Axel Grossmann

Genetische Algorithmen und SAT-Probleme

Untersuchungen über den Einsatz evolutionärer Verfahren zur Lösung des Erfüllbarkeitsproblems für die Aussagenlogik

Technische Universität Dresden
Fakultät Informatik
Wissensverarbeitung

Großer Beleg

zum Thema

Genetische Algorithmen und SAT-Probleme

Untersuchungen über den Einsatz evolutionärer Verfahren
zur Lösung des Erfüllbarkeitsproblems für die Aussagenlogik

von
Axel Großmann
geb. am 29. Dezember 1971 in Halle

Technische Universität Dresden
Fakultät Informatik

Eingereicht am 18. September 1995

Betreuer: Prof. Dr. Steffen Hölldobler

Inhaltsverzeichnis

Abkürzungsverzeichnis

Beschreibung des Erfüllbarkeitsproblems

CNF	Konjunktive Normalform einer aussagenlogischen Formel
$3CNF$	Konjunktive Normalform einer Formel mit genau drei verschiedenen Literalen pro Klausel
SAT	Erfüllbarkeitsproblem für die Aussagenlogik
$3SAT$	Erfüllbarkeitsproblem für $3CNF$-Formeln
$\mathcal{NP}$	Komplexitätsklasse
F, G, H	Aussagenlogische Formeln
Φ	CNF-Formel
L	Literal einer CNF-Formel
N	Anzahl von Variablen in einer CNF-Formel
m	Anzahl von Klauseln in einer CNF-Formel
k	Anzahl von Literalen pro Klausel in einer CNF-Formel
A	Belegung für eine aussagenlogische Formel
$\top$	Wahrheitswert true oder 1
$\bot$	Wahrheitswert false oder 0
$\mathcal{V}(\Phi)$	Menge aller aussagenlogischen Variablen, die in der Formel Φ auftreten
$F[A]$	Anwendung der Belegung A auf die Formel Φ
$\mathcal{C}(\Phi)$	Menge aller Klauseln, die in der Formel Φ auftreten
$\mathcal{C}_u(\Phi, A)$	Menge aller Klauseln der Formel Φ, die durch die Belegung A nicht erfüllt werden
$\mathcal{C}(\Phi, V)$	Menge aller Klauseln, die in der Formel Φ vorkommen und in denen die Variable V als Literal auftritt
$\mathcal{C}_u(\Phi, A, V)$	Menge der in der Formel Φ vorkommenden Klauseln mit der Variablen V, die unter der Belegung A unerfüllt bleiben.

$A[V]$	Belegung einer aussagenlogischen Formel entsprechend A, in der jedoch der Wahrheitswert der Variablen V umgekehrt wird
$\mathrm{score}(\Phi, A, V)$	Variablen-Score der Variablen V bezüglich der Formel Φ und der Belegung A

Beschreibung der evolutionären Algorithmen

SGA	*Simple genetic algorithm*
EGA	*Elitist genetic algorithm*
CGA	*Crowded genetic algorithm*
MGA	*MetaGen algorithm*
$GenEA$	*Generisches Modell für evolutionäre Algorithmen*
RWS	*Roulette-wheel selection*
SUS	*Stochastic universal sampling*
LS	*Linear fitness scaling*
$1X$	*One-point crossover*
$2X$	*Two-point crossover*
UX	*Uniform crossover*
MX	*MetaGen crossover*
UM	*Uniform mutation*
CM	*Unsatisfied clause mutation*
λ	Populationsgröße
p_s	Erwartungswert für die Selektion eines Individuums
p_c	Crossover-Wahrscheinlichkeit
p_m	Mutationswahrscheinlichkeit
$\mathcal{H}(t)$	Population zum Zeitpunkt t
t	Generation, Zeitpunkt
t_{max}	Maximale Anzahl von Generationen
f_{obj}	Bewertungsfunktion
f_{fit}	Fitneßfunktion
GG	*Generation gap*
CF	*Crowding factor*
h	Größe des *Halting window*

r_{max}	Maximale Anzahl von Versuchen
P_e	*On-line performance*
P_e^*	*Off-line performance*
p_{model}^+	Erfolgsrate
t_{avg}^+	Mittlere Anzahl von Generationen bei erfolgreicher Suche

Kapitel 1

Einleitung

Bei der Betrachtung des seit etwa vier Milliarden Jahren andauernden Evolutionsprozesses auf der Erde stellt man fest, daß viele der hervorgebrachten Formen gut, vielleicht sogar optimal an ihre Umwelt angepaßt sind. Lebewesen erscheinen oftmals als vollendete Problemlöser. In der Vielzahl der Aufgaben, die sie bewältigen, übertreffen sie die besten Computerprogramme bei weitem. Angesichts dieser Tatsache erscheint es durchaus sinnvoll, die algorithmischen und mathematischen Prinzipien der Evolutionsprozesse und ihre Anwendbarkeit zu untersuchen.

In den letzten Jahren haben genetische Algorithmen große Aufmerksamkeit auf sich gezogen, weil sie ein neuartiges Verfahren der stochastischen Optimierung darstellen, das eine Effizienzsteigerung verspricht [12,29]. Bei dieser Klasse von Algorithmen wird im Computer ein Evolutionsprozeß simuliert, in dessen Verlauf die Parameter bestimmt werden, die ein Minimum oder Maximum einer Funktion produzieren.

In dieser Arbeit soll untersucht werden, ob genetische Algorithmen zur Bestimmung der Modelle erfüllbarer aussagenlogischer Formeln einsetzbar sind. Um diese Frage zu beantworten, sind genetische Algorithmen für diese Problemklasse zu entwickeln und hinsichtlich ihrer Effizienz und Erfolgsrate zu beurteilen.

Die Lösung des Erfüllbarkeitsproblems für aussagenlogische Formeln ist von großem praktischen Interesse. Insbesondere ist man an Verfahren interessiert, die auch Formeln mit mehreren hundert Variablen bewältigen können. Aus diesem Grund gab es eine Reihe von Aktivitäten zum Entwurf geeigneter Lösungsverfahren. Zu den bekanntesten Verfahren gehören *GSAT* [45] und *Simulated annealing* [1]. In der letzten Zeit wurden auch einige Verfahren auf der Basis von genetischen Algorithmen veröffentlicht [14,48].

Im ersten Teil der Arbeit wird das Erfüllbarkeitsproblem aussagenlogischer Formeln behandelt. Nach der Einführung grundlegender Begriffe und Beschreibungsmittel wird das Erfüllbarkeitsproblem in seinen verschiedenen Formen vorgestellt und werden deren Eigenschaften erläutert. Danach folgt eine Beschreibung der bekanntesten Lösungsverfahren. Es werden Möglichkeiten für einen Vergleich der Verfahren diskutiert.

Der zweite Teil der Arbeit beinhaltet eine allgemeinverständliche Einführung in die Problematik der genetischen Algorithmen. Ausgehend von dem biologischen Vorbild werden die grundlegenden Prinzipien erläutert. Am Ende des Kapitels werden noch einige theoretische Aspekte behandelt, die für das weitere Vorgehen wichtig sind.

Der eigentliche Entwurf und die anschließende Beurteilung der genetischen bzw. evolutionären Algorithmen geschieht in den folgenden beiden Teilen der Arbeit.

Im letzten Kapitel werden einige ausgewählte Aspekte der Algorithmen eingehender behandelt und die Erkenntnisse abschließend zusammengefaßt.

Diese Arbeit liefert in erster Linie experimentelle Ergebnisse. Obwohl einzelne Hypothesen aufgestellt werden und Verweise auf die theoretische Literatur angegeben sind, enthält die Arbeit keine abstrakte Analyse der gewonnenen Ergebnisse. Es wäre zweifellos höchst wünschenswert, wenn man die genauen Ursachen für das Verhalten der Algorithmen, eine genaue Angabe der für sie geeigneten Problemklassen oder genaue Angaben zur benötigten Laufzeit angeben kann. Unglücklicherweise sind solche Aussagen zu unvollständigen Algorithmen für $\mathcal{NP}$-vollständige Probleme außerordentlich selten und schwer zu erhalten. Neben der experimentellen muß auch theoretische Arbeit betrieben werden. Dabei können die experimentellen Ergebnisse genutzt werden, um zu entscheiden, ob eine formale Analyse bestimmter Problemlösungsmethoden sinnvoll ist.

In der Literatur zu evolutionären Verfahren ist leider keine einheitliche Darstellung für die entworfenen Algorithmen und Operatoren anzutreffen. Eine formale Darstellung hat sich nicht gegenüber Pseudokode oder verbalen Beschreibungen durchgesetzt. In dieser Arbeit werden, wenn möglich, formale Beschreibungen benutzt. Das ermöglicht eine einheitliche, präzise und kompakte Darstellung der verwendeten Verfahren.

Danksagungen

Mein Dank gilt zuerst Prof. Erwin Stoschek, der bei mir das Interesse an den genetischen Algorithmen vor zwei Jahren in seiner Vorlesung *Algorithmen-*

konstruktion geweckt hat. Um seine Worte zu gebrauchen, ich halte genetische Algorithmen für ein Paradebeispiel, wie Zusammenhänge und Analogien auch zwischen scheinbar weit auseinanderliegenden Problemkreisen erkannt sowie Schlüsselideen, -lösungen und -algorithmen verwertet und kombiniert werden können. Durch Prof. Steffen Hölldobler habe ich die *GSAT*-Verfahren kennengelernt, und er gab die Anregung zur Entstehung dieser Arbeit. Antje Strohmaier hat freundlicherweise die verwendeten Testformeln zur Verfügung gestellt. Für das sorgfältige Lesen eines großen Teils des Manuskriptes bin ich Markus Färber zu besonderem Dank verpflichtet. Wertvolle Diskussionen zum Thema habe ich mit Torsten Seiler führen können. Die kritischen Anmerkungen von Antje Strohmaier waren der Anlaß für eine abschließende Überarbeitung. Weiterhin danke ich Michael Hohmuth und Hans-Peter Störr für ihre Hilfe am Rechner.

Dresden, September 1995

Axel Großmann [1]

[1] E-Mail: axel@inf.tu-dresden.de

Kapitel 2

Das Erfüllbarkeitsproblem für die Aussagenlogik

2.1 Einige Begriffe der Aussagenlogik

In diesem Abschnitt sollen einige grundlegende Begriffe aus der Aussagenlogik eingeführt werden. Dabei werden Notation und Terminologie aus [18,39] verwendet.

$F = F(x_1, x_2, \ldots, x_N)$ sei eine *Formel* in der Aussagenlogik, bestehend aus den aussagenlogischen *Variablen* $x_1, x_2, \ldots, x_N$ und den logischen Operatoren $\vee$, $\wedge$, $\neg$. Σ bezeichnet die Menge aller aussagenlogischen Variablen.

Definition 1 (Syntax aussagenlogischer Formeln) *Formeln werden induktiv wie folgt definiert:*

1. *Alle Variablen aus Σ sind Formeln.*
2. *Für zwei Formeln F und G sind $(F \vee G)$ und $(F \wedge G)$ Formeln.*
3. *Für jede Formel F ist $\neg F$ eine Formel.*

Eine Formel der Form $\neg F$ heißt *Negation* von F, $(F \vee G)$ heißt *Disjunktion* von F und G, $(F \wedge G)$ heißt *Konjunktion* von F und G. Eine Formel F, die als Teil einer Formel G auftritt, heißt *Teilformel* von G.

Die Elemente $\{\perp, \top\}$ heißen *Wahrheitswerte*. Eine *Belegung* ist eine Funktion $A : \mathcal{V}(F) \rightarrow \{\perp, \top\}$, wobei $\mathcal{V}(F)$ die Menge der in F vorkommenden Variablen bezeichnet.

Definition 2 (Semantik aussagenlogischer Formeln) *Die Anwendung $F[A]$ einer Belegung A auf eine Formel F ist induktiv wie folgt definiert:*

1. *Wenn F eine Variable ist, dann ist $F[A]$ der Wahrheitswert, der F durch A zugewiesen wird.*
2. *Wenn F von der Form $\neg G$ ist, dann gilt*
   $$F[A] = \begin{cases} \top & \text{falls } G[A] = \bot, \\ \bot & \text{sonst.} \end{cases}$$
3. *Wenn F von der Form $(G \vee H)$ ist, dann gilt*
   $$F[A] = \begin{cases} \bot & \text{falls } G[A] = H[A] = \bot, \\ \top & \text{sonst.} \end{cases}$$
4. *Wenn F von der Form $(G \wedge H)$ ist, dann gilt*
   $$F[A] = \begin{cases} \top & \text{falls } G[A] = H[A] = \top, \\ \bot & \text{sonst.} \end{cases}$$

Definition 3 (Modell, Erfüllbarkeit) *Sei F eine Formel und A eine Belegung für F. A wird Modell für F genannt, falls gilt $F[A] = \top$.*

Eine Formel F heißt erfüllbar, falls F mindestens ein Modell besitzt, andernfalls heißt F unerfüllbar.

Ein *Literal* ist entweder eine Variable $V \in \Sigma$ oder die Negation $\neg V$ einer Variablen $V \in \Sigma$. Im ersten Fall spricht man von einem *positiven*, im zweiten von einem *negativen Literal*.

Eine *Klausel* ist eine endliche Disjunktion von Literalen. Eine Klausel heißt *trivial*, wenn sie eine Variable V und deren Komplement $\neg V$ enthält.

Definition 4 (konjunktive Normalform) *Eine aussagenlogische Funktion in konjunktiver Normalform, kurz CNF-Formel, ist eine endliche Konjunktion von Klauseln.*

Eine CNF-Formel Φ besitzt also die Form

$$\Phi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n_i} L_{ij} = \bigwedge_{i=1}^{m} \Phi_i,$$

wobei jedes L_{ij} ein Literal und jedes Φ_i eine Klausel bezeichnet.

Φ ist eine *k-CNF-Formel*, falls Φ in konjunktiver Normalform ist und jede Klausel Φ_i aus genau k verschiedenen Literalen besteht.

Im folgenden sei Φ eine CNF-Formel und A eine Belegung für Φ.

$\mathcal{C}(\Phi)$ bezeichnet die Menge aller Klauseln, die in Φ auftreten.

$\mathcal{C}_u(\Phi, A) = \{c \in \mathcal{C}(\Phi) \mid c[A] = \perp\}$ ist die Menge der in Φ vorkommenden Klauseln, die unter der Belegung A unerfüllt bleiben.

$\mathcal{C}(\Phi, V)$ bezeichnet die Menge aller Klauseln, die in Φ vorkommen und in denen die Variable V als Literal auftritt.

$\mathcal{C}_u(\Phi, A, V) = \{c \in \mathcal{C}(\Phi, V) \mid c[A] = \perp\}$ ist die Menge der in Φ vorkommenden Klauseln mit der Variablen V, die unter der Belegung A unerfüllt bleiben.

$A[V]$ bezeichnet die Belegung einer Formel entsprechend A, in der jedoch der Wahrheitswert der Variablen V umgekehrt wird. Das kann man präzise formulieren als

$$V[A[W]] = \begin{cases} \neg V[A] & \text{falls } W = V, \\ V[A[W]] & \text{falls } W \neq V. \end{cases}$$

Definition 5 (Variablen-Score) *Der Score einer Variablen V bezüglich einer CNF-Formel Φ und einer zugehörigen Belegung A ist die Differenz zwischen der Anzahl von Klauseln, die unter A unerfüllt sind, und der Anzahl von Klauseln, die unter $A[V]$ unerfüllt bleiben.*

Ein *positiver Variablen-Score* von V repräsentiert eine Abnahme, wohingegen ein *negativer Variablen-Score* einen Anstieg der Anzahl der unerfüllten Klauseln beim Umkehren des Wahrheitswertes von V repräsentiert.

2.2 SAT-Probleme

In diesem Abschnitt ist es nun möglich, das Erfüllbarkeitsproblem für die Aussagenlogik zu definieren und Angaben zur Berechenbarkeitskomplexität zu machen.

2.2.1 Die Definition des Erfüllbarkeitsproblems

Definition 6 (SAT-Problem) *Unter dem Erfüllbarkeitsproblem, oder kurz SAT-Problem (Boolean satisfiability problem), versteht man die Entscheidung, ob eine aussagenlogische Formel F ein Modell besitzt oder nicht.*

Das Erfüllbarkeitsproblem bezieht sich auf beliebige aussagenlogische Formeln. Schränkt man die Menge der zulässigen Formeln ein, so erhält man Spezialfälle des SAT-Problems. Ein Spezialfall sind die *k-SAT-Probleme*, die die Erfüllbarkeit von k-CNF-Formeln behandeln.

Definition 7 ($3SAT$-Problem) *Unter dem $3SAT$-Problem versteht man die Entscheidung, ob eine aussagenlogische Formel Φ in konjunktiver Normalform mit genau drei Literalen pro Klausel ein Modell besitzt oder nicht.*

2.2.2 Die Berechenbarkeitskomplexität des Erfüllbarkeitsproblems

Das Erfüllbarkeitsproblem für die Aussagenlogik besitzt eine fundamentale Bedeutung in der Informatik. Neben seinen praktischen Anwendungen bei der Verifikation von Programmen und Schaltungen stellt es ein Kernproblem in der Komplexitätstheorie dar.

Das SAT-Problem ist $\mathcal{NP}$-vollständig. Es war überhaupt die erste Berechnungsaufgabe, deren $\mathcal{NP}$-Vollständigkeit gezeigt werden konnte [9]. Diese theoretische Eigenschaft von SAT bildet die Grundlage für die Komplexitätsbeweise vieler anderer Probleme [34]. Nach dem Satz von Cook [9] sind alle Probleme aus $\mathcal{NP}$ auf das Erfüllbarkeitsproblem reduzierbar.

Satz 1 *Das Erfüllbarkeitsproblem ist $\mathcal{NP}$-vollständig.*

Beweis:
Da der Beweis des Satzes den Rahmen dieser Arbeit übersteigt, soll hier nur auf [34] Seite 237 ff. verwiesen werden. □

Obwohl das SAT-Problem $\mathcal{NP}$-vollständig ist, gibt es syntaktisch eingeschränkte Varianten des Problems, für die Erfüllbarkeit einer aussagenlogischen Formel in polynomialer Zeit bestimmt werden kann. Zu diesen Varianten gehören unter anderem $HORNSAT$ und $2SAT$. Bei $HORNSAT$ ist die aussagenlogische Formel eine Konjunktion von Horn-Klauseln. Wenn also in einer Anwendung die syntaktische Form der benutzen Formeln eingeschränkt werden kann, ist es möglich, das Erfüllbarkeitsproblem effizient zu lösen.

Die Erkenntnis, daß eine erfüllende Belegung A für eine aussagenlogische Formel F existiert, liefert noch nicht zwangsläufig solch ein A. Es stellt sich die Frage, welcher Zusammenhang zwischen diesen beiden Problemstellungen besteht.

Satz 2 *Das konstruktive Problem, eine erfüllende Belegung für eine aussagenlogische Formel zu finden, ist mindestens so schwierig wie das Entscheidungsproblem, ob eine solche Belegung überhaupt existiert.*

Beweis:
Der Beweis beruht auf der Eigenschaft der Selbstreduzierbarkeit. Siehe hierzu [34] Seite 239 f. □

Als nächstes soll gezeigt werden, daß das Erfüllbarkeitsproblem bereits für eine Teilmenge von einfach strukturierten aussagenlogischen Formeln $\mathcal{NP}$-vollständig ist, nämlich für CNF-Formeln. Das Ergebnis kann man dahingehend interpretieren, daß die Schwierigkeit des Erfüllbarkeitsproblems nicht in der syntaktischen Struktur der Formeln liegt, sondern allein in der Frage, welcher Wahrheitswert den einzelnen Variablen zugeordnet werden soll. Die Anzahl aller möglichen Belegungen (2^N) wächst exponentiell in der Zahl der Variablen N. Daher benötigt ein Lösungsverfahren, das alle Möglichkeiten der Reihe nach prüft, bis eine erfüllende Belegung gefunden wird, exponentielle Zeit. Dies gilt im *worst case* für erfüllbare Formeln und generell für unerfüllbare Formeln. Bislang sind keine Verfahren mit subexponentieller Laufzeit bekannt, die die Erfüllbarkeit von Formeln, selbst unter der Einschränkung der $3CNF$-Form, *immer korrekt* entscheiden [34].

