turnierselektion werden aus der Population der zu selektierenden Individuen mit einer Zufallsselektion  $S_{s,q}$   $q$  Individuen ausgewählt. Diese  $q$  Individuen bilden die Population  $M_q$ .

Die Individuen aus  $M_q$  treten nun in einem gedachten Turnier gegeneinander an. Normalerweise treten dabei immer zwei Individuen gegeneinander an und das Bessere der beiden Individuen kommt weiter. Für diese Entscheidung muss der Fitnesswert der beiden Individuen nicht berechnet werden. Es muss nur entschieden werden, welches der beiden Individuen eine höhere Fitness als das Andere hat.

Nach dem Ende des Turniers werden die  $x > 0$  besten Individuen selektiert. Anschließend wird eine neue Population von Individuen als  $M_r$  für das nächste Turnier ausgewählt. Bei jedem Turnier werden immer  $x$  Individuen bestimmt, die in die Population  $M_s$  aufgenommen werden. Es müssen also  $\lfloor \frac{s}{x} \rfloor$  viele Turniere durchgeführt werden. Die Zufallsvariable  $X_s(\omega)$  mit dem Ereignis  $\omega =$  Selektion eines Individuums ist in diesem Algorithmus implizit enthalten. Die

Wahrscheinlichkeit für die Auswahl eines Individuums kann aber aus der obigen Beschreibung zunächst nicht direkt angegeben werden. Sie kann aber berechnet werden.

Diese Wahrscheinlichkeit soll nun für das Beispiel einer Turnierselektion, bei der nur der Gewinner des Turniers selektiert wird berechnet werden.

Die Selektionen können also durch Beobachtung der Selektionen nicht mehr unterschieden werden. Sie verhalten sich gleich. Die  $q$ -Turnierselektion umgeht allerdings den Aufwand, der zur Sortierung der Individuen bei der rangbasierten Selektion betrieben werden muss, um den Rang explizit zu berechnen. Dieser Wert wird bei der  $q$ -Turnierselektion nicht explizit berechnet. Die Anzahl der Vergleiche, bis  $s$  Individuen selektiert sind ist  $O(s \cdot q)$ , da bei jedem Turnier das beste Individuum gefunden werden muss. Dies geschieht  $s$  mal.

### Elternselektion

Die Elternselektion entscheidet, welche Individuen die Möglichkeit erhalten ihre im Genotyp gespeicherten Informationen an die Kinder, und somit indirekt an die nächste Generation weiterzugeben. Diese Selektion ist die erste Stelle im Evolutionszyklus, an der der Selektionsdruck eingestellt werden kann. Je nachdem wie stark die hier eingesetzte Selektion die Individuen mit einer schlechten Fitness bestraft fällt der Selektionsdruck an dieser Stelle höher oder niedriger aus. Die Elternselektion steht in einem direkten Zusammenhang mit der gewählten Rekombination, da normalerweise der Genotyp der Elternindividuen im weiteren Ablauf rekombiniert werden wird. Deshalb sollten in der Elternselektion Eltern ausgewählt werden, deren Chancen hoch sind, durch die Rekombination mit anderen selektierten Eltern ein fites Kind zu erzeugen.

Bei der Elternselektion werden normalerweise Tupel (im Normalfall 2-Tupel) von Eltern zusammengestellt, aus denen später mit der Rekombination Kindindividuen erzeugt werden. Es werden normalerweise entweder gleich viele Kinder wie Eltern erzeugt, oder genau ein Kind pro Rekombination, also aus einem selektierten Elternpaar erzeugt. Diese Selektion wird also mehrfach auf die Elternpopulation ausgeführt, bis genug Elternpaare ausgewählt wurden, um die gewünschte Anzahl von Kindern erzeugen zu können.

### Umweltselektion

Die Umweltselektion wird auf die Population der erzeugten Kindindividuen angewendet. Diese Population hat  $\lambda$  viele Individuen. Normalerweise ist  $\lambda$  ein vielfaches von  $\mu$ , also der Anzahl der Individuen in der Population, aus der die Eltern ausgewählt werden.

Die Umweltselektion erzeugt aus einer Population mit  $\lambda$  vielen Individuen eine Population mit  $\mu$  vielen Individuen. Die erzeugte Population wird normalerweise im nächsten Evolutionszyklus als Population verwendet, aus der die Eltern mit der Elternselektion ausgewählt werden. Die Umweltselektion soll die Umwelteinflüsse, die auf die Individuen einwirken nachbilden. Sie soll sicherstellen, dass die Fitness der Individuen, die in die nächsten Evolutionszyklus übernommen werden eine gewisse Güte erreicht. Durch die Rekombination können Kindindividuen entstanden sein, die eine schlechte Fitness haben. Diese werden normalerweise mit der Umweltselektion mit hoher Wahrscheinlichkeit nicht in die Population für den nächsten Evolutionszyklus ausgewählt. Durch die Umweltselektion wird ein Selektionsdruck erzeugt. Der Selektionsdruck durch die Umweltselektion darf aber nicht zu hoch sein, damit die Population nicht von Individuen mit gleichem Genotyp übernommen werden kann.

### 4.7.2 Rekombination

Eine Rekombination  $R^{r:s} : \mathcal{G}^r \rightarrow \mathcal{G}^s$  erzeugt aus einer Menge von  $r$  Elternindividuen eine von  $s$  Kindern. Dabei wird der Genotyp der Kindindividuen  $k_i, i \in \{1, \dots, s\}$  aus den Genotypen der Elternindividuen zusammengestellt. Im Folgenden wird davon ausgegangen, dass sich der Genotyp aufteilen lässt, in

$$\mathcal{G} = g_1, g_2, \dots, g_n$$

wobei es sich bei den  $g_i$  um die Gene des Genotyps handelt. Es gibt viele verschiedene Rekombinationen, die sich in verschiedenen Optimierungsproblemen bewährt haben. Da in dieser Arbeit ein Genetischer Algorithmus in Binärkodierung verwendet wird, werden nun Rekombinationen vorgestellt, die auf dem Crossover-Prinzip basieren. Ein Crossover setzt ein Kindindividuum aus den unveränderten Genen seiner Eltern zusammen. Für ein Gen des Kindes gilt also, dass es gleich dem Gen einer seiner Eltern ist. Motiviert ist dieses Prinzip aus Beobachtungen der natürlichen Vermehrung von Organismen in der Natur, bei der auch crossing-over Effekte geschehen können.

Für Crossover-Operatoren werden häufig zwei Bezugsgrößen herangezogen, um diese zu bewerten. Es handelt sich hierbei um den *positional bias* und den *distributional bias*.

Der positional bias bewertet, wie stark die Abweichung der Wahrscheinlichkeit eines Gens, von einem Elternindividuum übernommen zu werden, von der Gleichverteilung ist. Der distributional bias bewertet die Abweichung der Wahrscheinlichkeiten der Längen der Teilstücke, die übernommen werden von der Gleichverteilung. Wenn keine Abweichung vorhanden ist, dann ist der bias jeweils 0. Dieser Zustand wird normalerweise angestrebt.

Die problemabhängigen Rekombinationen bzw. Crossover, die im Rahmen dieser Arbeit entstanden sind werden im Kapitel 5 vorgestellt.

#### 1-Punkt-Crossover

Der 1-Punkt-Crossover ist die Standard-Rekombination für den Genetischen Algorithmus. Ursprünglich wurde sie für eine Rekombination mit zwei Elternindividuen verwendet. Es können entweder ein oder zwei Kinder rekombiniert werden. Der Genotyp der beiden Eltern sei  $\mathcal{G}_1$  und  $\mathcal{G}_2$ . Die Rekombination bestimmt zufällig einen Punkt  $k \in \{1, \dots, n\}$  und erzeugt die Kinder:

$$K_1 = \begin{cases} (k_1)_i = (g_1)_i & i \leq k \\ (k_1)_i = (g_2)_i & i > k \end{cases}$$

$$K_2 = \begin{cases} (k_2)_i = (g_2)_i & i \leq k \\ (k_2)_i = (g_1)_i & i > k \end{cases}$$

wobei das zweite Kind  $K_2$  nur erzeugt wird, wenn zwei Kinder bei der Rekombination erzeugt werden sollen. Die Rekombination kann auf  $r$  Elternindividuen erweitert werden. Dann muss entschieden werden, von welchen Eltern die Teilstücke für die Kinder übernommen werden. In dieser Arbeit wurde die Variante mit zwei Elternindividuen und zwei Kindern implementiert siehe 5.3.1.

Diese Rekombination ist dann für den Evolutionären Algorithmus geeignet, wenn eine hohe Fitness des Individuums daraus resultiert, dass der Genotyp des Individuums aus guten Teilstücken, sogenannten Building-Blocks aufgebaut ist. Die Rekombination kombiniert nun die Building-Blocks der beiden Elternindividuen zu einem Kindindividuum. Durch die Vertauschung können im Kindindividuum die Building-Blocks besser verteilt sein. Dadurch hat das Kindindividuum eine höhere Fitness (mehr dazu siehe Schema-Theorem in [Wei02d]).

Der positional bias des 1-Punkt-Crossovers ist hoch, da für die ersten Gene der Kindindividuen die Wahrscheinlichkeit sehr hoch ist, von einem bestimmten Elternindividuum übernommen zu werden (z. B. für  $K_1$  von dem ersten Elternindividuum). Der distributional bias des 1-Punkt-Crossovers ist hingegen 0, es liegt also kein distributional bias vor. Auch bei der Erweiterung auf mehr als zwei Elternindividuen ändern sich diese Eigenschaften des 1-Punkt-Crossovers nicht.

Bildlich gesprochen werden die Elternindividuen bei diesem Crossover an einer Stelle aufgeteilt, und dann aus den Teilstücken wieder die Kindindividuen zusammengesetzt. Siehe Abbildung 4.8.

### Diagonal-Crossover

Der diagonal-Crossover ist für  $m$  Elternindividuen anwendbar und erzeugt dann  $m$  Kindindividuen ( $m \geq 2$ ). Er bestimmt  $n > 1$  Schnittpunkte  $s_1, s_2, \dots, s_n$  mit  $0 \leq s_1 \leq s_2 \leq \dots \leq s_n$ . Nummeriert man die von Crossover-Punkten  $s_i$  bzw. Anfang und Ende begrenzten Abschnitte der Individuen von links beginnend fortlaufend mit  $1, 2, \dots, n+1$ , so wird für das  $j$ . Kindindividuum der  $i$ . Abschnitt vom  $(i+j) \bmod m$ -ten Elternindividuum übernommen. Hierbei nummeriert man die Eltern und Kindindividuen von 0 beginnend fortlaufend auf. Es wird jedes  $\bmod m$ -te Teilstück von dem gleichen Elternindividuum übernommen. In Abbildung 4.9 ist ein Beispiel mit 3 Elternindividuen und  $n = 2$  Schnittpunkten gegeben.

Für zwei Elternindividuen und einem Diagonal-Crossover mit  $n = 2$  ergibt sich dann:

$$K_1 = \begin{cases} (k_1)_i = (g_1)_i & i > s_2 \vee i < s_1 \\ (k_1)_i = (g_2)_i & s_1 \leq i \leq s_2 \end{cases}$$

und

$$K_2 = \begin{cases} (k_2)_i = (g_2)_i & i > s_2 \vee i < s_1 \\ (k_2)_i = (g_1)_i & s_1 \leq i \leq s_2 \end{cases}$$

Der Diagonal-Crossover hat einen niedrigeren positional bias als der 1-Punkt-Crossover (wenn man diesen auf mehr als zwei Eltern erweitert). Je größer der Wert  $n$  ist, umso niedriger wird der positional bias. Allerdings steigt für größeres  $n$  der distributional bias an.

### Uniform-Crossover

Der uniform-Crossover ist eine Verallgemeinerung des 1-Punkt-Crossovers. Statt an einer Stelle  $k$  zu entscheiden, ob die Gene  $g_i$  von dem einen, oder dem anderen Eltern-Individuum übernommen werden, wird diese Entscheidung für jedes Gen  $g_i$  einzeln getroffen. Hierbei ist die Wahrscheinlichkeit eines Gens, vom ersten Elternindividuum übernommen zu werden für das erste Kind  $p_{ux}$  und für das zweite Kind  $1 - p_{ux}$ . Ursprünglich war dieser Crossoveroperator auch für zwei Elternindividuen gedacht, kann jedoch leicht erweitert werden. Die Erweiterung erfolgt, indem für jedes Gen zufällig eines der Elternindividuen ausgewählt wird, von dem es übernommen werden soll.

Bei  $r$  Elternindividuen wird nun für jedes der  $s$  Kindindividuen für jedes Gen  $(k_i)_j$  entschieden, von welchem der  $r$  Eltern es übernommen wird, indem eine Zufallszahl  $z \in \{1, \dots, r\}$  gezogen wird. Insgesamt werden  $n$  Zufallszahlen gezogen.

$$K_i = \left\{ (k_i)_j = (g_{z_i})_j \right.$$

Die Gene des Individuum haben bei diesem Crossoveroperator keinerlei räumlichen Zusammenhang mehr. Das bedeutet, wenn das  $i$ . Gen von Individuum  $i$  übernommen wird, so hat dies keinerlei Einfluss auf die Wahrscheinlichkeit, dass das  $i+1$  Gen von diesem Individuum übernommen wird.

Der positional bias des uniform-Crossovers ist 0, da die Position eines Bits keine Auswirkung auf die Zufallszahl hat, die benutzt wird, und auch alle Bits gleich behandelt werden.

Der distributional bias des uniform-Crossovers ist sehr hoch. Die Länge der von einem Elternindividuum übernommenen Teilstück des Genotyps folgt einer Binomialverteilung mit dem Erwartungswert  $p_{ux} \cdot L$ .

Als Varianten kann man die Wahrscheinlichkeiten, mit denen ein Gen von einem bestimmten Elternindividuum übernommen wird variieren. So kann man z. B. spezielle Individuen mit einer zweiten Fitnessfunktion bevorzugen, durch diese Änderung erhält man wieder einen distributional bias. Dies wurde im Rahmen dieser Arbeit allerdings nicht implementiert.

### 4.7.3 Mutation der Kinder

Nachdem im Evolutionszyklus die Rekombination abgeschlossen ist wird der Mutationsschritt durchgeführt. Beim Mutationsschritt wird für jedes Kindindividuum ( $\lambda$  viele) entschieden, ob eine Mutation durchgeführt werden soll (siehe Abbildung 4.10). Dieses Verfahren ist normalerweise auch stochastisch und nicht deterministisch. Für den Evolutionären Algorithmus ist eine sogenannte Mutationswahrscheinlichkeit  $p_{mut}$  festgelegt. Mit dieser Wahrscheinlichkeit wird durchschnittlich an einem Kindindividuum die Mutation durchgeführt.

Wenn die Mutation für ein Individuum  $I$  durchgeführt werden soll, so wird eine spezielle Funktion

$$M : \mathcal{G} \rightarrow \mathcal{G}$$

auf dem Genotyp  $\mathcal{G}_i$  dieses Individuums angewendet. Im Allgemeinen ist diese Funktion nicht deterministisch. Diese Mutation wird für jedes Individuum in der Kindpopulation separat durchgeführt, im Gegensatz zur Rekombination, bei der zwangsläufig mehrere Individuen beteiligt waren. Es wird auch immer nur ein Individuum bei der Mutation erzeugt. Erst die mutierten Individuen bzw. die Individuen, bei denen keine Mutation durchgeführt wurde, werden mit der Fitnessfunktion bewertet und dann der Umweltselektion unterzogen.

Wie bei der Rekombination und der Selektion gibt es auch hier mehrere Verfahren, die als Mutation angewendet werden können.

#### Bit-Flip-Mutation

Die Bit-Flip-Mutation ist die Standard-Mutation des Genetischen Algorithmus. Bei dieser Mutation wird davon ausgegangen, dass das Genom des Individuums binär codiert vorliegt. Für die Bit-Flip-Mutation wird nun eine Wahrscheinlichkeit angegeben, mit der ein Bit in einem Individuum **geflippt**, also geändert wird. Diese Wahrscheinlichkeit kann fest sein (z. B. 0.1) oder in Abhängigkeit zur Länge  $n$  des Genoms eines Individuums festgelegt werden (z. B. 5 Bits werden **geflippt** oder  $\sqrt{n}$  werden **geflippt**).

Nun wird für jedes Bit des Genoms eine Zufallszahl gezogen und anhand dieser entschieden, ob das Bit verändert werden soll, oder nicht. Die Wahrscheinlichkeit für eine Änderung ist dabei genau die Wahrscheinlichkeit, die für die Bit-Flip-Mutation eingestellt wurde. Um ein Individuum zu mutieren müssten  $n$  Zufallszahlen gezogen werden. Dies kann man umgehen, indem man pro

Individuum entscheidet, ob es mutiert werden soll. Anschließend wird das Individuum mutiert (wobei mehr als ein Bit **gefilp**pt werden kann) oder direkt übernommen.

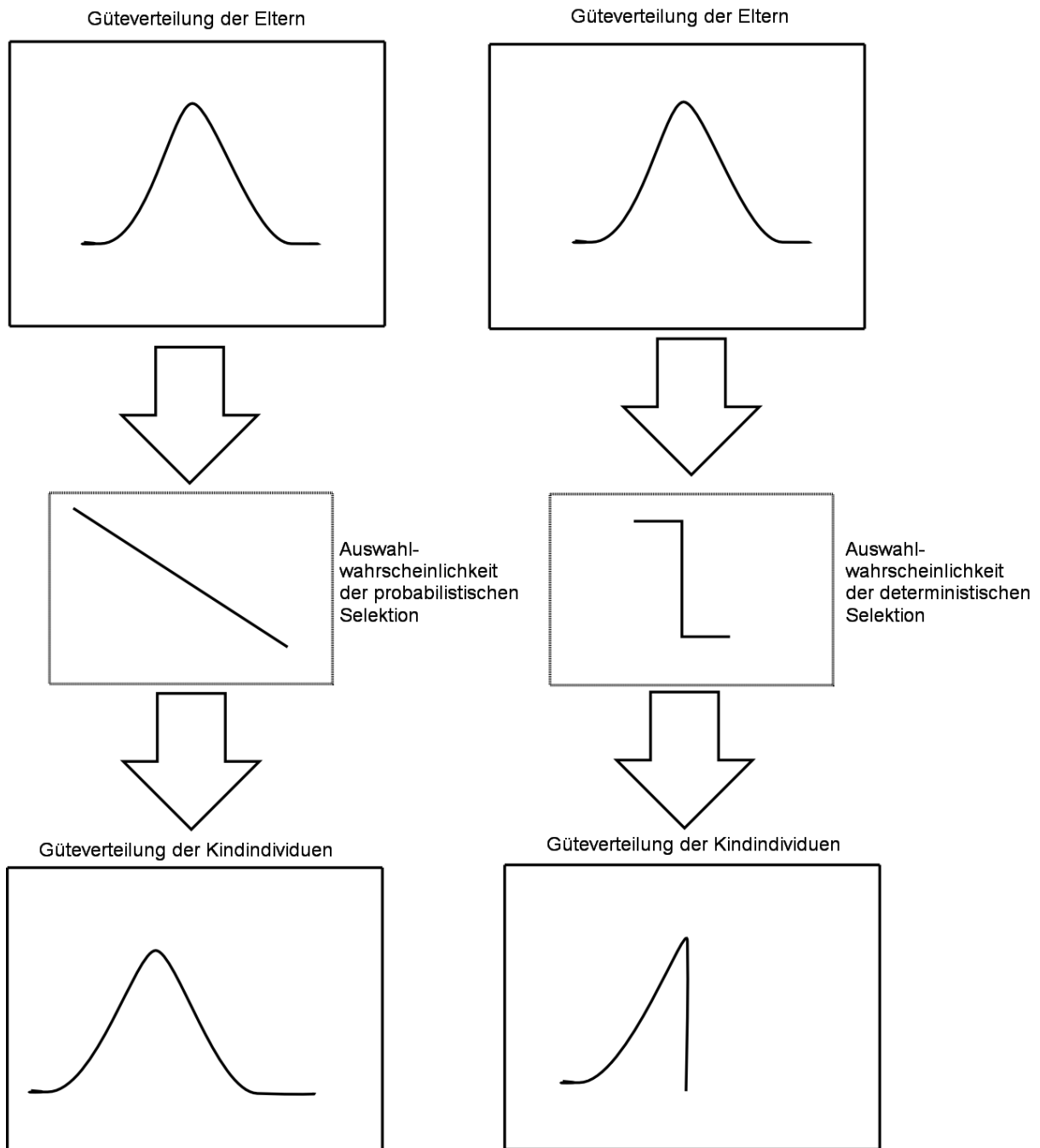


Abbildung 4.3: Güteverteilung vor und nach probabilistischen und deterministischen Selektionen (nach [Wei02c]).

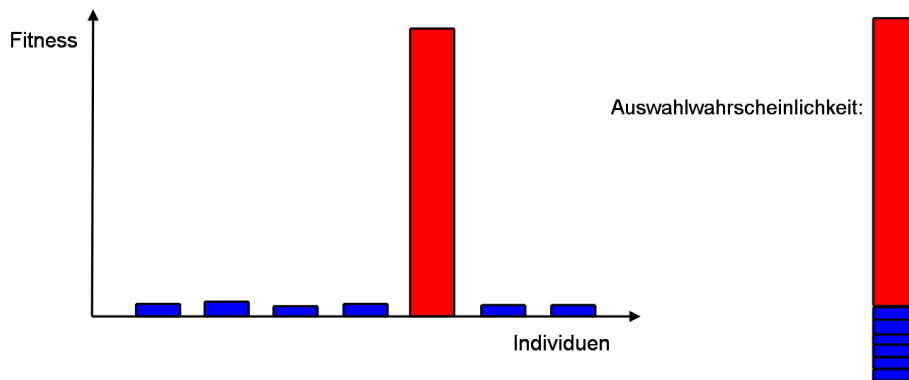


Abbildung 4.4: Eine Population, die von einem Superindividuum dominiert wird.

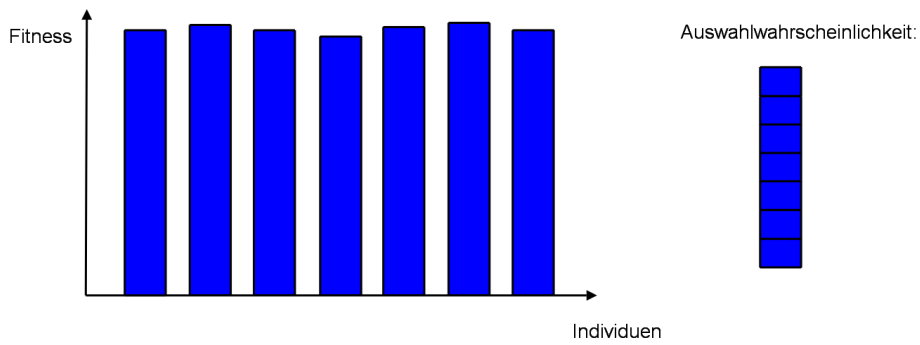


Abbildung 4.5: Eine Population, deren Individuen alle eine sehr große Fitness haben.

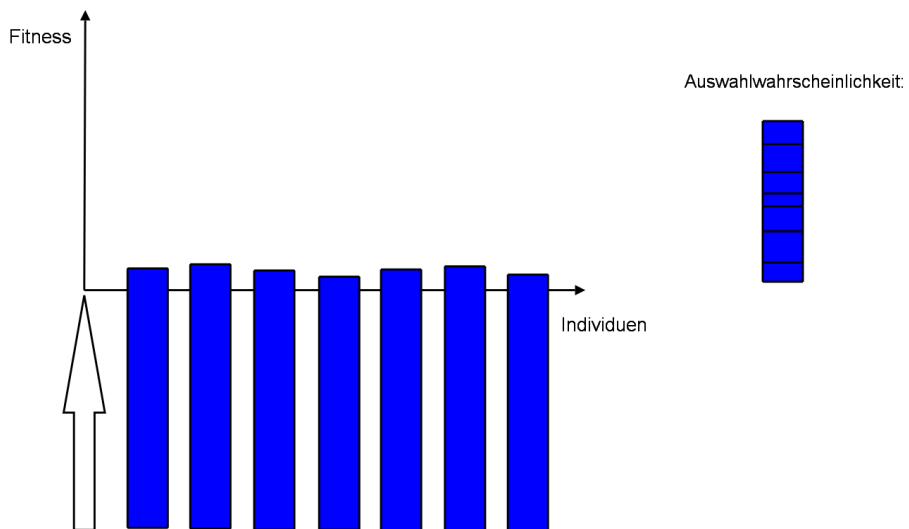


Abbildung 4.6: Durch die Verschiebung der Koordinatenachse werden die Selektionswahrscheinlichkeiten der Individuen wieder unterschiedlicher.

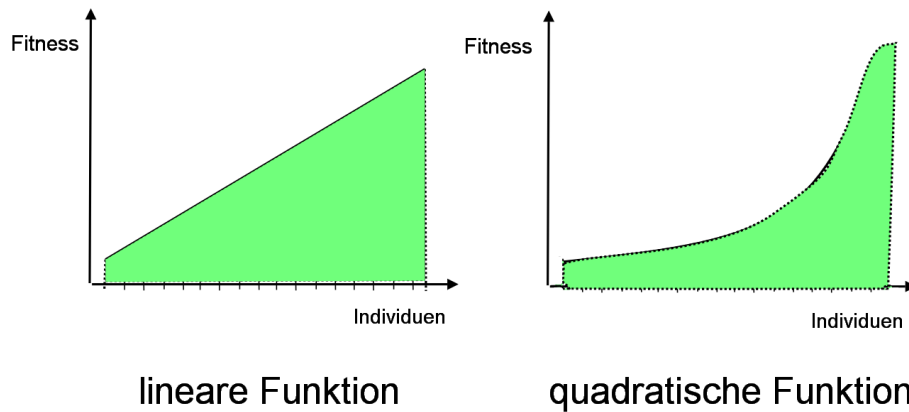


Abbildung 4.7: In dieser Abbildung werden die sortierten Individuen einer Population in Verhältnis zu der Auswahlwahrscheinlichkeit der rangbasierten Selektion gesetzt. Für die Auswahlwahrscheinlichkeit kann eine lineare, oder z. B. eine quadratische Funktion benutzt werden.

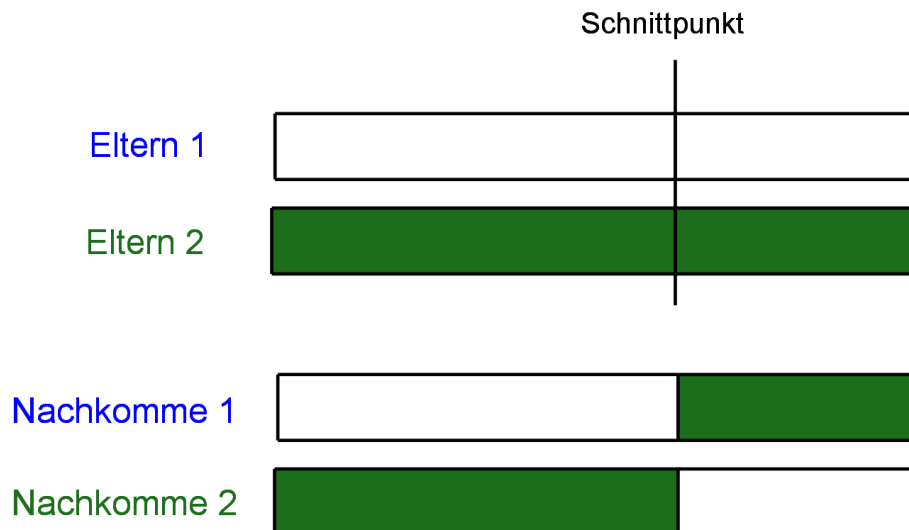


Abbildung 4.8: Bildliche Veranschaulichung des 1-Punkt-Crossovers. Der erste Abschnitt wird beim ersten Nachkommen vom ersten Elternindividuum übernommen, der zweite vom Zweiten. Für den zweiten Nachkommen wird das jeweils andere Elternindividuum gewählt.

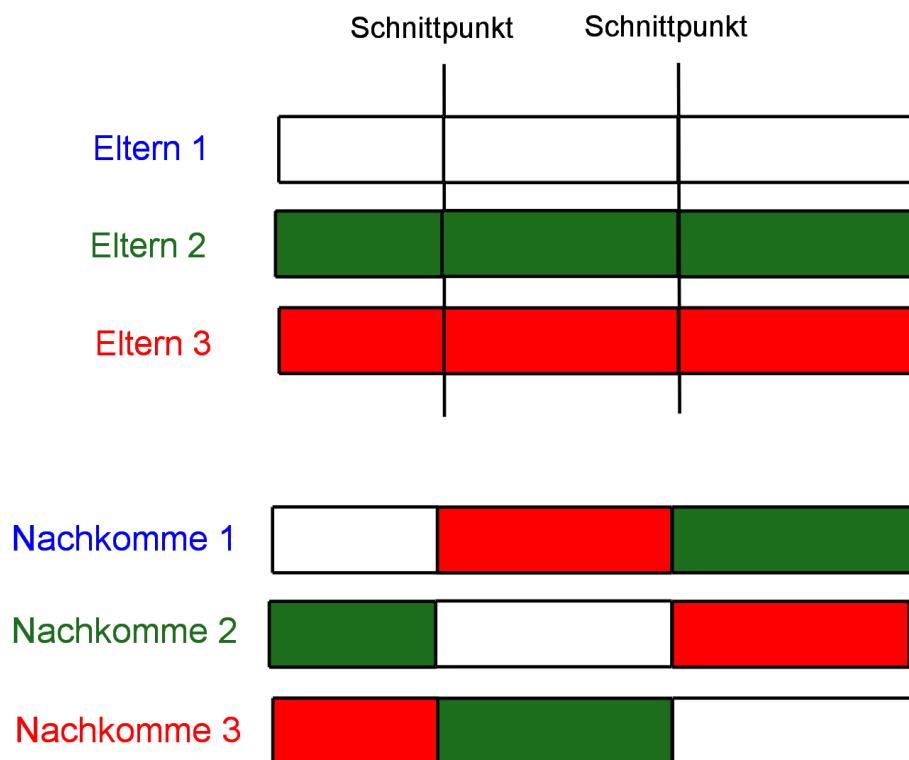


Abbildung 4.9: Bildliche Veranschaulichung des Diagonal-Crossovers. Für das  $i$ . Kindindividuum wird für den ersten Abschnitt das Teilstück vom  $i$ . Elternindividuum übernommen. Für den zweiten Abschnitt das Teilstück vom  $i + 1 \bmod m$  ten Elternindividuum.