Satz 3 *Für jedes $k \geq 3$ ist das k-SAT-Problem $\mathcal{NP}$-vollständig.*

Beweis:
Es genügt zu zeigen, daß man das SAT-Problem auf k-SAT reduzieren kann. Dazu überführt man eine beliebige Formel F in eine *äquivalente* Formel Φ in CNF-Form. In Φ werden in der Regel zusätzliche, sogenannte *interne*, Variablen benötigt. Ohne dies wäre es nicht immer möglich, die Länge der CNF-Formel polynomial beschränkt zu halten. *Äquivalent* bedeutet hier, daß Φ genau dann erfüllbar ist, wenn dies für F gilt.

Mit Hilfe der Regeln von De Morgan können alle Negationen direkt vor die Variablen gebracht werden. Um nun eine Formel, in der $\vee$ und $\wedge$ beliebig geschachtelt sein können, in eine Konjunktion von Klauseln zu überführen benutzt man die folgende Beobachtung: Ist

$$F = \Phi_1 \vee \Phi_2$$

und sind die Teilformeln Φ_i bereits in CNF, so läßt sich F mit Hilfe einer neuen internen Variablen y, die in Φ_i nicht vorkommt, durch eine äquivalente Formel

$$F' = (\Phi_1 \lor y) \land (\Phi_2 \lor \neg y)$$

ersetzen. Der Beweis der Erfüllbarkeitsäquivalenz von F und F' läßt sich leicht durch eine Wahrheitswerttabelle erbringen.

Für

$$\Phi_i = \Phi_{i1} \land \Phi_{i2} \land \ldots \land \Phi_{in}$$

mit den Klauseln Φ_{ij} kann man dann $(\Phi_i \lor y)$ umformen in

$$\Phi_i = (\Phi_{i1} \lor y) \land (\Phi_{i2} \lor y) \land \ldots \land (\Phi_{in} \lor y),$$

so daß sich wieder eine Konjunktion von Klauseln ergibt. Zwei Teilformeln in CNF, die durch $\land$ verknüpft sind, lassen sich ebenfalls in diese Form überführen. Dabei braucht man nur die Klammern zu streichen, die die Formeln begrenzen.

Führt man dieses Verfahren rekursiv durch, so wird für jedes $\lor$ eine neue Variable eingeführt. Hat die ursprüngliche Formel die Länge l, so wächst die Länge höchstens um den Faktor l, und die Transformation kann in $O(l^2)$ Schritten ausgeführt werden.

Schließlich muß noch gezeigt werden, daß man eine Formel in beliebiger konjunktiver Normalform in eine äquivalente Formel in k-CNF-Form transformieren kann. Dazu wandelt man eine Klausel $\Phi_\nu = (x_1 \land x_2 \land \ldots \land x_n)$ durch Verwendung zusätzlicher interner Variablen in eine oder mehrere Klauseln um, deren Konjunktion äquivalent zu Φ_ν ist.

Im weiteren soll der Fall $k = 3$ betrachtet werden. Die y_i seien neue interne Variablen. In Abhängigkeit von der Länge n ersetzt man Φ_ν wie folgt:

$$\begin{aligned}
n = 1: \quad & (x_1 \lor y_1 \lor y_2) \land (x_1 \lor \neg y_1 \lor y_2) \land (x_1 \lor y_1 \lor \neg y_2) \land \\
& (x_1 \lor \neg y_1 \lor \neg y_2), \\
n = 2: \quad & (x_1 \lor x_2 \lor y_1) \land (x_1 \lor \neg x_2 \lor \neg y_1), \\
n > 3: \quad & (x_1 \lor x_2 \lor y_1) \land (\neg y_1 \lor x_3 \lor y_2) \land (\neg y_2 \lor x_4 \lor y_3) \land \ldots \land \\
& (\neg y_{n-3} \lor x_{n-1} \lor x_n).
\end{aligned}$$

Die Äquivalenz ergibt sich aus folgenden Überlegungen. Im Fall $n = 1$ können die vier Klauseln nur dann gleichzeitig gelten, wenn x_1 den Wert $\top$ hat, denn unabhängig davon, welche Werte man für die neuen internen Variablen y_1 und y_2 wählt, in einer der Klauseln liefern beide Variablen oder ihre Negationen den Wert $\perp$. Ähnliches gilt für $n = 2$.

Wird die ursprüngliche Formel im Fall $n > 3$ durch die Variable x_i erfüllt, so wähle man $y_j = \top$ für $j \leq i-2$ und $y_j = \bot$ für $j > i-2$. Dann werden in der Folge der neuen Klauseln die ersten $i-2$ Klauseln durch die y_j erfüllt und die letzten $n-(i-2)-3$ durch $\neg y_j$. Die $(i-1)$-te Klausel, die x_i enthält, wird durch x_i erfüllt. Liefert andererseits jede dieser Klauseln den Wert $\top$, so kann man drei Fälle unterscheiden. Haben alle y_j den Wert $\top$, so muß $x_{n-1} \vee x_n = \top$ gelten und damit wäre auch Φ_ν erfüllt. Analoges gilt für $x_1 \vee x_2$, falls alle y_j den Wert $\bot$ haben. Es verbleibt der Fall, daß bis zu einem i, $1 \leq i < k-3$, $y_1 = y_2 = \cdots = y_i = \top$ und $y_{i+1} = \bot$ gewählt wurden. Dann kann die $(i+1)$-te Klausel nur durch $x_{i+2} = \top$ erfüllt werden, das heißt, Φ_ν gilt. $\square$

Nachfolgend soll ein Beispiel betrachtet werden. Es sei

$$F = \neg(\neg(x_1 \vee x_2 \vee \neg x_3) \vee (x_4 \vee (x_3 \wedge \neg x_5)))$$

Nach dem Verschieben der Negationen erhält man zunächst die äquivalente Formel

$$F' = (x_1 \vee x_2 \vee \neg x_3) \vee (\neg x_4 \wedge (\neg x_3 \vee x_5))$$

Diese transformiert man dann mit Hilfe neuer interner Variablen y_1, y_2, y_3 in eine äquivalente $3CNF$-Formel

$$\begin{aligned} & (x_1 \vee x_2 \vee \neg x_3 \vee y_1) \wedge (\neg x_4 \vee \neg y_1) \wedge (\neg x_3 \vee x_5 \vee \neg y_1) \\ \equiv\ & (x_1 \vee x_2 \vee y_2) \wedge (\neg y_2 \vee \neg x_3 \vee y_1) \wedge (\neg x_4 \vee \neg y_1 \vee y_3) \wedge \\ & (\neg x_4 \vee \neg y_1 \vee \neg y_3) \wedge (\neg x_3 \vee x_5 \vee \neg y_1) = \Phi. \end{aligned}$$

2.3 Klassen von SAT-Algorithmen

Die Eigenschaft der $\mathcal{NP}$-Vollständigkeit wird in der künstlichen Intelligenz dazu benutzt, Aufgaben, die mit realistischem Einsatz von Ressourcen lösbar sind, von denen zu trennen, für die das nicht möglich ist. Ist ein Problem als $\mathcal{NP}$-vollständig klassifiziert worden, stellt sich die Frage nach dem weiteren Vorgehen. Dabei hat man in der Regel die Wahl zwischen zwei Möglichkeiten. Entweder strebt man *Optimalität* an, was mit dem Risiko eines sehr großen oder möglicherweise nichtpraktikablen Rechenzeitbedarfs verbunden ist, oder man sucht schnell erreichbare Lösungen mit dem Risiko der *Suboptimalität*.

SAT-**Algorithmen**	
Systematische Suchverfahren	**Nichtsystematische Suchverfahren**
Methoden:	Methoden:
Durchsuchen des gesamten Lösungsraumes, Enumeration u. a.	Lokale Suchalgorithmen, Hill-climbing-Verfahren, Probabilistische Suche u. a.
Beispiele:	Beispiele:
Davis-Putnam-Verfahren, Backtrack-Verfahren, 0/1-lineare Programmierung, Resolutionsverfahren	Lokale Suchverfahren, $GSAT$-Verfahren, Simulated annealing, Evolutionäre Algorithmen

Abbildung 2.1: Klassen von SAT-Algorithmen

Im Falle des Findens von Modellen für aussagenlogische Formeln ist man jedoch nur an den *optimalen* Lösungen, den erfüllenden Belegungen, interessiert.

Obwohl das SAT-Problem $\mathcal{NP}$-vollständig ist, wurden eine Reihe von Methoden zur Befriedigung der praktischen Bedürfnisse entwickelt. Diese Methoden kann man grob in zwei Klassen einteilen, *systematische* und *nichtsystematische Methoden.* Abbildung 2.1 gibt einen Überblick über gegenwärtig eingesetzte Verfahren. Unter einem *SAT-Algorithmus* soll im weiteren ein Lösungsverfahren für das Erfüllbarkeitsproblem verstanden werden.

Systematische Methoden sind *korrekt* und *vollständig.* Das heißt, sie haben den Vorteil, daß sie theoretisch in der Lage sind, alle Lösungen eines SAT-Problems zu finden, oder zu beweisen, daß keine Lösung existiert. Dieser Vorteil muß aber mit einem enormen Aufwand an Rechenzeit bezahlt werden. Somit ist die Anwendbarkeit von vollständigen Verfahren durch die Problemgröße beschränkt. Die Problemgröße wird bei SAT durch die Anzahl N der in der Formel enthaltenen Variablen repräsentiert. Das beste heute für das SAT-Problem bekannte systematische Suchverfahren kann eine Formel mit bis zu 400 Variablen auf einer Workstation in einer Stunde bearbeiten [41].

Auf der anderen Seite besitzen *nichtsystematische Methoden* effiziente Heuristiken, um den Suchraum so weit wie möglich einzuschränken. Tatsächlich wurde gezeigt, daß solche Verfahren in der Lage sind, größere SAT-Probleme zu lösen, als das mit systematischen Methoden möglich ist [18, 40–45]. Jedoch besitzen nichtsystematische Verfahren, wie zum Beispiel $GSAT$, einen entscheidenden Nachteil: Besitzt ein Problem keine Lösung, so ist es nicht möglich, das mit Gewißheit festzustellen. Deshalb sind nichtsystematische Verfahren *korrekt* und *nichtvollständig*.

Es ist nicht in jedem Fall möglich, einen SAT-Algorithmus in eine der beiden Klassen einzuordnen. Eine Reihe von Verfahren versuchen die positiven Eigenschaften aus beiden Klassen zu kombinieren, zum Beispiel *Dynamic backtracking* [11].

Das ursprüngliche SAT-Problem ist als Entscheidungsproblem formuliert. Es gibt jedoch eine Reihe verwandter Suchprobleme:

Modell-Suche. Finde eine erfüllende Belegung für die Variablen einer gegebenen Formel oder gib aus, daß eine solche nicht existiert.

Modell-Zählung. Bestimme die Anzahl von unterschiedlichen Modellen für eine gegebene Formel.

Theorem-Beweis. Finde einen formalen Beweis von der *Negation* einer gegebenen Formel oder gib aus, daß es keinen Beweis gibt. Existiert ein Modell für die negierte Formel, dann ist die originale Formel unerfüllbar.

Im nächsten Abschnitt sollen zwei bekannte SAT-Algorithmen vorgestellt werden. Das ist das problemspezifische $GSAT$-Verfahren und Simulated annealing als ein universell einsetzbares Verfahren zur kombinatorischen Optimierung.

2.4 Lokale Suchverfahren

2.4.1 Der GSAT-Algorithmus

Selman et al. [45] entwarfen ein nichtsystematisches lokales Suchverfahren, $GSAT$ genannt (*Greedy satisfiabiblity tester*), zur Lösung des

Erfüllbarkeitsproblems für aussagenlogische Formeln. In Experimenten konnte gezeigt werden, daß mit *GSAT* erheblich größere zufällig generierte Formeln behandelt werden können, als das mit traditionellen Methoden möglich ist.

```
GSAT(Φ)
for i ← 1 to MaxTries
do A ← zufällige Belegung für alle Variablen in 𝒱(Φ)
    for j ← 1 to MaxFlips
    do if Φ[A] = ⊤
          then return A
        ν ← Index der Variablen mit dem
                größten Score bezüglich Φ und A
        A ← A[V_ν]
return „Kein Modell gefunden."
```

Algorithmus 2.1: Der *GSAT*-Algorithmus

GSAT führt eine lokale Suche nach einer erfüllenden Belegung durch. Das Verfahren startet mit einer zufällig generierten Belegung. Danach wird iterativ immer der Wahrheitswert der Variablen umgekehrt, mit der jeweils der größte Anstieg in der Zahl der erfüllten Klauseln erreicht werden kann. Das heißt, es wird immer die Variable mit dem größten Variablen-Score ausgewählt und invertiert. Diese Schritte werden solange wiederholt, bis eine erfüllende Belegung gefunden wurde oder eine zuvor festgelegte maximale Anzahl von Schritten *MaxFlips* erreicht wird. Dieser Prozeß wird, falls nötig, maximal *MaxTrials* Male wiederholt.

Der Suchprozeß bei *GSAT* läßt sich mit dem Modell der kombinatorischen Optimierung und einer Energiefunktion veranschaulichen. *GSAT* sucht ein globales Minimum in der Anzahl der unerfüllten Klauseln als eine Funktion über den Wahrheitswerten. Dabei fällt *GSAT* jedoch in das nächste lokale Minimum und hat praktisch keine Möglichkeit dort hinauszugelangen, außer die Suche an einem zufällig gewählten Punkt in der Energielandschaft erneut zu beginnen.

2.4.2 GSAT mit random walk

Der Nachteil von *GSAT*, besonders anfällig für lokale Minima zu sein, kann durch die Einführung der sogenannten *random walk*-Strategie erheblich gemildert werden.

```
GWSAT(Φ)
for i ← 1 to MaxTries
do A ← zufällige Belegung für alle Variablen in 𝒱(Φ)
    for j ← 1 to MaxFlips
    do if Φ[A] = ⊤
          then return A
        r ← Zufallszahl aus [0, 1]
        if r ≤ p
          then ν ← Index der zufälligen gewählten
                   Variablen aus 𝒞_u(Φ, A)
          else  ν ← Index der Variablen mit dem
                   größten Score bezüglich Φ und A
        A ← A[V_ν]
return „Kein Modell gefunden.“
```

Algorithmus 2.2: Der *GWSAT*-Algorithmus als *GSAT* mit *random walk*

Mit einer von außen festgelegten Wahrscheinlichkeit p darf auch eine Variable mit einem geringeren als dem maximalen Score ausgewählt und invertiert werden. Die *random walk*-Strategie läßt sich wie folgt beschreiben:

- Wähle mit der Wahrscheinlichkeit p eine Variable in irgendeiner unerfüllten Klausel aus und kehre ihren Wahrheitswert um.

- Verfahre mit der Wahrscheinlichkeit $(1-p)$ nach dem ursprünglichen *GSAT*-Verfahren, das heißt, führe die bestmögliche lokale Bewegung in der Energielandschaft aus.

Der *GSAT*-Algorithmus mit *random walk* wird im weiteren *GWSAT* genannt. Weitere Modifikationen von *GSAT* sowie die Anwendung der Verfahren zur Lösung anderer kombinatorischer Probleme werden in [18, 40–45] vorgestellt und ausführlich diskutiert.

2.4.3 Simulated annealing

In der Optimierungspraxis wird seit langem eine Methode verwendet, die unter dem Namen *Simulated annealing* (simuliertes Ausglühen) bekannt geworden ist. Ausglühen ist ein metallurgischer Prozeß, bei dem ein Material stark erhitzt und dann langsam abgekühlt wird. Auf diese Weise wird die kristalline Struktur des Materials zu einem Energieminimum geführt.

Der Grund dafür, daß Simulated annealing lokale Minima umgeht, besteht darin, daß die Teilchendynamik eine temperaturabhängige Zufallskomponente, das sogenannte *thermische Rauschen*, enthält. Die Teilchen verlieren im Laufe der Abkühlung nicht nur Energie, wie es der Energiefunktion des Materials entspricht, sondern können gelegentlich auch einen höheren Energiezustand erreichen. Auf diese Weise können nicht allzu tief liegende lokale Minima übergangen werden.

Simulated annealing konnte bei einer Reihe von kombinatorischen Optimierungsproblemen erfolgreich angewandt werden [1]. Es wurden auch Experimente zur Anwendung auf SAT-Probleme unternommen [18, 40–44]. Im folgenden soll kurz ein SAT-Algorithmus auf der Basis von Simulated annealing beschrieben werden.

Den Ausgangspunkt bildet wieder eine zufällig erzeugte Belegung für die gegebene aussagenlogische Formel. Danach wird wiederholt eine Variable zufällig ausgewählt und ihr Variablen-Score $score$ berechnet. Ist ihr Score positiv (eine Abwärtsbewegung in der Energielandschaft), so wird ihr Wahrheitswert umgekehrt. Andernfalls wird der Wert der Variablen nur mit der Wahrscheinlichkeit $\mathrm{e}^{-score/T}$ umgekehrt, wobei T ein formaler Parameter ist, die sogenannte *Temperatur*. Die Temperatur kann entweder konstant gehalten oder langsam von einem hohen Wert gegen Null reduziert werden. Dabei wird die Abnahme der Temperatur vom sogenannten *cooling schedule* festgelegt. Häufig benutzt man geometrische *cooling schedules*, wobei die Temperatur wiederholt durch Multiplikation mit einem konstanten Faktor kleiner als Eins stetig verringert wird.

Simulated annealing besitzt die Eigenschaft der theoretischen *Konvergenz*, das heißt, es ist *vollständig*. Diese theoretische Eigenschaft besagt, daß bei beliebig kleinen Temperaturschritten ein globales Energieminimum garantiert gefunden wird. Das ist aber nur sehr bedingt von praktischem Interesse, da immer nur ein endlicher *cooling schedule* eingesetzt werden kann. Deshalb wird eine erfüllende Belegung nicht in jedem Fall gefunden, und man ist wieder darauf angewiesen, den Algorithmus mehrfach zu starten.

2.5 Die Schwierigkeit von SAT-Problemen

Wie in den vorangegangenen Abschnitten dargestellt wurde, existieren eine Reihe von Verfahren zur Lösung des Erfüllbarkeitsproblems für aussagenlogische Formeln. Es stellt sich nun die Frage, wie die einzelnen Verfahren miteinander verglichen werden können. Mögliche Vergleichskriterien wären sicherlich die *Effizienz* und die *Erfolgsrate* bezüglich des Findens der Modelle.

Ein Vergleich bezüglich der Laufzeiteffizienz ist nur sehr schwer möglich, da dort Implementierungsgesichtspunkte eine entscheidende Rolle spielen. Eine mögliche Alternative dafür ist die Abschätzung der Berechnungskomplexität.

Ein Vergleich der Verfahren hinsichtlich der Erfolgsrate bezüglich des Findens der Modelle ist möglich, wenn die gleichen Testdaten benutzt werden. Daraufhin ergibt sich die Frage, welche Testdaten man benutzen soll. Zum Vergleich von SAT-Algorithmen wurden bisher entweder zufällig generierte aussagenlogische Formeln oder kombinatorische Probleme genutzt, die in SAT-Probleme transformiert wurden.

2.5.1 Zufällig erzeugte Formeln

Zufällig erzeugte CNF-Formeln werden oft zur Evaluierung von SAT-Algorithmen benutzt, weil sie einfach zu erzeugen sind und keine verborgene innere Struktur besitzen, wie das häufig bei von Hand erzeugten Testformeln der Fall ist. Jedoch haben Experimente ergeben, daß die Qualität der erzeugten Formeln entscheidend von bestimmten Faktoren abhängt. Das sind unter anderem der benutzte Zufallsgenerator und das Verhältnis von Klausel- und Variablenanzahl. Bei ungünstiger Wahl der Parameter können die zufällig erzeugten Formeln so entarten, daß sie trivial zu lösen sind. In einem solchen Fall erreicht auch ein SAT-Algorithmus respektable Erfolgsraten, der nur zufällige Belegungen erzeugt. Daraufhin wurden zahlreiche empirische Untersuchungen gemacht, welche Arten von Formeln für SAT-Algorithmen am schwersten zu bearbeiten sind [25, 30].

In [25] wird festgestellt, daß ein Schwellenwert bei der Erfüllbarkeit von zufällig erzeugten $3CNF$-Formeln existiert. Diese Schwelle ist abhängig vom Verhältnis der Klauselanzahl m zur Variablenanzahl N und liegt bei $m/N \approx 4.3$. Bei dieser Untersuchung wurden $3CNF$-Formeln zufällig erzeugt. Danach hat man durch ein vollständiges Suchverfahren bestimmt, ob die generierten Formeln ein Modell besitzen oder nicht. Im Ergebnis war die Wahrscheinlichkeit, daß eine zufällig generierte $3CNF$-Formel der Art

$m/N = 4.3$ erfüllbar ist, am geringsten. Diese Ergebnisse stimmen mit [30] überein.

Ist die Erfüllbarkeitswahrscheinlichkeit für diese Art von Formeln am geringsten, so besitzen sie statistisch gesehen auch die geringste Anzahl von Modellen. Daher ist es naheliegend, zufällig generierte $3CNF$-Formeln mit $4.3 \cdot N$ Klauseln zur Evaluierung von SAT-Algorithmen zu benutzen. Formeln mit dieser Eigenschaft werden auch als *hard random*-Formeln bezeichnet.

Unabhängig vom Verhältnis m/N unterscheidet man noch zwischen sogenannten *forced*-Formeln und *unforced*-Formeln. *Forced*-Formeln werden so erzeugt, daß sie erfüllbar sind, wohingegen *unforced*-Formeln völlig zufällig generiert und die unerfüllbaren dann durch ein vollständiges Suchverfahren herausgefiltert werden. In dieser Arbeit wurden CNF-Formeln beider Arten benutzt, die mit dem Formelgenerator von B. Selman und H. Kautz erzeugt wurden.

Es sei N die Anzahl der Variablen, k die Anzahl der Literale pro Klausel und m die Anzahl von Klauseln in einer zufällig generierten CNF-Formel. Dann existieren $\binom{N}{k} \cdot 2^k$ nichtriviale Klauseln der Größe k.

Man erhält eine *unforced*-Formel, durch zufälliges Generieren von m Klauseln mit jeweils k Literalen. Die k Literale werden gebildet, indem man gleichverteilt k Variablen aus den N möglichen auswählt. Jede dieser Variablen wird dann nach mit einer Wahrscheinlichkeit von 0.5 negiert. Dagegen bildet man eine *forced*-Formel, indem man zuerst eine Belegung zufällig erzeugt. Danach werden Klauseln ebenfalls zufällig erzeugt, aber nur solche Klauseln für die Formel verwendet, die von der Belegung erfüllt werden. Das geschieht so lange, bis die benötigte Anzahl von Klauseln vorliegt. Durch dieses Vorgehen wird erzwungen, daß mindestens ein Modell für die *forced*-Formel existiert.

2.5.2 Kombinatorische Optimierungsprobleme

Zufällig generierte CNF-Formeln besitzen keine verborgene innere Struktur, wie es bei den meisten praktischen Problemen der Fall ist. Es stellt sich die Frage, welchen Einfluß die innere Struktur auf die Schwierigkeit einer Formel für die SAT-Algorithmen hat. Tatsächlich zeigen Experimente, daß probabilistische Suchverfahren, wie $GSAT$ oder Simulated annealing, Schwierigkeiten bei der Bearbeitung von manchen Klassen strukturierter Formeln haben. Das ist der Fall bei aussagenlogischen Darstellungen von *Blocks-world*-Planungsproblemen oder von *Graph coloring*-Problemen mit einem hohen Grad an Asymmetrie [40–44]. Deshalb ist es sehr wohl von Bedeutung, neben zufällig generierten Formeln auch SAT-Instanzen von praktischen Problemen zur Evaluierung der Algorithmen zu benutzen.

Über die Verwendung von *forced*-Formeln als Experimentiergrundlage für SAT-Verfahren gab es in der Vergangenheit eine recht kontroverse Diskussion. Vielfach ist man der Meinung, daß *forced*-Formeln keine objektive Basis zur Bewertung von Algorithmen sind, weil sie erstens zu einfach sind und zweitens durch die zufällige Erzeugung keine innere Struktur besitzen.

Kapitel 3

Genetische Algorithmen

3.1 Das natürliche Vorbild

Genetische Algorithmen sind adaptive Verfahren, die zur Lösung von Such- und Optimierungsproblemen eingesetzt werden können. Sie beruhen auf den Prinzipien der biologischen Evolution.

Bei der Betrachtung des seit etwa vier Milliarden Jahren andauernden Evolutionsprozesses auf der Erde stellt man fest, daß viele der hervorgebrachten Formen gut, vielleicht sogar optimal an ihre Umwelt angepaßt sind. Es gibt heute mit der Bionik eine Wissenschaft, die Lösungen aus der Natur für die Technik zu nutzen versucht und damit sehr erfolgreich ist [37]. Warum sollte also eine Nachahmung von Prinzipien der biologischen Evolution nicht auch zu guten Optimierungsverfahren führen?

Die Grundlagen der *genetischen Algorithmen* wurden Anfang der sechziger Jahre von John H. Holland [16] entwickelt. Zur gleichen Zeit begründeten Ingo Rechenberg und Hans-Paul Schwefel die *Evolutionsstrategien* und Lawrence J. Fogel die *evolutionäre Programmierung*. Diese unabhängig voneinander entstandenen Verfahren, die heute unter dem Sammelbegriff *evolutionäre Algorithmen* zusammengefaßt werden, haben die gemeinsame Eigenschaft, daß sie bewußt Prinzipien der biologischen Evolution nachahmen, um sie im Sinne von Optimierungsregeln einzusetzen.

In der Natur konkurrieren die Individuen einer Population miteinander um Ressourcen, wie Nahrung, Wasser und Lebensraum. Dieser Wettbewerb äußert sich auch bei der Fortpflanzung der Vertreter einer Art. Die Individuen, welche am erfolgreichsten im Überlebenskampf sind und die meisten Paarungspartner finden, haben relativ betrachtet die meisten Nachkommen.

An die Umgebungsbedingungen schlecht angepaßte Individuen werden nur wenige oder überhaupt keine Nachkommen hervorbringen. Folglich werden die Gene von Individuen hoher Qualität in den nachfolgenden Generationen vermehrt auftreten. Dieses Prinzip wird *survival of the fittest* genannt. Unter der *Fitneß* versteht man die Angepaßtheit eines Individuums an seine Umgebungsbedingungen. Durch die Kombination der guten Eigenschaften von den Vorfahren kann manchmal ein neues Individuum entstehen, das eine größere Fitneß als seine Eltern besitzt. Die Vertreter einer Art passen sich also immer besser an ihre Umwelt an.

Genetische Algorithmen benutzen eine direkte Analogie zum natürlichen Vorbild. Sie arbeiten auf einer *Population* von *Individuen*, wobei jedes Individuum eine mögliche Lösung des gegebenen Problems repräsentiert. Jedem Individuum wird ein Fitneßwert zugeordnet, der beschreibt, wie gut die Lösung für das Problem ist. Den Individuen mit einer großen Fitneß wird dann die Möglichkeit gegeben, sich zu *reproduzieren*. Das geschieht durch die Rekombination der Gene mit anderen Individuen der Population. Dadurch werden neue Individuen hervorgebracht, die einige Eigenschaften ihrer Eltern aufweisen. Die schlechtesten Mitglieder einer Population werden mit der geringsten Wahrscheinlichkeit zur Reproduktion ausgewählt und sterben aus.

Eine neue Population möglicher Lösungen wird also erzeugt durch *Selektion* der besten Individuen der aktuellen Generation und einer anschließenden genetischen *Reproduktion*. Die neue Generation enthält einen höheren Anteil der Eigenschaften der guten Mitglieder aus der vorigen Generation. Durch dieses Vorgehen werden über viele Generationen hinweg gute Eigenschaften der Individuen innerhalb der Population verteilt, kombiniert und mit anderen guten Eigenschaften ausgetauscht. Durch die Bevorzugung von Individuen mit einer hohen Fitneß sollen nur die erfolgversprechenden Gebiete des Suchraumes erkundet werden. Falls der genetische Algorithmus richtig implementiert ist, konvergiert die Population in eine optimale Lösung des Problems.

3.2 Grundlegende Prinzipien

Der Pseudokode eines traditionellen genetischen Algorithmus ist in Abbildung 3.1 dargestellt.

Notwendige Voraussetzungen für die Anwendung eines genetischen Algorithmus ist das Finden einer geeigneten Repräsentation (Kodierung) für das Optimierungsproblem. Weiterhin wird eine Bewertungs- und eine Fitneßfunktion benötigt. Diese Aspekte werden im folgenden näher beschrieben.

- Erstellen und zufälliges Initialisieren der Anfangspopulation
- Berechnen der Bewertung und Fitneß jedes Individuums
- Wiederholen bis Fitneß zufriedenstellend oder Abbruchbedingung erreicht ist
  - Selektieren von Paaren aus der alten Generation zur Reproduktion
  - Reproduktion von jeweils einem Paar zu zwei neuen Individuen (durch *Crossover* und *Mutation*)
  - Berechnen der *Bewertung* und *Fitneß* der Nachkommen
  - Ersetzen der Individuen der aktuellen Generation durch die Nachkommen

Abbildung 3.1: Pseudokode eines traditionellen genetischen Algorithmus

3.2.1 Das Kodierungsproblem

John H. Holland und die anderen Vertreter der genetischen Algorithmen haben dem Kodierungsproblem besondere Aufmerksamkeit geschenkt. Das heißt, sie haben vor allem die Frage der Kodierung der genetischen Information in den Vordergrund ihrer Untersuchungen gestellt und die Konsequenzen, Vorteile und Grenzen bestimmter Kodierungsformen analysiert [12, 16].

Es wird vorausgesetzt, daß eine potentielle Problemlösung als eine Menge von Parametern repräsentiert wird. Diese Parameter, *Gene* genannt, werden zusammengefaßt zu einer Folge von Werten, den sogenannten *Chromosomen*.

In genetischen Algorithmen werden die Chromosomen einer Population in der Regel als *binäre Vektoren* kodiert. Aus theoretischen Gesichtspunkten ist es wichtig, die Informationen nicht so kompakt wie möglich (beispielsweise als reelle Zahlen – wie bei den Evolutionsstrategien), sondern so breit wie möglich (binär) zu kodieren.

In der Genetik bezeichnet man die Gesamtheit aller Gene eines Lebewesens als *Genotyp*. Ein Genotyp enthält die zur Konstruktion eines Organismus benötigten Informationen. Der Organismus selbst heißt dann *Phänotyp*. Man benutzt diese Begriffe auch bei genetischen Algorithmen. Im Falle des Erfüllbarkeitsproblems ist der Genotyp die Menge der Parameter, die die Belegung der Formel beschreibt. Die daraus konstruierte Formel mit den instantiierten Variablen wird dann Phänotyp genannt.

Die Fitneß eines Individuums hängt von den Eigenschaften des Phänotyps ab. Sie kann aber aus dem Genotyp mit Hilfe der Fitneß- bzw. Bewertungsfunktion berechnet werden.

3.2.2 Die Fitneß- und Bewertungsfunktionen

Für jedes zu lösende Problem muß eine Fitneß- und eine Bewertungsfunktion definiert werden.

Die *Bewertungsfunktion* legt das eigentliche Optimierungskriterium und -ziel fest. Ein genetischer Algorithmus versucht, durch systematisches Erzeugen von (in der Regel suboptimalen) Lösungen das Optimum der Bewertungsfunktion (je nach Problemstellung ein Minimum oder Maximum) zu finden. Die Bewertungsfunktion mißt dabei, wie nahe ein Chromosom dem oder den gesuchten optimalen Werten ist. Sie ist ein Maß für die Güte. An die Bewertungsfunktion muß in der Regel keine besondere Anforderung gestellt werden.

Bei genetischen Algorithmen unterscheidet man im allgemeinen zwischen der *Bewertung* eines Individuums und seiner *Fitneß*. Die Bewertung eines Individuums wird mit einer *Fitneßfunktion* zunächst in einen Fitneßwert umgerechnet. Auf der Basis dieses Wertes wird dann entschieden, mit welcher Wahrscheinlichkeit ein Individuum an dem Prozeß der Erzeugung von Nachkommen teilhaben darf. Aus diesem Grund sollte der Wertebereich der Fitneßfunktion auf positive Werte beschränkt sein.

Die Bewertung und die Fitneß eines Individuums haben eine unterschiedliche Bedeutung. Die Bewertungsfunktion mißt, wie gut ein Chromosom das Optimum approximiert. Die Fitneßfunktion hingegen gibt an, wie sich aus der Güte – also der Bewertung des Chromosoms – die Chance für das Chromosom errechnet, sich in der nächsten Generation zu reproduzieren. Daß man dennoch oft zwischen den Begriffen nicht unterscheidet, liegt daran, daß man für die Bewertung und die Fitneß häufig die gleiche Funktion verwenden kann (aber nicht muß) oder daran, daß die Fitneßfunktion der Einfachheit halber meist eine simple Transformation der Bewertung darstellt.

3.2.3 Reproduktion von Individuen

Während der Reproduktionsphase eines genetischen Algorithmus werden die Individuen der Population *selektiert* und *rekombiniert*, um neue Individuen für die nächste Generation zu erzeugen. Die *Selektion* der Elternpaare

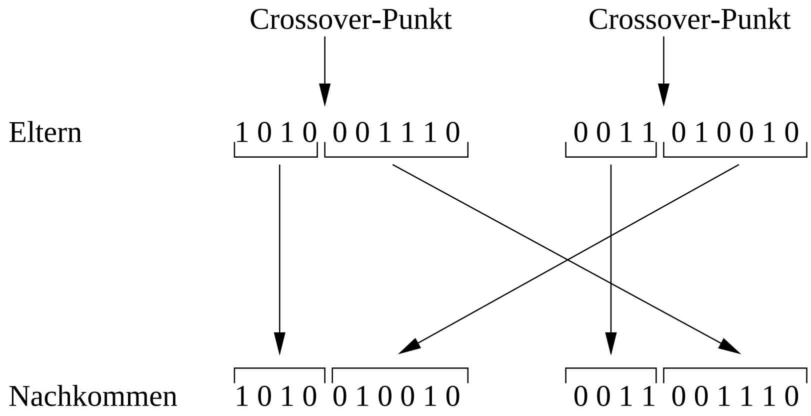


Abbildung 3.2: *One-point crossover* als einfachstes Crossover-Verfahren

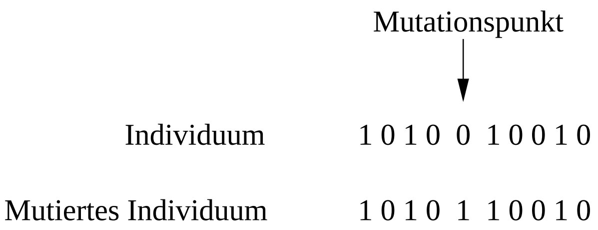


Abbildung 3.3: Die gleichverteilte Mutation

aus der Population geschieht mit einem zufälligen Verfahren, das Individuen mit einer großen Fitneß bevorzugt. Dadurch werden die guten Individuen wahrscheinlich mehrere Male ausgewählt, schlechte dagegen möglicherweise überhaupt nicht.

Nach der Selektion der Eltern werden deren Chromosomen *rekombiniert*. Das geschieht typischerweise durch *Crossover* und *Mutation*. Im folgenden sollen die einfachsten Formen dieser Operatoren kurz beschrieben werden.

Die einfachste Form der Vererbung von Chromosomensätzen ist das *One-point Crossover*. Dabei werden die Chromosomenketten der Elternteile an einer zufällig gewählten Position getrennt, wodurch zwei Anfangssegmente und zwei Endsegmente entstehen. Die Endsegmente werden dann vertauscht, und es entstehen zwei vollständige neue Chromosomensätze. Die Nachkommen besitzen dann Gene von beiden Elternteilen. Siehe Abbildung 3.2.

Die Crossover-Operation wird gewöhnlich nicht auf alle selektierten Individuenpaare angewandt. Die Entscheidung, ob ein Crossover durchgeführt wird, geschieht zufällig mit einer Wahrscheinlichkeit typischerweise zwischen 0.6

und 1.0. Findet kein Crossover statt, dann werden die Nachkommen einfach durch Duplizieren ihrer Eltern erzeugt. Dadurch hat ein Individuum auch die Möglichkeit, seine gesamten Gene ohne Veränderung durch Crossover zu vererben.

Nach dem Crossover wird jedes neue Individuum *mutiert*. Die *gleichverteilte Mutation* ist die Standardvariante des Mutationsoperators in genetischen Algorithmen. Mit einer kleinen Wahrscheinlichkeit, typischerweise um 0.001, wird dabei jedes Gen verändert. Abbildung 3.3 zeigt, wie das fünfte Gen des Chromosoms mutiert wird.

Mutationen dienen in genetischen Algorithmen vorwiegend und fast ausschließlich dem Zweck, eine frühzeitige Konvergenz zu verhindern und für eine gewisse Divergenz und Inhomogenität innerhalb der Population zu sorgen. Die Mutation liefert einen kleinen Beitrag zur zufälligen Suche und hilft sicherzustellen, daß kein Punkt im Suchraum eine Erreichbarkeitswahrscheinlichkeit von Null hat. Das Ziel der Crossover-Verfahren ist es, den Suchraum möglichst effizient zu durchschreiten. Aus diesem Grund wird der Crossover-Operator häufig als der bedeutendere der beiden Reproduktionsmechanismen angesehen.

3.2.4 Konvergenz einer Population

Ein genetischer Algorithmus ist korrekt implementiert, wenn sich die Population sukzessive über den Generationen so entwickelt, daß sich die maximale und die mittlere Fitneß der Individuen in jeder Generation vergrößert und zwar in Richtung des globalen Optimums. Der Selektionsdruck und die Bewertungen der Individuen führen im Laufe der Generationen zu immer homogeneren Populationen. Dieser Prozeß der Vereinheitlichung der Gene in der Population wird *Konvergenz* genannt.

Man sagt ein *Gen* ist konvergiert, wenn es in 95% der Population denselben Wert hat [21]. Eine *Population* ist konvergiert, wenn alle ihre Gene konvergiert sind.

3.2.5 Ein Beispiel für einen genetischen Algorithmus

Zur Veranschaulichung der Arbeitsweise eines genetischen Algorithmus soll an dieser Stelle ein Beispiel betrachtet werden. Dabei soll mit Hilfe eines genetischen Algorithmus ein Modell für die aussagenlogische Formel

Nr.	Zufällig erzeugte Anfangspopulation	Bewertung f_{obj}	p_s	$p_s \cdot \lambda$	Anzahl Selektionen
(1)	1 1 0 0 0 0	6	0.24	0.96	1
(2)	1 0 0 1 0 0	7	0.28	1.12	2
(3)	1 1 0 0 1 0	7	0.28	1.12	1
(4)	0 1 0 0 0 0	5	0.20	0.80	0
Summe		25	1.00	4.00	4
Mittel		6.25			

Tabelle 3.1: Simulation eines einfachen genetischen Algorithmus: Erzeugung der Anfangspopulation und Selektion

$$
\begin{aligned}
&(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_4 \vee \neg x_1) \wedge \\
&(\neg x_1 \vee x_5 \vee x_3) \wedge (\neg x_6 \vee x_1 \vee x_4) \wedge \\
&(x_4 \vee x_3 \vee \neg x_2) \wedge (x_3 \vee x_1 \vee x_5) \wedge \\
&(\neg x_4 \vee x_5 \vee x_3) \wedge (x_4 \vee \neg x_2 \vee x_3) \wedge \\
&(x_1 \vee x_5 \vee x_6) \wedge (x_6 \vee x_4 \vee \neg x_2)
\end{aligned}
$$

gefunden werden. Die Formel enthält sechs aussagenlogische Variablen und besteht aus zehn Klauseln. Die Population soll in diesem Beispiel aus vier Individuen bestehen. Die Formel und die Größe der Population wurden so gewählt, daß die Arbeitsweise des Algorithmus Schritt für Schritt nachvollzogen werden kann. In praktischen Anwendungsfällen wäre sowohl die Variablen- und Klauselanzahl als auch die Anzahl der Individuen in der Population größer.