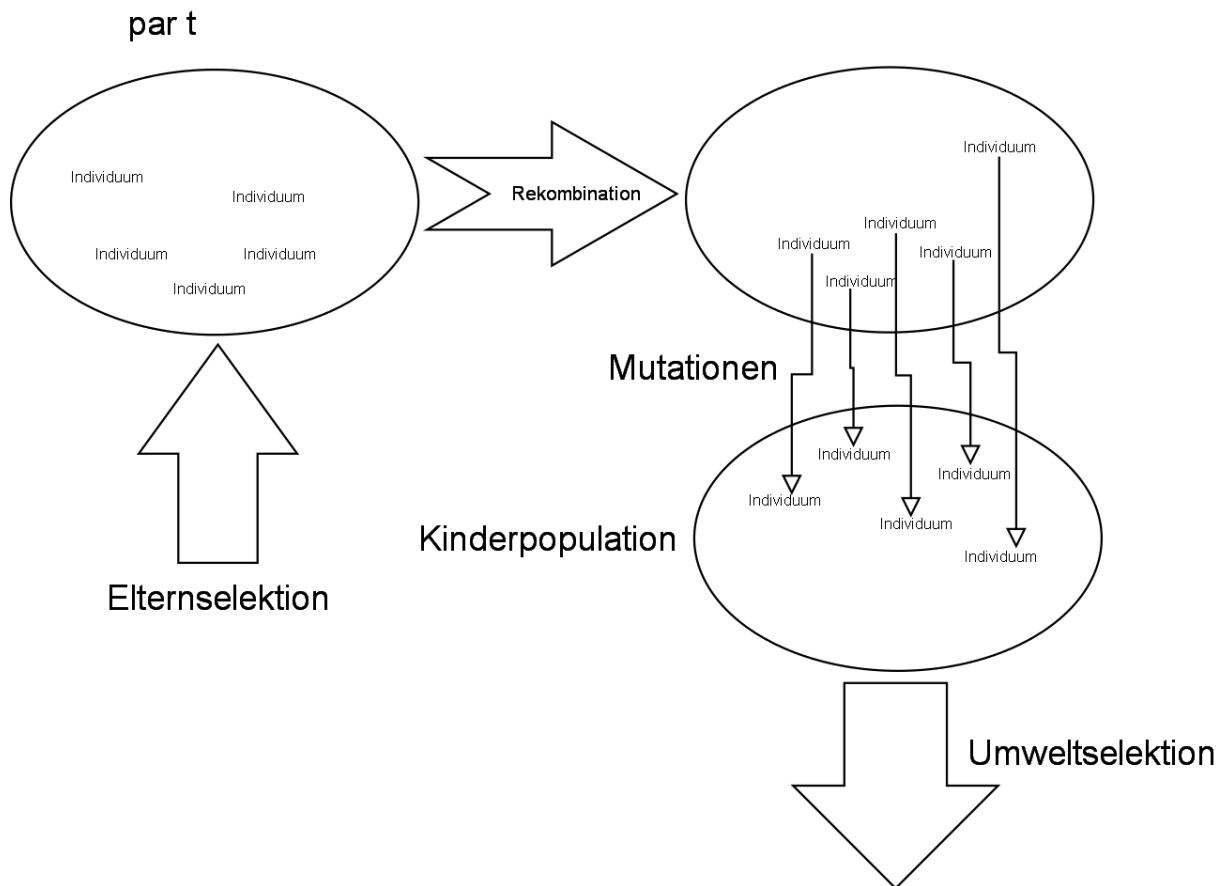


Abbildung 4.10: Nach der Elternselektion und der Rekombination wird die Mutation durchgeführt. Anschließend wird die Umweltselektion durchgeführt. Die Mutation arbeitet nicht mit Populationen, sondern mit einzelnen Individuen.

## 4.8 Funktion von Mutation und Rekombination

In einem Evolutionären Algorithmus müssen die Evolutionären Operatoren (Mutation und Rekombination) zwei Rollen übernehmen. Während der Laufzeit des Evolutionären Algorithmus können die beiden Operatoren die Rollen tauschen.

**explorativ** Der evolutionäre Operator erkundet den Raum aller möglichen Lösungen des Optimierungsproblems. Das von ihm erzeugte Individuum kann dabei im Bezug auf eine Nachbarschaft von Genotypen und auf die Güte des Individuums weit von den Eingabe-Individuen entfernt sein.

**feinabstimmend** Der evolutionäre Operator versucht in der Nähe des ursprünglichen Individuums ein besseres Individuum zu finden. Dabei wird der Genotyp nur wenig verändert. Auch die Änderung des Gütewerts des entstandenen Individuums sollte geringfügig sein.

Ist die Diversität in der Population hoch, so ist die Wahrscheinlichkeit, dass durch die Rekombination zwei gleiche Individuen kombiniert werden ziemlich gering. In diesem Fall gilt:

Die Rekombination erstellt neue Individuen, deren Genotyp aus relativ guten Individuen der Vorgängergeneration zusammengesetzt sind. Durch dieses Zusammensetzen des Genotyps des Kindindividuum wird der Raum der möglichen Lösungen des Optimierungsproblems  $\Omega$  erkundet. Die Rekombination übernimmt die Rolle des explorativen Operators der Evolution, durch sie soll sichergestellt werden, dass alle möglichen Lösungen des Problems vom Genetischen Algorithmus auch erreicht werden können.

Die Mutation ändert beim Genetischen Algorithmus nur wenige Bits eines Individuums (typischerweise jedes 10. bis 100. Bit). Durch die Änderung weniger Bits sollte sich bei einem Genetischen Algorithmus die Güte (der Fitnesswert) des Individuums auch nur gering ändern. Dann übernimmt die Mutation die Rolle der Feinabstimmung. Wenn schon viele Lösungen gefunden wurden, die nahe an einem Optimum liegen sorgt sie dafür, dass durch kleine Änderungen schließlich auch das Optimum gefunden wird.

Wenn die Diversität in der Population niedrig ist, wenn z. B. ein Individuum die Population übernommen hat (die Population nur noch aus Individuen mit gleichem Genotyp besteht) ändern sich die Rollen der evolutionären Operatoren:

Nun werden meistens gleiche Individuen rekombiniert. Bei der Rekombination entstehen also nur noch selten neue Individuen (Individuen, mit anderem Genotyp), die noch nicht in der Population enthalten waren. Die Rekombination verliert ihre explorativen Rolle.

Die Mutation ist nun der einzige evolutionäre Operator, der eine Erkundung des Suchraumes leisten kann!

Wegen dieses Verhaltens muss in einem Genetischen Algorithmus, wie in Evolutionären Algorithmen allgemein, dafür gesorgt werden, dass der zweite Fall nur sehr selten eintritt. Um dies zu erreichen gibt es verschiedene Ansätze:

- Veränderung der Mutations- und Rekombinationswahrscheinlichkeiten, wenn die Population von einem Individuum übernommen wird.
- Verbieten von gleichen Individuen in der Population.
- Einstellung der Mutations- und Rekombinationswahrscheinlichkeit, damit die Population nicht von einem Individuum übernommen werden kann.

# Der Genetische Algorithmus zur Wortsuche

---

Im Folgenden werden verschiedene Aspekte des Genetischen Algorithmus zur Wortsuche angesprochen. In den Beschreibungen werden die Individuen häufig synonym zu den Suchwörtern benutzt, die sie repräsentieren, um die Beschreibungen nicht unnötig zu komplizieren.

## 5.1 Unterschiede zum allgemeinen Genetischen Algorithmus

Bei der Umsetzung des Genetischen Algorithmus für dieses konkrete Problem wurden einige Entwurfsentscheidungen getroffen, die in diesem Abschnitt angesprochen werden.

### 5.1.1 $\lambda > \mu$ Kinder

Im implementierten Genetischen Algorithmus werden  $\lambda$  viele Kinder erzeugt. Im Gegensatz zur Beschreibung in 4.2 gilt  $\lambda > \mu$ , es können mehr Kinder erzeugt werden, als Eltern in der vorhergehenden Population vorhanden waren. Die Größe von  $\lambda$  ist hierbei nicht festgelegt und kann für jeden Durchlauf den Genetischen Algorithmus zur Wortsuche neu festgelegt werden. Dieser Parameter wurde bei den Parametereinstellungen nicht ausgemessen, sondern aus der Literatur der Evolutionsstrategien ([Nis97e]) mit  $\lambda = 5\mu$  übernommen.

Dadurch, dass die Population der Kindindividuen 5-mal größer ist, als die Population, aus der die Eltern selektiert werden ergibt sich der Zwang, neben der ersten Selektion des Genetischen Algorithmus eine zweite Selektion zu bestimmen. Diese Selektion wählt aus der Population der Kinder wieder  $\mu$  Individuen aus, die für den nächsten Evolutionszyklus zur Auswahl der Eltern benutzt werden sollen. Diese zweite Selektion ist die Umweltselektion.

### 5.1.2 Der Generationenschritt

Im allgemeinen Genetischen Algorithmus werden Elternpaare aus der aktuellen Population  $pop_t$  ausgewählt, die zur Erzeugung der Kindindividuen benutzt werden. Im Genetischen Algorithmus zur Wortsuche wurde diese Paarung der Elternindividuen vermieden. Stattdessen wird eine zweite Population  $par_t$  benutzt, in die die Individuen eingetragen werden, die von der Elternselektion ausgewählt wurden. Die Individuen in dieser Population werden für die weiteren Schritte des Evolutionszyklus benutzt. Die Individuen aus  $pop_t$ , die nicht in  $par_t$  eingetragen wurden, sind für

die weiteren Schritte des Evolutionszyklus nicht mehr benutzbar.  $pop_e$  ist somit die Population der Elternindividuen. Die Population wird auch Elternpopulation genannt.

Eine weitere Besonderheit des Genetischen Algorithmus zur Wortsuche ist, dass immer die  $n$  besten Individuen aus der Vorgängergeneration in die Population  $pop_{t+1}$  übernommen werden. Dieses Verhalten ist sinnvoll, da durch den Genetischen Algorithmus zur Wortsuche das beste Suchwort gefunden werden soll. Es wird durch die Übernahme der  $n$  Elite-Individuen aus  $pop_t$  sichergestellt, dass ein einmal gefundenes sehr gutes Suchwort während des Genetischen Algorithmus zur Wortsuche nicht mehr verloren geht. Der Ablauf lässt sich in 5 Schritte unterteilen:

1. Aus  $pop_t$  werden mit der Elternselektion solange Individuen in  $par_t$  hinzugefügt, bis diese eine vorgegebene Größe erreicht hat (normalerweise  $\mu$  Individuen).
2. Als  $par_t$  werden mit der Zufallsselektion die Elternindividuen ausgewählt, die mit dem Rekombinationsoperator zur Erzeugung von Kindindividuen verwendet werden. Dies wird solange durchgeführt, bis  $\lambda$  Kinder erzeugt wurden.
3. Mit der Umweltselektion werden aus diesen  $\lambda$  vielen Individuen  $\mu$  viele ausgewählt.
4. Die  $\mu$  Individuen werden in die Population  $pop_{t+1}$  eingefügt.
5. Die schlechtesten Individuen aus  $pop_{t+1}$  werden durch Kopien der  $n$  besten Individuen aus  $pop_t$  ersetzt.

Es kann also ein Elternindividuum aus  $par_t$  nicht, einmal oder mehrmals zur Rekombination mit anderen Elternindividuen herangezogen werden, wenn die Auswahl aus der Elternmenge mit zurücklegen ausgeführt wird. Der Ablauf wird nochmal im Algorithmus 5 verdeutlicht.

```

Algorithmus 5 while (abbruchbedingung.weitermachen(this, generation))
{
  // Aus der Population die Eltern auswaehlen
  Population eltern = fuehreElternselektionAus(pop);
  // Aus der Elternpopulation die Kinder auswaehlen
  Population kinder = new Population();
  while (kinder.size() < lambda)
  {
    // Erzeuge Menge der Eltern fuer eine Rekombination.
    Collection<AbstractIndividuum> e = waehleEltern(par);
    Collection<AbstractIndividuum> k = rekombination.
      berechneRekombination(e, factory);

    // mutiere Kinder
    foreach (AbstractIndividuum i in k)
    {
      mutation.berechneMutation(i);
      // jetzt das Kind noch in die neue Population einfuegen.
      kinder.add(i);
    }
  }
  // Kinderpopulation fertig.
  // Die Umweltselektion auf die neue Population anwenden und
  // Ausgangspopulation fuer naechsten durchlauf erzeugen.
  Population popNext = this.fuehreUmweltselektionAus(kinder, popSize);
  // Hier ElitenEvolution, also die best Individuen direkt
  // aus pop uebernehmen.
  popNext.entferneDieNKleinstenIndividuen(n,bewertungsfunktion);
  List<AbstractIndividuum> list = pop.gibIndividuumListe();
  list.Sort(this.bewertungsfunktion);
  for (int i = 1 ; i <= best ; i++)
  {
    System.Console.Out.WriteLine(list[list.Count - i]);
    popNext.add(list[list.Count - i]);
  }
  pop = popNext;
  // Eine Generation mehr.
  generation++;
}

```

Mit `fuehreElternselektionAus` werden die Individuen in die Elternpopulation `par` selektiert. Dann werden solange mit `waehleEltern` so viele Individuen aus `par` ausgewählt, wie für einen Rekombination erforderlich sind bis  $\lambda$  viele Kinder erzeugt wurden. In `berechneRekombination` werden die Kinder aus den Elternindividuen mit einer Crossover-Rekombination erzeugt und schließlich, in `berechneMutation` mutiert. Anschließend wird mit der Umweltselektion `fuehreUmweltselektionAus` die Population für den nächsten Durchlauf durch den Evolutionszyklus erzeugt. Dann werden die  $n$  besten Individuen aus  $pop_t$  in  $pop_{t+1}$  übernommen. Wichtig ist, dass bei der Rekombination immer neue Individuen entstehen, auch wenn der Genotyp der Elternindividuen ohne Rekombination übernommen wird. Es werden neue Individuen erzeugt, die sich in der Zusatzinformation von den Elternindividuen unterscheiden. Es wird hier eine Komma-Strategie verwendet, da die Individuen andere sind, obwohl der Genotyp gleich dem der Elternindividuen sein kann.

In `berechneRekombination` und `berechneMutation` wird probabilistisch entschieden, ob eine Rekombination oder eine Mutation durchgeführt werden soll oder nicht. Wenn nicht, werden die Individuen ohne Mutations- oder Rekombinationsoperation erzeugt.

### 5.1.3 Hillclimbing

Wenn der Evolutionäre Algorithmus seine Berechnungen beendet hat, wird mit dem besten Individuum noch eine weitere Suche durchgeführt. Vom besten gefundenen Individuum aus (und nur von diesem) werden alle Nachbarindividuen betrachtet und das beste Individuum aus der Nachbarschaft übernommen. Als Nachbarindividuen betrachten wir alle Individuen, die durch das Austauschen eines Zeichens des Suchwortes entstehen können (siehe 3.4.1). Anschließend wird von diesem Individuum aus die gesamte Nachbarschaft dieses Individuums betrachtet bis in der Nachbarschaft, des aktuellen Individuums kein besseres Individuum mehr vorhanden ist. Dann bricht die Suche ab. Man kann sich diese Suche wie den Weg eines Bergsteigers vorstellen, der immer den Berg hinauf geht, daher auch der Name dieses Verfahrens. So wird ein lokales Maximum gefunden.

Neben der obigen Definition der Nachbarschaft sind für die Suche noch andere Nachbarschaften denkbar, wurden aber im Rahmen dieser Arbeit nicht untersucht:

- Individuen, die durch Verschiebungen des Suchwortes nach links und Auffüllen mit einem bestimmten Buchstaben des Alphabets entstehen.
- Individuen, die durch zyklische Verschiebung des Suchwortes nach links entstehen.

Auch die Entscheidung welches Individuum das Beste ist, hängt prinzipiell von der Bewertungsfunktion ab. Die Bewertungsfunktion ist aber durch die Größe des Boyer-Moore-Automaten für das gegebene Suchwort fest vorgegeben.

### 5.1.4 Die Elternpopulation

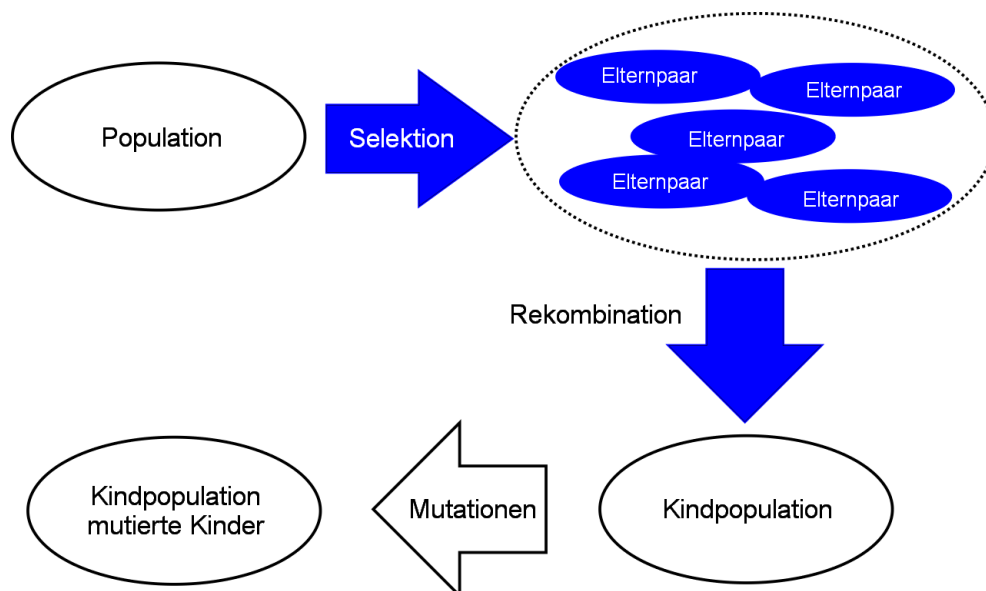


Abbildung 5.1: Der traditionelle Ablauf der Auswahl der Eltern.

Im Gegensatz zum allgemeinen Genetischen Algorithmus werden bei der Elternselektion keine Tupel von Individuen (z. B. 2-Tupel oder 3-Tupel wenn 3 Eltern bei der Rekombination erwartet werden) aus der Population gezogen, dies wird in Abbildung 5.1 gezeigt.

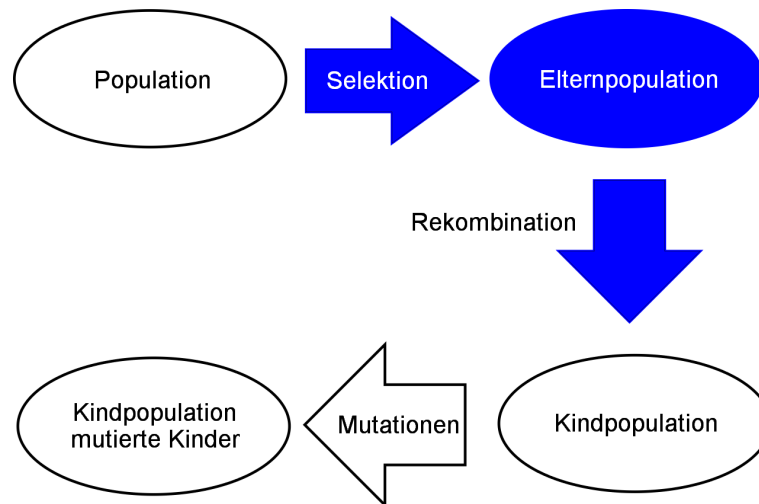


Abbildung 5.2: Die Elternpopulation im Ablauf des Evolutionszyklus.

Stattdessen wird eine weitere Population von Eltern eingefügt, wie in Abbildung 5.2 zu sehen (im Quellcode 5 par genannt). Die Elternselektion baut diese Population der Eltern (Elternpopulation) auf. Anschließend werden aus dieser Elternpopulation bei der Rekombination zufällig Individuen gezogen, die für die Rekombination als Elternindividuen benutzt werden. Dieses Ziehen der Elternindividuen kann mit oder ohne zurücklegen erfolgen. Durch die separate Elternpopulation ist die Selektion nicht mehr direkt mit der Rekombination verbunden. Die Selektion wählt eine bestimmte Anzahl an Individuen als Eltern und die Rekombination wählt aus diesen Eltern wieder aus. Wieviele Elternindividuen für eine Rekombination benötigt werden ist für die Selektion nicht wichtig.

## 5.2 Mutationen

In den Testläufen werden einige verschiedene Mutationen getestet, um den geeigneten Operator für die Wortsuche zu finden. Der Genetischen Algorithmus zur Wortsuche entscheidet für jedes Individuum, ob es mutiert werden soll oder nicht. Dazu wird eine Zufallszahl zwischen 0 und 1 gezogen. Wenn die gezogene Zufallszahl größer ist als die Mutationswahrscheinlichkeit, so wird das Individuum mutiert, wenn die Zahl kleiner ist, so wird das Individuum nicht mutiert. In den folgenden Abschnitten werden die verschiedenen Abläufe beschrieben, die auf einem Individuum durchgeführt werden, wenn es mutiert werden soll.

### 5.2.1 Bit-Flip-Mutation

Bei der **Bit-Flip-Mutation** wird im Allgemeinen eine bestimmte Anzahl von Bits des Individuums geändert. Das bedeutet, dass sich der Wert dieser Bits invertiert ( $x_{neu} = \neg x_{alt}$ ).

Bei dem Genetischen Algorithmus zur Wortsuche wird der Genotyp eines Individuums mutiert. Dieser Genotyp ist das Suchwort  $w$ , das dieses Individuum darstellt. Es werden eines oder mehrere Zeichen dieses Suchwortes geändert und dadurch das Suchwort mutiert. Die Standardeinstellung ist, dass 10 Prozent aller Zeichen des Suchwortes geändert werden. Wenn  $x$  Zeichen des Suchwortes geändert werden sollen, werden  $x$  Zufallszahlen zwischen 1 und der Länge des Suchwortes  $m$  gezogen. An diesen Stellen wird das Zeichen, das an dieser Stelle im Suchwort  $w$  stand, durch ein anderes Zeichen aus  $\Sigma$  ersetzt. Der Code zu dieser Mutation ist im Algorithmus 6 zu sehen.

```
Algorithmus 6 void mutiere(AbstractIndividuum a)
{
// Die Buchstaben holen.
Collection<long> buchstaben = a.gibAlphabet();
long[] bArray = new long[buchstaben.Count - 1];
// Stellen fuer die Mutation suchen und dort zufaellig Buchstaben
// eintragen. Es sollen prozentual viele Stellen getauscht werden.
for (int i = 0 ; i < (((double)a.gibLaenge() * mutationsatz)) ; i++)
{
    int x = (int)Zufallsgenerator.getInstance().next(a.gibLaenge() - 1);
    // Das Zeichen aussuchen, das eingesetzt werden soll.
    // zuerst das Zeichen entfernen, das an der Stelle noch steht.
    long altesZeichen = a.gibZeichenLong(x);
    buchstaben.Remove(altesZeichen);
    buchstaben.CopyTo(bArray, 0);
    long zeichen = Zufallsgenerator.getInstance().gibZufaelligesZeichen(bArray);
    buchstaben.Add(altesZeichen);
    a.setzeZeichen(x, zeichen);
}
}
```

Durch das Ziehen der  $x$  Zufallszahlen kann es vorkommen, dass weniger als  $x$  Zeichen des Suchwortes geändert werden. Dies wird in Kauf genommen, um den Algorithmus einfach zu halten.

### 5.2.2 Wurzel-BitFlip-Mutation

Die **Wurzel-BitFlip-Mutation** ist eine spezielle Variante der **BitFlip-Mutation**. Auch bei der **Wurzel-BitFlip-Mutation** werden Zeichen einzeln verändert.

Bei der **Wurzel-BitFlip-Mutation** wird für jedes Zeichen mit einer Zufallszahl entschieden, ob dieses Zeichen geändert werden muss. Die Wahrscheinlichkeit für ein einzelnes Zeichen wird in `mutationswkeit` abgespeichert als  $mutationswkeit = \frac{\sqrt{n}}{n}$ , wobei  $n$  die Länge des Suchwortes ist.

Wenn nun die gezogene Zufallszahl für das  $i$ . Zeichen (`zg.nextDouble()`) kleiner als `mutationswkeit` ist, wird das  $i$ . Zeichen auf ein zufälliges Zeichen gesetzt. Dabei kann das Zeichen an  $i$ . Stelle geschrieben werden, das vorher an der  $i$ . Stelle stand.

Dieser Vorgang wird, wie im Algorithmus 7 zu sehen für alle Zeichen des Suchwortes des Individuums durchgeführt.

**Algorithmus 7** *protected override void mutiere(AbstractIndividuum i)*

```

{
    double mutationswkeit = Math.Sqrt(i.gibLaenge())/i.gibLaenge();
    long[] alphabet = new long[i.gibAlphabet().Count];
    i.gibAlphabet().CopyTo(alphabet,0);
    Zufallsgenerator zg = Zufallsgenerator.getInstance();
    for (int j=0; j < i.gibLaenge() ; j++)
    {
        if (zg.nextDouble() < mutationswkeit)
        {
            i.setzeZeichen(j, zg.gibZufaelligesZeichen (
                alphabet));
        }
    }
}

```

**5.2.3 Suffixmutation**

Die **Suffix-Mutation** wählt ein Suffix des aktuellen Wortes  $w$  aus, das frühestens ab der Mitte des Suchwortes beginnt und schreibt es an einer zufälligen Stelle wieder in das Individuum.

Diese Mutation beruht auf der notwendigen Bedingung für ein Suchwort mit großem BMA, dass es möglichst viele Teilstücke in diesem Wort gibt, die gleich einem Suffix des Wortes sind (siehe 2.2.3). Durch die Mutation sollen viele solche Symmetrien im Suchwort entstehen. Diejenigen, die sinnvoll sind werden dann durch die Selektion in die nächste Generation übernommen. Im Algorithmus 8 ist die Berechnung der Mutation zu sehen.

**Algorithmus 8** *void mutiere(AbstractIndividuum a)*

```

{
    Zufallsgenerator zg = Zufallsgenerator.getInstance();
    int laenge = zg.next(a.gibLaenge() / 2);
    int pos = zg.next(a.gibLaenge() - 1);
    long[] tmp = new long[laenge];
    Array.Copy(a.gibPatternLong(), a.gibLaenge() - laenge - 1, tmp, 0, laenge);
    for (int i = 0 ; i < laenge ; i++)
    {
        a.setzeZeichen(i + pos % (a.gibLaenge() - 1), tmp[i]);
    }
}

```

Mit dem Zufallsgenerator werden über die Methoden `next(max)` Zufallszahlen im Bereich  $\{0, \dots, max\}$  erzeugt. Mit dieser Methode wird die Länge des Suffix `laenge` und die Position `pos`, ab der der Suffix wieder in das Suchwort eingetragen werden soll bestimmt. Anschließend wird der Suffix in einen Zwischenspeicher kopiert und ab der Stelle `pos` wieder in das Suchwort geschrieben. Wenn beim Rückschreiben über die letzte Position des Suchwortes hinaus geschrieben wird, wird am Anfang das Suchwortes weitergeschrieben.

**5.2.4 Verschiebemutation**

Die **Verschiebemutation** verschiebt das Suchwort  $w$  zyklisch um eine zufällige Anzahl von Stellen.

Im Gegensatz zur **Bit-Flip-Mutation** bleiben die Nachbarschaften zwischen den einzelnen Zeichen des Suchwortes bestehen, da für alle benachbarten Zeichen im verschobenen Suchwort

$w'$  gilt, dass sie auch in  $w$  benachbart waren, bis auf das Paar, das in  $w$  noch an der ersten und letzten Stelle gestanden ist.

Die Hoffnung bei dieser Mutation ist, dass sich die Bewertung von derart mutierten Individuen nicht so stark ändert wie bei der **Bit-Flip-Mutation**. Im Algorithmus 9 ist die Berechnung dieser Mutation zu sehen.

```
Algorithmus 9 void mutiere(AbstractIndividuum i)
{
    Zufallsgenerator zg = Zufallsgenerator.getInstance();
    // mutation durchfuehren.
    int laenge = i.gibLaenge();
    int verschiebung = zg.next(0, laenge - 1);
    long[] tmp = new long[laenge];
    for (int j = 0; j < laenge; j++)
    {
        tmp[(j + verschiebung) % (laenge - 1)] = i.gibZeichenLong(j);
    }
    for (int x = 0; x < tmp.Length; x++)
    {
        i.setzeZeichen(x, tmp[x]);
    }
}
```

Auch hier wird mit der Methode `next` des Zufallsgenerators eine Zufallszahl zwischen 0 und der Länge des Suchwortes erzeugt. Diese Zufallszahl gibt an, um wie viele Zeichen das Suchwort zyklisch nach rechts verschoben werden soll. Die Verschiebung wird zunächst in einem Zwischenspeicher (`tmp`) durchgeführt und dann in das Individuum zurückgeschrieben.

### 5.2.5 Bestenmutation

Bei der **Bestenmutation** werden zuerst mit einem Genetischen Algorithmus die Besten, (oder hoffentlich wenigstens gute) Suchwörter  $w'$  bis zu einer bestimmten kürzeren Länge gefunden. Anschließend werden bei einer Mutation diese besten Suchwörter mit kürzerer Länge an einer beliebigen Stelle in das zu mutierende Suchwort  $w$  eingetragen. Standardmäßig werden mit einem Genetischen Algorithmus die besten Suchwörter der Längen 4 bis 7 gesucht. Es wird ein Genetischer Algorithmus mit den Einstellungen, wie in Tabelle 6.1 zu sehen, benutzt.

Populationsgröße	20
Elternpopulationsgröße	40
Kinderpopulationsgröße	100
Elternselektion	2-Turnierselektion
Umweltselektion	4-Turnierselektion
Mutation	Bit-Flip-Mutation
Rekombination	1-Punkt-Crossover
MutW'keit pro Individuum	0.5
RekombinationsW'keit	0.7
Anzahl Generationen	100

Tabelle 5.1: Parametereinstellungen des Genetischen Algorithmus für die Bestenmutation.

Es werden keine Eliteindividuen zusätzlich in die nächste Generation übernommen.

Wir gehen bei der Mutation davon aus, dass die besten gefundenen Individuen der kürzeren Längen in einem 2-dimensionalen Array `best` abgespeichert sind. Die erste Dimension gibt die Länge der Suchwörter an, in der zweiten Dimension sind die gefundenen Suchwörter abgelegt. Im Algorithmus 10 ist der Code zur Berechnung der Mutation zu sehen.

```
Algorithmus 10 void mutiere(AbstractIndividuum ind)
{
    // Zufällig auswählen, welche laenge eingefuegt wird.
    // Anschliessend an einer zufaelligen Stelle einfuegen.
    Zufallsgenerator zg = Zufallsgenerator.getInstance();
    // Einfuegepunkt auswählen.
    int laenge = zg.next(min, max);
    int punkt = zg.next(0, ind.gibLaenge() - 1 - laenge);
    // Zeichen austauschen.
    // Ein beliebiges gutes Pattern auswählen.
    int j = zg.next(best[laenge - min].Count - 1);
    for (int i = punkt; i < (punkt + laenge); i++)
    {
        ind.setzeZeichen(i, best[laenge - min][j].gibZeichenLong(i - punkt));
    }
}
```

Mit dem Zufallsgenerator wird zuerst die Länge des Suchworts ausgewählt, dann die Stelle, ab der es in das Suchwort des Individuums eingetragen werden soll. Schließlich wird eines der besten gefundenen Individuen ausgewählt, das in das Suchwort des zu mutierenden Individuums eingetragen wird.

Diese Mutation beruht auf der Beobachtung, dass die besten Suchwörter manchmal die besten (oder gute) Suchwörter kürzerer Länge beinhalten.

## 5.3 Rekombinationen und Crossoveroperatoren

Die Rekombination kombiniert die Elternindividuen zu neuen Kindindividuen. Da es sich hier um einen Genetischen Algorithmus handelt, sind die Rekombinationen in diesem Kapitel alle Crossoveroperatoren. Im Genetischen Algorithmus zur Wortsuche wird eine Population mit Elternindividuen übergeben, und es wird eine Population mit Kindern erzeugt. In den meisten Fällen kann der Crossover mit 2 oder mehr Eltern ausgeführt werden und erzeugt bei jeder Ausführung so viele Kindindividuen, wie er Elternindividuen übergeben bekam. Bei Rekombinationen, die sich anders verhalten, wird dies extra erwähnt.