Der erste Schritt für die Anwendung eines genetischen Algorithmus ist das Finden einer geeigneten binären Kodierung für die möglichen Lösungen. Die Lösungen sind in diesem Fall Belegungen für eine aussagenlogische Formel. Eine Belegung läßt sich am einfachsten als Bitkette darstellen. Für jede Variable ist in der Bitkette genau eine Stelle vorhanden. Dabei entspricht 0 dem Wahrheitswert $\perp$ und 1 dem Wahrheitswert $\top$.

Der nächste Schritt ist die Definition einer Fitneß- und einer Bewertungsfunktion. In diesem einfachen Beispiel wird nicht zwischen Fitneß und Bewertung eines Individuums unterschieden. Deshalb wird nur die Bewertungsfunktion benötigt. Diese Funktion soll beschrieben, wie nahe ein Chromosom der gesuchten Lösung, in diesem Fall der erfüllenden Belegung, ist. Für das Erfüllbarkeitsproblem kann die Bewertungsfunktion am einfachsten als die Anzahl der durch die Belegung erfüllten Klauseln definiert werden.

Selektierte Individuen	Crossover-Partner	Crossover-Punkt	Neue Population	Bewertung f_{obj}
1 1 — 0 0 0 0	(2)	2	1 0 0 0 0 0	9
1 0 — 0 1 0 0	(1)	2	1 1 0 1 0 0	8
1 1 0 0 — 1 0	(3)	4	1 0 0 1 1 0	9
0 1 0 0 — 0 0	(2)	4	1 1 0 0 0 0	7
Summe				33
Mittel				8.25

Tabelle 3.2: Simulation eines einfachen genetischen Algorithmus: Crossover

Nach dem Festlegen einer geeigneten Kodierung und der Definition der Bewertungsfunktionen sind alle Voraussetzungen für die Anwendung des genetischen Algorithmus gegeben. Die Abarbeitung des Algorithmus beginnt mit dem Erstellen und zufälligen Initialisieren der Anfangspopulation. Das heißt für diesen Fall, es müssen vier Individuen erzeugt werden. Die Belegungen werden dabei durch gleichverteilte Auswahl von Nullen und Einsen erzeugt. Für jedes Individuum wird danach die Bewertung als Resultat der Bewertungsfunktion bestimmt. Die erzeugten Individuen und ihre Bewertungen sind im linken Teil der Tabelle 3.1 dargestellt.

Der genetische Algorithmus erzeugt eine neue Generation möglicher Lösungen durch Selektion der besten Individuen aus der aktuellen Generation (in diesem Fall aus der Anfangspopulation) und einer anschließenden genetischen Reproduktion. In unserem Fall müssen vier Individuen (zwei Paare) aus der aktuellen Generation selektiert werden. Die Selektionswahrscheinlichkeit eines Individuums soll dabei proportional zur Bewertung des Individuums sein. In diesem Fall werden Individuum (1) einmal, Individuum (2) zweimal und Individuum (3) einmal ausgewählt. Individuum (4) darf sich aufgrund seiner geringen Bewertung nicht an der Reproduktion beteiligen. Siehe Tabelle 3.1.

Die Reproduktion besteht aus der Anwendung des Crossover-Verfahrens und einer anschließender Mutation der Nachkommen. In diesem Beispiel soll *One-point crossover* zur Anwendung kommen. Als Crossover-Wahrscheinlichkeit wird 1.0 angenommen. Das heißt, für jedes selektierte Paar werden die Chromosomenketten an einer zufällig gewählten Position getrennt und die Endsegmente vertauscht. Die Ergebnisse des Crossover-Verfahrens sind in Tabelle 3.2 dargestellt.

Danach werden einzelne Bits der Nachkommen mutiert. In diesem Beispiel wird eine Mutationswahrscheinlichkeit von 0.01 angenommen. Bei den insgesamt vorhandenen 24 Bit-Positionen kann man erwarten, daß $20 \cdot 0.01 = 0.2$ Bits während einer Generation mutiert werden. In unserem Beispiel, das eine Simulation eines realen genetischen Algorithmus darstellen soll, wird kein einziges Bit mutiert. Das heißt, die durch Crossover erzeugten Nachkommen bilden unverändert die neue Generation von Belegungen für die gegebene Formel.

Die Erzeugung neuer Generationen von Belegungen wird solange wiederholt, bis entweder ein Indivuum mit der Fitneß 10 gefunden wurde (eine erfüllende Belegung) oder ein anderes Abbruchkriterium (Konvergenz, maximale Anzahl von Generationen) erfüllt wird.

Obwohl man keine allgemeinen Aussagen von einem einzigen Simulationszyklus ziehen kann, soll an dieser Stelle ein Blick auf das Verhalten der Mittelwerte der Individuen-Bewertungen geworfen werden. Die mittlere Bewertung der Population hat sich während eines Generationsschrittes von 6.25 auf 8.25 verbessert.

Das vorgestellte Beispiel sollte einen ersten Eindruck von der Arbeitsweise eines einfachen genetischen Algorithmus vermitteln. In den folgenden Abschnitten werden eine Reihe von genetischen Operatoren vorgestellt und ihre Nützlichkeit zur Lösung des SAT-Problems für $3CNF$-Formeln diskutiert.

3.3 Theoretische Grundlagen

Ein Großteil der Forschung auf dem Gebiet der genetischen Algorithmen war darauf gerichtet, empirische Regeln zur Effizienzverbesserung zu finden. Es gibt noch keine akzeptierte allgemeine Theorie zur Erklärung der Eigenschaften von genetischen Algorithmen. Trotzdem existieren einige Hypothesen, mit denen es möglich ist, ihren Erfolg teilweise zu erklären.

3.3.1 Schemata und das Schema-Theorem

An dieser Stelle soll nochmal auf das Kodierungsproblem zurückgekommen werden. In der Theorie der genetischen Algorithmen hat man sich ausführlich damit beschäftigt, welche Auswirkungen die Anwendung der genetischen Operatoren (Selektion, Crossover und Mutation) auf eine Population von Chromosomen haben kann.

Das *Schema-Theorem* von John H. Holland [16] war die erste gründliche Erklärung der Arbeitsweise genetischer Algorithmen. Unter einem *Schema* versteht man ein Muster von Genen, das (bei binärer Kodierung) als eine Zeichenkette über dem Alphabet $\{0, 1, \#\}$ dargestellt werden kann. Man sagt, ein gegebenes Chromosom enthält ein bestimmtes Schema, wenn es mit dem Schema übereinstimmt, wobei das *don't care*-Symbol „#" als eine Variable aufgefaßt werden kann. Das Chromosom „1010" enthält beispielsweise neben anderen die Schemata „10##", „#0#0", „##1#" und „101#". Unter der *Ordnung* eines Schemas versteht man die Anzahl der Positionen, die nicht das Variablensymbol „#" enthalten (2, 2, 1, 3 im Beispiel). Die *definierende Länge* eines Schemas ist der Abstand zwischen den äußersten Nichtvariablensymbolen (2, 3, 1, 3 im Beispiel).

Das *Schema-Theorem* beschreibt die Verarbeitung der Schemata durch den genetischen Algorithmus. Dabei wird unter anderem die Frage behandelt, welche „Überlebenschancen" ein Schema hat und wovon diese abhängen.

Den Individuen in der Population wird die Möglichkeit gegeben, sich zu reproduzieren. Dabei verhält sich die Anzahl dieser Möglichkeiten proportional zur Fitneß der Individuen. Die nächste Generation enthält also vorwiegend Gene der besseren Individuen. Dabei soll nun angenommen werden, daß ein Individuum genau dann eine gute Fitneß besitzt, wenn es gute Schemata enthält. Folglich enthält die neue Generation auch mehr gute Schemata als die alte. Leider ist damit die Situation noch nicht vollständig erfaßt. Die Schemata können nämlich durch die genetischen Operatoren Crossover und Mutation auch wieder zerstört werden.

John H. Holland hat gezeigt, daß der Lösungsraum optimal durchsucht wird, wenn man den Individuen proportional zu ihrer Fitneß die Möglichkeit gibt, sich zu reproduzieren. Bei diesem Selektionsverfahren werden gute Schemata wesentlich (sogar exponentiell) häufiger zur Erzeugung der Nachkommen herangezogen als schlechte Schemata. Die Wahrscheinlichkeit, daß gute Schemata wieder zerstört werden, ist offensichtlich von ihrer Länge und Ordnung abhängig.

Satz 4 (Schema-Theorem – verbal) *Bei genetischen Algorithmen erhalten Schemata mit überdurchschnittlicher Fitneß, kurzer definierender Länge und niedriger Ordnung in den Folgegenerationen steigende Chancen sich zu reproduzieren.*

Der Wert des Theorems besteht nicht so sehr darin, daß es verständlich macht, wie ein genetischer Algorithmus arbeitet, sondern eher Hinweise für eine geeignete Kodierung gibt [38].

Die Anzahl der Schemata, die bei einer Population von λ Chromosomen pro Generationen indirekt verarbeitet werden, liegt bei $O(\lambda^3)$ [12]. Diese Zahl ist im Vergleich zu der tatsächlich pro Generation erzeugten und bewerteten Individuenanzahl λ verblüffend hoch. Man nennt diese Eigenschaft eines genetischen Algorithmus auch *implizite Parallelität*. Sie ist eine mögliche Erklärung für dessen Effizienz.

3.3.2 Die Building block-Hypothese

Entsprechend dem Schema-Theorem benutzt ein genetischer Algorithmus kompakte Schemata mit überdurchschnittlicher Fitneß, auch anschaulich *Building blocks* genannt, um aus ihnen optimale Chromosomen zu rekombinieren. Diese Eigenschaft wurde von David E. Goldberg als *Building block-Hypothese* formuliert [12].

Hypothese 1 *Die Verwendung von building-blocks als Grundlage eines allgemeinen Such- und Optimierungsverfahrens ist sinnvoll und effizient, wenn das Kodierungsschema die folgenden beiden Bedingungen erfüllt:*

1. *Die Teile eines Chromosoms, die inhaltlich zusammengehörige Informationen kodieren, liegen eng zusammen.*
2. *Die Wechselwirkung zwischen den Genen ist gering.*

Die Bedingung (2) ist eine Voraussetzung für (1). Eine *Wechselwirkung* zwischen Genen bedeutet, daß der Beitrag eines Gens zur Fitneß von den Werten anderer Gene in dem Chromosom abhängt. Bei praktischen Problemen, wie bei der Optimierung multimodaler Funktionen, gibt es jedoch immer einer derartige Wechselwirkung.

Soll ein genetischer Algorithmus eingesetzt werden, muß man eine Kodierung des Problems in eine eindimensionale binäre Zeichenkette finden. Die Wahl dieser Kodierung beeinflußt die Effizienz des Algorithmus entscheidend. Dabei kann man davon ausgehen, daß eine Kodierung entsprechend der *Building block*-Hypothese eine maximale Effizienz sichert.

3.3.3 Exploration und Exploitation

Jeder effiziente Optimierungsalgorithmus muß zwei Methoden benutzen, um ein globales Optimum zu finden: *Exploration* (Erforschung, Erkundung) zur

Untersuchung neuer und unbekannter Gebiete im Suchraum sowie *Exploitation* (Ausnutzung) zur Verwertung des schon über den Suchraum gesammelten Wissens zum Finden besserer Punkte. Diese beiden Forderungen sind konträr, und ein guter Suchalgorithmus muß einen Kompromiß zwischen beiden finden.

Ein reines zufälliges Suchverfahren hat gute Explorationseigenschaften, aber überhaupt keine Exploitation, wohingegen reine *hillclimbing*-Methoden ein gutes Exploitationsverhalten, aber kaum Exploration besitzen. Eine Kombination dieser beiden Strategien (wie *GWSAT* oder Simulated annealing) ist am erfolgversprechendsten, wobei es aber schwierig ist, eine Balance zwischen beiden zu finden.

John H. Holland [16] und David E. Goldberg [12] haben gezeigt, daß ein genetischer Algorithmus *gleichzeitig* Exploration und Exploitation optimal ausführt. Dieses Ergebnis hat aber nur eine theoretische Bedeutung, weil dabei vereinfachende Annahmen getroffen wurden, darunter, daß:

1. die Populationsgröße unendlich ist,
2. die Fitneßfunktion die Nützlichkeit der Lösung genau wiedergibt und
3. keine signifikante Wechselwirkung zwischen den Genen eines Chromosoms auftritt.

Die Annahme (1) ist in der Praxis unerfüllbar. Deshalb werden bei jedem genetischen Algorithmus stochastisch Fehler beim Finden des Optimums auftreten. Auch die Annahmen (2) und (3) sind für reale Probleme nur schwer zu erfüllen. In diesem Zusammenhang ist es sicher interessant zu erörtern, welche Bedeutung die genetischen Operatoren und ihre Wahrscheinlichkeiten für das Explorations- und Exploitationsverhalten eines genetischen Algorithmus haben. Es soll an dieser Stelle auf [20] verwiesen werden.

3.4 Anwendungen

Genetische Algorithmen wurden erfolgreich angewandt zur Optimierung von Funktionen, Plänen und physikalischen Problemen. Ferner existieren erste vielversprechende Ergebnisse beim Einsatz zur Lösung von Klassifikationsproblemen, beim maschinellen Lernen und bei der Simulation von biologischen, ökologischen, ökonomischen und sozialen Systemen. Eine Zusammenstellung aktueller Anwendungen genetischer Algorithmen findet man in [5,17]. Der Einsatz genetischer Algorithmen zur Steuerung eines Pipeline-Systems, zur Optimierung von Kommunikationssoftware oder zum Entwurf eines Mantelstrom-Düsentriebwerks sind dabei herausragende Beispiele.

Kapitel 4

Genetische Algorithmen für das 3SAT-Problem

4.1 Die Zielstellung

Die Zielstellung für das weitere Vorgehen kann aus den bereits gewonnenen Erkenntnissen abgeleitet werden. Deshalb soll an dieser Stelle kurz zusammengefaßt werden.

1. Das Erfüllbarkeitsproblem für aussagenlogische Formeln SAT ist $\mathcal{NP}$-vollständig. Das trifft auch auf die syntaktisch eingeschränkte Variante $3SAT$ zu. Siehe Abschnitt 2.2.

2. Es besteht ein großes Interesse an effizienten Lösungsverfahren für SAT. Diese Verfahren sollen auch noch für Formeln mit mehreren Hundert Variablen praktisch anwendbar sein. Siehe Abschnitt 2.3.

3. Es wurde erläutert, daß unvollständige Suchverfahren, wie $GSAT$, $GWSAT$ oder Simulated annealing, bei Formelklassen mit vielen Variablen vollständigen Entscheidungsverfahren in praktischen Gesichtspunkten überlegen sind. Siehe Abschnitt 2.4.2.

4. Genetische Algorithmen konnten erfolgreich für verschiedene $\mathcal{NP}$-vollständige Optimierungsprobleme eingesetzt werden, darunter sind *Traveling salesman problem*, *Subset sum problem*, *Maximum cut problem* und *Maximum tardy task*. Siehe [23, 29].

Angesichts dieser Aussagen ist es sicherlich naheliegend, die Verfahren der simulierten Evolution auch auf das Erfüllbarkeitsproblem anzuwenden. Von

N	Anzahl von Variablen in einer Formel
$m = 4.3 \cdot N$	Anzahl von Klauseln in einer Formel
$k = 3$	Anzahl von Literalen pro Klausel

Formelklasse	N	Anzahl von Testformeln
$(forced, 100)$	100	100
$(forced, 200)$	200	100
$(forced, 300)$	300	50
$(unforced, 100)$	100	100

Tabelle 4.1: Die verwendeten Formelklassen, ihre Parameter und die Anzahl der jeweils vorhandenen Testformeln

besonderem Interesse wird dabei der Vergleich mit anderen unvollständigen Suchverfahren sein. Nur dadurch kann man erkennen, ob die Anwendung der genetischen Algorithmen Vorteile bietet oder nicht.

In den folgenden Abschnitten werden eine Reihe von genetischen und evolutionären Algorithmen für das $3SAT$-Problem vorgestellt. Zur Evaluierung der entworfenen Algorithmen wurden die Testformeln aus Hölldobler et al. [18] verwendet. Damit ist ein direkter Vergleich der evolutionären Verfahren mit den zur Zeit erfolgreichsten unvollständigen SAT-Algorithmen, den $GSAT$-Verfahren, möglich.

Die entworfenen genetischen und evolutionären Algorithmen wurden mit schwierigen zufällig generierten $3CNF$-Formeln, sogenannten *hard random*-Formeln, getestet. Dabei wurde zusätzlich zwischen *forced*- und *unforced*-Formeln unterschieden. Siehe Abschnitt 2.5.1. Die Parameter der verwendeten Formelklassen sind in Tabelle 4.1 zusammengefaßt.

Eine aussagenlogische Formel einer bestimmten Formelklasse kann dann mit Hilfe des Tripels (Typ, N, Formelnummer) identifiziert werden, beispielsweise bezeichnet $(forded, 100, 10)$ die Formel mit der Nummer 10 aus der Formelklasse $(forced, 100)$.

4.2 Zwei Problemlösungsstrategien

Der klassische genetische Algorithmus, wie er in Abschnitt 3.2 vorgestellt wurde, benutzt außer in der Bewertungsfunktion kein problemspezifisches Wissen zum Finden einer optimalen Lösung. In diesem Fall besteht die eigentliche Schwierigkeit darin, eine geeignete binäre Kodierung des Problems zu finden, damit die klassischen Bit-Operatoren für Crossover und Mutation effizient arbeiten.

Auf der anderen Seite hat es sich in vielen Fällen als nützlich erwiesen, problemspezifische genetische Operatoren zu benutzen [29]. Dabei wird die Schwierigkeit des Kodierungsproblems auf den Entwurf der genetischen Operatoren verlagert.

Im allgemeinen werden unter dem Begriff „evolutionäre Algorithmen“ alle Verfahren der simulierten Evolution, einschließlich der genetischen Algorithmen, zusammengefaßt. In dieser Arbeit wird der Begriff jedoch aus methodischen Gründen mit einer speziellen Bedeutung verwendet, nämlich zur Unterscheidung, ob bei den genetischen Operatoren auch problemspezifische Informationen genutzt werden.

Definition 8 (Genetischer Algorithmus) *Ein genetischer Algorithmus ist ein Verfahren der stochastischen Optimierung, bei dem im Computer ein Evolutionsprozeß simuliert wird. Durch die Anwendung der genetischen Operatoren Selektion, Crossover und Mutation wird über einer Population von möglichen Lösungen, diejenige Lösung gesucht, deren Parameter ein Minimum oder Maximum der zu optimierenden Funktion produzieren. Die genetischen Operatoren benutzen außer der Fitneß- und Bewertungsfunktion keine problemspezifischen Information.*

Definition 9 (Evolutionärer Algorithmus) *Ein evolutionärer Algorithmus ist ein Verfahren der stochastischen Optimierung, bei dem im Computer ein Evolutionsprozeß simuliert wird. Durch die Anwendung der genetischen Operatoren Selektion, Crossover und Mutation wird über einer Population von möglichen Lösungen, diejenige Lösung gesucht, deren Parameter ein Minimum oder Maximum der zu optimierenden Funktion produzieren. Die genetischen Operatoren werden für den konkreten Anwendungsfall definiert. Das heißt, sie können problemspezifische Informationen zur Konstruktion der Lösungen benutzen.*

Zur Lösung des Erfüllbarkeitsproblems mit evolutionären Verfahren ergeben sich aus dieser Sicht als mögliche Problemlösungsstrategien:

- der Entwurf *traditioneller genetischer Algorithmen* unter Verwendung universeller genetischer Operatoren für Crossover und Mutation sowie
- der Entwurf *evolutionärer Algorithmen* unter Verwendung problemspezifischer genetischer Operatoren.

Für das Erfüllbarkeitsproblem ist nicht abzusehen, welche der beiden Strategien erfolgreicher sein wird. Aus diesem Grund werden beide Varianten betrachtet, in Kapitel 4 die genetischen und in Kapitel 5 die evolutionären Algorithmen.

4.3 Der einfache genetische Algorithmus

Für ein gegebenes Problem, in diesem Falle $3SAT$, kann man verschiedene genetische Algorithmen konstruieren. Die einzelnen Algorithmen können sich dabei in den verwendeten Selektions- und Crossover-Verfahren sowie in den Populationsmodellen unterscheiden. Die erste und einfachste Variante für einen genetischen Algorithmus zur Lösung von $3SAT$ entspricht dem Algorithmus, der bereits in Abschnitt 3.2 beschrieben wurde.

Sei Φ eine $3CNF$-Formel mit m Klauseln und N aussagenlogischen Variablen. Eine Belegung A weist jeder der in Φ vorkommenden Variablen einen Wahrheitswert zu. Ferner sei $X = \{0,1\}^N$ die Menge der Binärfolgen mit der Länge N. Dann läßt sich die Belegung A darstellen als $x = x_1 x_2 \ldots x_N$ mit $x \in X$. Dabei entspricht 0 dem Wahrheitswert $\perp$ und 1 dem Wahrheitswert $\top$.

Eine *Bewertungsfunktion* $f_{obj} : X \to \mathbb{N}$ weist jeder Belegung x eine natürliche Zahl zu, die sogenannte *Bewertung*. Für das $3SAT$-Problem kann die Bewertungsfunktion am einfachsten definiert werden als die Anzahl der durch die Belegung x erfüllten Klauseln:

$$f_{obj}(x) \stackrel{def}{=} m - \|\mathcal{C}_u(\Phi, x)\|$$

Eine *Fitneßfunktion* $f_{fit} : X \to \mathbb{R}^+$ weist jeder Belegung x ausgehend von ihrer Bewertung eine positive reelle Zahl zu, die sogenannte *Fitneß*. Wenn nicht anders angegeben gilt

$$f_{fit}(x) \stackrel{def}{=} f_{obj}(x).$$

Eine *Population* ist eine (indizierte) Multimenge über X. Sei $\mathcal{H}(t)$ eine Population zum Zeitpunkt t, dann gilt $\lambda = \|\mathcal{H}(t)\|$ für alle $t \geq 0$. Das heißt, λ bezeichnet die konstante Größe der Population. Ferner wird vorausgesetzt, daß λ eine gerade Zahl ist.

Der *einfache genetische Algorithmus SGA* wird charakterisiert durch drei Operationen: *Selektion* T_{RWS}, *Crossover* T_{1X} und *Mutation* T_{UM}. Diese Operationen sind wie folgt definiert:

Selektion. Zur Berechnung von $T_{RWS}(\mathcal{H})$ werden λ Elemente aus $\mathcal{H}$ ausgewählt, wobei jedes Element $x \in \mathcal{H}$ eine Selektionswahrscheinlichkeit $p_s(x)$ besitzt.

$$p_s(x) = \frac{f_{fit}(x)}{\sum\limits_{y \in \mathcal{H}} f_{fit}(y)}$$

Der Operator T_{RWS} wird *Roulette-wheel selection* genannt.

Crossover. Zur Berechnung von $T_{1X}(\mathcal{H})$ werden die folgenden Schritte $\lambda/2$ Male wiederholt. Wähle ein $x = x_1 x_2 \dots x_N \in \mathcal{H}$ und ein $y = y_1 y_2 \dots y_N \in \mathcal{H}$ zufällig und gleichverteilt aus. Wähle mit der Wahrscheinlichkeit p_c einen zufälligen Index i mit $1 \leq i \leq N+1$, bilde $z_1 = x_1 \dots x_{i-1} y_i \dots y_N$ und $z_2 = y_1 \dots y_{i-1} x_i \dots x_N$ sowie füge z_1 und z_2 in $T_{1X}(\mathcal{H})$ ein. Anderenfalls, also mit der Wahrscheinlichkeit $1 - p_c$, füge x und y in $T_{1X}(\mathcal{H})$ ein. Der Operator T_{1X} wird *One-point crossover* genannt. Der Parameter p_c heißt *Crossover-Wahrscheinlichkeit.*

Mutation. Zur Berechnung von $T_{UM}(\mathcal{H})$ werden die folgenden Schritte für jedes $x \in \mathcal{H}$ mit der Wahrscheinlichkeit p_m ausgeführt. Wähle einen Index i mit $1 \leq i \leq N$ zufällig und gleichverteilt aus, und invertiere das i-te Bit x_i. Der Operator T_{UM} wird *gleichverteilte Mutation* oder *Uniform mutation* genannt. Der Parameter p_m heißt Mutationswahrscheinlichkeit.

Sei T die komposite Abbildung $T = T_{UM}(T_{1X}(T_{RWS}(\mathcal{H})))$ und $\mathcal{H}(t)$ die Population zum Zeitpunkt t. Dann gilt $\mathcal{H}(t+1) = T(\mathcal{H}(t))$. Ein solcher Berechnungsschritt wird *Generation* genannt. Der einfache genetische Algorithmus SGA kann nun als iterative Ausführung von T definiert werden.

Bei Beginn der Abarbeitung von SGA wird die Population $\mathcal{H}$ zufällig initialisiert (INITIALIZE). Die Abarbeitung kann beendet werden, wenn

1. ein Modell gefunden wurde, das heißt, es gilt $\max\{f_{obj}(x) \mid x \in \mathcal{H}\} = m$, oder

2. eine bestimmte Anzahl von Generationen t_{max} erreicht ist, das heißt, die Suche wird erfolglos abgebrochen.

Diese Abbruchbedingungen werden durch die Funktion DONE realisiert. Falls nichts anderes angegeben ist, wurde bei den Testläufen $t_{max} = 1500$ benutzt.

Bei *Roulette-wheel selection* geschieht die Auswahl der einzelnen Individuen proportional zu ihrer Fitneß. Die Bezeichnung *Roulette-wheel* bezieht sich

SGA()
$t \leftarrow 0$
$\mathcal{H} \leftarrow$ INITIALIZE()
while $\neg$DONE$(t, \mathcal{H})$
do $t \leftarrow t + 1$
    $\mathcal{H} \leftarrow T_{RWS}(\mathcal{H})$
    $\mathcal{H} \leftarrow T_{1X}(\mathcal{H})$
    $\mathcal{H} \leftarrow T_{UM}(\mathcal{H})$
return $\mathcal{H}$

Algorithmus 4.1: Der einfache genetische Algorithmus SGA

auf die Implementierung des Verfahrens. Dabei stellt man sich ein Roulette-Rad aufgerollt als Band vor. Die Länge des Bandes ist die Summe der Fitneß aller Individuen, also die Güte der Population. Je besser ein Individuum ist, desto breiter ist der Streifen, den es auf dem Band einnimmt. Nun wird eine zufällige Zahl zwischen Null und der Gesamtfitneß der Population ermittelt. Dasjenige Individuum, in dessen Bereich die Zahl liegt, wird selektiert.

Jeder genetische Algorithmus, somit auch SGA, besitzt die folgenden *exogenen*, von außen vorgegebenen, Parameter:

- Populationsgröße λ
- Crossover-Wahrscheinlichkeit p_c
- Mutationswahrscheinlichkeit p_m

Die *Populationsgröße* λ hat einen entscheidenden Einfluß auf die Qualität der Ergebnisse. Leider ist eine große Population auch immer mit einem großen Berechnungsaufwand verbunden. Deshalb muß die Wirkung der Populationsgröße für jedes zu bearbeitende Problem untersucht werden. Dabei sollte ein Kompromiß zwischen Effizienz und Qualität angestrebt werden. Ein empfohlener Wert ist $\lambda \approx 2 \cdot N$, wobei N die Größe eines Individuums in Bit bezeichnet. Zur Lösung des Erfüllbarkeitsproblems für Formeln mit 100 Variablen müssen dann 200 Individuen pro Generation verarbeitet werden.

Zur Bestimmung geeigneter Wahrscheinlichkeiten für Crossover und Mutation gab es für den einfachen genetischen Algorithmus zahlreiche experimentelle Untersuchungen [4]. Typische Einstellungen für die *Crossover-Wahrscheinlichkeit* sind:

$p_c \approx 0.6$	(De Jong, 1975)
$p_c \in [0.75, 0.95]$	(Schaffer et al., 1989)
$p_c \approx 0.95$	(Grefenstette, 1986)

Entsprechende Angaben werden auch für die *Mutationswahrscheinlichkeit* gemacht:

$p_m \approx 0.001$	(De Jong, 1975),
$p_m \in [0.005, 0.01]$	(Schaffer et al., 1989),
$p_m \approx 0.01$	(Grefenstette, 1986)

Zur Darstellung der Arbeitsweise von genetischen Algorithmen benutzt man gewöhnlich den Verlauf

1. der Bewertungen des besten Individuums und
2. des Mittelwertes der Bewertungen aller Individuen in der Population

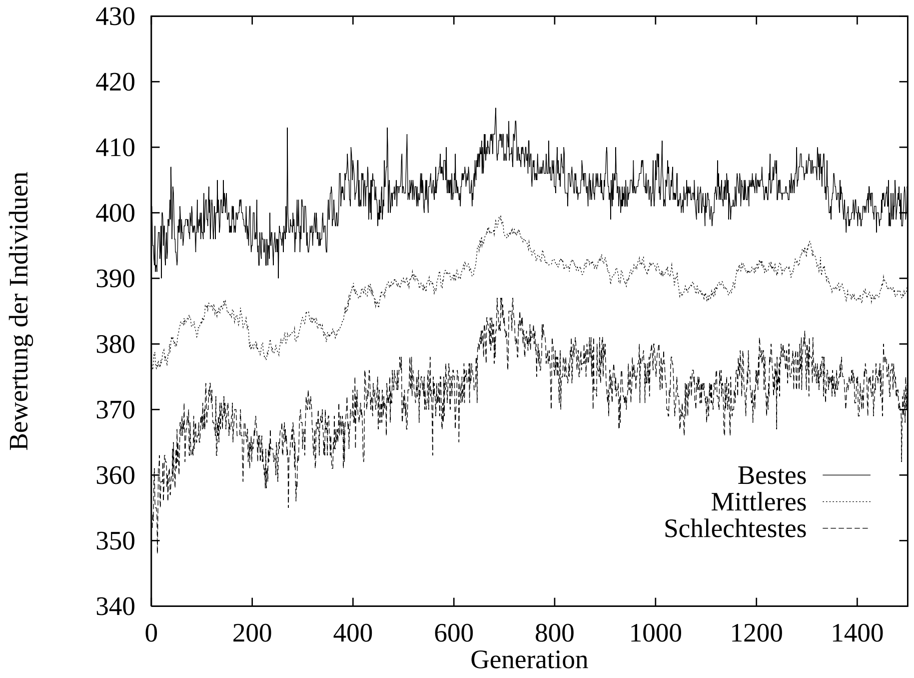
über den Generationen.

Zwei weitere Beschreibungsmittel für die Verbesserung bzw. Konvergenz einer Population wurden von Kenneth A. De Jong eingeführt [21]. Das sind *On-line performance* $P_e(t)$ und *Off-line performance* $P_e^*(t)$ für eine Generation t.

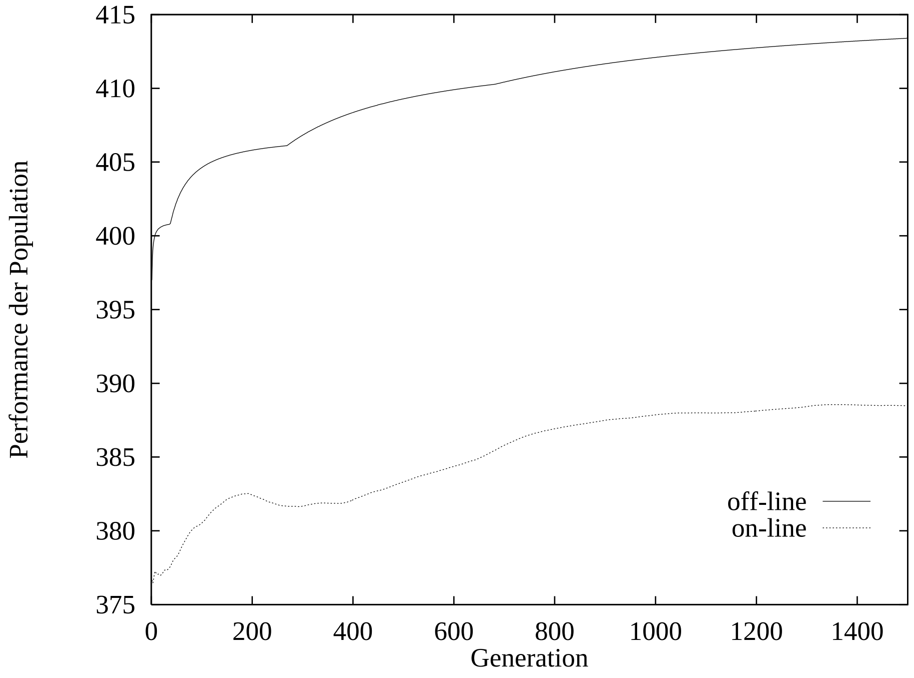
$$P_e(t) \stackrel{def}{=} \frac{1}{t}\sum_{\tau=1}^{t}\left[\frac{1}{\lambda}\sum_{x\in\mathcal{H}(\tau)} f_{obj}(x)\right]$$

$$P_e^*(t) \stackrel{def}{=} \frac{1}{t}\sum_{\tau=1}^{t}\max\{f_{obj}(x) \mid x \in \mathcal{H}(\tau') \text{ mit } 1 \leq \tau' \leq \tau\}$$

Im Algorithmus *SGA* benutzt man das denkbar einfachste *Populationsmodell*, das sogenannte *Generational replacement*. Dabei wird die aktuelle Population vollständig durch ihre Nachkommen ersetzt. Dieses Vorgehen bringt Gefahren mit sich, denn durch die Ersetzung der gesamten Population kann es vorkommen, daß sowohl die Bewertung des besten Individuums der Nachfolgegeneration als auch die durchschnittliche Bewertung der neuen Generation schlechter ist als in der Elterngeneration. Die Bewertungsfunktion ist dann weder für das beste Individuum noch für den Durchschnitt über die Generationen monoton steigend. Das heißt, die Population konvergiert nicht wie gewünscht in Richtung der optimalen Lösung. Siehe Abschnitt 3.2.4.



(a) Verlauf der Individuen-Bewertungen über den Generationen



(b) Verlauf der Performance über den Generationen

Abbildung 4.1: Eine Anwendung des genetischen Algorithmus SGA mit $\lambda = 200$, $p_c = 0.9$ und $p_m = 0.002$ auf die Formel $(forced, 100, 1)$

In Abbildung 4.1 ist ein typischer Verlauf der Bewertungen bzw. der Performance für den *SGA* dargestellt. Man kann deutlich erkennen, daß die Qualität der Belegungen nicht monoton zunimmt und ein Modell nicht gefunden wird.

Um eine Erhaltung der besten Elemente einer Population zu erreichen und damit ein monotones Ansteigen der Bewertung der besten Elemente zu gewährleisten wurde das Populationsmodell mit *Elitismus* entwickelt. Dabei wird eine bestimmte Anzahl der besten Elemente der aktuellen Population unverändert in die nächste Generation übernommen. Aus diesem Grund wird ein weiterer exogener Parameter eingeführt, das sogenannte *Generation gap* GG mit $0 < GG \leq 1$. Im Falle des Populationsmodells mit Elitismus werden dann $\lambda \cdot GG$ Individuen selektiert und entsprechend viele Nachkommen erzeugt. Beim Einfügen der neuen Individuen in die nächste Generation, bleiben dann die besten $\lambda \cdot (1 - GG)$ Elemente aus der alten Generation erhalten.

Bei zu kleinen Werten von GG kann der gewünschte Effekt des Elitismus aber auch nachteilige Wirkungen haben. Durch die Erhaltung der besten $\lambda \cdot (1 - GG)$ Elemente kann es zu einem Dominanzeffekt kommen. Eine zu homogene Population und damit suboptimale Lösungen können die Folge sein.

Zum dargestellten Verhalten von *Generational replacement* und Elitismus existieren auch Beweise aus der Theorie der genetischen Algorithmen [36].

Satz 5 *Der einfache genetische Algorithmus mit Mutation, Crossover und Fitneß-proportionaler Selektion konvergiert nicht.*

Satz 6 *Ein genetischer Algorithmus mit Elitismus konvergiert.*

Die Beweise dieser Sätze sind in [36] angeben.

4.4 Ein generisches Modell für evolutionäre Algorithmen

In der Literatur zu genetischen und evolutionären Algorithmen ist keine einheitliche Darstellung für die entworfenen Algorithmen, Operatoren und Verfahren anzutreffen. Eine formale Darstellung hat sich noch nicht gegenüber Pseudokode oder verbalen Beschreibungen durchgesetzt.

An dieser Stelle soll ein *generisches Modell* für evolutionäre Algorithmen eingeführt werden. Sämtliche in dieser Arbeit entworfenen Algorithmen lassen sich als Instanzen dieses Modells darstellen. Durch dieses Vorgehen soll insbesondere ein Vergleich der einzelnen Algorithmen erleichert werden.

GenEA()
$t \leftarrow 0$
$\mathcal{H} \leftarrow$ Initialize()
while $\neg$Done$(t, \mathcal{H})$
do $t \leftarrow t + 1$
    $\mathcal{H}' \leftarrow T_S(\mathcal{H})$
    $\mathcal{H}' \leftarrow T_X(\mathcal{H}')$
    $\mathcal{H}' \leftarrow T_M(\mathcal{H}')$
    $\mathcal{H} \leftarrow$ Replace$(\mathcal{H}, \mathcal{H}')$
return $\mathcal{H}$

Algorithmus 4.2: Das generische Modell für evolutionäre Algorithmen $GenEA$

Das generische Modell $GenEA$ beinhaltet

- die generischen Operatoren
  - T_S als Selektionsoperator
  - T_X als Crossover-Operator
  - T_M als Mutationsoperator
- die generischen Funktionen
  - Initialize zur Initialisierung der Anfangspopulation
  - Done zur Realisierung der Abbruchkriterien
  - Replace als Ersetzungsverfahren

Durch die Verwendung der generischen Funktion Replace ist es möglich, verschiedene Populationsmodelle zu beschreiben. Der bereits vorgestellte Algorithmus SGA kann beispielsweise mit Hilfe von $GenEA$ wie folgt definiert werden.

$$\begin{aligned} GenEA &\longrightarrow SGA \\ T_S &\longmapsto T_{RWS} \\ T_X &\longmapsto T_{1X} \\ T_M &\longmapsto T_{UM} \\ \text{Replace} &\longmapsto \text{Replace-G} \end{aligned}$$

Dabei wird das *Generational replacement* druch die Funktion REPLACE-G realisiert. Die Elterngeneration $\mathcal{H}$ wird vollständig durch ihre Nachkommen ersetzt. Folglich kann REPLACE-G$(\mathcal{H}, \mathcal{H}')$ definiert werden als **return** $\mathcal{H}'$.

Die Funktionen INITIALIZE und DONE werden, wenn nichts anderes angegeben ist, immer so verwendet, wie sie für SGA beschrieben wurden.

4.5 Genetische Algorithmen mit Elitismus

Wie im vorangegangenen Abschnitt festgestellt wurde, ist der einfache genetische Algorithmus SGA nicht zur Lösung des $3SAT$-Problems geeignet. Für keine der 100 Formeln aus der Formelklasse $(forced, 100)$ konnte innerhalb von 1500 Generationen eine erfüllende Belegung gefunden werden. Die Hauptursache für dieses Verhalten besteht in der fehlenden Konvergenz des Verfahrens. Dieser Nachteil kann durch die Verwendung eines Populationsmodells mit Elitismus aufgehoben werden. Als erste Variante mit Elitismus soll ein genetischer Algorithmus mit *Two-point crossover* und *Stochastic universal sampling* untersucht werden.

Der genetische Algorithmus EGA beinhaltet zwei neue genetische Operatoren, T_{SUS} und T_{2X}, sowie ein neues Ersetzungsschema, REPLACE-E.

Roulette-wheel selection (RWS), wie es bei SGA verwendet wurde, ist das Standard-Selektionsverfahren bei den genetischen Algorithmen. Es wählt die Kandidaten für die Erzeugung der Nachkommen mit einer Wahrscheinlichkeit p_s aus, die proportional zu ihrer Fitneß ist. Eine neue, verbesserte Variante dieses Verfahrens wurde als *Stochastic universal sampling* (SUS) bekannt.