### 5.3.1 1-Punkt-Crossover

Diese Rekombinationsoperation wird als Vergleich implementiert. Er entspricht dem in 4.7.2 besprochenen 1-Punkt-Crossover. Die anderen Verfahren müssen mindestens so gut sein, wie dieser Crossoveroperator, um noch näher betrachtet zu werden.

Bei dem 1-Punkt-Crossover handelt es sich um einen Standard-Operator. Er verwendet kein problemspezifisches Wissen. Bei dem 1-Punkt-Crossover wird ein Schnittpunkt im neu entstehenden Kindindividuum definiert. Links von diesem Schnittpunkt werden die Informationen des ersten Elternindividuum übernommen, rechts vom Schnittpunkt werden die Informationen des

zweiten Elternindividuum übernommen. Bei diesem Crossover gehen wir davon aus, dass nur zwei Elternindividuen zur Erzeugung der Kindindividuen zur Verfügung stehen.

Wenn nun  $e^1 = e_0^1 \dots e_{m-1}^1$  und  $e^2 = e_0^2 \dots e_{m-1}^2$  die Elternindividuen sind, dann gibt es ein  $s \in 0..m-1$  so, dass für das Kind  $k = k_0 \dots k_{m-1}$  gilt:  $k_i = e_i^1$  für  $i \leq s$  und  $k_i = e_i^2$  für  $i > s$ .

Die Individuen, die rekombiniert werden sollen, werden als Menge (`Collection`) übergeben. Die `factory` wird zur Erzeugung der rekombinierten Kinder benötigt. Als Ergebnis wird eine Menge von rekombinierten Kindindividuen zurückübergeben. Dieser Ablauf ist im Algorithmus 11 genauer zu sehen.

**Algorithmus 11** *Collection<AbstractIndividuum> rekombiniere(Collection<AbstractIndividuum> eltern, AbstractIndividuumFactory factory)*

```
{
  IEnumerator<AbstractIndividuum> enumerator = eltern.GetEnumerator();
  enumerator.MoveNext();
  AbstractIndividuum a = enumerator.Current;
  enumerator.MoveNext();
  AbstractIndividuum b = enumerator.Current;
  Collection<AbstractIndividuum> erg = new Collection<AbstractIndividuum>();
  // Die Strings fuer die Erzeugung der Kindindividuen aufbauen.
  long[] kind1 = new long[a.gibLaenge()];
  long[] kind2 = new long[a.gibLaenge()];
  // Stelle fuer den ersten und einzigen Schnitt finden. Wir runden.
  int stelle = (int) Zufallsgenerator.getInstance().nextNormalverteilt
               (a.gibLaenge() / 2, a.gibLaenge() / 4);
  if (stelle < 0 || stelle > a.gibLaenge() - 1)
  {
    stelle = a.gibLaenge() / 2;
  }
  // jetzt die beiden Kindindividuen bauen und zurueckgeben.
  for (int i = 0; i < stelle; i++)
  {
    kind1[i] = a.gibZeichenLong(i);
    kind2[i] = b.gibZeichenLong(i);
  }
  // hier der Schnitt. Ab jetzt die Eltern vertauschen.
  for (int i = stelle; i < kind1.Length; i++)
  {
    kind2[i] = a.gibZeichenLong(i);
    kind1[i] = b.gibZeichenLong(i);
  }
  erg.Add(factory.ErzeugeIndividuum(kind1));
  erg.Add(factory.ErzeugeIndividuum(kind2));
  return erg;
}
```

Mit dem Zufallsgenerator wird die Stelle des Crossovers (`stelle`) bestimmt. Anschließend werden zwei neue Individuen (`kind1` und `kind2`) erzeugt und die Zeichen der Suchwörter der Elternindividuen in die Suchwörter der Kindindividuen übertragen. Aus den Suchwörtern für die Kindindividuen werden dann die Kindindividuen mit der `factory` erzeugt. Die erzeugten Kindindividuen werden in der Menge `erg` zurückgegeben.

### 5.3.2 Uniform-Crossover

Auch dieser Crossoveroperator wird als Vergleich implementiert. Er entspricht den in 4.7.2 besprochenem **Uniform-Crossover**. Es soll auch untersucht werden, ob dieses Verfahren besser ist als der 1-Punkt-Crossover.

Bei dem Uniform-Crossover wird für jedes Zeichen  $k_i$  des Kindindividuum  $k = k_0 \dots k_{m-1}$  zufällig ein Elternindividuum gewählt, von dem es übernommen wird. Die Wahrscheinlichkeit, dass das  $i$ . Zeichen von dem ersten, oder dem zweiten Elternindividuum übernommen wird ist dabei jeweils  $p_{ux} = 0.5$ . Der Aufwand für diesen Crossover ist im Vergleich zum **1-Punkt-Crossover** sehr hoch, da viele Zufallszahlen benutzt werden. Da allerdings im Genetischen Algorithmus zur Wortsuche der Aufwand zum Aufbau des Boyer-Moore-Automaten eindeutig überwiegt, könnte sich dieses Verfahren bei besseren Ergebnissen trotz des höheren Aufwands bei dem Crossover durchsetzen. Bei dem Uniform-Crossover werden sämtliche lokalen Zusammenhänge der Zeichen aus beiden Eltern zerstört, so dass dieses Verfahren nicht sehr vielversprechend zu sein scheint. Im Algorithmus 12 ist der Ablauf des **Uniform-Crossover** zu sehen.

**Algorithmus 12** *Collection<AbstractIndividuum> rekombiniere(Collection<AbstractIndividuum> eltern, AbstractIndividuumFactory factory)*

```

{
  Collection<AbstractIndividuum> erg = new Collection<AbstractIndividuum>();
  Zufallsgenerator zg = Zufallsgenerator.getInstance();
  AbstractIndividuum[] indArr = new AbstractIndividuum[eltern.Count];
  eltern.CopyTo(indArr,0);
  long[] pat = new long[indArr[0].gibLaenge()];
  // nun fuer jedes einzelne Zeichen des Patterns ein Eltern
  // aussuchen und das Zeichen kopieren.
  for (int i = 0 ; i < pat.Length ; i++)
  {
    pat[i] = indArr[zg.next(indArr.Length - 1)].gibZeichenLong(i);
  }
  erg.Add(factory.ErzeugeIndividuum(pat));
  return erg;
}

```

Hier wird für jedes Zeichen separat eine Zufallszahl gezogen, die angibt von welchem Elternindividuum dieses Zeichen des Kindindividuum übernommen werden soll. Bei dieser Implementierung können auch mehr als 2 Elternindividuen in der Elternmenge vorhanden sein, dann werden auch wieder so viele Kindindividuen erzeugt wie Elternindividuen in der Elternmenge vorhanden waren.

### 5.3.3 Linearer-Crossover

Bei dem linearen-Crossover handelt es sich um einen **Diagonal-Crossover** mit  $n = 2$  und  $m = 2$ , wie in 4.7.2 beschrieben. Es werden zwei Punkte im Suchwort bestimmt, die drei Abschnitte unterteilen. Für jeden dieser 3 Abschnitte wird entschieden von welchem Elternindividuum die Zeichen des Suchwortes in diesen Abschnitt übernommen werden.

**Algorithmus 13** *Collection<AbstractIndividuum> rekombiniere(Collection<AbstractIndividuum> eltern, AbstractIndividuumFactory factory)*

```

{
  Collection<AbstractIndividuum> erg = new Collection<AbstractIndividuum>();
  Zufallsgenerator zg = Zufallsgenerator.getInstance();

```

```

IEnumerator<AbstractIndividuum> en = eltern.GetEnumerator();
en.MoveNext();
AbstractIndividuum a = en.Current;
en.MoveNext();
AbstractIndividuum b = en.Current;
// Wo soll dieses Stueck anfangen?
int anfang = zg.next(a.gibLaenge() - 1);
// Wo soll dieses Teilstueck eingefuegt werden?
int pos = zg.next(a.gibLaenge() - 1);
// jetzt die beiden Patterns fuer die Kinder aufbauen.
long[] kind1 = erzeugeRekombinationsPattern(a, b, anfang, pos);
long[] kind2 = erzeugeRekombinationsPattern(b, a, anfang, pos);
// Kinderindividuen bauen und zurueckgeben.
erg.Add(factory.ErzeugeIndividuum(kind1));
erg.Add(factory.ErzeugeIndividuum(kind2));
return erg;
}

```

**Algorithmus 14** `long[] erzeugeRekombinationsPattern(AbstractIndividuum a, AbstractIndividuum b, int anfang, int pos)`

```

{
    long[] neu = new long[a.gibLaenge()];
    // jetzt dieses Stueck ausschneiden
    long[] tmp = new long[laenge];
    for (int i = 0 ; i < laenge ; i++)
    {
        // Wenn das Stueck ueber das Ende des Patterns laeuft geht es
        // vorne am Pattern weiter.
        tmp[i] = b.gibZeichenLong(i % (b.gibLaenge() - 1));
    }
    // jetzt das neue Pattern schreiben.
    for (int j = 0 ; j < neu.Length ; j++)
    {
        if (j >= pos && j <= pos + laenge - 1)
        {
            // hier das Patternstueck von b uebernehmen
            neu[j] = tmp[j - pos];
        }
        else
        {
            neu[j] = a.gibZeichenLong(j);
        }
    }
}
return neu;
}

```

In der Implementierung wird davon ausgegangen, dass ein Teilstück des Elternteils **a** in das erste Kindindividuum **kind1** übernommen wird. Die restlichen Zeichen werden durch die entsprechenden Zeichen aus dem zweiten Elternindividuum **b** aufgefüllt. Für das zweite Kind werden die Zeichen des Teilstücks von **b** übernommen, während die restlichen Zeichen von **a** aufgefüllt werden.

### 5.3.4 Abschnittsbewertungs-Crossover

Bei dem **Abschnittsbewertungs-Crossover** können mehr als 2 Elternindividuen rekombiniert werden. Wenn  $n$  Elternindividuen zur Rekombination übergeben werden. Es wird immer genau ein Kindindividuum bei diesem Crossover erzeugt.

Es werden die folgenden Schritte durchlaufen:

- Es werden  $n - 1$  zufällige Zahlen im Bereich zwischen 0 und der Länge der Suchwörter gezogen. Diese Zufallszahlen begrenzen, sortiert  $n$  Teilstücke des Suchworts.
- Jedes dieser Teilstücke wird bei jedem Elternindividuum bewertet und dann das Beste der bewerteten Teilstücke in das Kind übernommen.
- Das Kind wird der Menge der erzeugten Kinder hinzugefügt.

Das erzeugte Kind wird zurückgegeben. Dieser Crossover-Operator ist bei der Auswahl, von welchem Elternindividuum ein Teilwort übernommen wird, deterministisch. Er wählt immer das Teilstück, dessen Bewertung am besten (höchsten) ist. Der Bewertet-Lineare-Crossover ist dennoch probabilistisch, da die Schnittstellen der Teilstücke zufällig bestimmt werden. Der Code dieses Crossoveroperators ist im Algorithmus 15 zu sehen.

**Algorithmus 15** *Collection<AbstractIndividuum> rekombiniere(Collection<AbstractIndividuum> eltern, AbstractIndividuumFactory factory)*

```

{
  long[] kindPat = new long[eltern[0].gibLaenge()];
  LineareZustandszahlBewertung bewertung = new LineareZustandszahlBewertung();
  List<int> schnittListe = new List<int>();
  int[] bestenIndex = new int[elternZahl + 1];
  int[] bestenBewertung = new int[elternZahl + 1];
  Zufallsgenerator zg = Zufallsgenerator.getInstance();
  // Nun die Patterns der Eltern an zufaelligen Stellen "durchschneiden".
  // Noch den Anfang und das Ende des Patterns einfuegen.
  schnittListe.Add(0);
  schnittListe.Add(eltern[0].gibLaenge());
  for (int i = 1; i < elternZahl; i++)
  {
    schnittListe.Add(zg.next(eltern[0].gibLaenge() - 1));
  }
  schnittListe.Sort();
  // Nun sind die Schnittstellen aufsteigend sortiert.
  for (int j = 1; j < schnittListe.Count; j++)
  {
    // wenn wir einen leeren Schnitt haben, dann abbrechen.
    if (schnittListe[j - 1] != schnittListe[j])
    {
      // Fuer jedes Elternindividuum das Teilstueck holen und bewerten.
      for (int k = 0; k < elternZahl; k++)
      {
        long[] pat = eltern[k].gibPatternLong();
        long[] teilStueck = new long[schnittListe[j] - schnittListe[j - 1]];
        // teilstueck kopieren.
        kopiere(pat, teilStueck, schnittListe[j - 1], 0, teilStueck.Length - 1);
        // teilstueck bewerten.
        if (bewertung.bewerte
            (factory.ErzeugeIndividuum(teilStueck)) > bestenBewertung[j])
        {
          bestenIndex[j] = k;
        }
      }
      // Nun das beste Teilstueck kopieren.
      kopiere(eltern[bestenIndex[j]].gibPatternLong(), kindPat, schnittListe[j - 1],
              schnittListe[j - 1], schnittListe[j] - schnittListe[j - 1]);
    }
  }
  // Das Kind ist fertig. In Ergebnis einbauen
  Collection<AbstractIndividuum> erg = new Collection<AbstractIndividuum>();
  erg.Add(factory.ErzeugeIndividuum(kindPat));
  return erg;
}

```

Die Hilfsmethode `void kopiere(long[] a, long[] b, int anfangA, int anfangB, int anzahl)` kopiert `anzahl` viele Zeichen von `a` nach `b`, wobei die Zeichen in `a` ab der `anfangA`-ten Position gelesen werden und in `b` ab der `anfangB`-ten Position.

### 5.3.5 Palindrom-Crossover

Bei dem Palindrom-Crossover werden die Suffixe der Eltern nach Palindromen untersucht. Es wird das größte Suffix eines Elternindividuums in das Kindindividuum kopiert und die restlichen Zeichen des Kindes mit denen eines zufälligen Elternindividuums aufgefüllt. Dies geschieht für jedes Elternindividuum in der Menge.

Bei dem Palindrom-Crossover können mehr als 2 Elternindividuen rekombiniert werden. Normalerweise werden bei diesem Crossover 4 Elternindividuen rekombiniert. In dieser Implementierung werden genau so viele Kinder wie Eltern erzeugt. Diese Effizienzsteigerung ist nötig, da die Berechnung der Palindrome sehr aufwändig ist (Aufwand  $O(m^2)$ ). Durch diesen Crossover werden die Individuen bevorteilt, deren Suchwörter ein Palindrom beinhalten (z. B. `Updated-Palindromes`). Andere Klassen von Suchwörtern werden aber nicht komplett von der Suche ausgeschlossen und können immer noch gefunden werden. Die Hoffnung ist nun, durch die Begünstigung der Palindrom beinhaltenden Suchwörter schneller das beste Suchwort zu finden. Im Algorithmus 16 ist die Berechnung des Crossoveroperators zu sehen.

**Algorithmus 16** *Collection<AbstractIndividuum> rekombiniere(Collection<AbstractIndividuum> eltern, AbstractIndividuumFactory factory)*

```

{
  Collection<AbstractIndividuum> res = new Collection<AbstractIndividuum>();
  Zufallsgenerator zg = Zufallsgenerator.getInstance();
  foreach(AbstractIndividuum ind in eltern)
  {
    // Nach dem Palindrom suchen
    int start = suchePalindrom(ind);
    // nun das Individuum aussuchen, mit dessen Werten aufgefüllt werden soll.
    int zufallsZahl = zg.next(eltern.Count-1);
    long[] pat1 = ind.gibPatternLong();
    long[] pat2 = eltern[zufallsZahl].gibPatternLong();
    long[] kind = new long[ind.gibLaenge()];
    for (int i = 0; i < kind.Length; i++)
    {
      if (i < start)
      {
        kind[i] = pat2[i];
      }
      else
      {
        kind[i] = pat1[i];
      }
    }
    res.Add(factory.ErzeugeIndividuum(kind));
  }
  return res;
}

```

Die Hilfsmethode `suchePalindrom` gibt den Index des Beginns des größten Suffixes in dem übergebenen Individuum zurück, das gleichzeitig ein Palindrom ist.

## 5.4 Selektionen

Die Selektionen des Genetischen Algorithmus zur Wortsuche wählen aus einer Population  $pop_t$  von Individuen diejenigen aus, die im Weiteren verwendet werden sollen. Diese Individuen werden in einer Population gespeichert. Diese Population wird von der Selektion als Ergebnis zurückgegeben. Die implementierten Selektionen sind alle Selektionen mit Duplikaten, probabilistisch und allgemein.

### 5.4.1 Rangselektion

Diese Selektion funktioniert nach dem Prinzip der rangbasierten Selektion wie in 4.7.1 beschrieben. Die Wahrscheinlichkeit des Individuums  $I_i$  mit dem Rang  $i$  selektiert zu werden, ist in der implementierten Selektion

$$P(I_i) = a \cdot (i - |pop|) + k$$

Es wird also eine Geradengleichung für die verschiedenen Selektionswahrscheinlichkeiten angewendet.  $|pop|$  sei die Anzahl der Individuen der Population, aus der die Individuen selektiert werden sollen. Damit die Wahrscheinlichkeiten gleichmäßig verteilt sind, setzen wir  $k = a$ , so steigt die Wahrscheinlichkeit zwischen dem  $i$ . und dem  $i - 1$ . Individuum immer um die Wahrscheinlichkeit, mit der das schlechteste Individuum selektiert wird. Aus dieser Wahl folgt:

$$a = k = \frac{2.0}{|pop|^2 + 1.0}$$

Mit diesen Parametern berechnet der Algorithmus 17 die Rangselektion.

**Algorithmus 17** Population selektion (Population pop, int popSize)

```
{
// Setzen der Konstante k abhängig von der Populationsgroesse.
k = 2.0 / (((double)pop.size())*((double)pop.size()+1.0));
// trotzdem noch a berechnen, damit man später k noch ändern kann.
double a = 2.0 / (((double)pop.size())*((double)pop.size()-1.0))
           - 2*k/((double)pop.size()-1.0);
Population erg = new Population();
// Wenn in der Population schon jetzt weniger Individuen sind,
// dann sind wir schon fertig .
if (pop.size() <= popSize)
{
return pop;
}
// nun alle Individuen bewerten lassen und sortieren
List<AbstractIndividuum> indList = pop.gibIndividuumListe();
indList.Sort(bewertungsfunktion);
// Die Raenge sind nun hergestellt .
Zufallsgenerator zg = Zufallsgenerator.getInstance();
while (erg.size() < popSize)
{
// Nun die Zufallszahl ziehen.
double zZahl = zg.nextDouble();
// Die Zufallszahl liegt zwischen 0 und 1.
// Jetzt in der Schleife die Zufallszahl herunterzählen, bis der
// Rang des gezogenen Elements ermittelt ist.
int index = 0;
while (zZahl > 0)
```

```

    {
        zZahl -= a * (double)index + k;
        index++;
    }
    // Des Individuum der Ergebnispopulation hinzufuegen.
    erg.add(indList[index-1]);
}
return erg;
}

```

In der Implementierung wird keine beschleunigte Variante der Selektion wie z. B. das SUS ([Nis97b]) benutzt, da das Ziehen von Zufallszahlen durch die Laufzeit der Bewertungsfunktion (die Konstruktion des Boyer-Moore-Automaten) nicht ins Gewicht fällt. Es wird stattdessen die `Roulette wheel selection` durchgeführt. Die Individuen sind in `indList` sortiert abgelegt. Durch Abziehen der Wahrscheinlichkeiten der Individuen (die durch ihren Rang festgelegt sind) wird bestimmt, welches Individuum selektiert werden soll. Wenn die Zufallszahl kleiner 0 wird, wird das entsprechende Individuum aus der Liste gewählt. Da das am besten bewertete Individuum das Letzte in der Liste ist, wird der Index inkrementiert.

### 5.4.2 Turnierselektion

Die Turnierselektion funktioniert nach dem Prinzip der  $q$ -Turnierselektion in 4.7.1. Das  $q$ , also die Größe des Turniers, das zur Selektion durchgeführt werden soll, kann bei der Initialisierung der Selektion frei gewählt werden. Standardmäßig ist sie auf 2 eingestellt. In der Implementierung für den Genetischen Algorithmus zur Wortsuche wird immer nur der Gewinner des Turniers selektiert. Der Aufwand für die Selektion steigt dabei bei der Selektion von  $m$  Individuen mit  $O(m \cdot q)$ .

Der Code der Turnierselektion ist in Algorithmus 18 zu sehen.

#### Algorithmus 18 *Population selektion (Population pop, int size)*

```

{
    Population erg = new Population();
    for (int i = 0 ; i < size ; i++)
    {
        AbstractIndividuum best = pop.gibZufaelligesIndividuum();
        // Jeweils die gewuenschte Anzahl Individuen waehlen (ausserhalb
        // der Schleife haben wir schon eines gewaehlt, deshalb -1) ,
        // dann den Besten in die Ergebnismenge uebernehmen.
        for (int j = 0 ; j < (q-1) ; j++)
        {
            AbstractIndividuum tmp = pop.gibZufaelligesIndividuum();
            if (bewertungsfunktion.bewerte(tmp) > bewertungsfunktion.bewerte(best))
            {
                best = tmp;
            }
        }
        erg.add(best);
    }
    // Ergebnispopulation ist fertig .
    return erg;
}

```

In der äußeren `for`-Schleife wird dafür gesorgt, dass genug Individuen selektiert werden, während in der inneren `for`-Schleife das „Turnier“ durchgeführt wird. Es soll nur das beste Individuum selektiert werden, deshalb wird von den  $q$  Individuen des Turnieres nur das Beste abgespeichert. Dieses Individuum wird dann als Gewinner des Turniers in die Menge der selektierten Individuen eingetragen.

### 5.4.3 Fitnessproportionale Selektion

Diese Selektion funktioniert nach dem Prinzip der **fitnessproportionalen Selektion** wie in 4.7.1 beschrieben. Die Wahrscheinlichkeit eines Individuums, selektiert zu werden ist direkt proportional zu der Fitness des Individuums. Um diese Gewichtung berechnen zu können, wird dafür zunächst die Fitness aller Individuen der Population  $pop_t$ , aus der selektiert werden soll, aufsummiert:

$$sum = \sum_{i \in pop_t} f(i)$$

Wenn nun ein Individuum selektiert werden soll, so wird eine Zufallszahl  $z$  zwischen 0 und  $sum$  generiert. Nun wird über die Liste der Individuen der Population  $pop_t$  gelaufen und die Fitness des Individuums von der Zufallszahl abgezogen  $z = z - f(i)$ , bis die Zahl negativ wird. Das Individuum, durch dessen Fitness die Zahl  $z$  negativ wurde, wird selektiert. Gestartet wird dieser Durchlauf immer bei dem gleichen Individuum und die Reihenfolge mit der die Individuen abgearbeitet werden, ändert sich über die Durchläufe nicht. Im Algorithmus 19 ist der Code dieser Selektion zu sehen.

**Algorithmus 19** *Population selektion (Population pop, int popSize)*

```
{
    Population erg = new Population();
    // Die gesamtfitness der Population berechnen.
    double gesFit = 0.0;
    foreach (AbstractIndividuum ind in pop.gibIndividuumListe())
    {
        gesFit += bewertungsfunktion.bewerte(ind);
    }
    Zufallsgenerator zg = Zufallsgenerator.getInstance();
    while (erg.size() < popSize)
    {
        double zZahl = zg.nextDouble();
        // zZahl ist zwischen 0 und 1, deshalb skalieren.
        zZahl = zZahl * gesFit;
        // Nun solange die fitness der Individuen subtrahieren,
        // bis die Zahl nicht mehr positiv ist.
        List<AbstractIndividuum> iList = pop.gibIndividuumListe();
        IEnumerator<AbstractIndividuum> enumerator = iList.GetEnumerator();
        while (zZahl > 0 && enumerator.MoveNext())
        {
            zZahl -= bewertungsfunktion.bewerte(enumerator.Current);
        }
        // selektiertes Element hinzufuegen.
        erg.add(enumerator.Current);
    }
    return erg;
}
```

Mit dem Zufallsgenerator `zg` wird mit der Methode `nextDouble` eine Zufallszahl zwischen 0 und 1 erzeugt. Mit dieser Zufallszahl wird dann die Selektion wie oben beschrieben durchgeführt.

### 5.4.4 Zufällige Selektion

Die zufällige Selektion benutzt eine Zufallszahl, um ein zufälliges Individuum aus der Population  $pop_t$  zu selektieren. Es entspricht der Selektion, die in 4.7.1 beschrieben wurde.

**Algorithmus 20** *Population selektion (Population pop, int popSize)*

```
{
  Population erg = new Population();
  for (int i = 0 ; i < popSize ; i++)
  {
    erg.add(pop.gibZufaelligesIndividuum ());
  }
  return erg;
}
```

Mit der Methode `gibZufaelligesIndividuum` wird aus der Population ein zufälliges Individuum ausgewählt. Dieses Individuum wird in die Ergebnispopulation eingetragen, die von der Selektion zurückgegeben wird. Dies wird solange wiederholt, bis `popSize` Individuen selektiert wurden.

## 5.5 Bewertungen

In diesem Abschnitt werden verschiedene mögliche Bewertungsfunktionen beschrieben, die für den Genetischen Algorithmus in Betracht gezogen wurden. Die Funktionsweise der Bewertungsfunktionen wird beschrieben und anschließend eine Bewertung abgegeben.

### 5.5.1 Zählen der Zustände des BMA

Bei dieser Bewertung wird der Boyer-Moore-Automat komplett aufgebaut. Die Zustände werden gezählt. Diese Zustandszahl wird direkt als Bewertung für das Individuum (das Suchwort) benutzt. Für den Aufbau des Automaten wird der verbesserte Algorithmus von [BYCG94] benutzt. Der Aufwand des verbesserten Algorithmus ist  $O(|Q|m^2)$ .

Als Bewertungsfunktion ist diese Bewertung ideal, allerdings ist die Laufzeit für längere Suchwörter zu groß, so dass nach einem schnelleren Algorithmus gesucht werden muss.

### 5.5.2 Heuristische Abschätzung der Zustände des BMA

Die Erzeugung des gesamten Boyer-Moore-Automaten für ein Suchwort ist der aufwändigste Teil des gesamten Genetischen Algorithmus. Außerdem ist die Laufzeitentwicklung der Bewertung mit  $O(|Q|m^2)$  und einer vielleicht vorhandenen Abhängigkeit von  $|Q| = 2^{k*m}$  für große  $m$  nicht akzeptabel. Deshalb sollte eine Methode mit geringerem Aufwand gefunden werden, die die Anzahl der Zustände eines Boyer-Moore-Automaten für ein Suchwort berechnet.

Wenn nicht mehr die genaue Anzahl der Zustände berechnet werden muss, sondern ein groberer Richtwert, der zu einer bestimmten Wahrscheinlichkeit eine nur geringe Abweichung von der korrekten Anzahl der Zustände hat, braucht nicht der gesamte BMA berechnet zu werden. Das folgende Verfahren, soll diesen Richtwert liefern, ohne den Boyer-Moore-Automaten dabei komplett zu berechnen (in seltenen Fällen wird er doch komplett berechnet werden müssen):

1. Es wird eine bestimmte Anzahl an zufälligen Zuständen des Automaten erzeugt.

2. Für diese Zustände muss nun mit einem Verfahren bestimmt werden, ob sie vom Startzustand aus erreichbar sind oder nicht.
3. Wenn für alle zufällig generierten Zustände berechnet ist, ob sie erreichbar sind, oder nicht wird der Anteil der Zustände berechnet, die erreichbar waren.
4. Dieser Wert wird nun auf alle Zustände hochgerechnet und als Zustandszahl des Boyer-Moore-Automaten angenommen.

Dieses Verfahren ist nicht anwendbar, da sich die Wahrscheinlichkeit bei der zufälligen Auswahl einen Zustand im BMA zu treffen mit wachsender Suchwortgröße zunehmend unwahrscheinlicher wird. Bei der Suchwortlänge  $|w| = 28$  benötigt man durchschnittlich 18338 Versuche, um zufällig einen Zustand im BMA zu treffen, somit wird die Laufzeit, um eine aussagekräftige Hochrechnung anstellen zu können, zu groß.

# Auswertungen

---

In diesem Kapitel werden die Messreihen ausgewertet, die mit dem Evolutionären Algorithmus zur Wortsuche erzeugt wurden. Der Evolutionären Algorithmus zur Wortsuche soll ein (wenn möglich das) bestes Suchwort  $w \in \Sigma^*$  finden. Um diese Aufgabe zu erfüllen müssen die Ergebnisse des Evolutionären Algorithmus möglichst nahe an dem besten Suchwort liegen. Um solche Ergebnisse zu erhalten wurden vor den eigentlichen Messläufen mit dem Evolutionären Algorithmus Testläufe durchgeführt, um gute Einstellungen für die evolutionären Parameter zu finden.

Dazu wurden verschiedene Einstellungsmöglichkeiten mit dem gleichen Problem (bestes Suchwort  $w \in \Sigma = \{a, b\}$  der Länge 30 finden) gestartet und jeweils 20 Testdurchläufe mit diesen Einstellungen durchgeführt. Anschließend wurden die besten gefundenen Werte und die durchschnittliche BMA-Größe verglichen, um festzustellen, welche Einstellung bessere Ergebnisse liefert.

Die Selektionen waren hierbei duplikatfrei. Bei der Umweltselektion wurde ein Turnier mit 2 Individuen, bei der Elternselektion ein Turnier mit 4 Individuen durchgeführt. Außerdem wurde immer das beste Individuum einer Generation in die nächste Generation übernommen, um ein einmal gefundenes Optimum nicht zu verlieren.

Als Mutation ist, wenn nicht anders angegeben, die BitFlip-Mutation eingestellt. Wenn ein Individuum mutiert wird, so werden 10 % der Zeichen des Suchwortes geändert (wobei die gleichen Zeichen mehrmals geändert werden können, siehe 5.2.1).

Als Rekombination ist, wenn nicht anders angegeben, der Palindrom-Crossover, wie in 5.3.5 beschrieben, eingestellt.

Die Populationsgröße wurde nicht bei der Suche nach optimalen Parametereinstellungen betrachtet. Die Population ist bei allen Testläufen und auch bei den Suchläufen für große Suchwörter 200 Individuen groß. Die Elternpopulation umfasst auch immer 200 Individuen. Die Population mit den Kindindividuen ist immer 1000 Individuen groß, der Parameter  $\lambda$  ist also bei allen Läufen auf  $\lambda = 5$  eingestellt.

Zuerst wurden in diesen Testläufen die verschiedenen Komponenten des Evolutionären Algorithmus getestet. Dies sind die Mutation, die Rekombination und die Selektion. Bei diesen Testläufen wurde die Mutationswahrscheinlichkeit und die Rekombinationswahrscheinlichkeit konstant auf den Werten 0.7 und 0.8 gehalten, um die Ergebnisse vergleichbar zu halten.

In den Auswertungen wird die Fitness eines durchschnittlichen Individuums in der  $i$ . Generation angegeben. Außerdem wird das arithmetische Mittel der Fitness des besten Individuums der  $i$ . Generation über alle Testläufe berechnet und der Verlauf für jede Mutation und Rekombination besprochen. Der Durchschnittswert ist das arithmetische Mittel aller Fitnesswerte der  $i$ . Generation, der 20 Testdurchläufe. Das beste Individuum einer Generation ist das mit der maximalen Fitness aller Individuen der  $i$ . Generation, der 20 Testdurchläufe.