Stochastic universal sampling T_{SUS}. Zur Berechnung des Operators $T_{SUS}(\mathcal{H})$ werden $\lambda \cdot GG$ Elemente aus $\mathcal{H}$ ausgewählt. Jedes Element $x \in \mathcal{H}$ wird dabei genau zwischen $\lfloor \lambda \cdot GG \cdot p_s(x) \rfloor$ und $\lceil \lambda \cdot GG \cdot p_s(x) \rceil$ Male ausgewählt. Die unteren bzw. oberen Klammern bezeichnen die Funktion für das Ab- bzw. Aufrunden zur nächsten ganzen Zahl. Die Selektionswahrscheinlichkeit $p_s(x)$ ist wie für T_{RWS} definiert.

EGA()
$t \leftarrow 0$
$\mathcal{H} \leftarrow$ Initialize()
while $\neg$Done$(t, \mathcal{H})$
do $t \leftarrow t + 1$
    $\mathcal{H}' \leftarrow T_{SUS}(\mathcal{H})$
    $\mathcal{H}' \leftarrow T_{2X}(\mathcal{H}')$
    $\mathcal{H}' \leftarrow T_{UM}(\mathcal{H}')$
    $\mathcal{H} \leftarrow$ Replace-E$(\mathcal{H}, \mathcal{H}')$
return $\mathcal{H}$

Algorithmus 4.3: Der genetische Algorithmus *EGA* mit Elitismus und Two-point crossover

SUS besitzt gegenüber *RWS* die folgenden Vorteile:

- Die Fitneß-proportionale Auswahl ist gewährleistet. Sie geschieht für ein einzelnes Individuum mit der gleichen Wahrscheinlichkeit p_s wie bei *RWS*.
- Es wird sichergestellt, daß die besten Individuen einer Generation mindestens $\lfloor \lambda \cdot GG \cdot p_s \rfloor$ Male zur Erzeugung von Nachkommen herangezogen werden.
- Das Verfahren kann äußerst effizient realisiert werden.

Zur Veranschaulichung von *SUS* stelle man sich wieder ein Roulette-Rad vor, auf dem genau λ Felder markiert sind. Für jedes Element $x \in \mathcal{H}$ ist ein Feld vorhanden, dessen Breite $p_s(x)$ beträgt. Der Umfang des Rades beträgt $\sum_{x \in \mathcal{H}} p_s(x) = 1$. Anstatt wie bei *RWS* das Roulette-Rad λ Male mit einem Zeiger zu drehen, wird es bei *SUS* nur einmal gedreht. Dabei befinden sich λ abstandsgleiche Zeiger auf dem äußeren Rand der Rades.

Der traditionelle genetische Algorithmus benutzt *One-point crossover*. Dabei werden die Chromosomensätze an einer zufällig gewählten Position getrennt und die dadurch entstehenden Endsegmente vertauscht. Dieses Vorgehen stimmt mit dem biologischen Vorbild überein, in der Natur tritt nur *One-point crossover* auf. Das verwendete Crossover-Verfahren hat einen entscheidenden Einfluß auf die Exploitationseigenschaften des Algorithmus. Aus diesem Grund stehen die Crossover-Verfahren im Mittelpunkt des Interesses. Schon sehr früh hat man untersucht, ob sogenannte *Multi-point crossover*-Verfahren eine Effizienzsteigerung bei den genetischen Algorithmen bewirken können [12, 21]. Tatsächlich hat man in empirischen Untersuchungen festgestellt, daß in den meisten Fällen *Two-point crossover* eine Verbesserung bewirkt. Die Verwendung von mehr als zwei Crossover-Punkten ist dagegen im allgemeinen nicht sinnvoll. An dieser Stelle soll darauf hingewiesen werden, daß es keine allgemeingültigen theoretischen Erkenntnisse zur Wirkung der *Multi-point crossover*-Verfahren gibt. Aus diesem Grunde bleibt für den praktischen Anwendungsfall nur die Möglichkeit, verschiedene Verfahren zu untersuchen, und dasjenige Verfahren auszuwählen, mit dem die besten Ergebnisse erreicht werden.

Beim *Two-point crossover* (oder allgemein beim *Multi-point crossover*) stellt man sich den Chromosomensatz eines Individuums nicht als Kette sondern als Ring vor. Der Ring entsteht, wenn man Anfang und Ende des Chromomensatzes aneinanderlegt. Der Chromosomen-Ring wird dann an zwei (oder mehreren) Stellen auseinandergebrochen und die entstehenden Segmente mit dem jeweils anderen Individuum ausgetauscht. Das heißt, *Two-point crossover* erfüllt dieselbe Aufgabe wie *One-point crossover*, es wird genau ein Segment ausgetauscht, jedoch ist *Two-point crossover* allgemeiner. Ein Chromosom, das als Ring aufgefaßt wird, kann mehr *Building blocks* enthalten, weil diese auch über das Ende hinauslaufen können.

Das im Algorithmus *SGA* benutzte *One-Point crossover* läßt sich problemlos auf mehrere Crossover-Punkte erweitern.

Two-point crossover T_{2X}. Zur Berechnung von $T_{2X}(\mathcal{H})$ werden die folgenden Schritte $\lambda/2$ Male wiederholt. Wähle ein $x = x_1 x_2 \ldots x_N \in \mathcal{H}$ und ein $y = y_1 y_2 \ldots y_N \in \mathcal{H}$ zufällig und gleichverteilt aus. Wähle mit der Wahrscheinlichkeit p_c zwei zufällige Indizes i und j mit $1 \leq i < j \leq N + 1$, bilde $z_1 = x_1 \ldots x_{i-1} y_i \ldots y_{j-1} x_j \ldots x_N$ und $z_2 = y_1 \ldots y_{i-1} x_i \ldots x_{j-1} y_j \ldots y_N$ sowie füge z_1 und z_2 in $T_{2X}(\mathcal{H})$ ein. Anderenfalls, also mit der Wahrscheinlichkeit $1 - p_c$, füge x und y in $T_{2X}(\mathcal{H})$ ein.

Die Erhaltung einer bestimmten Anzahl der besten Elemente aus der Elterngeneration, das heißt die Elitismus-Eigenschaft, wird mit Hilfe der Funktion REPLACE-E realisiert.

Ersetzungschema mit Elitismus REPLACE-E. Durch die Funktion REPLACE-E$(\mathcal{H}, \mathcal{H}')$ werden genau die besten $\lambda \cdot (1 - GG)$ Elemente aus der Elterngeneration $\mathcal{H}$ ausgewählt. Diese Elemente

EGA mit $\lambda = 200$, $GG = 0.9$ und $p_m = 0.009$			
Crossover-Wahrscheinlichkeit p_c	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
0.55	200	29.0	432
0.60	200	26.0	423
0.65	200	29.5	399
0.70	200	29.5	392
0.75	200	27.0	330
0.80	200	28.0	360
0.85	200	30.0	349
0.90	200	27.5	293
0.95	200	34.0	377
1.00	200	28.0	357

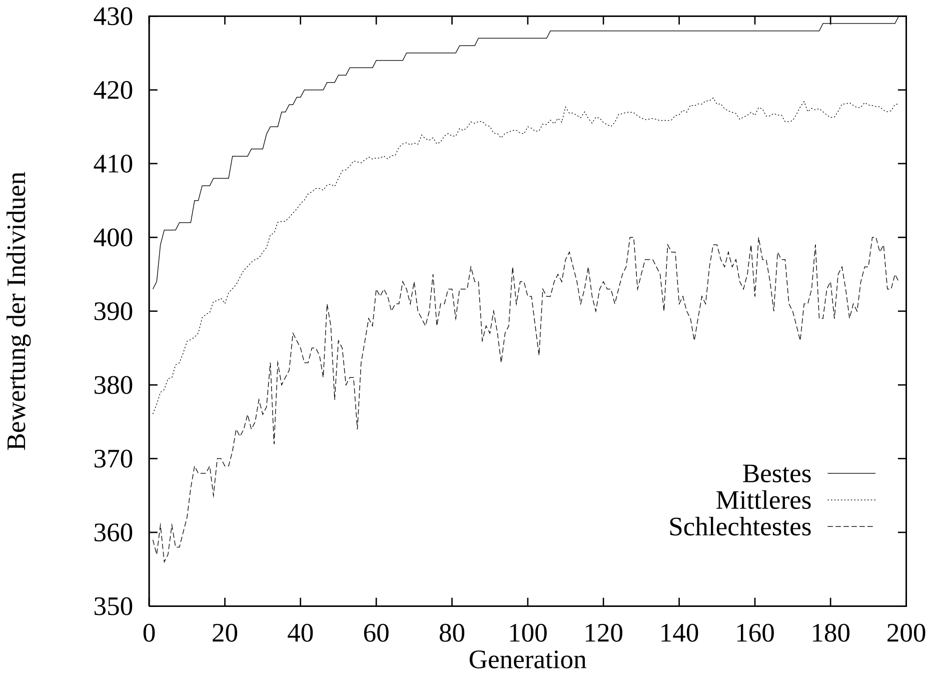
Tabelle 4.2: Testergebnisse zum Verhalten des genetischen Algorithmus mit Elitismus und *Two-point-crossover* in Abhängigkeit der Crossover-Wahrscheinlichkeit

werden dann gemeinsam mit den Elementen aus $\mathcal{H}'$ als Funktionswert zurückgegeben.

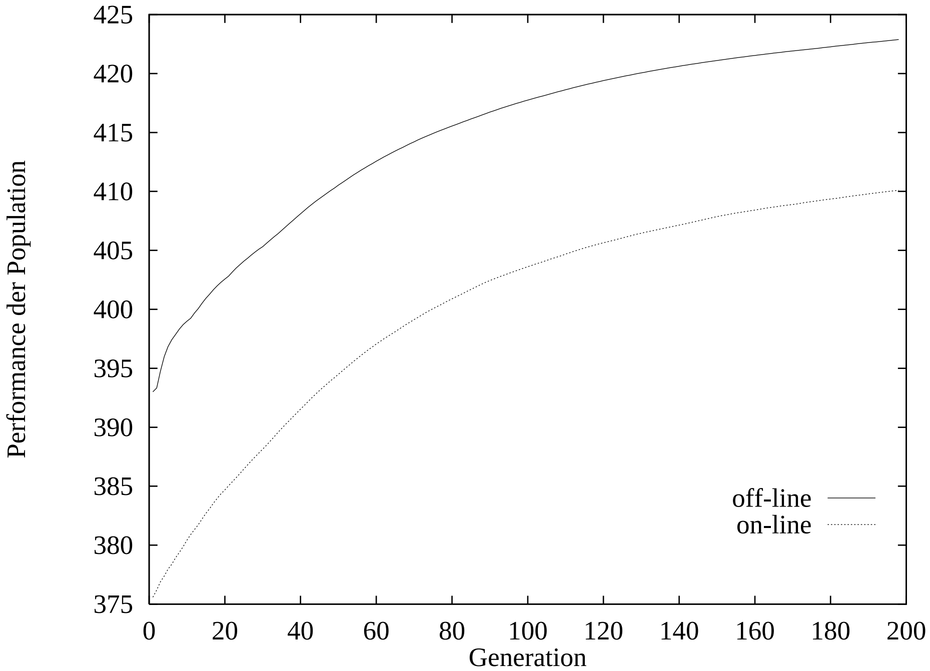
Wie man in den Abbildungen 4.2, 4.3 und 4.4 sehen kann, wurde das gewünschte Konvergenzverhalten erreicht. Die Population konvergiert in Richtung einer optimalen Lösung, und ein Modell wird gefunden. Trotzdem kann man nicht davon ausgehen, daß in jedem Fall eine Lösung gefunden wird. In Abbildung 4.5 ist ein solcher erfolgloser Testlauf dokumentiert. Die Population konvergiert nur in eine suboptimale Lösung.

Der nächste Schritt bei der Verbesserung des Algorithmus ist die Bestimmung der günstigsten Crossover- und Mutationswahrscheinlichkeiten. Bei den ersten Versuchen wurden $p_c = 0.9$ und $p_m = 0.01$ willkürlich gewählt.

Eine günstige (oder optimale) Crossover-Wahrscheinlichkeit kann nur empirisch bestimmt werden. Sie ist von der verwendeten Kodierung, der Fitneß- und Bewertungsfunktion und nicht zuletzt vom verwendeten Mutationsoperator abhängig. Unter der Voraussetzung, daß genügend Testläufe durchgeführt werden, würde man erwarten, daß eine optimaler Wert für p_c gefunden werden kann. Optimal heißt in diesem Fall, daß für einen bestimmten Wert die Erfolgsrate des Algorithmus maximal wird und daß für Werte größer



(a) Verlauf der Individuen-Bewertungen über den Generationen



(b) Verlauf der Performance über den Generationen

Abbildung 4.2: Eine Anwendung des genetischen Algorithmus EGA mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.9$ und $p_m = 0.01$ auf die Formel $(forced, 100, 10)$

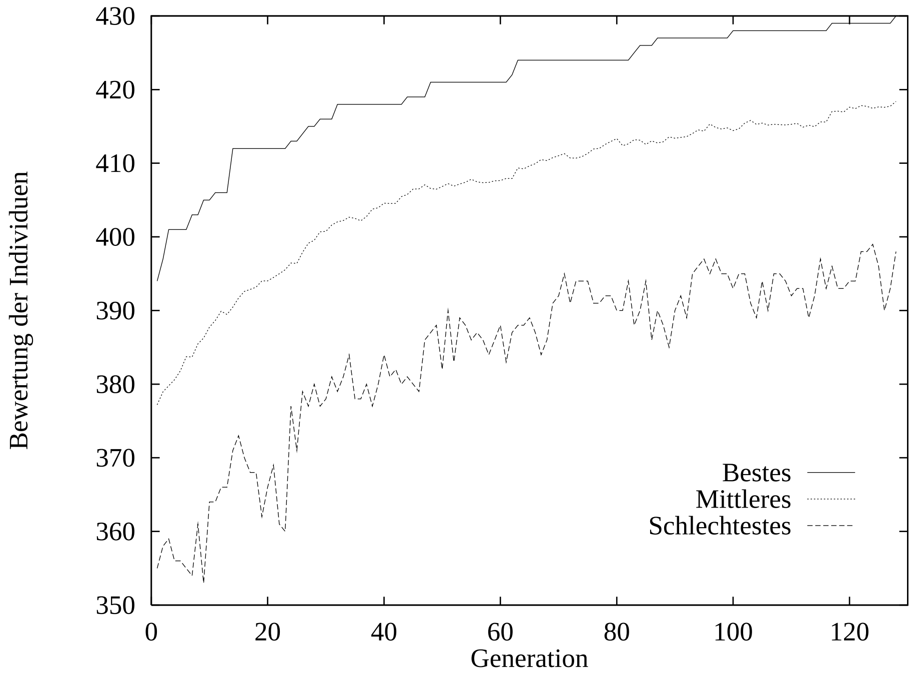


Abbildung 4.3: Ein Beispiel für schnelle Konvergenz bei der Anwendung des genetischen Algorithmus EGA mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.9$ und $p_m = 0.01$ auf die Formel $(forced, 100, 10)$

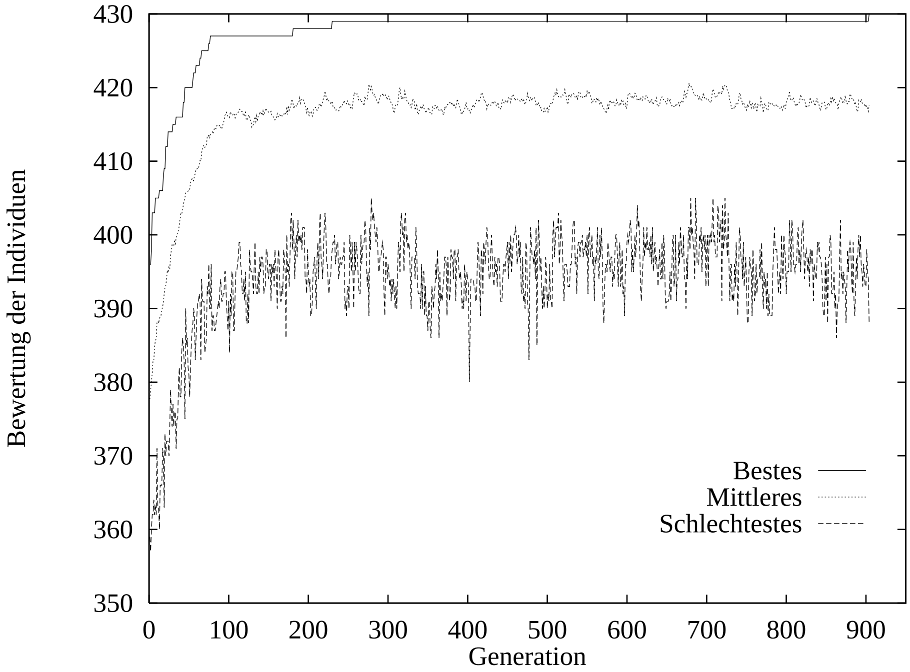


Abbildung 4.4: Ein Beispiel für langsame Konvergenz bei der Anwendung des genetischen Algorithmus EGA mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.9$ und $p_m = 0.01$ auf die Formel $(forced, 100, 10)$

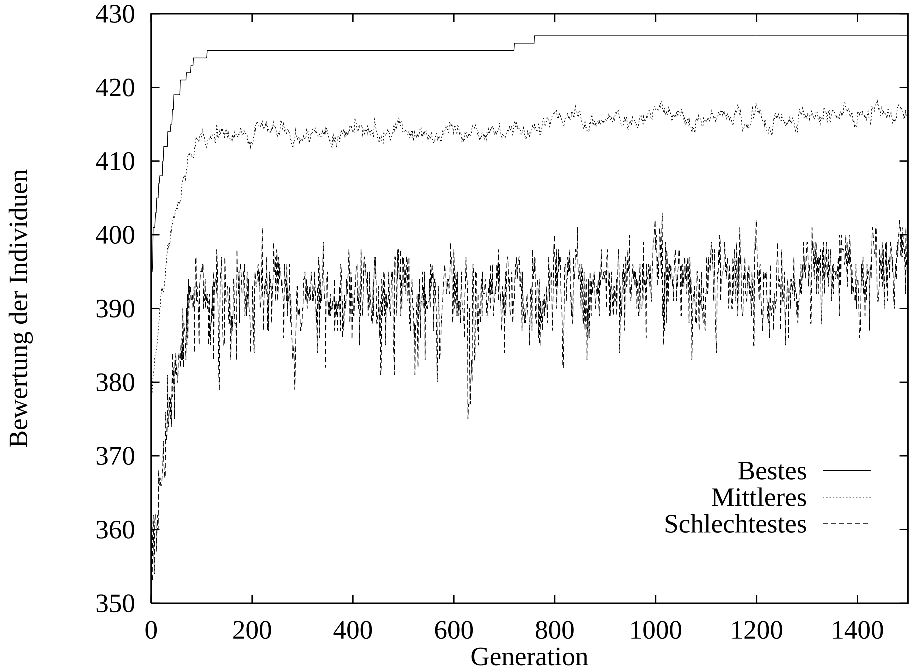


Abbildung 4.5: Ein Beispiel für erfolglose Suche bei der Anwendung des genetischen Algorithmus EGA mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.9$ und $p_m = 0.01$ auf die Formel $(forced, 100, 10)$

oder kleiner als dieses Optimum schlechtere Ergebnisse erzielt werden. Diese Erwartungen lassen sich mit dem intiutiven Verständnis des Crossover-Verfahrens begründen. Bei einer zu kleinen Crossover-Wahrscheinlichkeit sind die Exploitationseigenschaften des Algorithmus stark eingeschränkt. Ist sie wiederum zu groß, werden die guten Lösungen zerstört.

Die Testergebnisse zur Bestimmung einer optimalen Crossover-Wahrscheinlichkeit für den Algorithmus EGA sind in Tabelle 4.2 zusammengefaßt. Die erzielten Ergebnisse stimmen leider nicht mit den beschriebenen Erwartungen überein. Es ergibt sich kein klares Bild über die Abhängigkeit der Erfolgsrate von der verwendeten Crossover-Wahrscheinlichkeit. Die besten Ergebnisse wurden mit $p_c = 0.95$ erzielt. Über die Ursachen für diesen Verhalten können nur Vermutungen angestellt werden:

- Die probabilistischen Faktoren können einen starken Einfluß auf die Ergebnisse haben.
- Die gegensätzlichen Wirkungen des Operators (Exploitation versus Überlebenschancen der Lösungen) können lokale Maxima der Erfolgsrate hervorrufen.
- Es kann zu einer starken Wechselwirkung zwischen dem Crossover- und dem Mutationsoperator kommen.