Die Ergebnisse und Schlussfolgerungen aus den Testläufen finden sich in den jeweiligen Unterabschnitten.

Anschließend wurden mit den gefundenen Komponenten des Genetischen Algorithmus mehrere Testläufe durchgeführt, bei denen sich die Mutations- und Rekombinationswahrscheinlichkeit veränderte. Mit diesen Testläufen wurden die richtigen Einstellungen für diese beiden Parameter ermittelt.

Nachdem der Genetische Algorithmus mit allen seinen Parametern so für die Wortsuche eingestellt wurde, wurden die Messreihen durchgeführt. Hierbei wurden von der Suchwortlänge 30 ab jeweils 20 evolutionäre Läufe mit 50 Generationen durchgeführt, um ein gutes Suchwort zu finden. Die Suchwortlänge wurde zwischen zwei Messungen um 5 Zeichen erhöht. Also 35, 40, 45 usw. Die Ergebnisse dieser Messungen finden sich im Abschnitt 6.5.

### **6.1 Beurteilung der Mutationen**

In diesem Abschnitt werden zunächst die Ergebnisse der Testläufe für jede Mutation einzeln präsentiert und besprochen. Anschließend folgt ein Vergleich der verschiedenen Mutationen und die Bewertung der Mutationen (siehe Abschnitt 6.1.6).

### 6.1.1 Bestenmutation

Wie in Abbildung 6.1 zu sehen ist, findet die **Bestenmutation** (siehe 5.2.5) im Schnitt sehr schnell gute Individuen mit einer Zustandszahl von über 20000. Anschließend flacht der Anstieg ab und stagniert bei ca. 21000 Zuständen.

Die Fitness eines durchschnittlichen Individuums in der Population einer Generation steigt stetig bis knapp unter 16000 Zustände an und pendelt dann um diesen Wert. Dies ist in Abbildung 6.1 zu sehen.

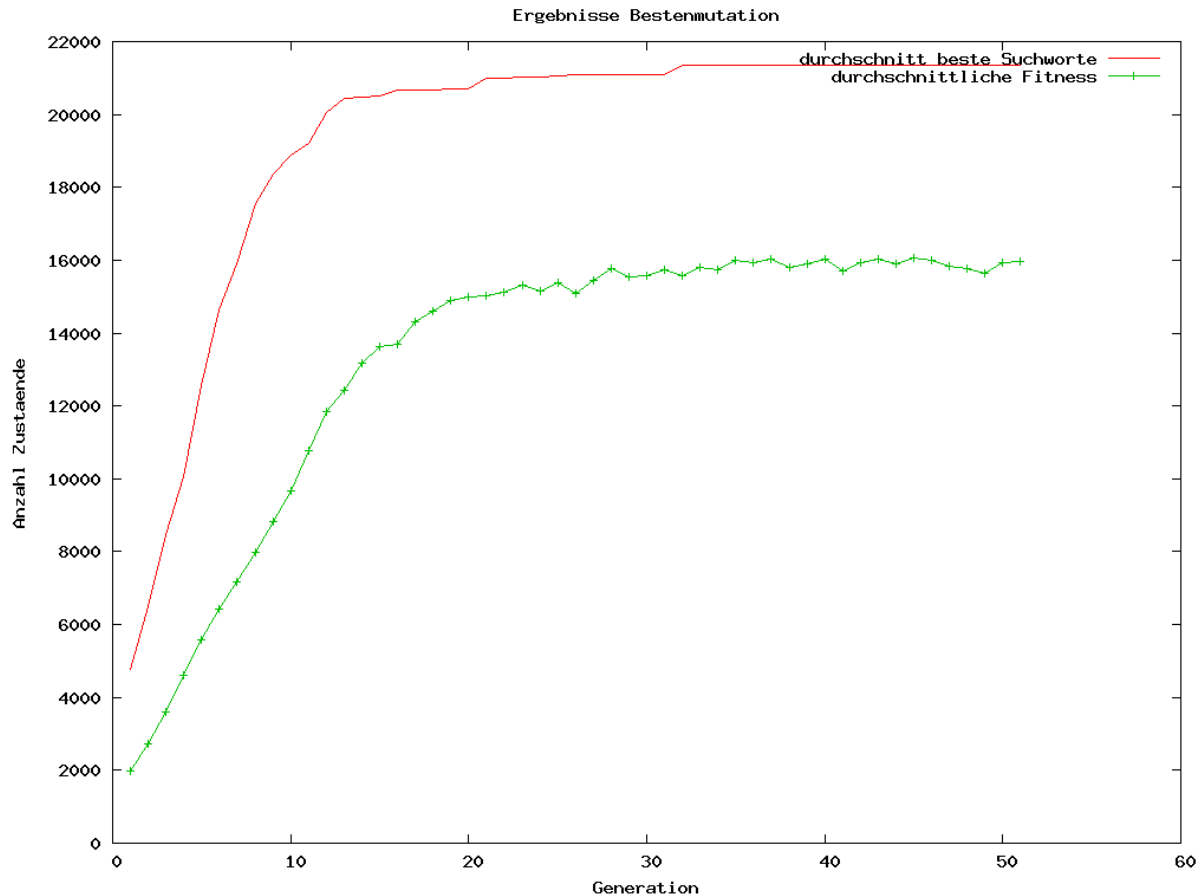


Abbildung 6.1: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit der Bestenmutation.

### 6.1.2 BitFlip-Mutation

Die Mittelwerte der besten Individuen einer Generation steigen bei der bitFlip-Mutation (siehe 5.2.1) bis ca. zur 14. Generation sehr stark an. Anschließend ist der Anstieg nicht mehr so stark, es folgen aber noch Verbesserungen der Fitness des Durchschnitts des besten Individuums bis nach der 40. Generation. Dieser Sachverhalt ist in Abbildung 6.2 zu sehen.

Bei den Werten der durchschnittlichen Fitness eines Individuums in einer Generation ist ein andauernder Anstieg zu erkennen. Allerdings unterbrochen von kleinen lokalen Verschlechterungen innerhalb weniger Generationen. Ab der 21. Generation verlangsamt sich der Anstieg der durchschnittlichen Fitness. Er erreicht etwa die 12000 Zustandsmarke.

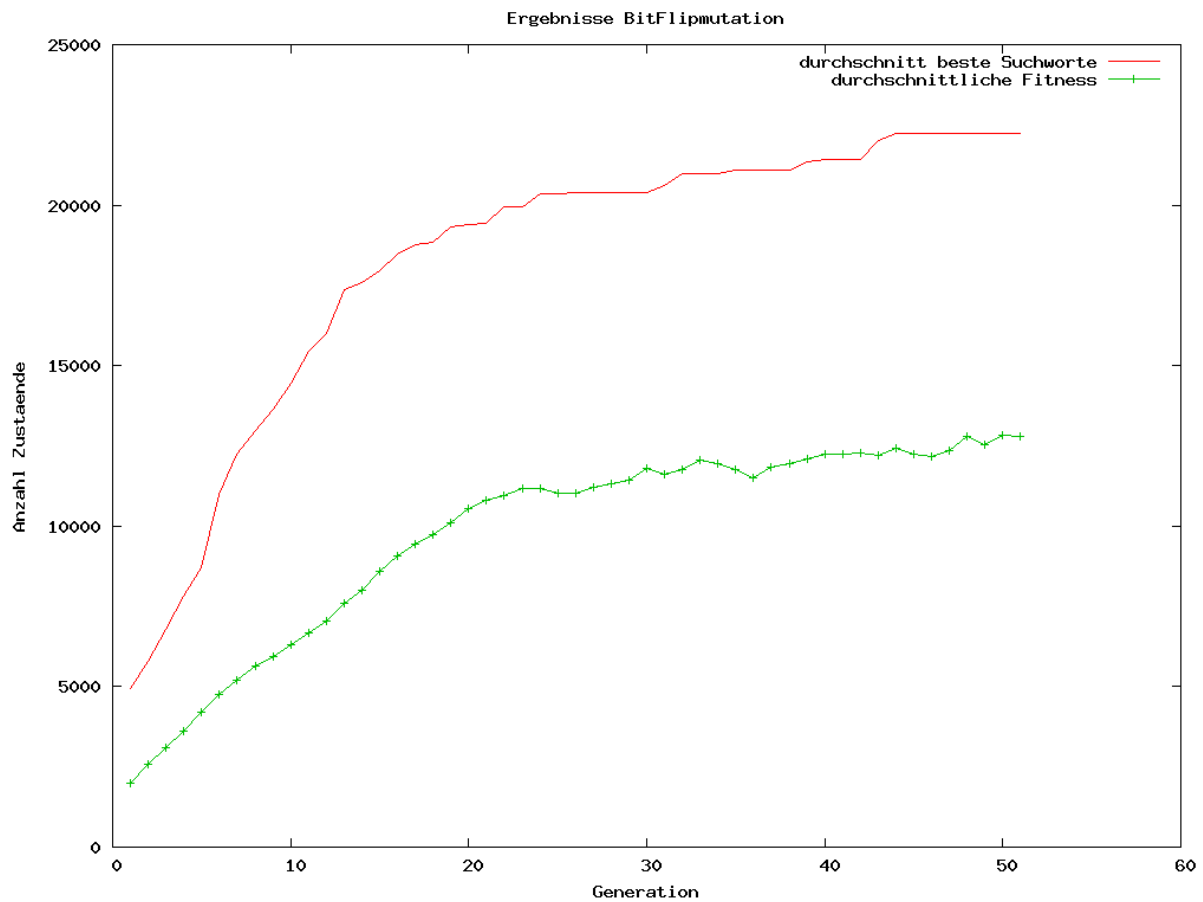


Abbildung 6.2: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit der bitFlip-Mutation.

### 6.1.3 Wurzel-BitFlip-Mutation

Nach einem anfänglich rasanten Anstieg flacht auch bei der Wurzel-BitFlip-Mutation (siehe 5.2.2) der Anstieg des Durchschnitts der besten Individuen etwa ab Generation 15 ab. Es folgen aber bis zur 50. Generation immer wieder Verbesserungen, wie in 6.3 zu sehen ist. Es ist außerdem zu vermuten, dass bei längerer Laufzeit auch weiter bessere Individuen gefunden werden.

Der Verlauf der durchschnittlichen Fitness eines Individuums in der  $i$ . Generation ist bis ca. zur 20. Generation fast linear. Danach pendelt der Wert um die Marke von 14000 Zuständen, wie in Abbildung 6.3 zu sehen. Hier scheint keine weitere Verbesserung möglich zu sein.

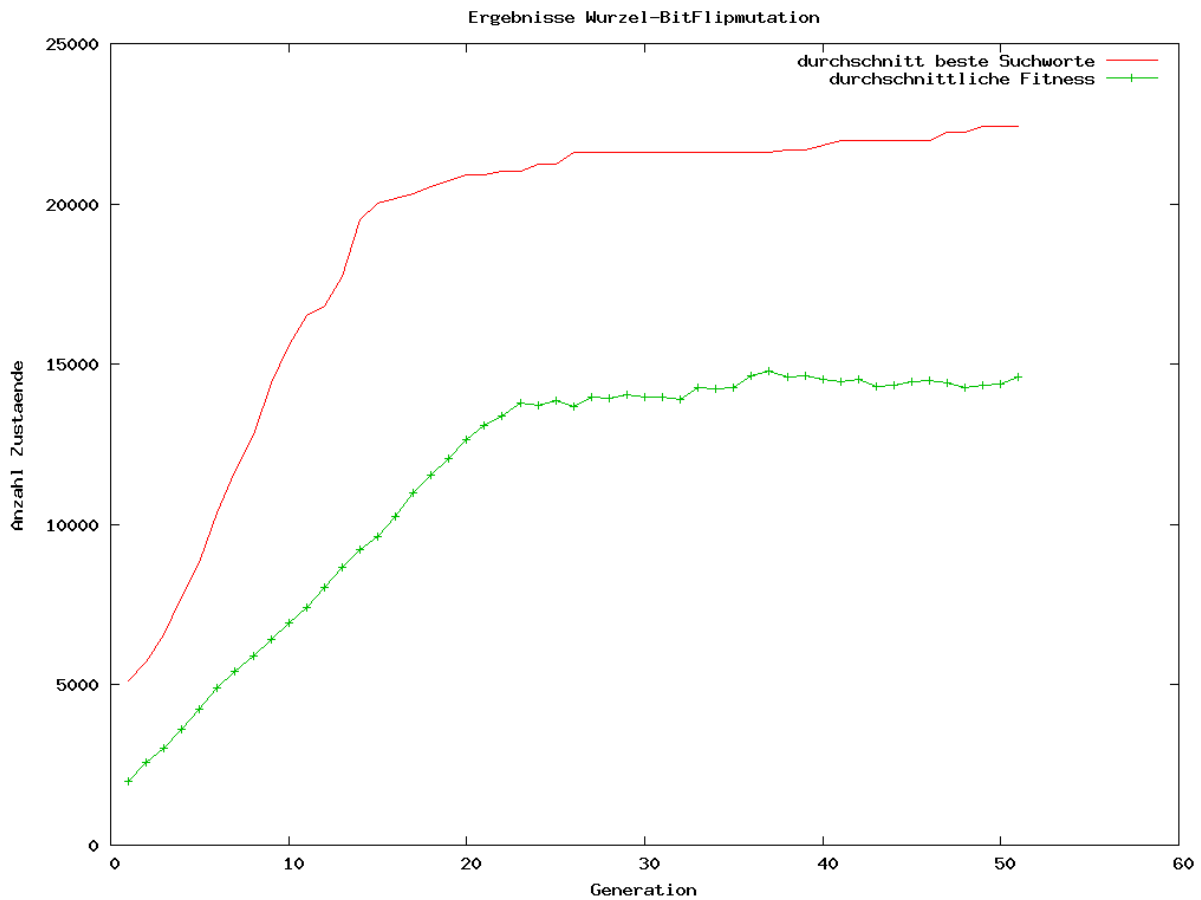


Abbildung 6.3: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit der Wurzel-BitFlip-Mutation.

### 6.1.4 Verschiebemutation

Bei der Verschiebemutation (siehe 5.2.4) ist der Anstieg der durchschnittlichen Fitness des besten Individuums einer Generation bis ca. zur 20. Generation recht stark. Anschließend nimmt der Anstieg stark ab. Bis zur 50. Generation ist noch ein leichter Anstieg zu erkennen, wobei der Anstieg zwischen der 49. und der 50. Generation durch das Hillclimbing in dem Genetischen Algorithmus erklärt werden kann und somit nicht direkt zur Bewertung dieser Mutation herangezogen werden darf. Der Anstieg ist in Abbildung 6.4 dargestellt.

Der Anstieg der durchschnittlichen Fitness eines Individuums in der  $i$ . Generation bei dieser Mutation ist in Abbildung 6.4 zu sehen. Der Anstieg verläuft relativ gleichmäßig. Es ist nicht erkennbar, warum er bei weiteren Generationen keinen höheren Wert erreichen sollte.

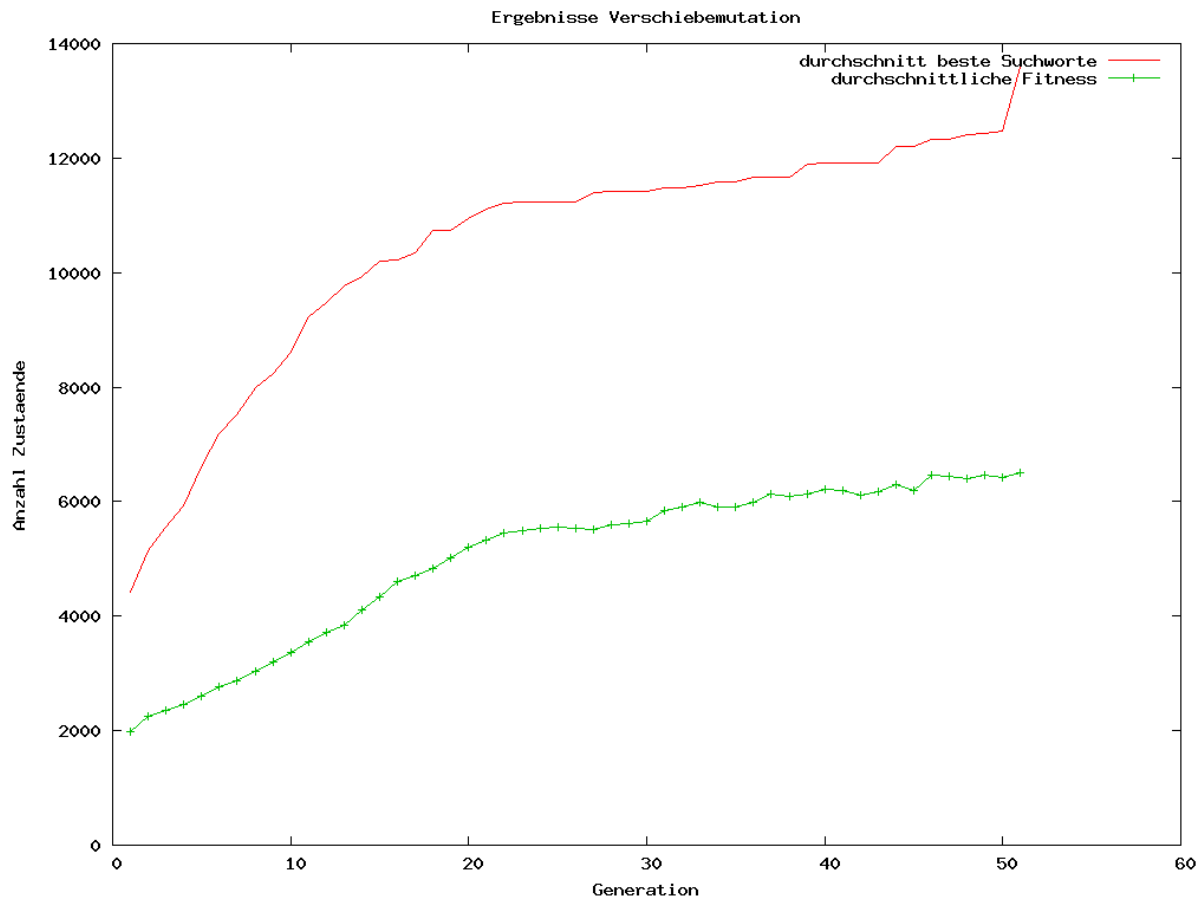


Abbildung 6.4: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit der Verschiebemutation.

### 6.1.5 Suffixmutation

Bei der Suffixmutation (siehe 5.2.3) ist der Anstieg des Durchschnitts des besten Individuums bis ca. zur 10. Generation sehr stark. Anschließend folgt zwischen der 10. und 20. Generation ein leichter Anstieg mit Abschnitten ohne Verbesserung. Nach der 20. Generation wird kein besseres Individuum mehr gefunden (siehe Abbildung 6.5). Der Anstieg zur 50. Generation kann auch hier mit dem Hillelimbing in dem Genetischen Algorithmus zur Wortsuche erklärt werden.

Der Anstieg der Fitness des durchschnittlichen Individuums der  $i$ . Generation ist bei der Suffixmutation bis ca. zur 15. Generation hoch. Anschließend flacht der Anstieg ab und pendelt schließlich zwischen 14000 und 15000 Zuständen. Dies ist in Abbildung 6.5 zu sehen.

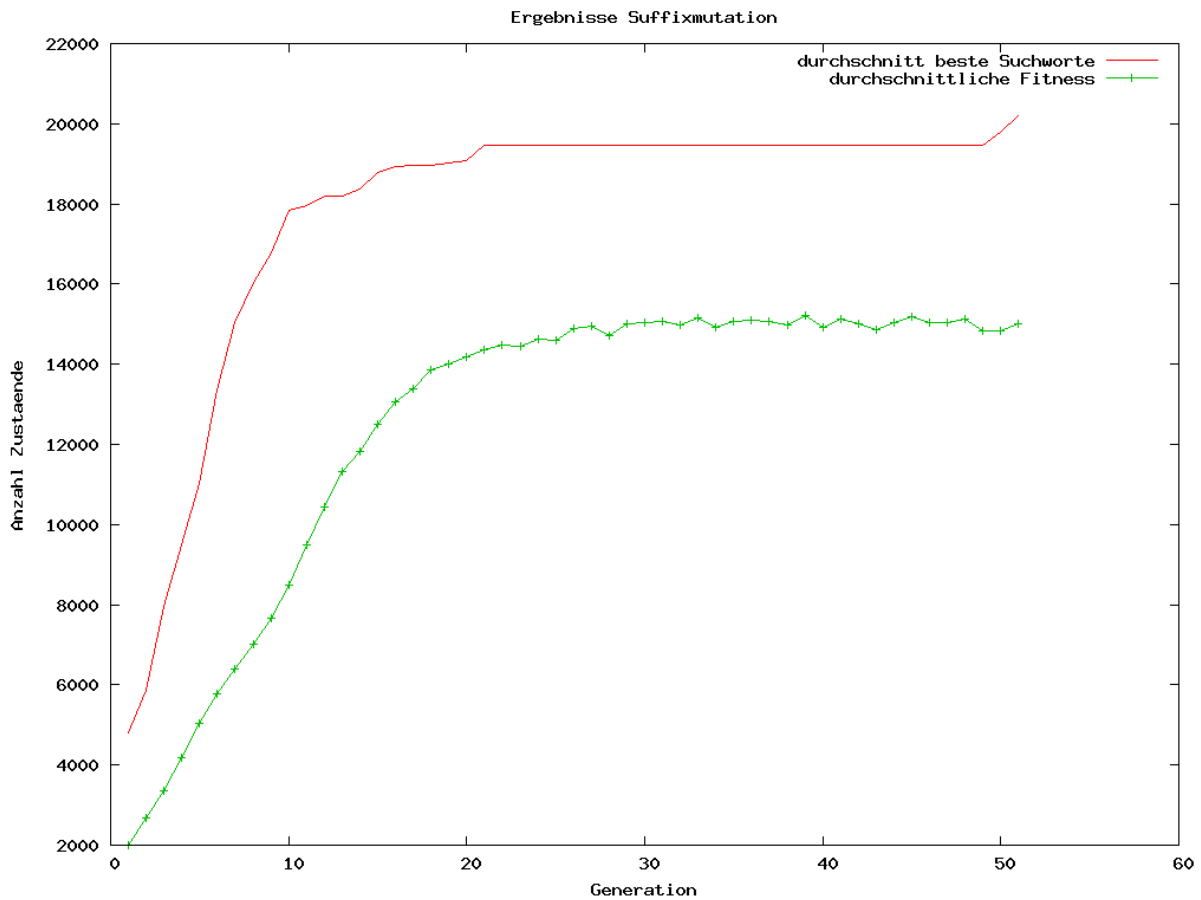


Abbildung 6.5: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit der Suffixmutation.

### 6.1.6 Fazit

In diesem Abschnitt sind die Ergebnisse aller Mutationen zusammengefasst. In Abbildung 6.6 sind die Werteverläufe der durchschnittlichen Fitness des besten Individuums eines Durchlaufs in der  $i$ . Generation zusammengefasst. In Abbildung 6.7 ist die durchschnittliche Fitness eines Individuums über alle Durchläufe in der  $i$ . Generation aufgetragen.

Die Wurzel-BitFlip-Mutation und die Bit-Flip-Mutation liefern die im Schnitt besten Ergebnisse. Die Wurzel-Bit-Flip-Mutation erreicht schneller gute Werte als die BitFlip-Mutation, nur kurzzeitig sind die Werte der Bit-Flip-Mutation besser.

Auch die Bestenmutation erreicht schnell hohe Werte bei den Durchschnitt der besten Suchwörter. Diese Mutation erzielt in den späteren Generationen jedoch keine Verbesserung mehr. Die Verschiebemutation liefert eindeutig die schlechtesten Ergebnisse. Die Suffixmutation scheint schneller als die besseren Mutationen in lokalen Maximas stecken zu bleiben, da sie in den ersten Generationen zwar gute Ergebnisse erzeugt, es dann aber keine weiteren Verbesserungen der Werte mehr gibt.

Betrachtet man die durchschnittlichen Fitness eines Individuums in der  $i$ . Generation, so erreicht hier die Bestenmutation die höchsten Werte, die Suffixmutation erzielt hier auch gute Ergebnisse. Die BitFlip- und die Wurzel-BitFlip-Mutationen erreichen bei dieser Betrachtung jedoch keine guten Werte. Die Ergebnisse der Verschiebemutation sind auch hier viel schlechter als die von allen anderen Mutationen.

Die speziellen problembezogenen Mutationen bewirken eine Verbesserung der durchschnittlichen Fitness eines Individuums. Sie scheinen, mit Ausnahme der Bestenmutation, aber nicht in der Lage zu sein bessere Suchwörter zu finden als die Standardmutationen des Genetischen Algorithmus, wie die BitFlip- und die Wurzel-BitFlip-Mutation.

Die durchschnittliche Fitness eines Individuums in der Population ist nicht von Interesse. Das Hauptkriterium für die Auswahl einer Mutation ist die durchschnittliche Fitness des besten gefundenen Suchwortes. Deshalb wird als Mutation die Wurzel-BitFlip-Mutation ausgewählt. Auch die BitFlip-Mutation hätte gewählt werden können, da die Unterschiede zwischen diesen beiden Mutationen nicht allzu groß sind. Die Wahl fiel wegen der besseren Fitness eines durchschnittlichen Individuums auf die Wurzel-BitFlip-Mutation.

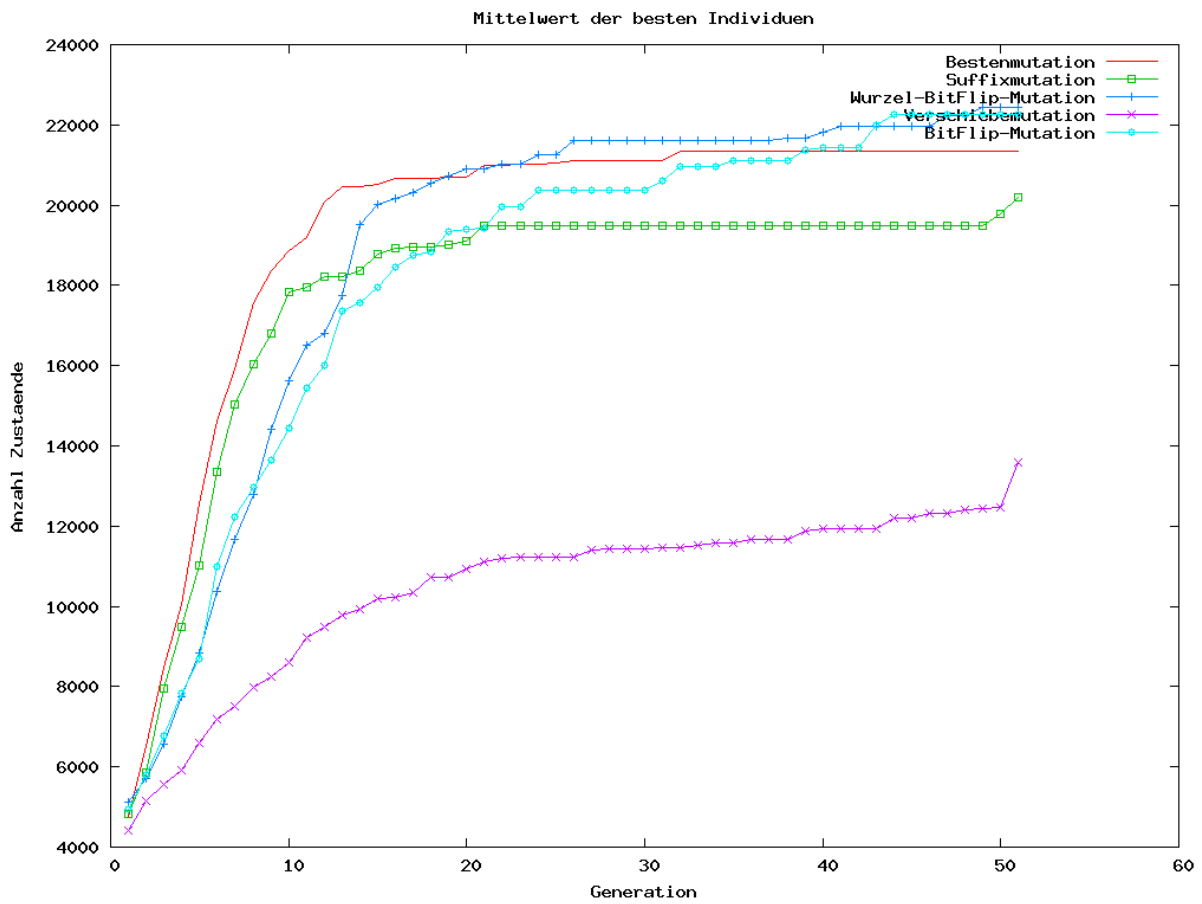


Abbildung 6.6: Der Verlauf der Fitness der besten Individuen im Vergleich der Mutationen.

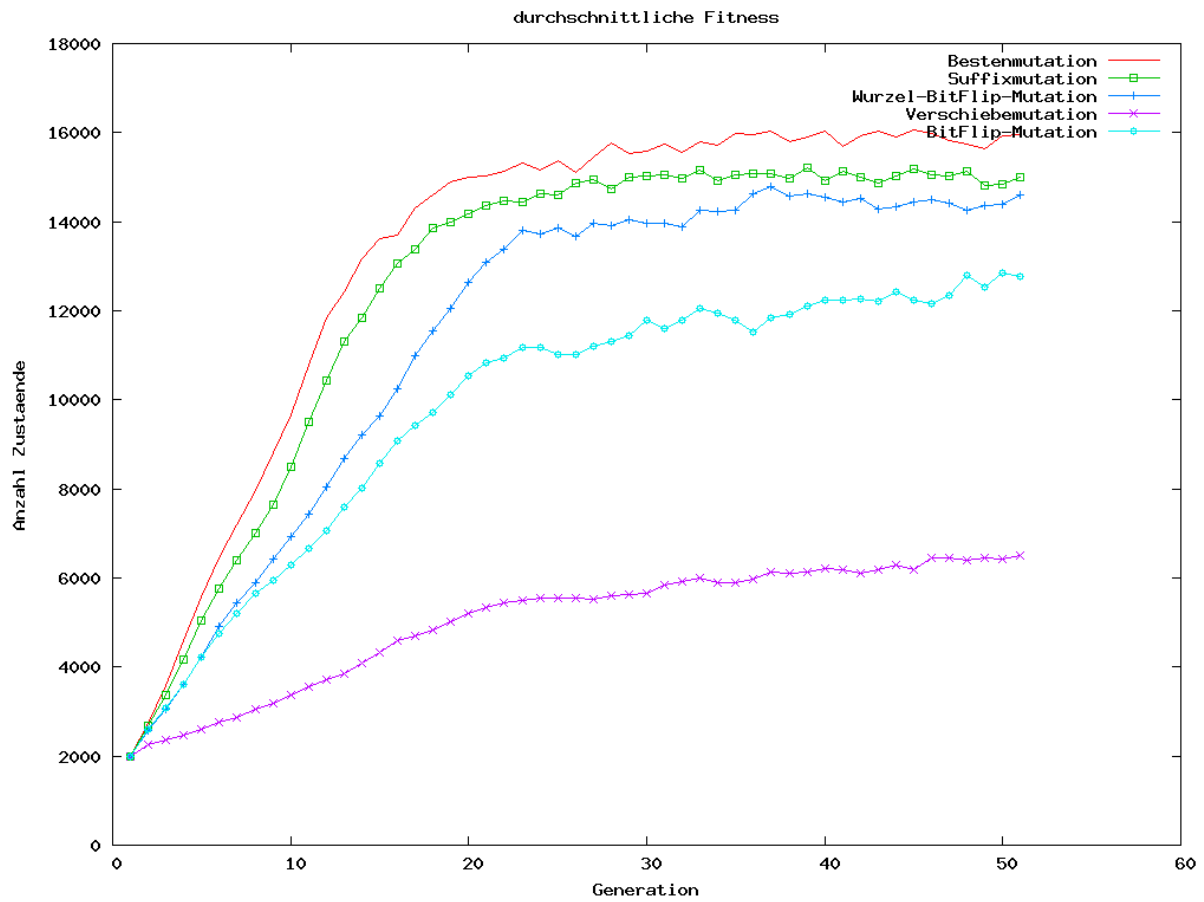


Abbildung 6.7: Der Verlauf der durchschnittlichen Fitness eines Individuums der  $i$ . Generation im Vergleich der Mutationen.