Neben der Erfolgsrate ist in Tabelle 4.2 noch eine weitere Testgröße angegeben: die mittlere Anzahl von Generationen, die zum Finden eines Modells benötigt wurde. Man kann bei Betrachtung dieser Größe deutlich erkennen, daß mit ansteigender Crossover-Wahrscheinlichkeit auch die Konvergenzgeschwindigkeit zunimmt. Dieses Verhalten ist plausibel.

Es wurden auch empirische Untersuchungen zur Bestimmung einer optimalen Mutationswahrscheinlichkeit unternommen. Über das Verhalten des Algorithmus in Abhängigkeit von der Mutationswahrscheinlichkeit kann man ähnliche Vermutungen wie bei der Crossover-Wahrscheinlichkeit anstellen. Man würde annehmen, daß es einen optimalen Wert für p_m gibt. Der Mutationsoperator hat einen entscheidenden Einfluß auf die Qualität der lokalen Suche. Wählt man die Mutationswahrscheinlichkeit zu klein, ist die lokale Suche ineffizient. Auf der anderen Seite ist man aber auch daran interessiert, nur punktuelle Veränderungen an den Chromosomen vorzunehmen, damit die guten Eigenschaften der Lösungen erhalten bleiben. Aus diesem Grund darf die Mutationswahrscheinlichkeit nicht zu groß gewählt werden. Wie man in Tabelle 4.3 sehen kann, ist die Bestimmung geeigneter Werte für p_m

EGA mit $\lambda = 200$, $GG = 0.9$ und $p_c = 0.9$			
Mutations-wahrscheinlichkeit p_m	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
0.001	100	20.0	285
0.002	100	25.0	256
0.003	100	28.0	383
0.004	100	29.0	380
0.005	200	35.0	298
0.006	200	28.5	338
0.007	200	33.5	383
0.008	200	30.0	344
0.009	200	39.0	349
0.010	200	27.5	367

Tabelle 4.3: Testergebnisse zum Verhalten des genetischen Algorithmus mit Elitismus und *Two-point-crossover* in Abhängigkeit der Mutationswahrscheinlichkeit

keine triviale Aufgabe. Es ergibt sich kein klares Bild über die Abhängigkeit von p_m und Erfolgsrate. Das Optimum wurde im Bereich $p_m \in [0.005, 0.01]$ vermutet. Daraufhin wurde für diese Werte der Testlauf wiederholt, um genauere Werte zu erhalten. Als Optimum ergab sich dann $p_m = 0.009$.

An dieser Stelle soll noch einmal auf die Selektionsverfahren RWS und SUS zurückgekommen werden. Die genannten Vorteile von SUS können auch experimentell bestätigt werden. Die entsprechenden Ergebnisse sind in Tabelle 4.4 dargestellt. Die Erfolgsrate bei Verwendung von SUS ist erheblich größer.

Den Crossover-Verfahren wird bei der Verbesserung von genetischen Algorithmen eine große Bedeutung zugeschrieben. Der Sinn der Crossover-Verfahren ist es, den Suchraum effizient zu durchschreiten. Die Varianten *One-point* und *Two-point crossover* wurden bereits diskutiert. Will man sicherstellen, daß jeder Punkt des N-dimensionalen Suchraumes konstruierbar ist, so kann man auch das sogenannte *Uniform crossover* verwenden.

Uniform crossover T_{UX}. Zur Berechnung von $T_{UX}(\mathcal{H})$ werden die folgenden Schritte $\lambda/2$ Male wiederholt. Wähle ein $x = x_1x_2 \ldots x_N \in \mathcal{H}$ und ein $y = y_1y_2 \ldots y_N \in \mathcal{H}$ zufällig und gleichverteilt aus. Generiere eine zufällige *Crossover-Maske* $m = m_1m_2 \ldots m_N \in \{0,1\}^N$. Erzeuge mit

EGA mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.95$ und $p_m = 0.009$			
Selektion	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
RWS	200	25.0	354
SUS	200	34.0	377

Tabelle 4.4: Testergebnisse zum Vergleich von *Roulette-wheel selection* (*RWS*) und *Stochastic universal sampling* (*SUS*)

der Wahrscheinlichkeit p_c ein Element z entsprechend der folgenden Regel und füge z in $T_{UX}(\mathcal{H})$ ein:

$$z_i = \begin{cases} x_i & \text{falls } m_i = 1 \\ y_i & \text{falls } m_i = 0 \end{cases} \quad \text{mit } 1 \leq i \leq N.$$

Anderenfalls, also mit der Wahrscheinlichkeit $1 - p_c$, füge x in $T_{UX}(\mathcal{H})$ ein. Anschließend werden x und y vertauscht, und wie oben angegeben wird ein weiteres Element für $T_{UX}(\mathcal{H})$ erzeugt.

Obwohl zahlreiche Untersuchungen zum Vergleich von *Two-point* und *Uniform crossover* angestellt wurden, konnte bisher nicht geklärt werden, welches Verfahren im allgemeinen Fall besser geeignet ist. Es gibt Anwendungen, in denen das eine Verfahren besser ist, und wiederum andere Fälle, in denen mit dem zweiten Verfahren bessere Ergebnisse erreicht werden. Mit Hilfe des Schema-Begriffs lassen sich Unterschiede zwischen beiden Crossover-Verfahren beschreiben. Bei *Uniform crossover* werden Schemata einer bestimmten Ordnung mit der gleichen Wahrscheinlichkeit zerbrochen, unabhängig von ihrer definierenden Länge. Beim *Two-point crossover* bestimmt die definierende Länge die Wahrscheinlichkeit der Zerstörung, nicht die Ordnung [6].

Für das $3SAT$-Problem sollen beide Crossover-Verfahren untersucht werden. Aus diesem Grund wurde der Algorithmus *EGA-UX* mit Elitismus und *Uniform crossover* implementiert und getestet.

Die Testergebnisse zur Bestimmung einer optimalen Crossover-Wahrscheinlichkeit für den Algorithmus *EGA-UX* sind in Tabelle 4.5

EGA-UX()

$t \leftarrow 0$

$\mathcal{H} \leftarrow$ INITIALIZE()

while $\neg$DONE$(t, \mathcal{H})$

do $t \leftarrow t + 1$

    $\mathcal{H}' \leftarrow T_{SUS}(\mathcal{H})$

    $\mathcal{H}' \leftarrow T_{UX}(\mathcal{H}')$

    $\mathcal{H}' \leftarrow T_{UM}(\mathcal{H}')$

    $\mathcal{H} \leftarrow$ REPLACE-E$(\mathcal{H}, \mathcal{H}')$

return $\mathcal{H}$

Algorithmus 4.4: Der genetische Algorithmus *EGA-UX* mit Elitismus und *Uniform crossover*

EGA-UX mit $\lambda = 200$, $GG = 0.9$ und $p_m = 0.009$			
Crossover-Wahrscheinlichkeit p_c	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
0.50	200	47.0	273
0.55	200	48.5	261
0.60	200	45.0	261
0.65	200	43.0	252
0.70	200	44.0	275
0.75	200	38.0	291
0.80	200	37.5	258
0.85	200	31.0	287
0.90	200	24.0	389
0.95	200	27.0	303
1.00	200	27.0	415

Tabelle 4.5: Testergebnisse zum Verhalten des genetischen Algorithmus mit Elitismus und *Uniform crossover* in Abhängigkeit der Crossover-Wahrscheinlichkeit

EGA-UX mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.55$ und $p_m = 0.009$			
Generation gap GG	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
0.1	200	43.0	415
0.2	200	46.0	274
0.3	200	53.0	229
0.4	200	59.5	177
0.5	200	53.5	177
0.6	200	58.0	152
0.7	200	52.5	160
0.8	200	57.0	212
0.9	200	39.0	277
1.0	200	0.0	—

Tabelle 4.6: Testergebnisse zum Verhalten des genetischen Algorithmus mit Elitismus und *Uniform crossover* in Abhängigkeit von *Generation gap*

zusammengefaßt. Hinsichtlich der Maximierung der Erfolgsrate beim Finden eines Modells ergibt sich $p_c = 0.55$ als optimale Einstellung. Damit konnte gezeigt werden, daß für das $3SAT$-Problem *Uniform crossover* günstiger geeignet ist. Der Algorithmus *EGA-UX* soll daher als Grundlage für die weiteren Untersuchungen dienen.

Der nächste Schritt bei der Verbesserung von *EGA-UX* ist die Untersuchung des Einflusses des Parameters *Generation Gap* (GG). Dieser Parameter beschreibt den Grad von Elitismus des Algorithmus. In jedem Generationsschritt werden $\lambda \cdot GG$ Individuen selektiert und entsprechend viele Nachkommen erzeugt. Beim Einfügen der neuen Individuen in die nächste Generation bleiben dann die besten $\lambda \cdot (1 - GG)$ Elemente von der alten Generation erhalten. Bei $GG = 1.0$ wird die Elterngeneration vollständig durch die Nachkommen ersetzt (*Generational replacement*). Der Fall $GG = 0$ ist nicht von praktischem Interesse, da hier keine neuen Individuen erzeugt werden. Je größer GG gewählt wird, umso größer ist der Elitismus und umso mehr wird die Breite der Suche eingeschränkt. Angesicht dieser Tatsache wird man erwarten, daß man einen optimalen Wert für GG finden kann, der einerseits die Erfolgsrate maximiert (Breite der Suche) und andererseits gute Konvergenzeigenschaften (Elitismus) zur Folge hat. Diese Vermutung kann experimentell bestätigt werden. Die entsprechenden Testergebnisse sind in Tabelle 4.6 dargestellt. Die besten Ergebnisse konnten mit $0.4 \leq GG \leq 0.6$

EGA-UX mit $\lambda = 200$, $GG = 0.4$, $p_c = 0.55$ und $p_m = 0.009$			
Fitneß-Skalierung	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
keine	200	59.5	177
linear	200	36.5	192

Tabelle 4.7: Testergebnisse zur Wirkung von *Linear fitness scaling* beim genetischen Algorithmus mit Elitismus und *uniform crossover*

erzielt werden. Diese Werte sind im Vergleich zu den üblicherweise verwendeten ungewöhnlich klein. Solch kleine Werte bewirken eine Dominanz der besten Individuen in der Population. Daduch wird aber auch die Breite der Suche stark eingeschränkt, und die Population konvergiert stark.

Kurz nach dem Start eines genetischen Algorithmus ist häufig die folgende Situation anzutreffen: In der Population sind nur einige wenige besonders gute Individuen enthalten, alle anderen besitzen nur mittelmäßige Bewertungen. Verfährt man in diesem Fall nach dem gewohnten Fitneß-proportionalen Selektionsverfahren, dann kann es schon in der Folgegeneration zu einem Dominanzeffekt dieser wenigen guten Individuen kommen. Das ist aber in einem solch frühen Stadium der Evolution nicht erwünscht. Die Breite der Suche würde dadurch zu stark eingeschränkt.

Nach einer gewissen Laufzeit des Algorithmus kann hingegen ein anderes Problem entstehen. Obwohl noch eine ausreichende Vielfalt in der Population vorhanden ist, unterscheidet sich häufig die beste Bewertung der Population nur noch kaum von der mittleren. In dieser Situation würden die mittelmäßigen Individuen im Auswahlprozeß dominieren und der Evolutionsprozeß zur zufälligen Suche in den mittelmäßigen Lösungen degenerieren.

Zur Vermeidung dieser beiden Probleme wird häufig das sogenannte *Linear fitness scaling* (*LS*) eingesetzt. Dabei wird eine lineare Fitneßfunktion über den Bewertungen der Individuen verwendet [12].

$$\begin{aligned}
f_{fit}(x) &\stackrel{def}{=} a \cdot f_{obj}(x) + b \qquad \text{mit} \\
\\
max &= \max\{f_{obj}(x) \mid x \in \mathcal{H}\} \\
min &= \min\{f_{obj}(x) \mid x \in \mathcal{H}\} \\
avg &= \frac{max + min}{2} \\
\Delta &= max - avg \\
a &= \frac{avg}{\Delta} \\
b &= \frac{avg \cdot (max - 2 \cdot avg)}{\Delta}
\end{aligned}$$

In den Algorithmen ohne Fitneß-Skalierung werden die Bewertung und die Fitneß eines Individuums als gleich definiert.

Da der Einsatz von *Linear fitness scaling* eine Verbessung der genetischen Algorithmen bewirken kann, wurde untersucht, ob das auch auf den Algorithmus *EGA-UX* zutrifft. Die Ergebnisse der Untersuchung sind in Tabelle 4.7 zusammengefaßt. Leider konnte durch die Verwendung der Fitneß-Skalierung keine Verbesserung in der Erfolgsrate erreicht werden.

Die Verwendung von *Linear fitness scaling* sollte einerseits eine frühe Dominanz von besonders guten Indiviuden verhindern helfen und andererseits einen gesunden Wettstreit unter ähnlich guten Individuen in den späteren Phasen ermöglichen. Dadurch sollte die Suche möglicher Lösungen möglichst breit und effizient gestaltet werden. Wie man an den Ergebnissen sehen kann, wurde dieses Ziel auch teilweise erreicht: Die Population konvergiert mit Fitneß-Skalierung langsamer. Im Falle des SAT-Problems hat das aber nicht zu einer Verbesserung der Erfolgsrate geführt, im Gegenteil zu einer merklichen Verschlechterung. Es stellt sich die Frage, welche Eigenschaft der Skalierung (die Wirkung in der frühen oder späten Phase) die Verschlechterung der Erfolgsrate zur Folge hat. Leider kann man angesichts des Experiments dazu keine Aussagen machen. Eine weitere Ursache für das Scheitern der Fitneß-Skalierung kann auch der starke Elitismus ($GG = 0.4$) sein. Um befriedigende Aussagen zu diesen Problemen zu finden, sind eine Reihe weitere Experimente nötig, die aber den Rahmen dieser Arbeit übersteigen.

4.6 Das Crowding factor-Modell

Beim einfachen genetischen Algorithmus SGA wird die aktuelle Population vollständig durch ihre Nachkommen ersetzt. Algorithmen mit einem ausgeprägten Elitismus hingegen erzeugen pro Generation nur einige wenige Nachkommen und nehmen sie dann in die neue Population auf. Im allgemeinen ist man jedoch an einem ausgewogenen Verhältnis zwischen Vielfalt und Elitismus interessiert. Es soll möglichst lange eine große Vielfalt der Population erhalten bleiben, aber gleichzeitig die Konvergenzeigenschaft nicht verloren gehen. Diese Erkenntnisse waren der Antrieb für die Entwicklung sogenannter *crowding*-Methoden. Dabei wird versucht, die Vielfalt der Population zu erhalten, indem beim Einfügen der Nachkommen nur die ähnlichen Individuen aus der Elterngeneration ersetzt werden.

Die Entwicklung der *crowding*-Methoden ist motiviert durch ein ökologisches Phänomen: den Wettbewerb unter ähnlichen Mitgliedern einer natürlichen Population um begrenzte Ressourcen. Wenn ähnliche Individuen einer Population (meistens Individuen derselben Art) anfangen in einem bestimmten Lebensraum zu dominieren, dann bewirkt der verstärkte Wettbewerb um die begrenzten Ressourcen eine Verringerung der Lebenserwartung und der Geburtsrate. In weniger dicht besiedelten Lebensräumen ist die Konkurrenz unter den Individiuen weniger stark, Lebenserwartung und Geburtsrate richten sich dann in erster Linie nach den Fähigkeiten der Individuen.

Ein typischer Vertreter der *crowding*-Methoden ist das *Crowding factor*-Modell [21]. Dieses Modell soll im folgenden kurz beschrieben werden.

Zuerst wird ein Teil der aktuellen Population durch Fitneß-proportionale Selektion zur Erzeugung von Nachkommen herangezogen. Dabei werden genau $\lambda \cdot GG$ neue Elemente durch Crossover und Mutation erzeugt. Die gleiche Anzahl von Individuen muß dann aus der aktuellen Population entfernt werden, damit die Nachkommen hinzugefügt werden können. Dazu werden für jedes neue Individuum CF Individuen aus der Elterngeneration zufällig ausgewählt. Die Anzahl CF wird *Crowding factor* genannt. Aus der Menge der CF Individuen wird dann dasjenige bestimmt, das dem neuen Individuum am ähnlichsten ist. Dieses bestimmte Individuum wird durch das neue ersetzt. Als Ähnlichkeitsmaß wird der Hamming-Abstand verwendet, das heißt, es wird ein bitweiser Vergleich der Individuen durchgeführt.

Der dargestellte Algorithmus CGA ist eine Implementation des *Crowding factor*-Modells für das $3SAT$-Problem.

Die Ersetzung der ähnlichen Individuen, das heißt die *crowding*-Eigenschaft, wird mit Hilfe der Funktion Replace-C realisiert.

CGA()
$t \leftarrow 0$
$\mathcal{H} \leftarrow$ Initialize()
while $\neg$Done$(t, \mathcal{H})$
do $t \leftarrow t + 1$
    $\mathcal{H}' \leftarrow T_{SUS}(\mathcal{H})$
    $\mathcal{H}' \leftarrow T_{UX}(\mathcal{H}')$
    $\mathcal{H}' \leftarrow T_{UM}(\mathcal{H}')$
    $\mathcal{H} \leftarrow$ Replace-C$(\mathcal{H}, \mathcal{H}')$
return $\mathcal{H}$

Algorithmus 4.5: Der genetische Algorithmus *CGA* mit *Crowding factor*-Modell

Ersetzungschema mit Crowding Replace-C. Erzeuge $\mathcal{H}''$ als eine zu $\mathcal{H}$ identische Multimenge. Die folgenden drei Schritte werden für jedes Element $x \in \mathcal{H}'$ ausgeführt. Wähle zufällig CF Individuen aus der Population $\mathcal{H}$ aus. Bestimme darunter das Individuum mit dem geringsten Hamming-Abstand bezüglich x, und entferne eben dieses Element aus $\mathcal{H}''$. Die Multimenge $\mathcal{H}' \cup \mathcal{H}''$ wird als Funktionswert zurückgegeben.

Der Ziel der *crowding*-Methoden ist es, die Vielfalt der Population so lange wie möglich zu erhalten. Genetische Algorithmen mit *Crowding* sollen in der Lage sein, mehrere optimale Lösungen zu finden und innerhalb einer Population zu erhalten. Gewöhnliche genetische Algorithmen werden dieser Forderung nicht gerecht, da die Vielfalt der Population zu schnell durch den Selektionsdruck, das probabilistische Rauschen bei der Selektion und durch zerstörende Operatoren beeinträchtigt wird.

Die genannten Eigenschaften machen die *crowding*-Methoden interessant für die Lösung des Erfüllbarkeitsproblems, insbesondere zur Lösung solcher Probleme, deren Fitneßfunktionslandschaft viele Extrema aufweist. Das vorgestellte *Crowding factor*-Modell wurde speziell zur Optimierung von multimodalen Funktionen entwickelt.