## 6.2 Beurteilung der Rekombinationen

In diesem Abschnitt werden zunächst die Ergebnisse der Testläufe für jede Rekombination einzeln präsentiert und besprochen. Anschließend folgt ein Vergleich der verschiedenen Rekombinationen und die Bewertung der Rekombinationen (siehe Abschnitt 6.2.6).

### 6.2.1 1-Punkt-Crossover

Beim 1-Punkt-Crossover (siehe 5.3.1) steigt die durchschnittliche Zustandszahl der besten Suchwörter bis ca. zur Generation 30 relativ stark an. Er findet im Schnitt ein bestes Suchwort, dessen BMA 21000 Zustände hat, bleibt dann aber gleich. Die Fitness eines durchschnittlichen Individuums verbessert sich nachdem kein besseres Individuen mehr gefunden werden nur noch wenige Generationen und schwankt dann um etwa 12000 Zustände. Der Verlauf der beiden Werte ist in Abbildung 6.8 zu sehen.

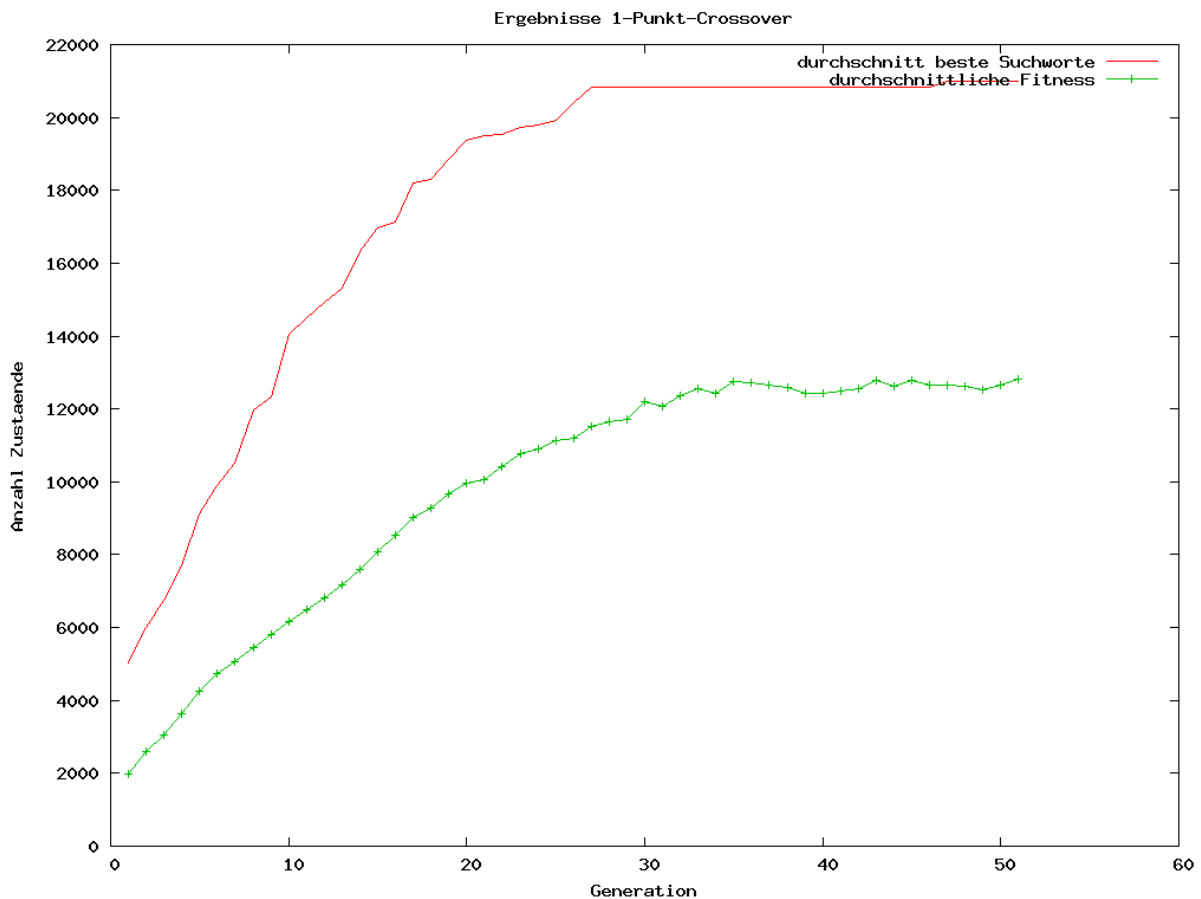


Abbildung 6.8: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit dem 1-Punkt-Crossover.

## 6.2.2 Linearer-Crossover

Beim Linearen-Crossover (siehe 5.3.3) verbessert sich der Wert der durchschnittlich gefundenen besten Suchwörter bis zur 40. Generation, die letzte Verbesserung der Fitness ist dem Hillclimbing des Genetischen Algorithmus zur Wortsuche zuzuschreiben. Die durchschnittliche Fitness eines Individuums in der  $i$ . Generation bleibt sehr niedrig. Bei diesem Wert ist über die Generationen fast keine Verbesserung zu erkennen. Diese Sachverhalte sind in Abbildung 6.9 zu sehen.

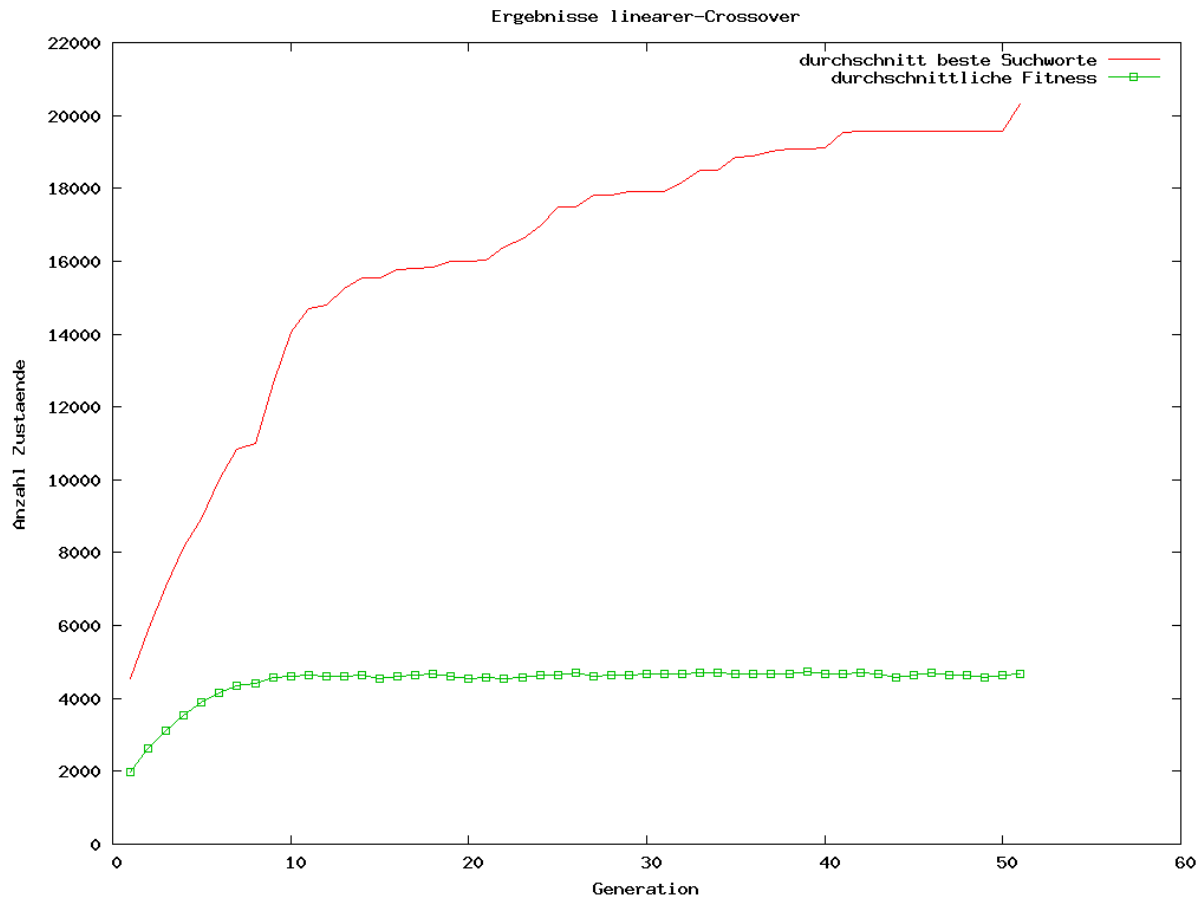


Abbildung 6.9: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit dem Linearen-Crossover.

### 6.2.3 Uniform-Crossover

Auch beim uniform-Crossover (siehe 5.3.2) steigt die Fitness der durchschnittlich gefundenen besten Individuen zunächst schnell an. Später im Testdurchlauf fällt der Anstieg ab und kommt zum Ende der 50 Generationen auf einen Wert um die 22500 Zustände. Von der 49. zur 50. Generation ist keine deutliche Verbesserung durch das Hillclimbing zu erkennen. Der uniform-Crossover scheint mit der BitFlip-Mutation gut zusammenzuwirken.

Die Fitness eines durchschnittlichen Individuums steigt bis ca. zur 30. Generation fast linear an und bleibt von dort an fast auf dem gleichen Wert.

Die Verläufe der beiden Werte sind in Abbildung 6.10 zu sehen.

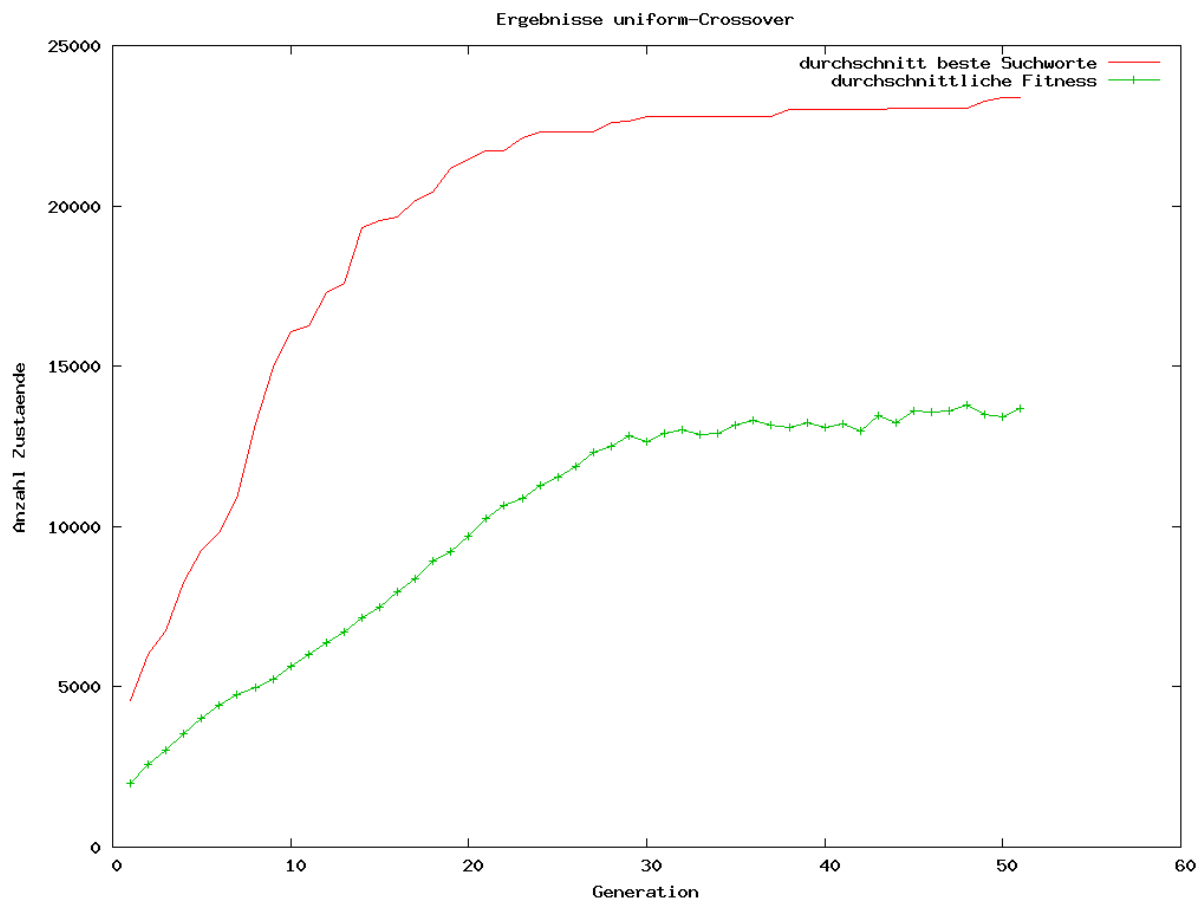


Abbildung 6.10: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit dem uniform-Crossover.

### 6.2.4 Abschnittsbewertungs-Crossover

Die Fitness der durchschnittlich gefundenen besten Individuen beim abschnittsbewertungs-Crossover (siehe 5.3.4) steigt in den ersten Generationen gleichmäßig an und findet anschließend nur nach einigen Generationen weitere Verbesserungen.

Die Fitness des durchschnittlichen Individuums der  $i$ . Generation steigt zunächst schnell, dann immer langsamer bis sie schließlich die letzten Generationen um die 12500 Zustände herum schwankt.

Die Kurven der beiden Werte sind in Abbildung 6.11 zu sehen.

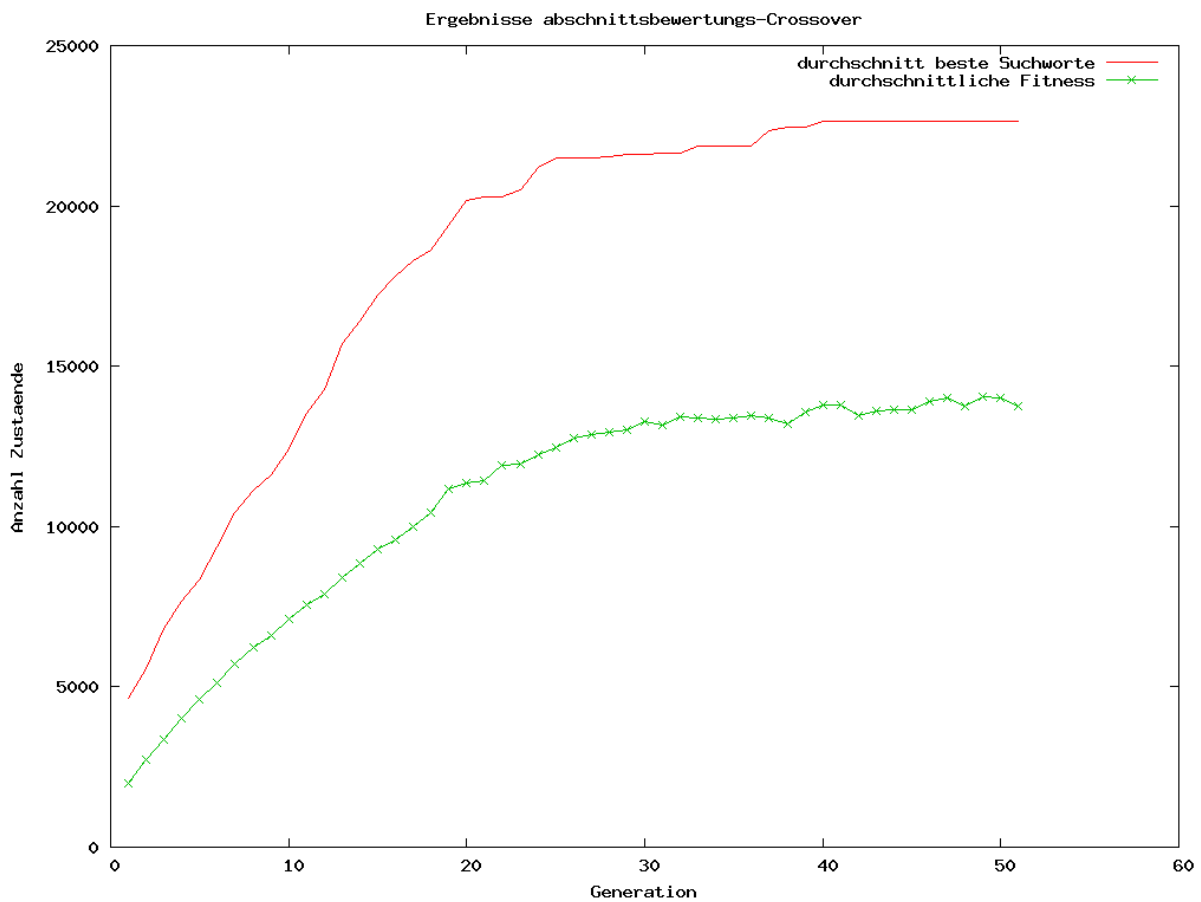


Abbildung 6.11: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit dem abschnittsbewertungs-Crossover.

### 6.2.5 Palindrom-Crossover

Beim Palindrom-Crossover (siehe 5.3.5) steigt die Fitness des durchschnittlich gefundenen besten Individuums gleichmäßig schnell an. Anschließend folgt ein Bereich, in dem der Anstieg stufenweise weiter erfolgt. Die Abstände zwischen den einzelnen Verbesserungen werden allerdings immer größer. Von der 40. bis zur 50. Generation werden schließlich keine Verbesserungen mehr gefunden.

Die Fitness des durchschnittlichen Individuums der  $i$ . Generation steigt bis zur 20. Generation fast linear an. Die Steigung ist fast so hoch wie die Steigung der Kurve der Fitness des durchschnittlich gefundenen besten Individuums. Nach der 20. Generation fällt der Anstieg schnell ab und der Wert bleibt fast gleich. Er schwankt um 11000 Zustände.

Die Kurven der beiden Werte sind in Abbildung 6.12 zu sehen.

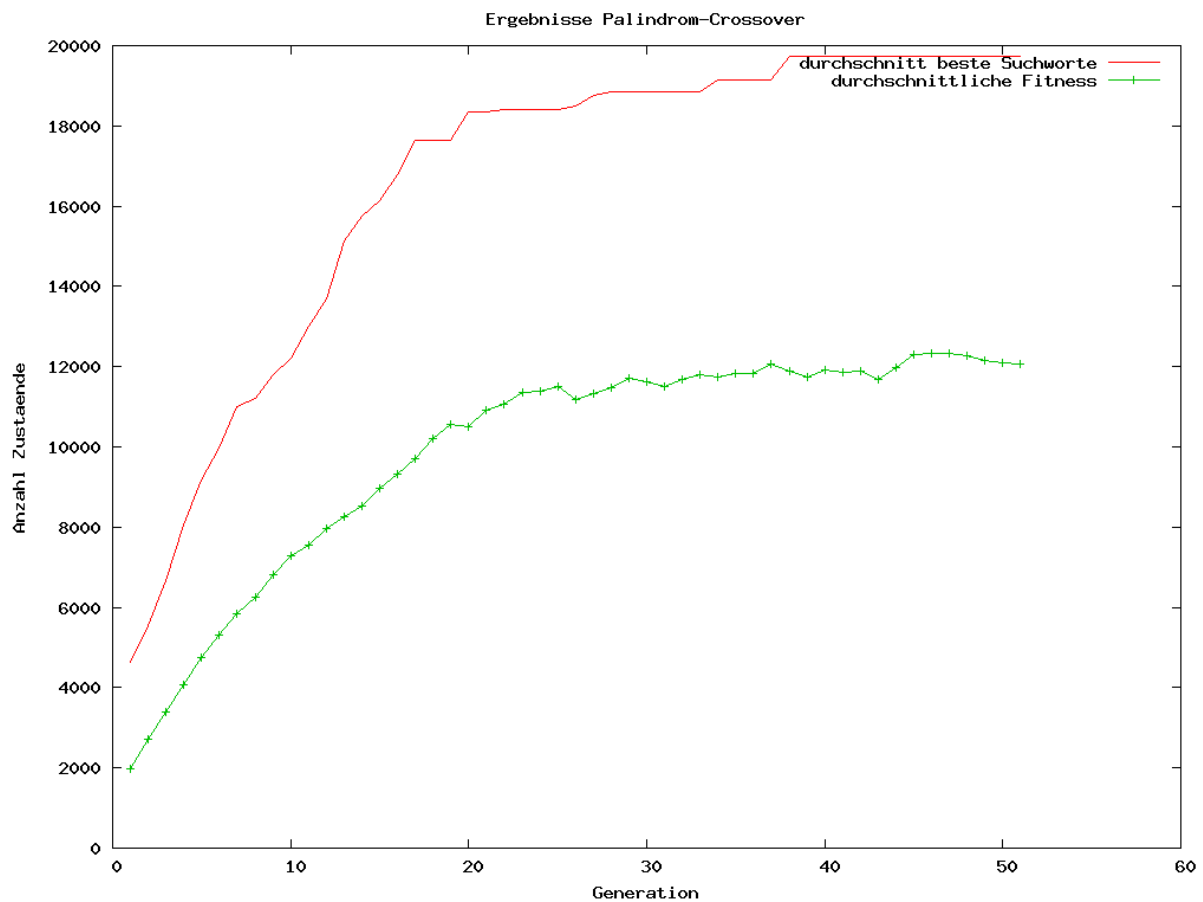


Abbildung 6.12: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation mit dem Palindrom-Crossover.

### 6.2.6 Fazit

Bei den Mittelwerten der besten Individuen für die  $i$ . Generation führen der uniform- und der abschnittsbewertungs-Crossover mit den besten Werten das Diagramm an. Der uniform-Crossover hat nach den ersten Generationen immer einen höheren Wert als der Abschnittsbewertungs-Crossover.

Der 1-Punkt-Crossover liefert bis zur 30. Generation Ergebnisse, die nahe an den beiden besten Crossover liegen. Anschließend fällt der 1-Punkt-Crossover hinter diese beiden Crossover zurück.

Der Palindrom-Crossover liefert nach den ersten Generationen durchweg schlechtere Ergebnisse, als die anderen Crossover. Er ist entgegen der früheren Hoffnungen nicht geeignet gute Individuen zu finden.

Die Verläufe der Kurven des Durchschnitts der besten Individuen in der  $i$ . Generation sind in Abbildung 6.13 zu sehen.

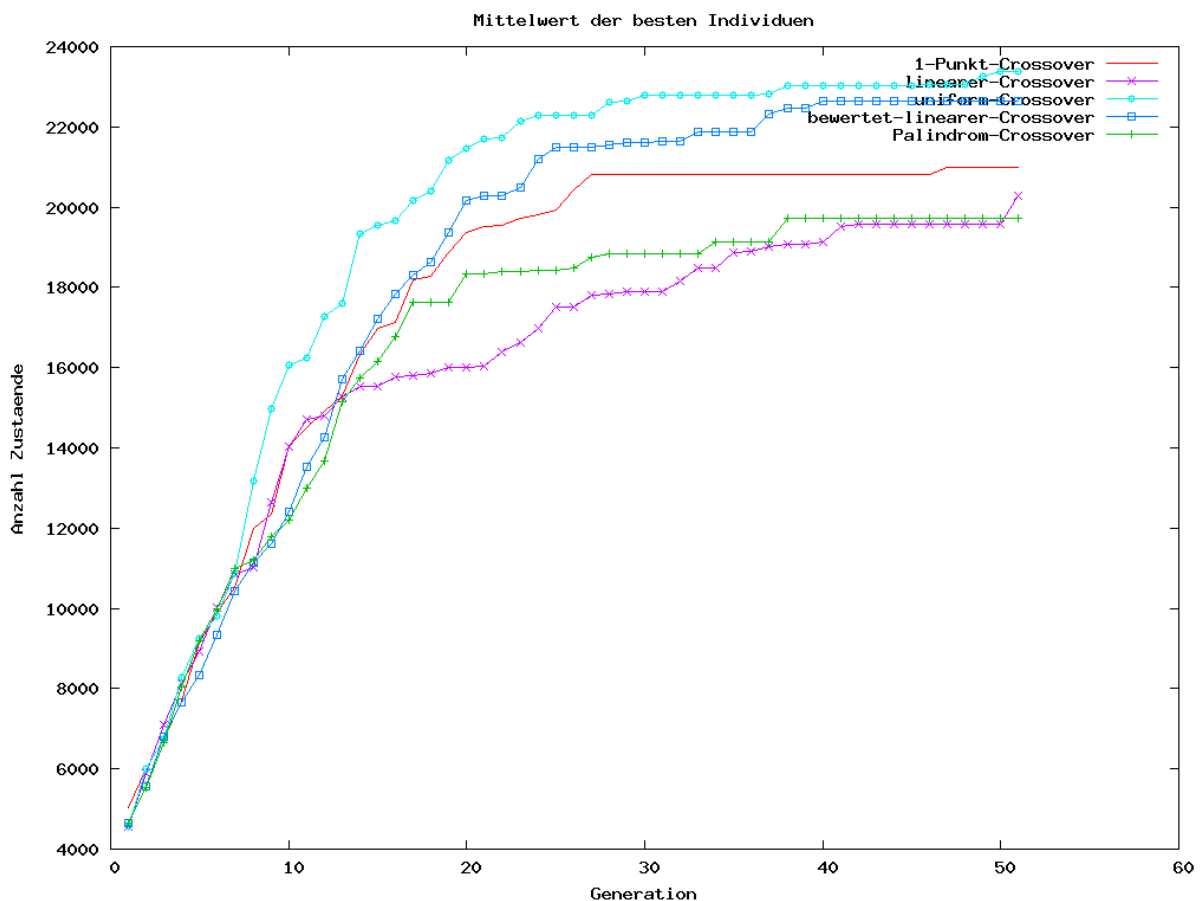


Abbildung 6.13: Der Verlauf der Fitness der besten Individuen im Vergleich der Rekombinationen.

Bei der durchschnittlichen Fitness eines Individuums in der  $i$ . Generation ist, wie in Abbildung 6.14 zu sehen, der abschnittsbewertungs-Crossover derjenige mit den besten Ergebnissen. Der uniform-Crossover liefert Ergebnisse, die nur minimal schlechter sind. Mit kleinem Abstand folgt dahinter der 1-Punkt-Crossover. Knapp hinter diesem liegt wiederum der Palindrom-Crossover, der auch hier die schlechtesten Ergebnisse liefert.

Diese Aufteilung bildet sich erst zwischen der 20. und der 30. Generation heraus. Davor sind die Ergebnisse der verschiedenen Crossover-Operatoren noch sehr ähnlich.

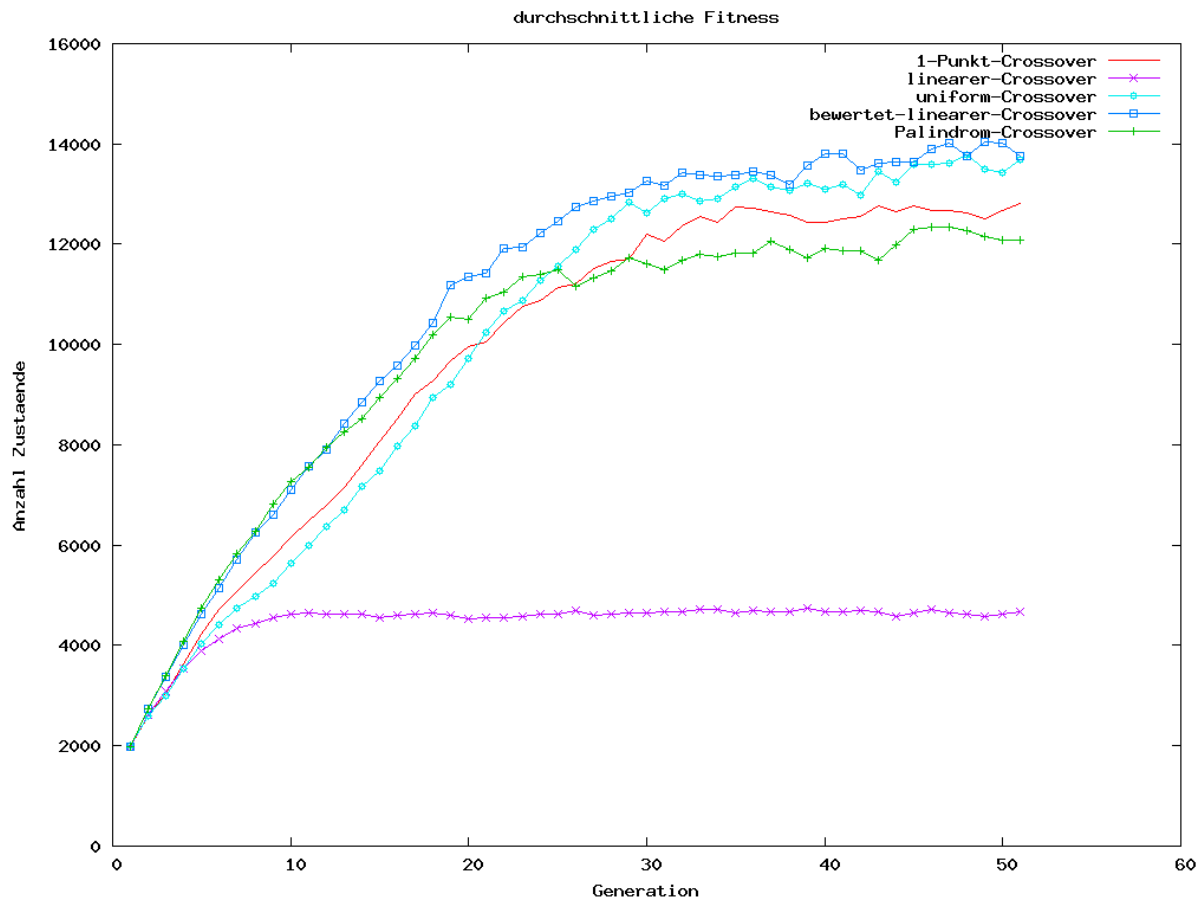


Abbildung 6.14: Der Verlauf der durchschnittlichen Fitness eines Individuums der  $i$ . Generation im Vergleich der Rekombinationen.

Auch bei dem Crossover-Operator kommen zwei Operatoren zur Auswahl in Frage, der Abschnittsbewertungs-Crossover und der uniform-Crossover. Die Wahl fällt wegen der besseren Ergebnisse bei den Durchschnitten der besten Individuen auf den uniform-Crossover.

## 6.3 Beurteilung der Selektionen

In diesem Abschnitt werden zunächst die Ergebnisse der Testläufe für jede Selektion einzeln präsentiert und besprochen. Anschließend folgt ein Vergleich der verschiedenen Selektionen und die Bewertung der Selektionen (siehe Abschnitt 6.3.4).

### 6.3.1 Fitnessproportionale Selektion

Bei der **fitnessproportionalen Selektion** (siehe 5.4.3) steigt die durchschnittliche Fitness des besten Individuums der  $i$ . Generation zunächst sehr schnell und fast linear an. Anschließend erhöht sich der Wert nur noch stufenweise und immer langsamer. Der Anstieg zwischen der 49. und der 50. Generation ist wieder auf das Hillclimbing des Genetischen Algorithmus zur Wortsuche zurückzuführen.

Die durchschnittliche Fitness eines Individuums in der  $i$ . Generation steigt bei der fitnessproportionalen Selektion zunächst schnell an. Sie verflacht dann aber entsprechend der Kurve des Durchschnitts der besten Individuen der  $i$ . Generation.

Der Verlauf der beiden Kurven ist in Abbildung 6.15 zu sehen.

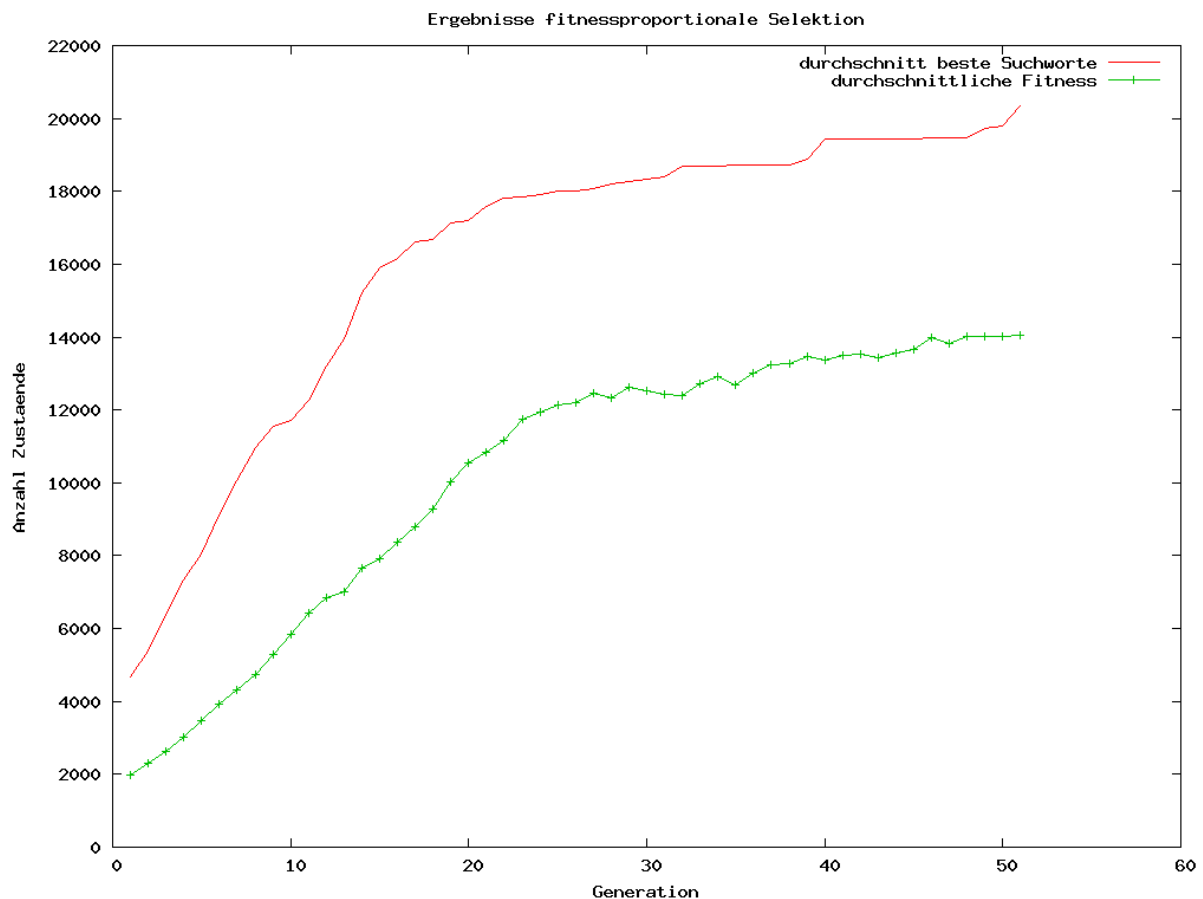


Abbildung 6.15: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation bei der fitnessproportionalen Selektion.

### 6.3.2 Rangbasierte Selektion

Der Anstieg der Fitness des Durchschnitts der besten Individuen der  $i$ . Generation verhält sich bei der **rangbasierten Selektion** (siehe 5.4.1) über die 50 Generationen fast linear, wobei der Anstieg manchmal stufenweise erfolgt. Der Anstieg von der 49. auf die 50. Generation ist auf das Hillclimbing des Genetischen Algorithmus zur Wortsuche zurückzuführen.

Die durchschnittliche Fitness eines Individuums in der  $i$ . Generation steigt bei der rangbasierten Selektion fast gar nicht an. Es gibt nur einen leichten Anstieg innerhalb der ersten Generationen. Anschließend bleibt der Wert unter 4000 Zuständen.

Die Verläufe der beiden Kurven ist in Abbildung 6.16 zu sehen.

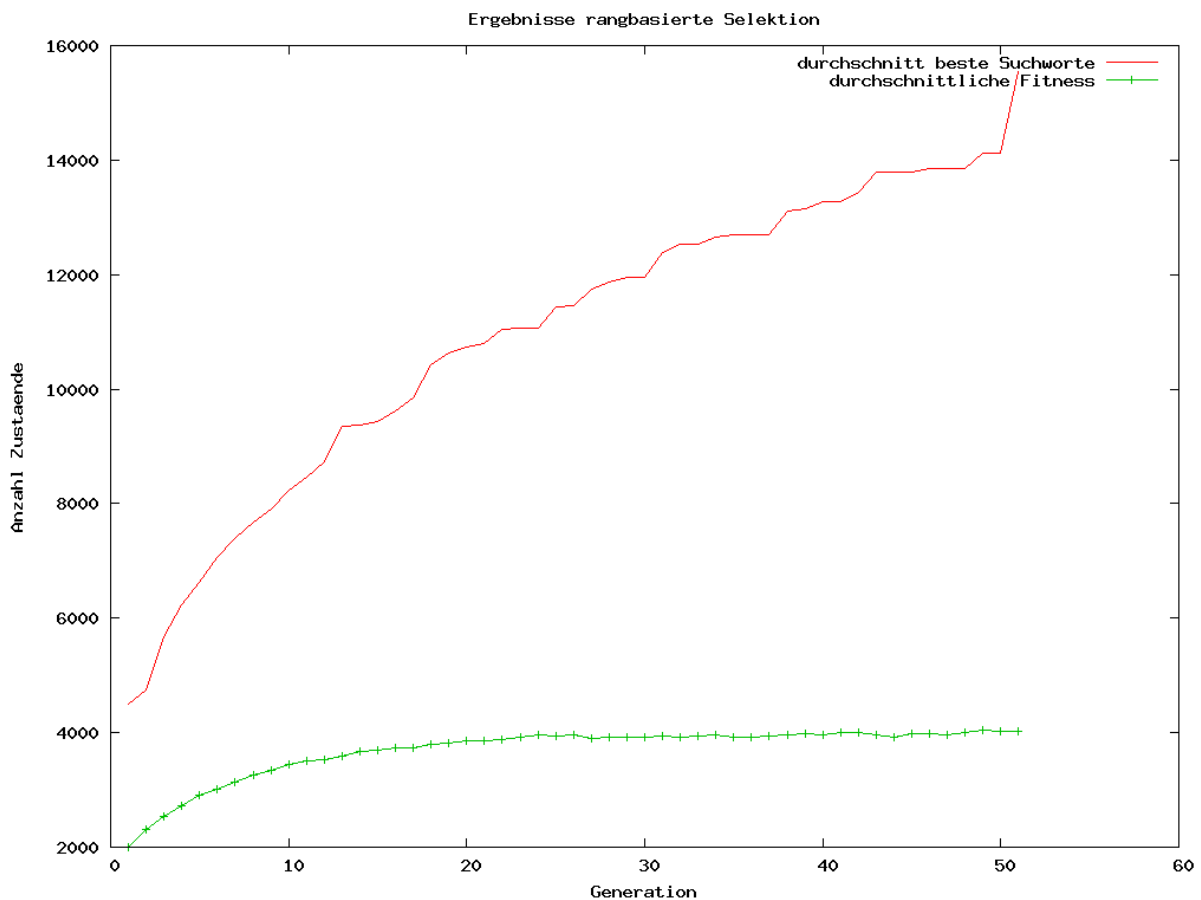


Abbildung 6.16: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation bei der rangbasierten Selektion.

### 6.3.3 Turniererlektion

Bei der Turniererlektion (siehe 5.4.2) steigt die durchschnittliche Fitness des besten Individuums einer Generation bis zur 20.-25. Generation gleichmäßig schnell an. Anschließend steigen die Werte stufenweise weiter an. Der Anstieg scheint bis zur 50. Generation nicht völlig zum Stillstand gekommen zu sein, so dass sich ein Lauf mit mehr Generationen lohnen würde. Auch das Hillclimbing zum Schluss des Genetischen Algorithmus zur Wortsuche findet kein Individuum mit einer höheren Fitness mehr.

Die durchschnittliche Fitness eines Individuums in der  $i$ . Generation steigt bei der Turniererlektion zunächst schnell, später immer langsamer an. Ab ca. der 30. Generation verbessert sich die Fitness der Individuen nicht mehr merklich. Sie bleibt unter 12000.

Die Verläufe der beiden Kurven ist in Abbildung 6.17 zu sehen.

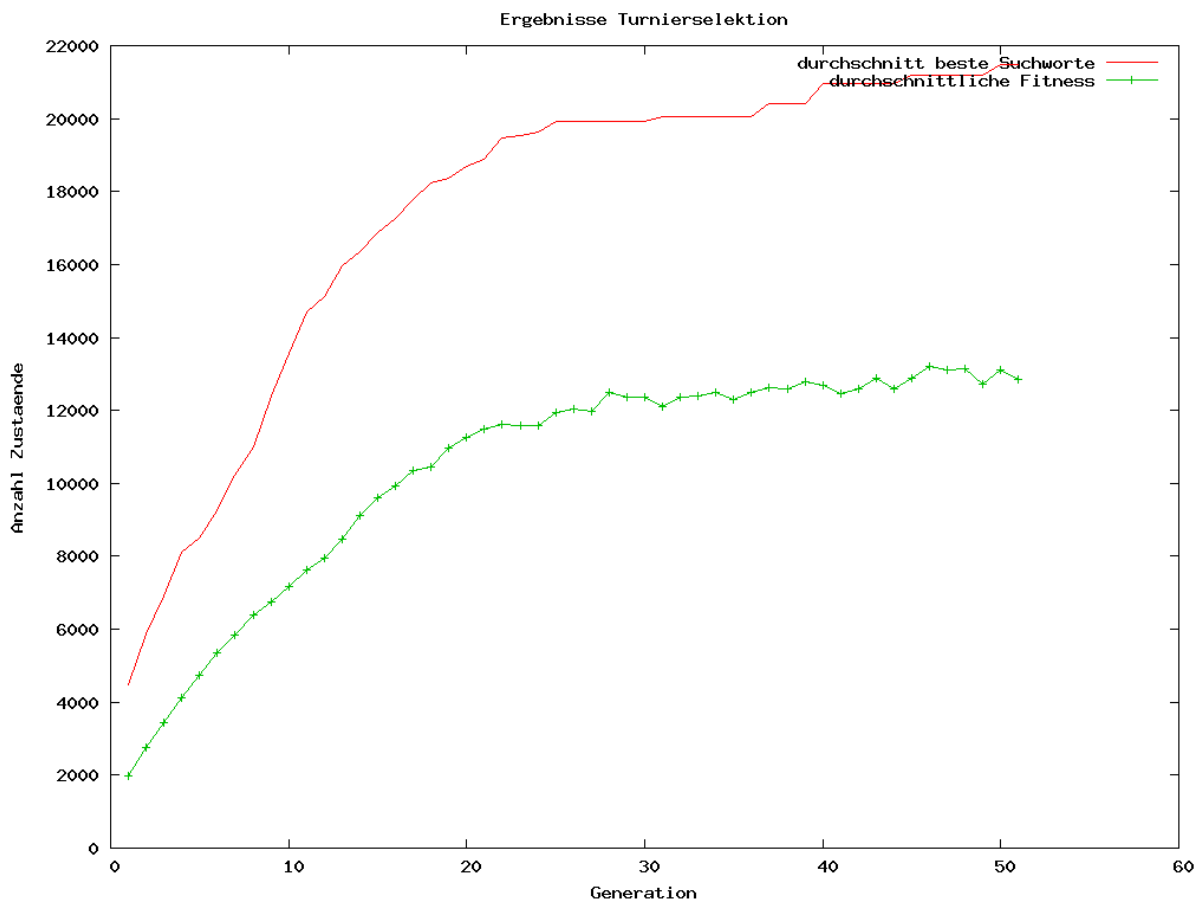


Abbildung 6.17: Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der  $i$ . Generation bei der Turniererlektion.

### 6.3.4 Fazit

Im Vergleich der drei verschiedenen Selektionen schneidet bei der durchschnittlichen Fitness der besten Individuen einer Generation die Turnierselktion am Besten ab. Sie berechnet die Individuen mit der besten Fitness. Die fitnessproportionale Selektion berechnet im Schnitt die zweitbesten Individuen. Die gefundenen besten Individuen in einer Generation haben bei der rangbasierten Selektion die niedrigsten Fitnesswerte im Vergleich aller getesteten Selektionen.

Die rangbasierte Selektion liefert mit Abstand schlechtere Ergebnisse als die beiden anderen Selektionen. Schon innerhalb der ersten Generationen ist dieses Verhalten zu erkennen. Die Werte der fitnessproportionalen und der Turnierselktion sind bis ca. zur 8. Generation sehr ähnlich. Anschließend trennen sie sich Werte auf, bleiben aber dennoch innerhalb einer Bandbreite.

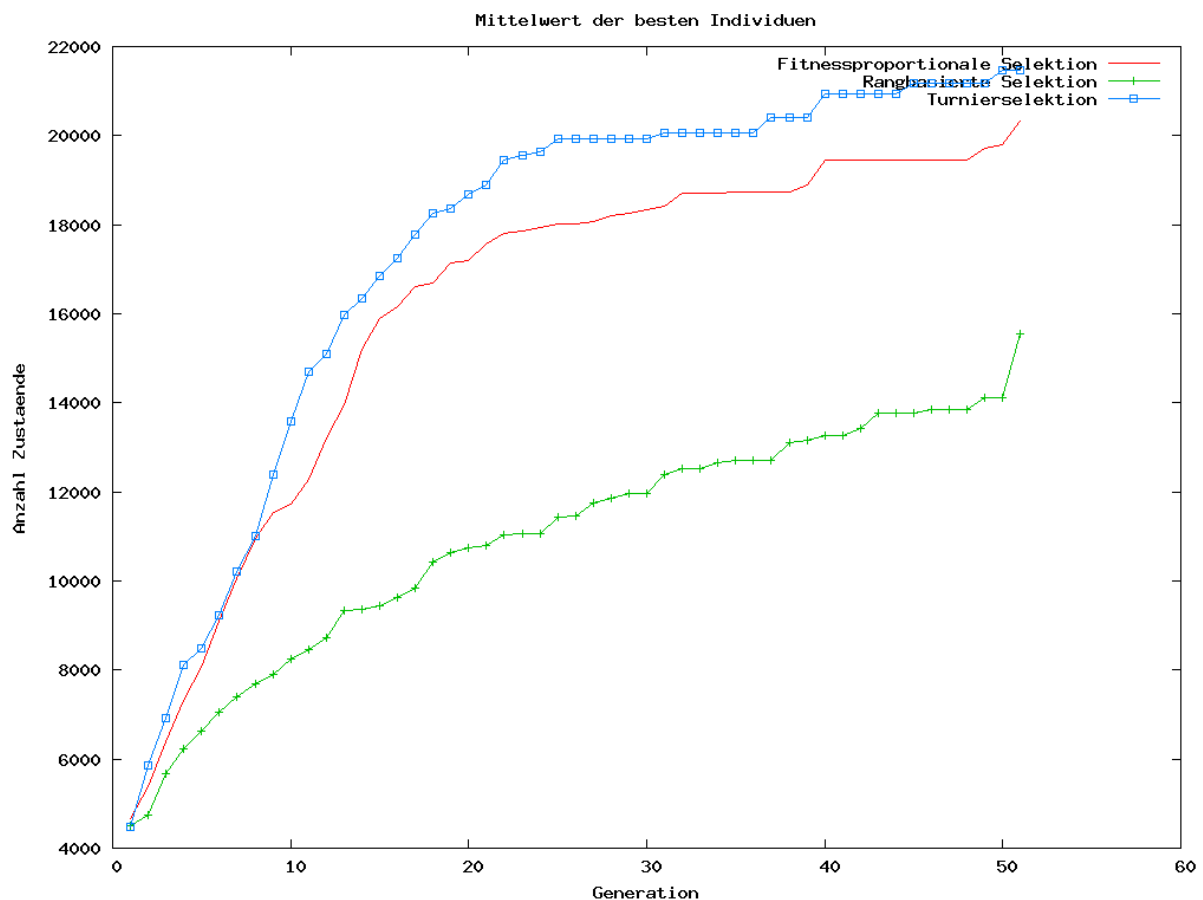


Abbildung 6.18: Der Verlauf der Fitness der besten Individuen im Vergleich der Selektionen.

Auch bei der durchschnittlichen Fitness eines Individuums der  $i$ . Generation berechnet die rangbasierte Selektion sehr viel schlechtere Individuen als die beiden anderen Selektionen. Die Ergebnisse der fitnessproportionalen und der Turnierselktion sind sehr ähnlich. Bis ca. zur 20. Generation sind die Fitnesswerte der Turnierselktion besser. Dann überholt die fitnessproportionale Selektion die Turnierselktion. Bis zur 50. Generation sind die Fitnesswerte der fitnessproportionalen Selektion höher als die der Turnierselktion.

Die Entscheidung, welche Selektion eingesetzt werden soll fällt zwischen der fitnessproportionalen und der Turnierselktion. Obwohl die durchschnittliche Fitness eines Individuums in der  $i$ . Gene-

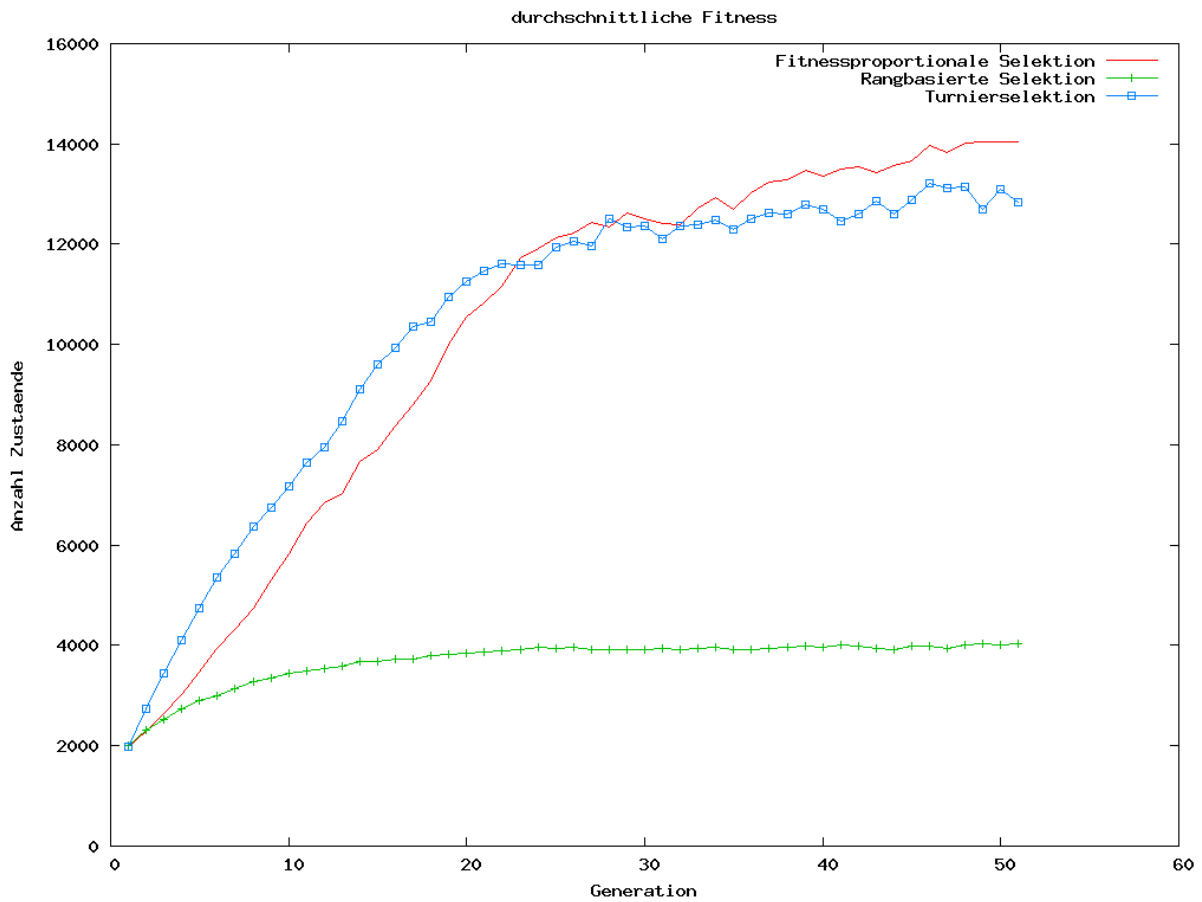


Abbildung 6.19: Der Verlauf der durchschnittlichen Fitness eines Individuums der  $i$ . Generation im Vergleich der Selektionen.

ration bei der fitnessproportionalen Selektion bessere Werte erreicht als bei der Turnierselktion wird die Turnierselktion benutzt, da die durchschnittliche Fitness der besten Individuen einer Generation bei der Turnierselktion höher ist als bei der fitnessproportionalen Selektion.

## 6.4 Parametereinstellungen

Damit die Mutationswahrscheinlichkeit und die Rekombinationswahrscheinlichkeit des Genetischen Algorithmus zur Wortsuche gut gewählt werden, wird eine Messreihe durchgeführt um geeignete Einstellungen zu finden. In der Testreihe werden im Genetischen Algorithmus zur Wortsuche die in den Abschnitten 6.1 und 6.2 gewählten Komponenten des Evolutionszyklus eingestellt und nur die Wahrscheinlichkeiten der Mutation und der Rekombination verändert. Ziel ist, **bester X** für ein Suchwort der Länge 30 zu finden. Die konkreten Einstellungen des Genetischen Algorithmus zur Wortsuche sind bei den Testläufen die folgenden:

Populationsgröße	200
Elternpopulationsgröße	200
Kinderpopulationsgröße	1000
Elternselektion	2-Turnierselektion
Umweltselektion	4-Turnierselektion
Mutation	Wurzel-BitFlip-Mutation
Rekombination	Uniform-Crossover
MutW'keit pro Individuum	variabel
RekombinationsW'keit	variabel
Anzahl Generationen	50

Tabelle 6.1: Parametereinstellungen des GA für die Bestenmutation.

Die Rekombinations- und Mutationswahrscheinlichkeiten wurden in 5 % Schritten geändert und mit jeder Kombination der Wahrscheinlichkeiten eine Messung durchgeführt. Aus zeitlichen Gründen konnten nur die Mutationswahrscheinlichkeiten von 65 % bis 75 % gemessen werden. Die Rekombinationswahrscheinlichkeiten reichen von 55 % bis 100 %.

Die Mittelwerte der gefundenen besten Individuen zum Ende der Durchläufe werden durch eine „Fitnesslandschaft“ in Abbildung 6.20 dargestellt. Dabei sind auf der x-Achse die Werte der Mutationswahrscheinlichkeit, in der y-Achse die Werte der Rekombinationswahrscheinlichkeit und in der z-Achse die durchschnittliche Bewertung des besten gefundenen Suchwortes mit den zugehörigen Wahrscheinlichkeiten für die Mutation und Rekombination gegeben.

Die gemessene „Fitnesslandschaft“ hat kein deutliches Maximum, das sich aus der Abbildung ablesen lässt. Die gemessenen Werte sind durch einzelne, in einem Testlauf gefundenen Werte, beeinflusst. Durch diese Werte entstehen lokale Maxima, die durch den Zufallseinfluss entstanden.

Es ist ein globaler Anstieg der Fitnesswerte mit steigenden Wahrscheinlichkeitswerten zu erkennen. Ab einer Rekombinationswahrscheinlichkeit von 85 % scheinen die Fitnesswerte wieder schlechter zu werden. Auch die Steigerung der Mutationswahrscheinlichkeit erhöht die Fitnesswerte.

Bei den Wahrscheinlichkeiten von 80 % für die Rekombinationswahrscheinlichkeit und 70 % für die Mutationswahrscheinlichkeit sind die Fitnesswerte hoch. Im Gegensatz zu anderen sehr guten gemessenen Werten gibt es in der Umgebung dieser Kombination auch noch andere gute Werte. Deshalb wird für die weiteren Messungen die Kombination aus 70 % Mutationswahrscheinlichkeit und 80 % Rekombinationswahrscheinlichkeit verwendet.

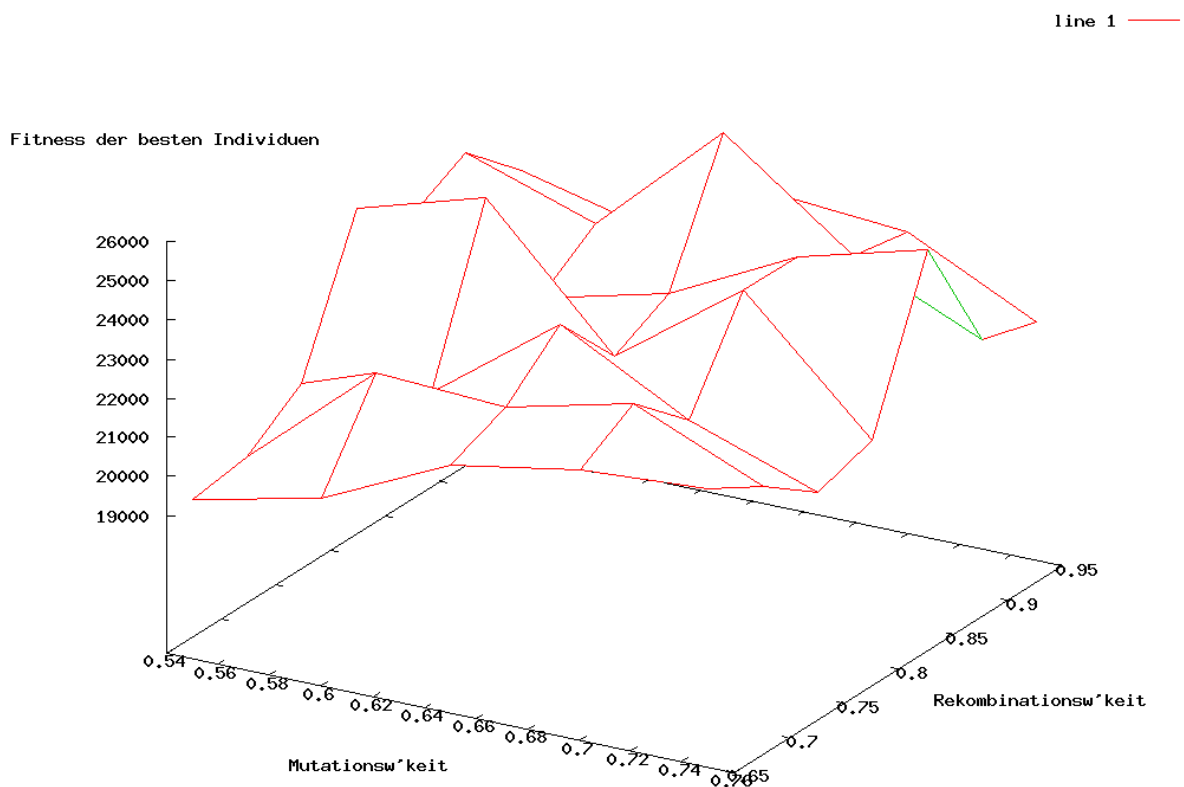


Abbildung 6.20: Die gefundenen besten Suchwörter in Abhängigkeit zur Mutationswahrscheinlichkeit und Rekombinationswahrscheinlichkeit.

## 6.5 Wachstumsverhalten der Suchwörter

In diesem Abschnitt werden die Ergebnisse des Genetischen Algorithmus zur Wortsuche besprochen. Die Ergebnisse der Suche für kurze Suchwortlängen aus Kapitel 3 sind in der Betrachtung einbezogen. Die Fragestellung, ob das Wachstum der BMA mit dem Wachstum der Suchwortlängen exponentiell oder polynomiell wächst wird dabei näher betrachtet.

### 6.5.1 2-elementiges Alphabet

Der Genetische Algorithmus zur Wortsuche wurde für längere Wortlängen als im Kapitel 3 gestartet, um beste Suchwörter für ein 2-elementiges Alphabet zu finden. Beginnend bei der Suchwortlänge 30 wurde die Suchwortlänge vergrößert.

In Tabelle 6.5.1 sind die gefundenen Ergebnisse der Suche des Genetischen Algorithmus zur Wortsuche aufgelistet.

Länge	maximales Suchwort	Zustandszahl
30	bbaabaaaabaaaaaaaaabaaaabaabb	26747
35	baaaaabaaaaaaaaababaaaaaaaaabaaaab	125029
40	bbaaaaaaaaabaaaabaaaaaaaaabaaaabaaaaaaaaabb	381154
45	bbabaaaaabaaaabaaaaababaaaaabaaaabaaaaababb	985363
50	abbaaaaabaabababaaaaabaaaaaaaaabaaaabababaaaaabb	2719023

Tabelle 6.2: Gefundene Suchwörter mit großen BMA für Alphabet  $\Sigma = \{a, b\}$ 

Im Diagramm 6.21 sieht man die durch den Genetischen Algorithmus zur Wortsuche gefundenen Ergebnisse im Vergleich zu den im Kapitel 3 durch vollständige Suche gefundenen besten Suchwörtern und den dort berechneten polynomiellen und exponentiellen Näherungsfunktionen. Das vom Genetischen Algorithmus zur Wortsuche gefundene Suchwort der Länge 30 ist dabei nicht das beste Suchwort. Deshalb gibt es einen leichten Versatz zwischen der Kurve der besten Suchwörter und der durch den Genetischen Algorithmus zur Wortsuche gefundenen Suchwörtern.

Das Diagramm zeigt die Größe der BMA's der Suchwörter in einer logarithmischen Skala, so dass besser über das Wachstumsverhalten der Werte entschieden werden kann.

Die Kurve der BMA-Größe, der vom Genetischen Algorithmus zur Wortsuche gefundenen Suchwörter, liegt zu Beginn unterhalb beider Näherungsfunktionen. Bis zur Suchwortlänge 35 verläuft die Kurve aber parallel zur exponentiellen Näherung und überholt damit die polynomielle Näherungskurve. Anschließend flacht sie etwas ab.

### 6.5.2 3-elementiges Alphabet

Der Genetische Algorithmus zur Wortsuche wurde für längere Wortlängen als im Kapitel 3 gestartet, um beste Suchwörter für ein 3-elementiges Alphabet zu finden. Beginnend bei der Suchwortlänge 20 wurde die Suchwortlänge vergrößert.

In Tabelle 6.5.2 sind die gefundenen Ergebnisse der Suche des Genetischen Algorithmus zur Wortsuche aufgelistet.

Länge	maximales Suchwort	Zustandszahl
20	aaabaaaaabaaaabaaaaac	3060
25	aaaabaaaaabaaaaaaaaabaaaaac	13188
30	aaabaaaaaaabaaaaaaaaacaaaaaab	40714
35	baaaaabaaaaaaaaababaaaaaaaaabaaaaac	297562
40	bbaaaaaaaaabaaaabaaaaabaaaabaaaaaaaaabc	495653

Tabelle 6.3: Gefundene Suchwörter mit großen BMA für Alphabet  $\Sigma = \{a, b, c\}$ 

In dem Diagramm 6.22 sieht man die durch den Genetischen Algorithmus zur Wortsuche gefundenen Ergebnisse im Vergleich zu den im Kapitel 3 durch vollständige Suche gefundenen besten Suchwörter für eine bestimmte Suchwortlänge.

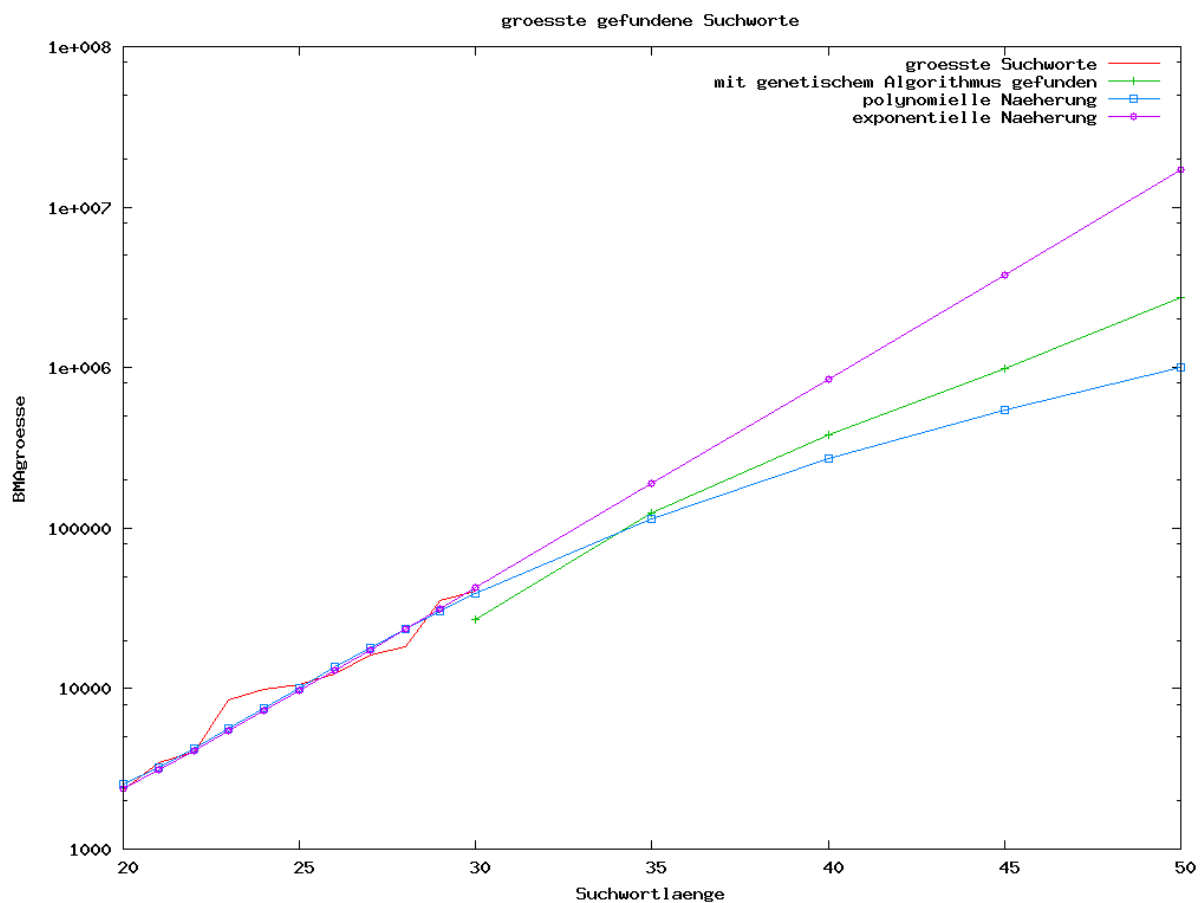


Abbildung 6.21: Das Wachstum der besten Suchwörter für Alphabet  $\Sigma = \{a, b\}$ .

Das Diagramm zeigt die Größe der BMA's der Suchwörter in einer logarithmischen Skala, so dass besser über das Wachstumsverhalten der Werte entschieden werden kann.

Bis zur Suchwortlänge 30 scheint das Wachstum die durch die vollständige Suche vorgegebene Steigung einzuhalten. Die Größe des BMA vervierfacht sich etwa bei einer Verlängerung des Suchworts um fünf Buchstaben. Der gefundene Wert für die Suchwortlänge 35 fällt auf, da die Steigerung im Vergleich zum Suchwort der Länge 30 stärker ist. Der für die Suchwortlänge 40 gefundene Wert steigert sich nicht mehr alzu stark im Vergleich zu dem Wert für die Suchwortlänge 35.

Der Verlauf der, mit dem Genetischen Algorithmus zur Wortsuche, gefundenen Suchwörter scheint die Kurve der durch vollständige Suche gefundenen Suchwörter mit größten BMA fortzusetzen. Das Wachstum scheint dabei eher exponentiell als polynomiell zu sein. Es kann aber auch hier keine definitive Aussage gemacht werden.

## 6.6 Fazit

Die Messungen, die für das 2-elementige und das 3-elementige Alphabet durchgeführt wurden zeigen, wie in den Verläufen des Wachstums in den Abbildungen 6.21 und 6.22 zu sehen, ein nicht einheitliches Bild.

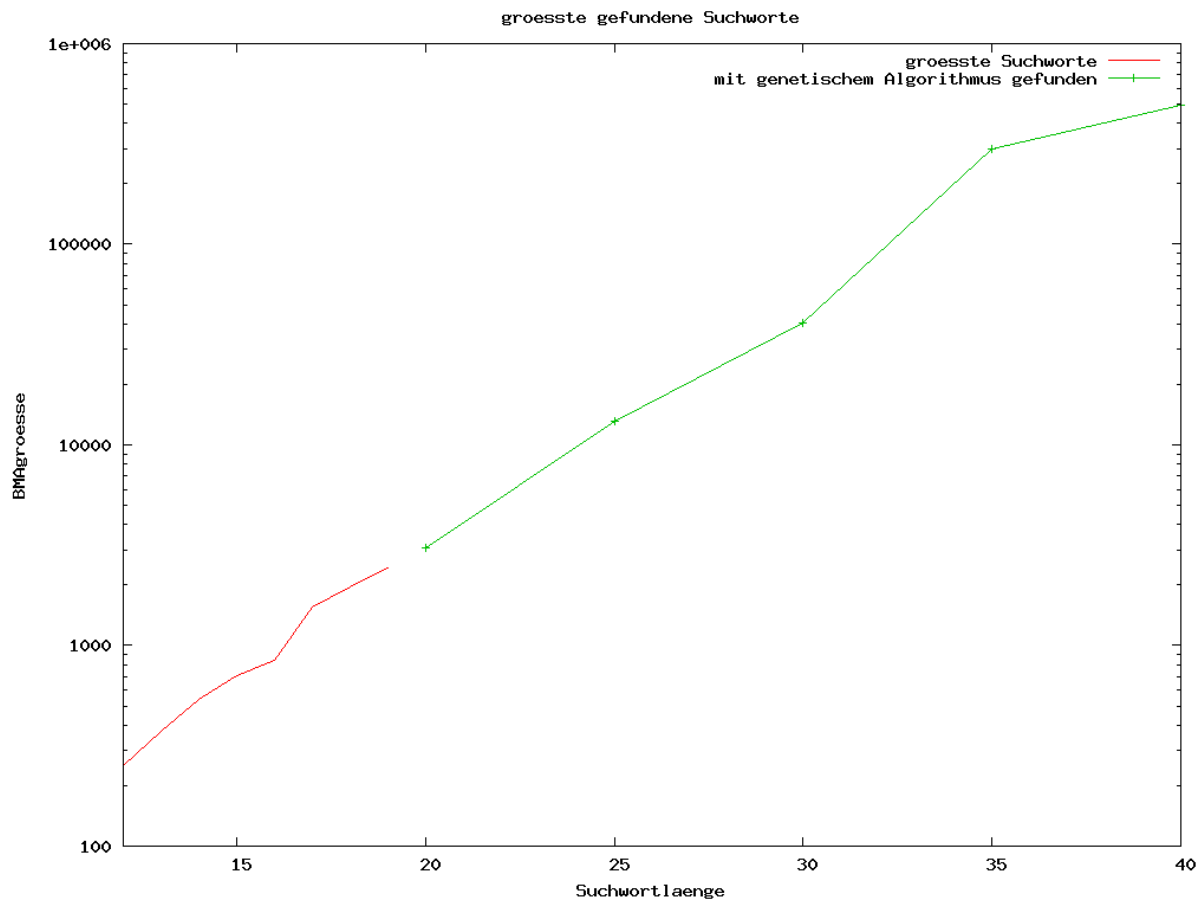


Abbildung 6.22: Das Wachstum der besten Suchwörter für Alphabet  $\Sigma = \{a, b, c\}$ .

Bei den Messungen für das 2-elementige Alphabet wird der Verlauf, der durch die besten Suchwörter bis zur Länge 30 vorgegeben ist mit einem leichten Versatz weitergeführt. Zunächst steigt die Messkurve parallel mit der exponentiellen Näherungsfunktion an, flacht dann aber ab. Dieses Abflachen kann daraus entstehen, dass der Genetische Algorithmus zur Wortsuche nicht das beste Suchwort für eine gegebene Länge findet, sondern nur ein sehr gutes Suchwort. Der Abstand in der Anzahl der Zustände zwischen diesen beiden Suchwörtern kann allerdings so groß sein, dass das Abflachen der Kurve dadurch erklärt werden kann. Die zweite Erklärung wäre, dass die besten Suchwörter selber nicht exponentiell mit der Suchwortlänge wachsen.

Auch die Messungen des 3-elementigen Alphabets zeigen ähnliche Ergebnisse. Wobei hier nur der letzte gefundene Wert nicht zu einem exponentiellen Wachstum passen würde. Die restlichen Messungen scheinen hier auf ein exponentielles Wachstum hinzudeuten.

Durch die im Abschnitt 3.4.1 beschriebenen Strukturen des Suchraums scheint die Annahme des exponentiellen Wachstums der Suchwörter stimmig. Auch bei 2-elementigen Alphabeten deutet das Wachstum bis zur Suchwortlänge 35, das parallel der exponentiellen Näherung verläuft, auf ein exponentielles Wachstum hin.

Aus zeitlichen Gründen konnten keine Berechnungen für längere Suchwörter durchgeführt werden. Der Effekt, nicht mehr das beste Suchwort zu finden, könnte sich aber bei längeren Suchwörtern immer weiter verstärken und so diese Messungen verfälschen, so dass die Messungen nur das

exponentielle Wachstum der Boyer-Moore-Automaten bestätigen können, nicht jedoch das polynomielle.

# Ausblick

---

In dieser Arbeit wurde der Genetische Algorithmus zur Wortsuche entwickelt, der zur Suche von Suchwörtern, die sehr großen Boyer-Moore-Automaten erzeugen, eingesetzt werden kann. Der Genetische Algorithmus zur Wortsuche findet dabei Suchwörter, die einen im Verhältnis zum durchschnittlichen Suchwort sehr großen Boyer-Moore-Automaten erzeugen. Er findet allerdings nicht immer das Suchwort, das den größten Boyer-Moore-Automaten für eine Suchwortlänge erzeugt. In diesen Fällen wird ein lokales Optimum gefunden, das nicht dem globalen Optimum entspricht.

Die bisherigen Berechnungen des Genetischen Algorithmus zur Wortsuche konnten keine eindeutige Aussage über das Wachstum der Boyer-Moore-Automaten erbringen. Durch aufwändigere und längere Messreihen des Genetischen Algorithmus zur Wortsuche könnte das gelingen. Auch weitere Arbeiten an den Komponenten (wie Selektionen, Mutationen) des Genetischen Algorithmus scheinen vielversprechend.

Der in dieser Arbeit implementierte Algorithmus zur Bewertung der BMA's ist exakt. Dadurch ist die Bewertung immer korrekt, allerdings ist die Laufzeit und der Speicheraufwand dieses Algorithmus abhängig von der Anzahl der Zustände. Die Laufzeit und der Speicherbedarf werden deshalb bei der Suchwortlänge 100 bereits sehr groß. Durch eine Verbesserung der Laufzeit und des Speicherbedarfs der Bewertung kann die berechenbare Suchwortlänge des Genetischen Algorithmus zur Wortsuche erheblich vergrößert werden. Die dann mögliche Berechnung sehr langer Suchwörter erleichtert die Entscheidung, in welchen Größenordnungen sich das Wachstum der Boyer-Moore-Automaten befindet, wenn die Qualität der Ergebnisse erhalten bleibt.

In dieser Arbeit wurde die Initialisierung mit zufälligen Suchwörtern durchgeführt. Die Initialisierung könnte z. B. auch mit Updated-Palindromen erfolgen. Dies könnte die Generationenzahl bis zum Finden eines guten Suchworts reduzieren, allerdings könnte es die Suche auch zu stark im Suchraum einschränken.

Für weitere Untersuchungen bietet sich außerdem die Implementierung von Individuen für 4-elementige Alphabete an, um auch auf diesem Alphabet den Genetischen Algorithmus zur Wortsuche benutzen zu können. Messungen mit größeren Suchwortlängen können weitere Erkenntnisse über das Wachstum der BMA's erbringen. Allerdings kann nicht ausgeschlossen werden, dass sich der Suchraum durch seine Beschaffenheit nicht für Evolutionäre bzw. Genetische Algorithmen eignet und selbst bei weiteren Verbesserungen der Komponenten keine eindeutige Aussage möglich ist.



# Entwurf des Genetischen Algorithmus zur Wortsuche

---

In diesem Kapitel soll ein Überblick über die Implementierung und die Klassenstruktur des Genetischen Algorithmus zur Wortsuche gegeben werden. Das Kapitel soll den Leser in die Lage versetzen, neue Klassen und Module für die Implementierung zu entwickeln, oder andere Wartungsaufgaben an der Software durchzuführen.

Die Beschreibung geht zuerst auf den Ablauf des Genetischen Algorithmus aus der Sicht der Implementierung ein. Damit wird klar welche Klassen welche Aufgaben während der Ausführung der Berechnung haben. Anschließend werden die einzelnen abstrakten Klassen und die Eigenschaften, die die abgeleiteten Klassen implementieren müssen, beschrieben.

## A.1 Die verschiedenen Komponenten des Genetischen Algorithmus in der Implementierung

In diesem Abschnitt werden die Komponenten eines Genetischen Algorithmus, wie z. B. die Selektion und die Rekombination zu den konkreten Klassen der Implementierung zugeordnet. Der Abschnitt A.1.1 beschreibt dazu zunächst, wie die Berechnung des Genetischen Algorithmus zur Wortsuche von einem Programm aus gestartet werden kann. Im Abschnitt A.1.2 wird dann Anhand des Evolutionszyklus die Zuordnung der Komponenten eines Genetischen Algorithmus zu den Klassen des Genetischen Algorithmus zur Wortsuche vorgenommen.

### A.1.1 Einbindung der Komponenten in den Ablauf

Die Implementierung des Genetischen Algorithmus ist allgemein gehalten. Sie geht dennoch davon aus, dass mit dem Genetischen Algorithmus ein Optimierungsproblem gelöst werden soll. Mit welchem Verfahren versucht wird Optima für das Optimierungsproblem zu finden ist allerdings nicht festgelegt. Die Klasse `Algorithmus` kapselt die Funktionalität der Suche nach Optima. Ein Aufrufendes Programm muss der Klasse `Algorithmus` eine Instanz einer von `Strategie` abgeleiteten Klasse zur Initialisierung übergeben. In `Strategie` wird die Strategie festgelegt, mit der nach Optima gesucht wird. Durch den Aufruf von `berechneOptimas` von `Algorithmus` wird die Berechnung gestartet.

### A.1.2 Die implementierten Komponenten des Genetischen Algorithmus zur Wortsuche

Wenn als Strategie eine Instanz der Klasse `EAStrategie` übergeben wurde, so müssen noch die Parameter für den Genetischen Algorithmus in dieser Klasse initialisiert werden. Diese Initialisierung entspricht der Einstellung der Parameter des Genetischen Algorithmus. Hier werden die Selektionen, die Rekombination, die Mutation, die Initialisierung und die Individuen des Genetischen Algorithmus zur Wortsuche eingestellt.

Bestimmte Einstellungen sind nicht, oder nur schlecht über diese Parameter möglich. Deshalb ist z. B. auch noch die Klasse `ElitenEvolutionStrategie` vorhanden, die dafür sorgt, dass immer die  $n$  ( $n$  wählbar) besten Individuen einer Generation in die nächste Generation übernommen werden. Es wäre auch möglich gewesen über spezielle Selektionen dieses Verhalten zu erzeugen, jedoch ist es so möglich alle schon bestehenden Selektionen weiter zu verwenden.

Der Implementierte Selektionszyklus der Klasse `EAStrategie` benutzt eine Implementierung einer Unterklasse von `AbstrakteInitialisierung`, um eine Initiale Population von Individuen herzustellen. Zur Verwaltung einer Population wird die Klasse `Population` verwendet. Ein Individuum ist eine Instanz einer Unterklasse von `AbstractIndividuum`. Um Individuen während des Evolutionszyklus zu erstellen, wird eine Instanz einer Unterklasse von `AbstractIndividuumFactory` benötigt.

Die Abbruchbedingung des Evolutionszyklus wird durch eine Implementierung einer Unterklasse von `AbstrakteAbbruchbedingung` dargestellt. Entscheidet diese Abbruchbedingung, dass der Evolutionszyklus durchgeführt werden soll, wird die Elternselektion durchgeführt. Hierfür wird eine Implementierung einer Unterklasse von `AbstrakteSelektion` verwendet. Die Elternselektion erzeugt die in 5.1.4 beschriebene Elternpopulation.

Aus der Elternmenge werden die Individuen zur Rekombination ausgewählt. Die rekombinierten Individuen werden anschließend mutiert. Diese beiden Schritte werden von einer Instanz einer Unterklasse von `AbstraktKinderErzeuger` durchgeführt.

Anschließend wird die Umweltselektion von einer Instanz einer Unterklasse von `AbstrakteSelektion` durchgeführt. Die hieraus hervorgehende Population wird  $pop_{t+1}$ .

Nun beginnt der Evolutionszyklus wieder beim Abprüfen der Abbruchbedingung.

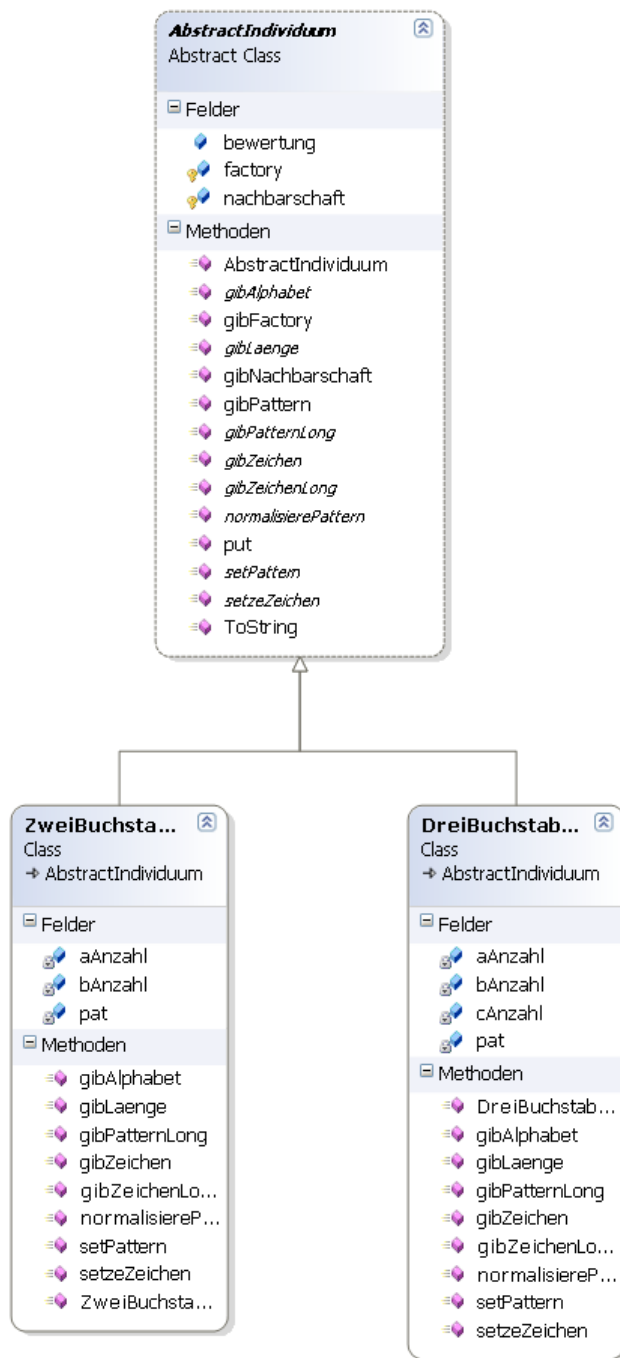
## A.2 Die abstrakten Klassen

In diesem Abschnitt werden die einzelnen abstrakten Klassen, die in A.1.2 erwähnt wurden kurz beschrieben und es wird auf einige Besonderheiten eingegangen.

### A.2.1 `AbstractIndividuum`

Eine Instanz einer Unterklasse von `AbstractIndividuum` repräsentiert ein Individuum. Im Rahmen dieser Arbeit wurden die Klassen für 2- und 3-elementige Individuen implementiert. Für weitere Untersuchungen können z. B. 4-elementige Individuen leicht integriert werden, indem eine Klasse für 4-elementige Individuen implementiert wird.

Eine Instanz von `AbstractIndividuum` wird nicht direkt über den Konstruktor der Klasse erzeugt. Es wird immer die `AbstractIndividuumFactory` (siehe A.2.2) verwendet, um Objekte dieser Klasse zu erzeugen.

Abbildung A.1: Die von `AbstractIndividuum` abgeleiteten Klassen.

Die abstrakte Klasse bietet einige Funktionen an, die von den Unterklassen implementiert werden müssen. Einige werden mit ihrer Bedeutung hier aufgeführt:

**gibAlphabet** Diese Methode liefert das Alphabet des Individuums zurück. Das Alphabet wird in einem Aufzählungstyp abgespeichert. Diese Methode ist notwendig, damit die Methoden des Genetischen Algorithmus allgemein programmiert werden können.

**gibPatternLong** Diese Methode liefert das Suchwort des Individuums (seinen Genotyp) als `long[]` Array zurück. Alle Aufzählungstypen können ihre Werte in `long` Werte umwandeln, somit ist diese Methode immer möglich.

**gibZeichen** Liefert das Zeichen zu einem in der Methode übergebenen Index als `long` zurück.

**setzeZeichen** Setzt das Zeichen des Suchworts an dem in der Methode übergebenen Index auf den Buchstaben, der in der Methode als `long` übergeben wird.

## A.2.2 AbstractIndividuumFactory

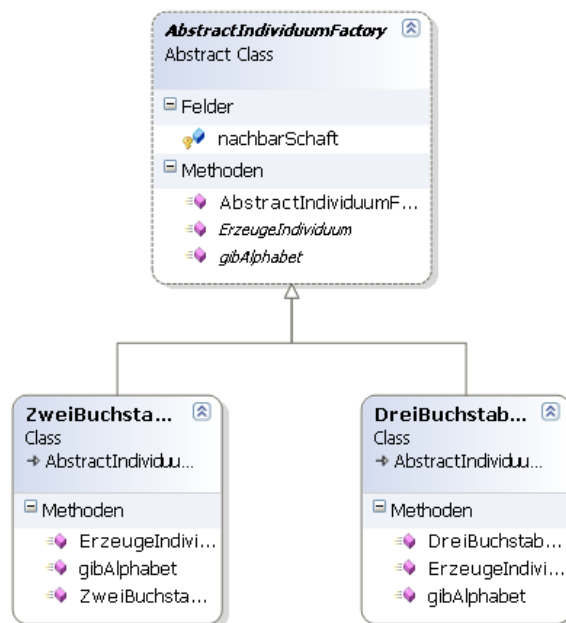


Abbildung A.2: Die von `AbstractIndividuumFactory` abgeleiteten Klassen.

Ein Objekt einer Unterklasse von `AbstractIndividuumFactory` hat die Aufgabe Objekte einer Unterklasse von `AbstractIndividuum` zu erzeugen. In der derzeitigen Implementierung gibt es für jede Unterklasse von `AbstractIndividuum` genau eine factory-Klassen, die Individuen erzeugen kann.

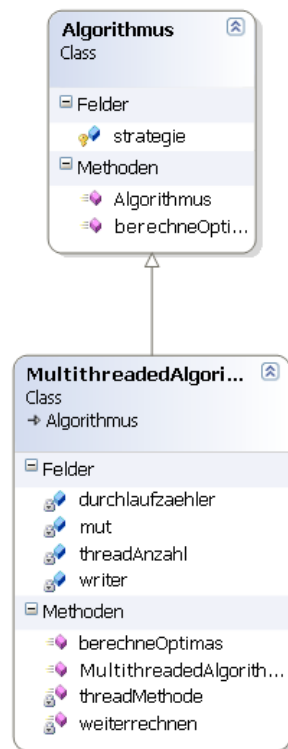
Die Methoden, die von Unterklassen implementiert werden müssen sind `ErzeugeIndividuum` und `gibAlphabet`.

**ErzeugeIndividuum** Erzeugt ein Objekt einer Unterklasse von `AbstractIndividuum` mit den Initialisierungen, die in dieser Methode mitgegeben werden (z. B. das Suchwort).

**gibAlphabet** Gibt das Alphabet der Objekte zurück, die von dieser Factory erzeugt werden können.

## A.2.3 Algorithmus

Diese Klasse wurde in die Klassenstruktur eingefügt, damit die **Strategie** von der Speicherung der berechneten Daten entkoppelt werden kann.

Abbildung A.3: Die Klasse `Algorithmus`.

In `Algorithmus` wird `berechneOptimas` von `Strategie` ausgeführt und anschließend werden die berechneten Ergebnisse in eine Datei ausgeschrieben. Es ist auch möglich, mehrere Durchläufe des Evolutionären `Algorithmus` durchzuführen und in mehrere Dateien zu schreiben. Diese Verarbeitung wird in `Algorithmus` durchgeführt. Sollten die Ausgabedateien geändert werden, oder eine andere Form der Ausgabe gewünscht werden, kann dies in dieser Klasse geändert werden.

#### A.2.4 Strategie

In denen von `Strategie` abgeleiteten Klassen wird die Suche nach Optima für das Optimierungsproblem (hier die Suche nach bester  $w$ ) durchgeführt. In dieser Arbeit wurden Unterklassen implementiert, die einen Genetischen `Algorithmus` für diese Suche implementieren, es könnten hier auch andere Ansätze implementiert werden. Die Unterklasse müssen die Methode `berechneOptimas` implementieren und als Ergebnis eine Menge von Individuen zurückgeben.

#### A.2.5 AbstrakteInitialisierung

Unterklassen von `AbstrakteInitialisierung` müssen die Methode `init` implementieren. In dieser Methode erhält das Objekt die Größe der zu erzeugenden Population und ein `Factory`-Objekt, um die Individuen zu erzeugen. Auch die Länge des Suchwortes, das die Individuen haben müssen wird mit übergeben. Als Ausgabe muss ein `Population`-Objekt zurückgegeben werden, das die geforderten Eigenschaften erfüllt.

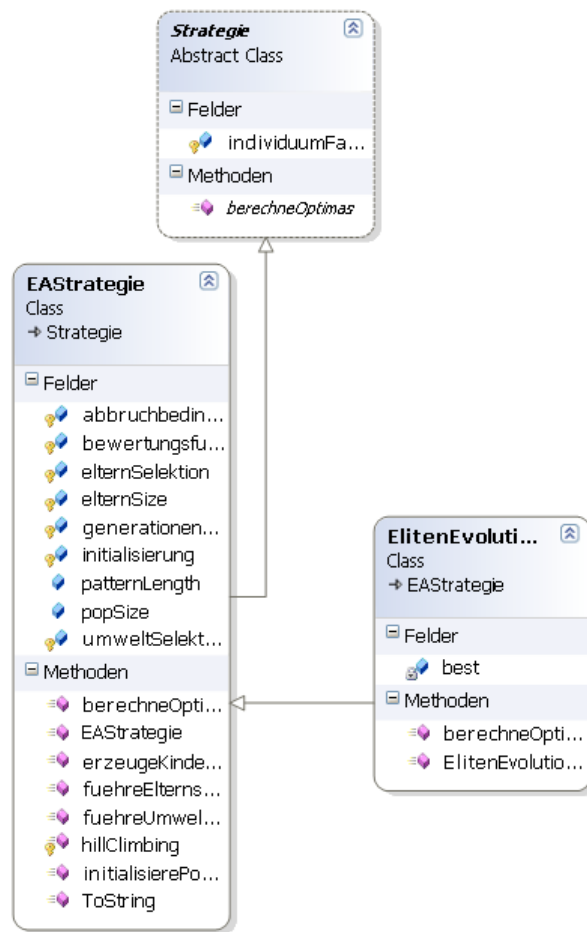


Abbildung A.4: Die von Strategie abgeleiteten Klassen.

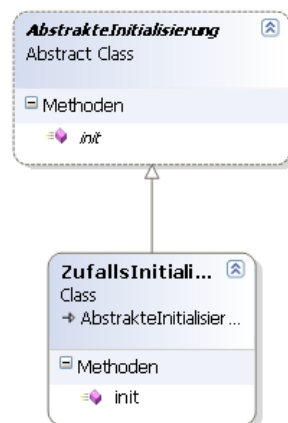


Abbildung A.5: Die von AbstrakteInitialisierung abgeleiteten Klassen.

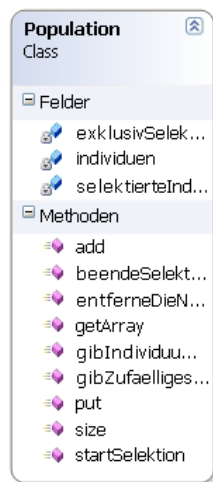


Abbildung A.6: Die Klasse Population.

## A.2.6 Population

Ein Objekt dieser Klasse enthält eine Menge von Individuen. In der Implementierung wird der Ansatz verwendet, dass jedes Individuum eine Zusatzinformation verwaltet, so dass Individuen mit gleichem Genotyp in einer Menge verwaltet werden können (siehe Abschnitt 4.4). Dies wird dadurch erreicht, dass die `equals` Methode bei `AbstractIndividuum` nicht überschrieben wird.

So kann eine Population die Individuen als `Collection` verwalten.

## A.2.7 AbstrakteAbbruchbedingung

Unterklassen von `AbstrakteAbbruchbedingung` müssen die Methode `weitermachen` implementieren. Der Rückgabewert entscheidet, ob der Evolutionszyklus weiter durchlaufen wird oder nicht. In dieser Arbeit wurde der Abbruch nach einer festgelegten Anzahl von Generationen implementiert. Es könnten auch andere Abbruchbedingungen implementiert werden (z. B. Zeitbegrenzung, Qualität der Population).

## A.2.8 AbstrakteSelektion

Unterklassen von `AbstrakteSelektion` sollen eine Selektion durchführen. Hierzu müssen sie die Methode `selektion` implementieren. In dieser Methode erhält die Selektion ein `Population`-Objekt und die Anzahl von Individuen, die selektiert werden sollen. Die Selektion gibt bei dieser Methode eine Population zurück, die aus den selektierten Individuen besteht. Mit diesem Konzept kann sowohl die Eltern- als auch die Umweltselektion durchgeführt werden.

Bei der Elternselektion wird die Elternpopulation  $par_t$  erzeugt, mit der dann die Rekombination durchgeführt wird und bei der Umweltselektion wird die Population  $pop_{t+1}$  zurückgegeben.

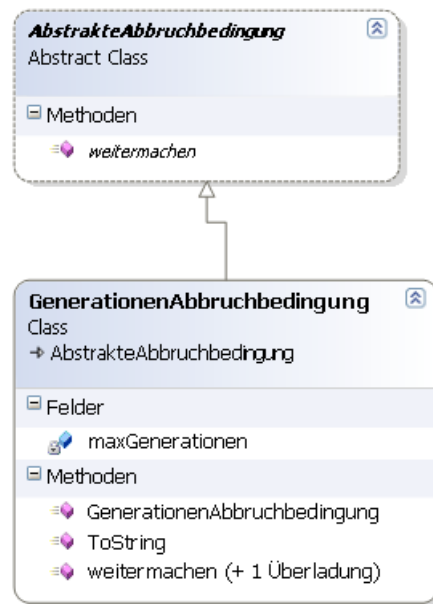


Abbildung A.7: Die von AbstrakteAbbruchbedingung abgeleiteten Klassen.

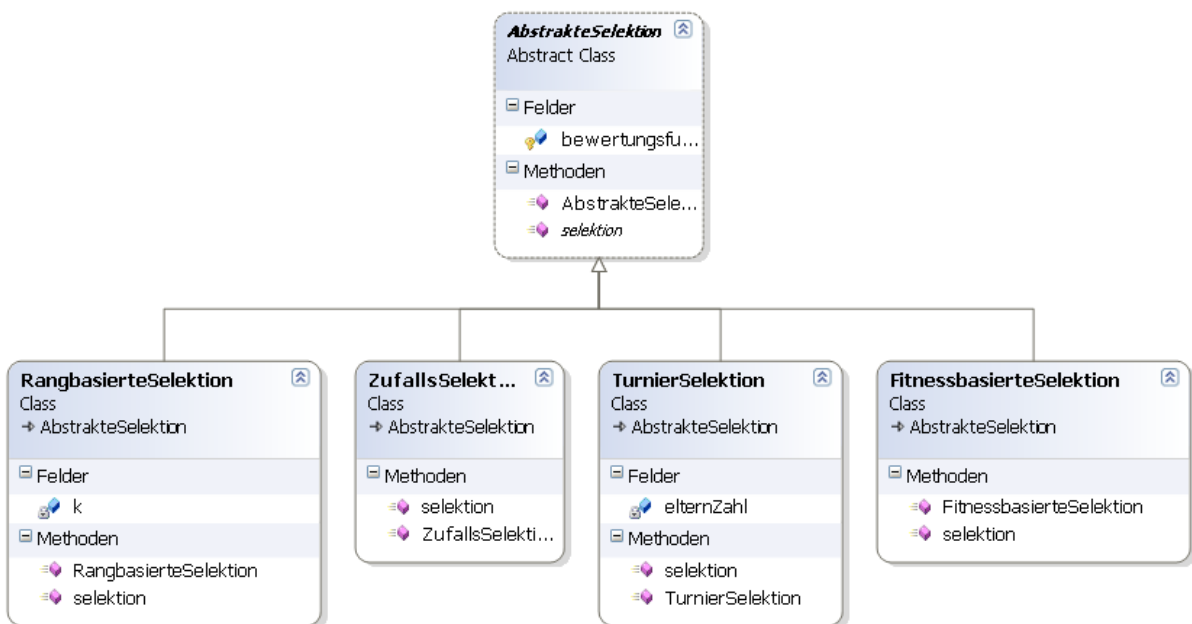
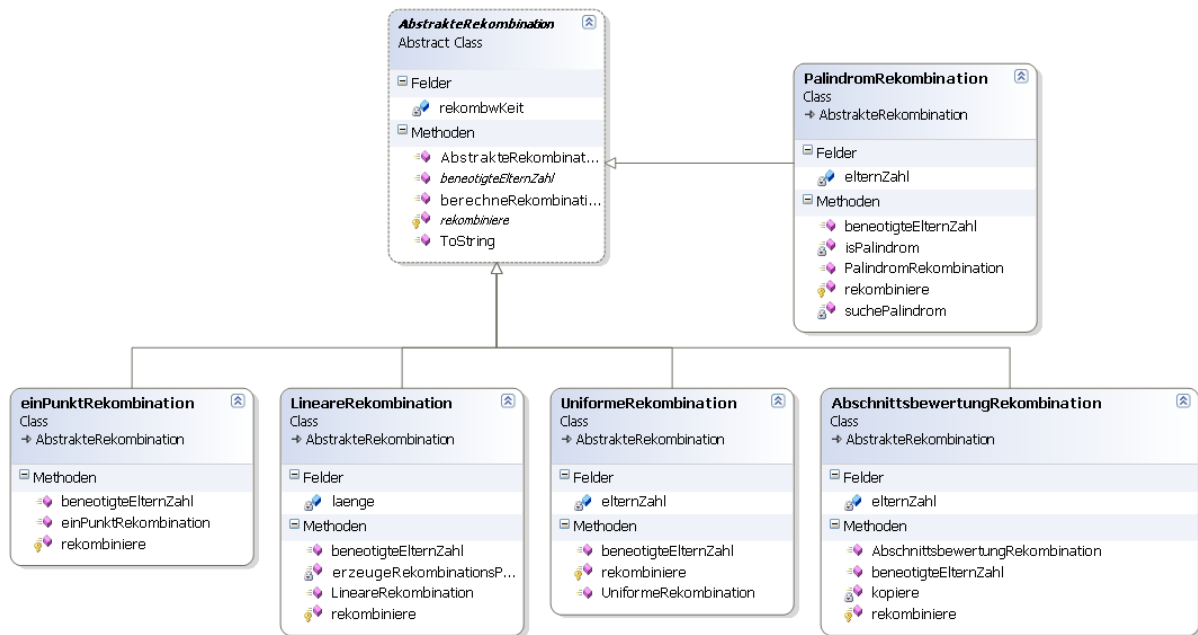


Abbildung A.8: Die von AbstrakteSelektion abgeleiteten Klassen.

### A.2.9 AbstrakteRekombination

Unterklassen der `AbstrakteRekombination` führen die Rekombination durch. Hierzu müssen sie die Methoden `rekombiniere` und `benoetigteElternzahl` implementieren. Das zufällige Entscheiden, ob eine Rekombination durchgeführt werden soll oder nicht wird von der abstrakten Klasse

Abbildung A.9: Die von `AbstrakteRekombination` abgeleiteten Klassen.

übernommen und nur dann `rekombiniere` aufgerufen, wenn eine Rekombination durchgeführt werden soll.

**rekombiniere** In dieser Methode wird eine Menge von Eltern übergeben, außerdem ein Factory-Objekt. Mit diesem Factory-Objekt sollen die Kindindividuen erzeugt werden. Anschließend werden die Kindindividuen als Menge zurückgegeben. Es dürfen 1 bis *anzahlEltern* viele Kindindividuen zurückgegeben werden.

**benoetigteElternzahl** Mit dieser Methode muss die Rekombination die Anzahl der Eltern zurückgeben, die für die Rekombination benötigt werden.

### A.2.10 AbstrakteMutation

Unterklassen von `AbstrakteMutation` führen die Mutation durch. Hierzu müssen sie die Methode `mutiere` implementieren. Mit dieser Methode wird ein Individuum mutiert. Die Entscheidung, ob ein Individuum mutiert wird oder nicht wird von der abstrakten Klasse getroffen und nur dann `mutiere` aufgerufen, wenn das Individuum mutiert werden soll.

### A.2.11 AbstraktKinderErzeuger

Unterklassen von `AbstraktKinderErzeuger` führen den Schritt von einer Elternpopulation ausgehend bis zur Population der mutierten Kinder aus. Dabei wird normalerweise die Rekombination zur Erzeugung der Kinder und anschließend gegebenenfalls eine Mutation durchgeführt. Unterklassen müssen dieses Verfahren natürlich nicht genau so durchführen, sondern können abweichen. Dieser Ablauf ist in `AbstraktKinderErzeuger` vorgegeben, dadurch können kleinere Abweichungen leicht realisiert werden.

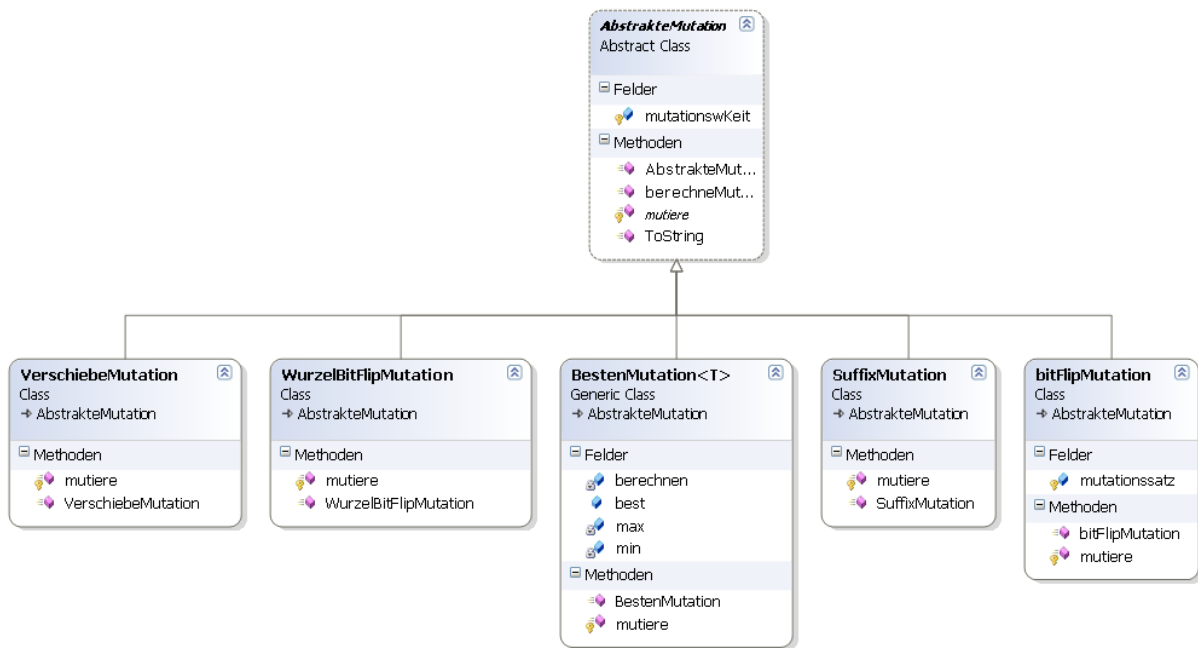


Abbildung A.10: Die von `AbstrakteMutation` abgeleiteten Klassen.

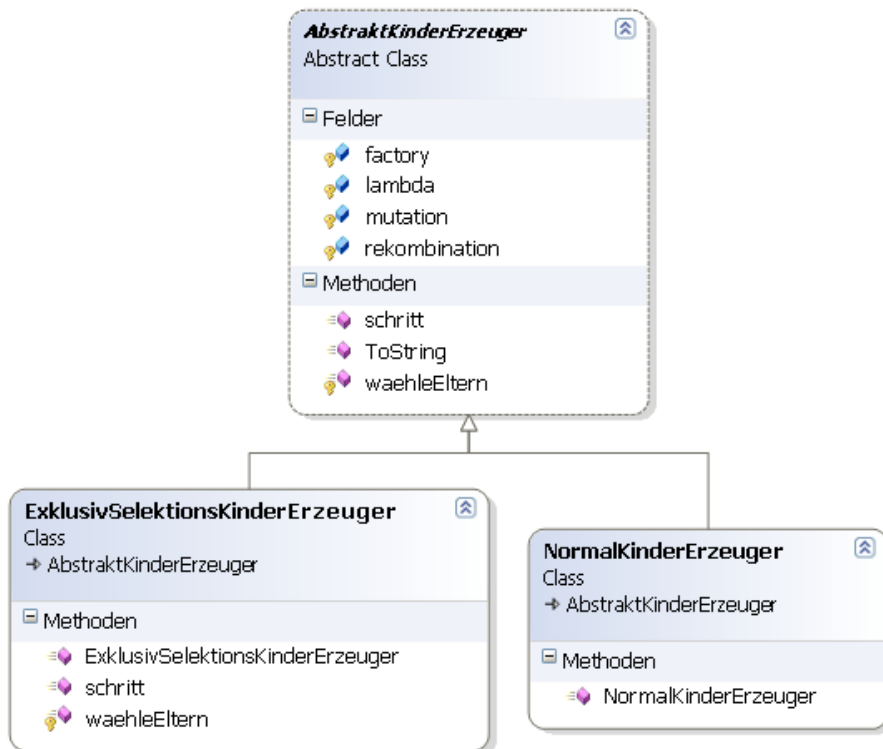


Abbildung A.11: Die von `AbstraktKinderErzeuger` abgeleiteten Klassen.

So kann z. B. die Auswahl der Elternindividuen aus der Elternmenge von einer zufälligen Auswahl auf eine Fitnessbasierte umgestellt werden.

Unterklassen können die Methoden `schritt` und `wahleEltern` überschreiben.

**schritt** Diese Methode führt den gesamten Ablauf, von der Elternpopulation zur Population der mutierten Kinder aus. Wenn diese Methode überschrieben wird, muss die Implementierung eine Population zurückgeben.

**wahleEltern** Mit dieser Methode werden die Individuen aus der Elternpopulation ausgewählt, die mit der Rekombination zur Erzeugung von Kindern verwendet werden sollen. Sie gibt die Menge von Individuen zurück, die für eine Rekombination ausgewählt wurden. `wahleEltern` wird so oft ausgeführt, wie Rekombinationen durchgeführt werden.

### A.2.12 AbstrakteBewertungsfunktion

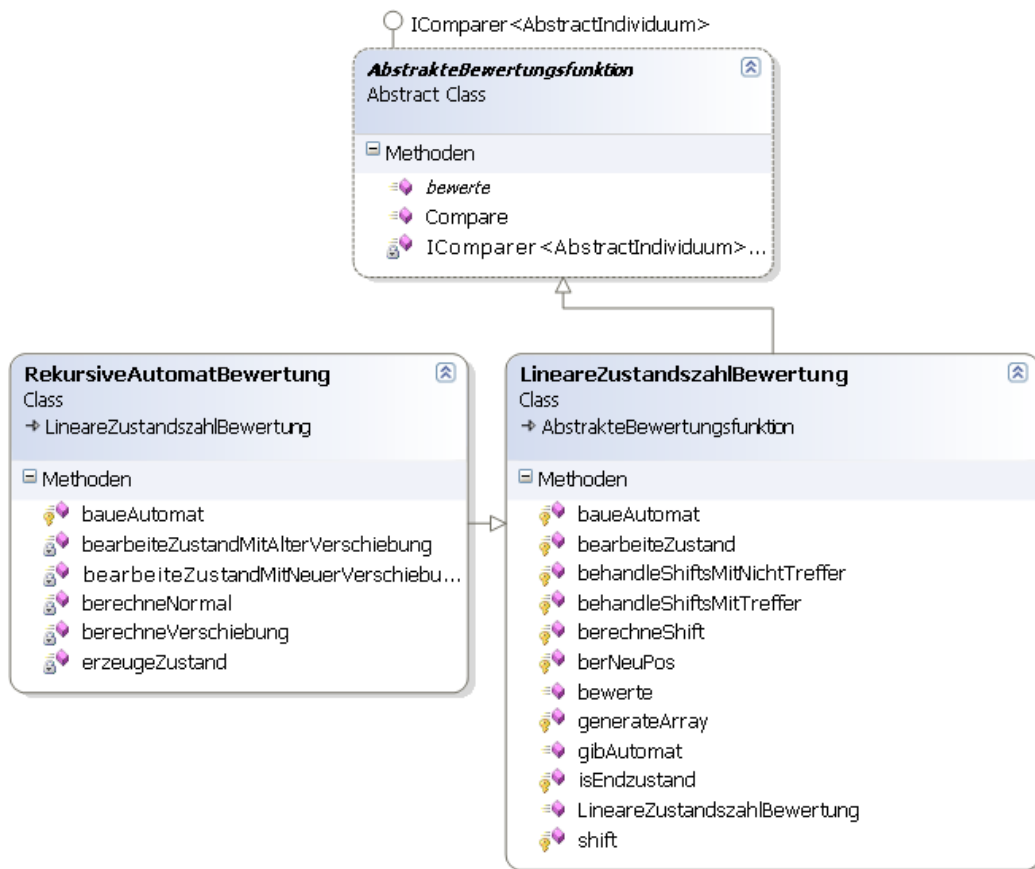


Abbildung A.12: Die von `AbstrakteBewertungsfunktion` abgeleiteten Klassen.

Unterklassen von `AbstrakteBewertungsfunktion` werden zur Bewertung von einem Individuum benutzt. In dieser Arbeit wurde die exakte Bewertung der Individuen nach der Anzahl der Zustände der von ihnen erzeugten Boyer-Moore-Automaten implementiert.

Unterklassen müssen die Methode `bewerte` implementieren. In dieser Methode wird die Fitness eines Individuums berechnet und als `double` zurückgegeben.



## ANHANG B

# CD

---

Auf der CD befinden sich die Quellcodes und Projekte, die während dieser Arbeit entstanden. Auch die Auswertungen sind auf der CD abgespeichert.

Genauere Beschreibungen finden sich in den `ReadMe.txt` Dateien auf der CD.



# Literaturverzeichnis

---

- [AC75] AHO, Alfred V. ; CORASICK, Margaret J.: Efficient String Matching: An Aid to Bibliographic Search. In: *Comm. ACM* 18 (1975), S. 333–340
- [BBYDS96] BRUYERE, Veronique ; BAEZA-YATES, Ricardo ; DELGRANGE, Olivier ; SCHEIHING, Rodrigo. *About the Size of Boyer-Moore Automata*. 1996
- [BM77] BOYER, R.S. ; MOORE, J.S.: A fast string searching algorithm. In: *Comm. ACM* 20 (1977), S. 762–772
- [BYCG94] BAEZA-YATES, R. A. ; CHOFFRUT, C. ; GONNET, G.H.: On Boyer-Moore Automata. In: *Algorithmica* 12 (1994), S. 268–292
- [Cla06] CLAUS, Volker. *Skript zur Vorlesung Evolutionäre Algorithmen*. 2006
- [Dav91] DAVIS, Lawrence: *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991
- [Dep] DEPARTMENT, Heikki H. *On Boyer-Moore Preprocessing*
- [Hen98a] HENNIG, Wolfgang: *Genetik*. Springer, 1998, S. 8–9
- [Hen98b] HENNIG, Wolfgang: *Genetik*. Springer, 1998, S. 685–686
- [Hen98c] HENNIG, Wolfgang: *Genetik*. Springer, 1998, S. 82
- [Hen98d] HENNIG, Wolfgang: *Genetik*. Springer, 1998, S. 450
- [KJP77] KNUTH, D.E. ; JR, J.H. M. ; PRATT, V.R.: Fast pattern matching in strings. In: *SIAM J. Comput.* 6 (1977), S. 323–350
- [LM03] LESCH, Harald ; MÜLLER, Jörn: *Big Bang, zweiter Akt*. Goldmann, 2003
- [May02] MAYR, Ernst: *what evolution is*. Weidenfeld & Nicolson, 2002, S. 118
- [Nis97a] NISSEN, Volker: *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997, S. 66–70
- [Nis97b] NISSEN, Volker: *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997, S. 65–67
- [Nis97c] NISSEN, Volker: *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997
- [Nis97d] NISSEN, Volker: *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997, S. 34–35
- [Nis97e] NISSEN, Volker: *Einführung in Evolutionäre Algorithmen*. Vieweg, 1997, S. 184
- [SCS05a] STEPHEN C. STEARNS, Rolf F. H.: *Evolution: an introduction*. Oxford University Press, 2005, S. 85
- [SCS05b] STEPHEN C. STEARNS, Rolf F. H.: *Evolution: an introduction*. Oxford University Press, 2005, S. 185–186
- [SCS05c] STEPHEN C. STEARNS, Rolf F. H.: *Evolution: an introduction*. Oxford University Press, 2005, S. 106–107
- [Sey98a] SEYFFERT, Wilhelm: *Genetik*. Gustav Fischer Verlag, 1998, S. 596

- [Sey98b] SEYFFERT, Wilhelm: *Genetik*. Gustav Fischer Verlag, 1998, S. 362
- [Sey98c] SEYFFERT, Wilhelm: *Genetik*. Gustav Fischer Verlag, 1998, S. 626
- [Wei02a] WEICKER, Karsten: *Evolutionäre Algorithmen*. Teubner, 2002
- [Wei02b] WEICKER, Karsten: *Evolutionäre Algorithmen*. Teubner, 2002, S. 43
- [Wei02c] WEICKER, Karsten: *Evolutionäre Algorithmen*. Teubner, 2002, S. 68
- [Wei02d] WEICKER, Karsten: *Evolutionäre Algorithmen*. Teubner, 2002, S. 95–99
- [Zie96] ZIEGLER, Bernhard: ESS - Ein schneller Algorithmus zur Mustersuche in Zeichenfolgen. In: *Informatik Forsch. Entw.* 11 (1996), S. 69–83

# Tabellenverzeichnis

---

2.1	Das Suchwort $w$ und der Text $t$ des Beispiels. . . . .	13
2.2	Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet $\Sigma = \{a, b\}$ . .	33
3.1	Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet $\Sigma = \{a, b\}$ . .	41
3.2	Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet $\Sigma = \{a, b\}$ . .	42
3.3	Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet $\Sigma = \{a, b, c\}$ .	44
3.4	Suchwörter mit den größten Boyer-Moore-Automaten für Alphabet $\Sigma = \{a, b, c, d\}$	44
3.5	Wachstum des Boyer-Moore-Automaten für Einstufen-Palindrome $ab^n ab^n a$ mit wachsendem $n$ . . . . .	45
3.6	Wachstum der durchschnittlichen Zustandszahl aus allen Boyer-Moore-Automaten für ein Suchwort $w$ bestimmter Länge. . . . .	49
5.1	Parametereinstellungen des Genetischen Algorithmus für die Bestenmutation. . .	82
6.1	Parametereinstellungen des GA für die Bestenmutation. . . . .	117
6.2	Gefundene Suchwörter mit großen BMA für Alphabet $\Sigma = \{a, b\}$ . . . . .	119
6.3	Gefundene Suchwörter mit großen BMA für Alphabet $\Sigma = \{a, b, c\}$ . . . . .	119



# Abbildungsverzeichnis

---

1.1	Prinzipieller Ablauf der Suche eines Wortes $w$ im Text $t$ . . . . .	6
2.1	Die Tabelle $shift_1$ speichert $s$ , den Abstand des Zeichens $a$ vom rechten Rand. . . . .	11
2.2	Die Tabelle $shift_2$ speichert $s + m - j$ für jede Stelle in $w$ . . . . .	11
2.3	Berechnung von $liste'$ aus $liste$ bei angenommenen gelesenen Zeichen $a$ im Zustand $q$ . . . . .	20
2.4	Die Suffixe für ein gegebenes $w$ . . . . .	22
2.5	Einige Zustände, die die notwendige Bedingung für das Suchwort $w$ erfüllen. . . . .	22
2.6	BMA mit Startzustand. . . . .	23
2.7	BMA nach hinzufügen von 100. . . . .	23
2.8	BMA nach dem 000 komplett verarbeitet wurde. . . . .	24
2.9	BMA nach dem Bearbeiten von $a$ im Zustand 001. . . . .	25
2.10	BMA nach Verarbeitung von 001. . . . .	25
2.11	Der BMA nach Verarbeitung von 011. . . . .	26
2.12	Der BMA nach Verarbeitung von 110. . . . .	26
2.13	Der BMA nach Verarbeitung von 100. . . . .	27
2.14	Der BMA nachdem die Konstruktion abgeschlossen wurde. . . . .	27
2.15	Der Beispiel-BMA . . . . .	29
2.16	Größe der Boyer-Moore-Automaten für Suchwörter, die durch sukzessives Anhängen an einer beliebigen Stelle im Wort erzeugt wurden. . . . .	32
3.1	Vergleich zwischen den durch vollständige Suche gefundenen größten Boyer-Moore-Automaten und den Funktionswerten der Näherungsfunktionen . . . . .	43
3.2	Häufigkeiten von Boyer-Moore-Automaten einer bestimmten Größe für Suchwörter der Länge 30. . . . .	46
3.3	Häufigkeiten von Boyer-Moore-Automaten einer bestimmten Größe für Suchwörter der Länge 20. . . . .	47
3.4	Maximale und durchschnittliche Änderung der Größe der BMA's für Suchwörter verschiedener Längen. . . . .	48
3.5	Das Wachstumsverhalten der durchschnittlichen BMA-Größe für verschiedengroße Alphabete. . . . .	50
4.1	Unterdisziplinen der Evolutionären Algorithmen. . . . .	52
4.2	Der Evolutionszyklus (aus [Wei02b]). . . . .	58
4.3	Güteverteilung vor und nach probabilistischen und deterministischen Selektionen (nach [Wei02c]). . . . .	69
4.4	Eine Population, die von einem Superindividuum dominiert wird. . . . .	70
4.5	Eine Population, deren Individuen alle eine sehr große Fitness haben. . . . .	70

---

4.6	Durch die Verschiebung der Koordinatenachse werden die Selektionswahrscheinlichkeiten der Individuen wieder unterschiedlicher. . . . .	70
4.7	In dieser Abbildung werden die sortierten Individuen einer Population in Verhältnis zu der Auswahlwahrscheinlichkeit der rangbasierten Selektion gesetzt. Für die Auswahlwahrscheinlichkeit kann eine lineare, oder z. B. eine quadratische Funktion benutzt werden. . . . .	71
4.8	Bildliche Veranschaulichung des 1-Punkt-Crossovers. Der erste Abschnitt wird beim ersten Nachkommen vom ersten Elternindividuum übernommen, der zweite vom Zweiten. Für den zweiten Nachkommen wird das jeweils andere Elternindividuum gewählt. . . . .	71
4.9	Bildliche Veranschaulichung des Diagonal-Crossovers. Für das $i$ . Kindindividuum wird für den ersten Abschnitt das Teilstück vom $i$ . Elternindividuum übernommen. Für den zweiten Abschnitt das Teilstück vom $i + 1 \bmod m$ ten Elternindividuum. . . . .	72
4.10	Nach der Elternselektion und der Rekombination wird die Mutation durchgeführt. Anschließend wird die Umweltselektion durchgeführt. Die Mutation arbeitet nicht mit Populationen, sondern mit einzelnen Individuen. . . . .	73
5.1	Der traditionelle Ablauf der Auswahl der Eltern. . . . .	78
5.2	Die Elternpopulation im Ablauf des Evolutionszyklus. . . . .	79
6.1	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit der Bestenmutation. . . . .	97
6.2	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit der bitFlip-Mutation. . . . .	98
6.3	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit der Wurzel-BitFlip-Mutation. . . . .	99
6.4	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit der Verschiebemutation. . . . .	100
6.5	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit der Suffixmutation. . . . .	101
6.6	Der Verlauf der Fitness der besten Individuen im Vergleich der Mutationen. . . . .	103
6.7	Der Verlauf der durchschnittlichen Fitness eines Individuums der $i$ . Generation im Vergleich der Mutationen. . . . .	104
6.8	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit dem 1-Punkt-Crossover. . . . .	105
6.9	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit dem Linearen-Crossover. . . . .	106
6.10	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit dem uniform-Crossover. . . . .	107
6.11	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit dem abschnittsbewertungs-Crossover. . . . .	108
6.12	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation mit dem Palindrom-Crossover. . . . .	109
6.13	Der Verlauf der Fitness der besten Individuen im Vergleich der Rekombinationen. . . . .	110

6.14	Der Verlauf der durchschnittlichen Fitness eines Individuums der $i$ . Generation im Vergleich der Rekombinationen. . . . .	111
6.15	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation bei der fitnessproportionalen Selektion. . . . .	112
6.16	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation bei der rangbasierten Selektion. . . . .	113
6.17	Der Verlauf der durchschnittlichen Fitness der besten Individuen und der durchschnittlichen Fitness eines Individuums der $i$ . Generation bei der Turnierselktion. . . . .	114
6.18	Der Verlauf der Fitness der besten Individuen im Vergleich der Selektionen. . . . .	115
6.19	Der Verlauf der durchschnittlichen Fitness eines Individuums der $i$ . Generation im Vergleich der Selektionen. . . . .	116
6.20	Die gefundenen besten Suchwörter in Abhängigkeit zur Mutationswahrscheinlichkeit und Rekombinationswahrscheinlichkeit. . . . .	118
6.21	Das Wachstum der besten Suchwörter für Alphabet $\Sigma = \{a, b\}$ . . . . .	120
6.22	Das Wachstum der besten Suchwörter für Alphabet $\Sigma = \{a, b, c\}$ . . . . .	121
A.1	Die von <code>AbstractIndividuum</code> abgeleiteten Klassen. . . . .	127
A.2	Die von <code>AbstractIndividuumFactory</code> abgeleiteten Klassen. . . . .	128
A.3	Die Klasse <code>Algorithmus</code> . . . . .	129
A.4	Die von <code>Strategie</code> abgeleiteten Klassen. . . . .	130
A.5	Die von <code>AbstrakteInitialisierung</code> abgeleiteten Klassen. . . . .	130
A.6	Die Klasse <code>Population</code> . . . . .	131
A.7	Die von <code>AbstrakteAbbruchbedingung</code> abgeleiteten Klassen. . . . .	132
A.8	Die von <code>AbstrakteSelektion</code> abgeleiteten Klassen. . . . .	132
A.9	Die von <code>AbstrakteRekombination</code> abgeleiteten Klassen. . . . .	133
A.10	Die von <code>AbstrakteMutation</code> abgeleiteten Klassen. . . . .	134
A.11	Die von <code>AbstraktKinderErzeuger</code> abgeleiteten Klassen. . . . .	134
A.12	Die von <code>AbstrakteBewertungsfunktion</code> abgeleiteten Klassen. . . . .	135



## **Erklärung**

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

---

(Ralf Hantschel)