Wie man an den Untersuchungsergebnissen in Tabelle 4.8 sehen kann, stellt *CGA* gegenüber *EGA-UX* keine Verbesserung dar. Die beste Erfolgsrate wurde mit $GG = 0.5$ und $CF = 2$ erreicht. Die Ergebnisse konnten auch

CGA mit $\lambda = 200$, $p_c = 0.55$ und $p_m = 0.009$				
Crowding factor CF	*Generation gap* GG	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
2	0.1	200	26.0	451
	0.2	200	33.0	308
	0.3	200	30.5	207
	0.4	200	31.5	181
	0.5	200	33.0	171
3	0.1	200	14.0	480
	0.2	200	22.0	304
	0.3	200	18.0	200
	0.4	200	26.5	162
	0.5	200	26.5	179
4	0.1	200	13.5	474
	0.2	200	16.5	341
	0.3	200	24.0	292
	0.4	200	22.5	226
	0.5	200	25.0	228

Tabelle 4.8: Testergebnisse zum Verhalten des *Crowding factor*-Modells in Abhängigkeit von *Crowding factor* und *Generation gap*

CGA mit $\lambda = 200$, $GG = 0.2$, $CF = 2$, $p_c = 0.55$ und $p_m = 0.009$			
Fitneß-Skalierung	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
keine	200	33.0	308
linear	200	15.5	332

Tabelle 4.9: Testergebnisse zur Wirkung von *Linear fitness scaling* im *Crowding factor*-Modell

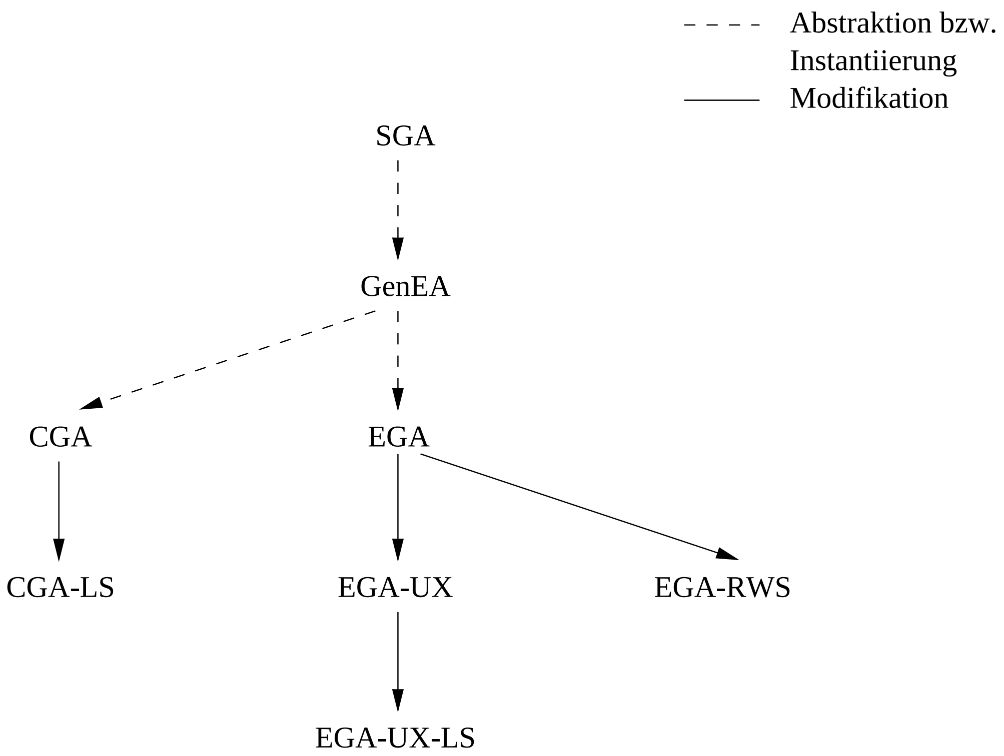


Abbildung 4.6: Die genetischen Algorithmen für das $3SAT$-Problen

durch die Verwendung von *Linear fitness scaling* nicht verbessert werden. Die zugehörigen Ergebnisse sind in Tabelle 4.9 dargestellt.

An dieser Stelle soll darauf hingewiesen werden, daß es eine Reihe ähnlicher Modelle gibt [27]. Die Entwicklung von *crowding*-Methoden ist Gegenstand der aktuellen Forschung. Es kann nicht ausgeschlossen werden, daß andere Methoden erfolgreicher sind. Weitere Untersuchungen zu dieser Problematik übersteigen den Rahmen dieser Arbeit.

4.7 Zusammenfassung der Ergebnisse

In diesem Kapitel wurden eine Reihe von genetischen Algorithmen vorgestellt. Die einzelnen Algorithmen unterscheiden sich in den verwendeten genetischen Operatoren und Populationsmodellen. Das Ziel bei den Entwürfen war es, ein evolutionäres Verfahren zu finden, welches ohne Verwendung problemspezifischer Operatoren das Erfüllbarkeitsproblem für große aussagenlogische Formeln mit einer möglichst hohen Erfolgsrate löst. Alle entwor-

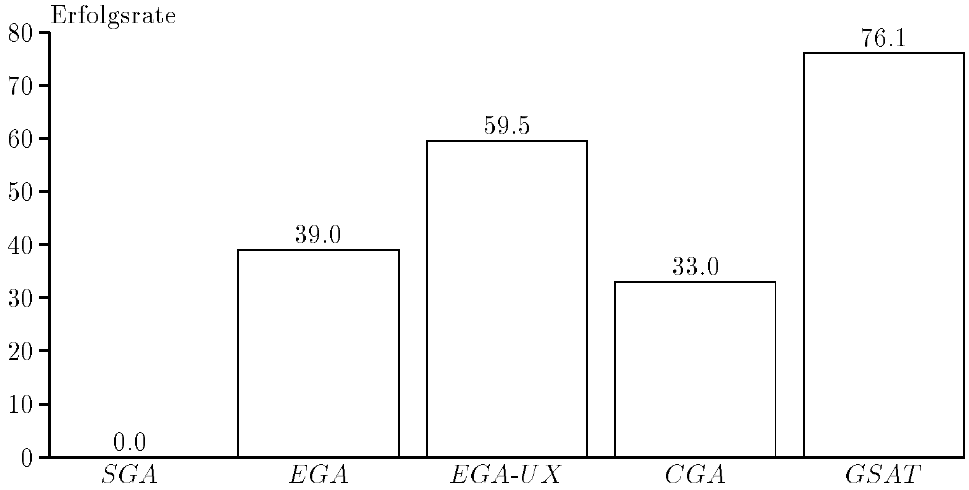


Abbildung 4.7: Der Vergleich von SAT-Algorithmen bezüglich der Erfolgsrate beim Finden von Modellen für die Formelklasse $(forced, 100)$. Die Angabe für $GSAT$ bezieht sich dabei auf [18].

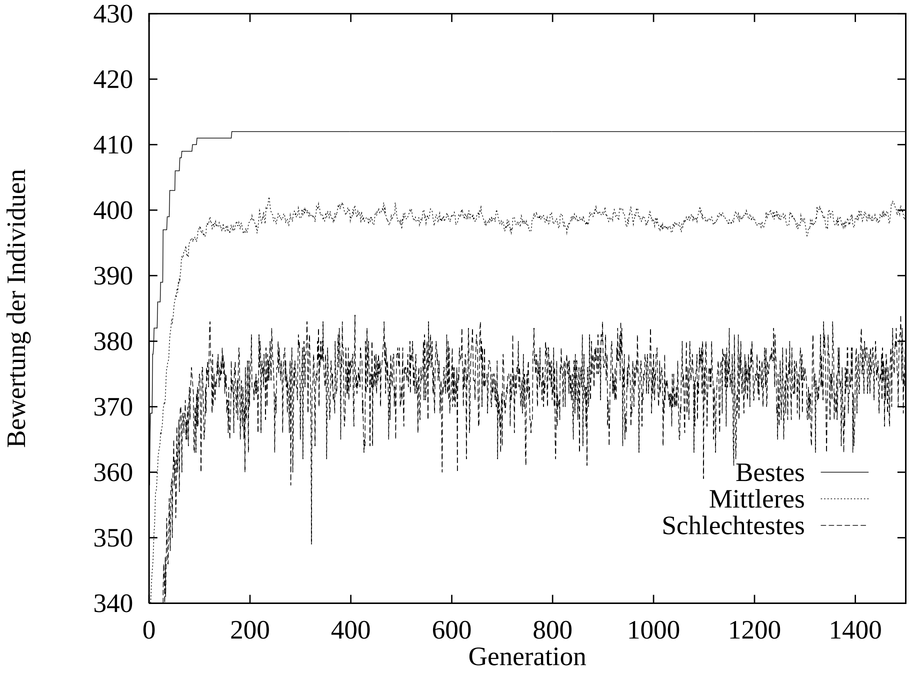
fen genetischen Algorithmen lassen sich als Instanz des generischen Modells $GenEA$ darstellen, siehe Abbildung 4.6.

Obwohl man zeigen konnte, daß durch die Wahl geeigneter Operatoren und Einstellungen die Erfolgsrate erheblich gesteigert werden kann, ist das Erreichte im Vergleich mit anderen SAT-Algorithmen, beispielsweise $GSAT$, nur wenig befriedigend. Die einzelnen Ergebnisse sind in Abbildung 4.7 zusammengefaßt. Die größte Erfolgsrate wurde von EGA-UX, einem genetischen Algorithmus mit Elitismus, *Stochastic universal sampling* und *Uniform crossover*, erreicht. Die Erhaltung der besten Individuen ist eine notwendige Voraussetzung für das Finden von Modellen. Der einfache genetische Algorithmus SGA, ohne diese Eigenschaft, findet daher überhaupt keine Modelle.

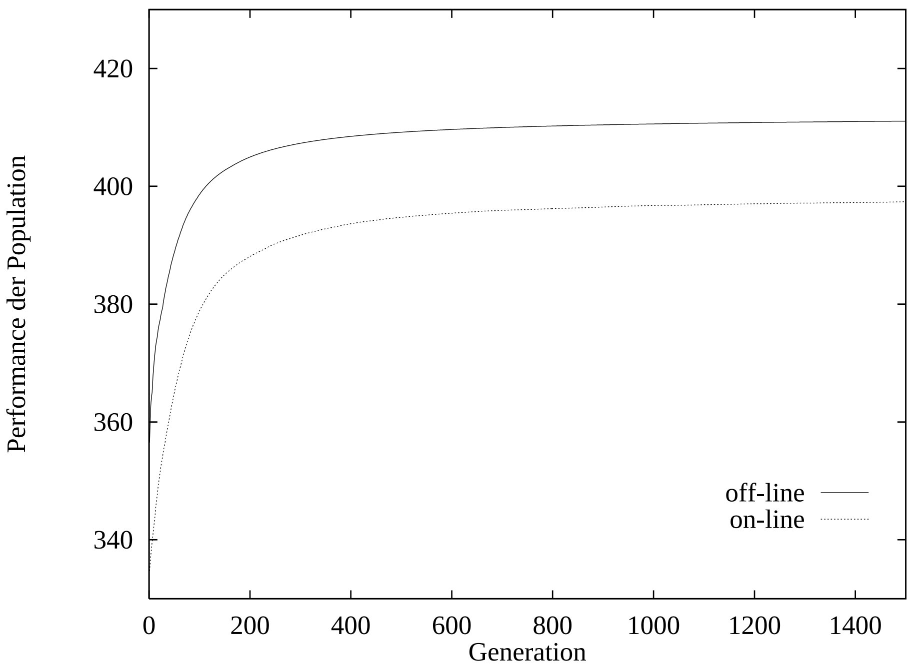
In den empirischen Untersuchungen für das $3SAT$-Problem wurde ermittelt, daß

- *Uniform crossover* den *One-* und *Two-point*-Varianten überlegen ist,
- mit dem *Crowding factor*-Modell gegenüber Elitismus keine Verbesserung bewirkt werden kann,
- die Verwendung von *Linear fitness scaling* den Evolutionsprozeß insgesamt nicht positiv beeinflußt hat.

Obwohl die einfachste Formelklasse, *forced*-Formeln mit 100 Variablen, verwendet wurde, liegt die größte erreichte Erfolgsrate nur bei 58 Prozent. Für



(a) Verlauf der Individuen-Bewertungen über den Generationen



(b) Verlauf der Performance über den Generationen

Abbildung 4.8: Eine Anwendung des genetischen Algorithmus *EGA-UX* mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.55$ und $p_m = 0.009$ auf die Formel $(unforced, 100, 39)$

unforced-Formeln sind die entworfenen genetischen Algorithmen sogar völlig ungeeignet. In keinem einzigen Testlauf konnte ein Modell gefunden werden. Ein typischer Testlauf für eine *unforced*-Formel ist in Abbildung 4.8 dargestellt. Die Population konvergiert dabei nur in eine suboptimale Lösung. Die gefundene Belegung liegt mit 412 erfüllten Klauseln noch weit hinter den geforderten 430 zurück.

Die Verfolgung der ersten Problemlösungsstrategie war damit nur wenig erfolgreich. Daraufhin sollen im nächsten Kapitel evolutionäre Algorithmen entworfen werden, deren genetische Operatoren problemspezifisches Wissen zum Finden der Modelle nutzen.

Kapitel 5

Evolutionäre Algorithmen für das 3SAT-Problem

5.1 Der MetaGen-Algorithmus

Den Gegenstand der weiteren Untersuchungen bilden evolutionäre Algorithmen mit problemspezifischen genetischen Operatoren. Als Ausgangspunkt dient dabei der *MetaGen*-Algorithmus (*Metalogical genetic algorithm*) von R. A. Young and A. Reel. Dieser Algorithmus wurde speziell zur Lösung des $3SAT$-Problems entwickelt und erreicht laut Angaben von Young et al. [48] bei Formeln mit 80 Variablen Erfolgsraten um 100 Prozent.

Der MetaGen-Algorithmus ist verhältnismäßig einfach. Er arbeitet über einer festen Anzahl von Individuen, die jeweils eine Belegung für die gegebene Formel repräsentieren. Als Bewertungsfunktion dient die Anzahl der durch die Belegung erfüllten Klauseln. Die Population wird am Anfang der Abarbeitung zufällig initialisiert. In jeder weiteren Generation werden für jedes Element der Population zwei neue Individuen erzeugt. Diese beiden neuen Individuen sind direkte Konkurrenten für das jeweilige Element der Elterngeneration. Der erste Konkurrent entsteht durch Mutation aus dem ursprünglichen Element und der zweite Konkurrent durch Rekombination (Crossover) von zwei zufällig aus der Elternpopulation gewählten Elementen.

Wird das Element aus der Elterngeneration durch einen seiner Konkurrenten in der Bewertung übertroffen, dann wird es durch diesen ersetzt. Dabei wird der Konkurrent mit der größten Bewertung von beiden benutzt. Sind die Bewertungen gleich, kommt der durch Mutation entstandene Konkurrent zum Zuge.

Das beschriebene Ersetzungsschema sowie die Trennung von Crossover und Mutation hinsichtlich der Elternindividuen erschweren die Beschreibung des Algorithmus mit Hilfe des *GenEA*-Modells. Aus diesem Grund müssen bei der Definition von MetaGen gegenüber den vorangegangenen Varianten kleine Änderungen vorgenommen werden. Der ursprüngliche MetaGen-Algorithmus wird im weiteren *MGA* genannt.

MGA()
$t \leftarrow 0$
$\mathcal{H} \leftarrow$ Initialize()
while $\neg$Done$(t, \mathcal{H})$
do $t \leftarrow t + 1$
    $\mathcal{H}' \leftarrow T_{MX}(\mathcal{H})$
    $\mathcal{H}'' \leftarrow T_{MM}(\mathcal{H})$
    $\mathcal{H} \leftarrow$ Replace-M$(\mathcal{H}, \mathcal{H}', \mathcal{H}'')$
return $\mathcal{H}$

Algorithmus 5.1: Der MetaGen-Algorithmus *MGA*

Den eigentlichen Kern von MetaGen bilden die problemspezifischen Crossover- und Mutationsoperatoren. Die beiden genetischen Operatoren benutzen zur Erzeugung der Konkurrenten die Kardinalität der beiden folgenden Klauselmengen:

- $\mathcal{C}(\Phi, V)$ als Menge aller Klauseln, die in der Formel Φ auftreten und in denen die Variable V als Literal vorkommt, sowie
- $\mathcal{C}_u(\Phi, V, A)$ als Menge der in Φ vorkommenden Klauseln mit der Variablen V, die unter der Belegung A unerfüllt bleiben.

MetaGen-Mutation T_{MM}. Zur Berechnung von $T_{MM}(\mathcal{H})$ werden die folgenden Schritte für jedes $x \in \mathcal{H}$ und jede Variable V_i in der gegebenen CNF-Formel Φ mit $1 \leq i \leq N$ ausgeführt. Wähle eine zufällige ganze Zahl r aus $[0, \|\mathcal{C}(\Phi, V_i)\|]$. Wenn $r \leq \|\mathcal{C}_u(\Phi, V_i, x)\|$ gilt, dann negiere den Wahrheitswert für V_i in der Belegung x.

Je größer $\|\mathcal{C}_u(\Phi, V_i, x)\|$ ist, um so größer ist die Wahrscheinlichkeit, daß der Wahrheitswert von V_i geändert wird. Mit einer kleinen Wahrscheinlichkeit kann aber auch der Fall eintreten, daß der Wahrheitswert von V_i geändert wird, obwohl alle Klauseln mit V_i erfüllt sind (bei $r = 0$).

MetaGen-Crossover T_{MX}. Zur Berechnung von $T_{MX}(\mathcal{H})$ werden die folgenden Schritte λ Male wiederholt. Wähle ein $x \in \mathcal{H}$ und ein $y \in \mathcal{H}$ zufällig und gleichverteilt aus. Erzeuge entsprechend der angegebenen Regel ein neues Element z und füge z in $T_{MX}(\mathcal{H})$ ein. Für alle $1 \leq i \leq N$ gilt:

$$\begin{array}{ll} x_i = y_i: & z_i = x_i = y_i, \\ x_i \neq y_i: & z_i = \begin{cases} y_i & \text{falls } r \leq \|\mathcal{C}_u(\Phi, V_i, x)\|, \\ x_i & \text{sonst.} \end{cases} \\ & \text{Dabei ist } r \text{ eine zufällige ganze Zahl aus} \\ & [0, \|\mathcal{C}_u(\Phi, V_i, x)\| + \|\mathcal{C}_u(\Phi, V_i, y)\| + 2]. \end{array}$$

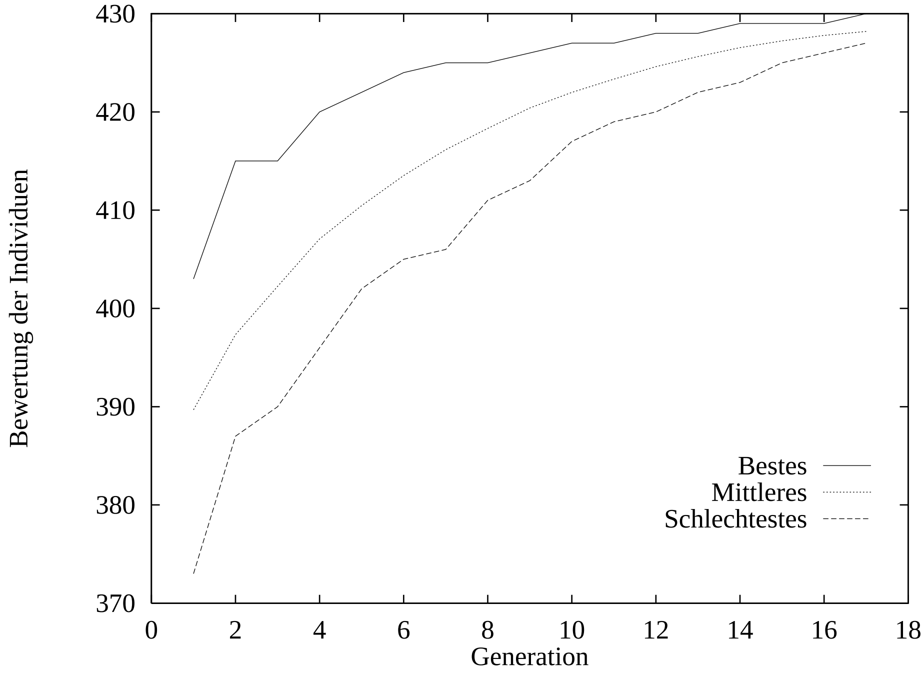
Je größer $\|\mathcal{C}_u(\Phi, V_i, x)\|$ gegenüber $\|\mathcal{C}_u(\Phi, V_i, y)\|$ ist, um so wahrscheinlicher ist, daß ein y_i ausgewählt wird. Wiederum gibt es die Möglichkeit, daß y_i ausgewählt wird, obwohl alle Klauseln mit V_i durch x erfüllt werden (bei r = 0). Der MetaGen-Crossover-Operator kann somit als ein problemspezifisches *Uniform crossover* betrachtet werden.

Das beschriebene Konkurrenten-Populationsmodell wird mit Hilfe der Funktion REPLACE-M$(\mathcal{H}, \mathcal{H}', \mathcal{H}'')$ realisiert.

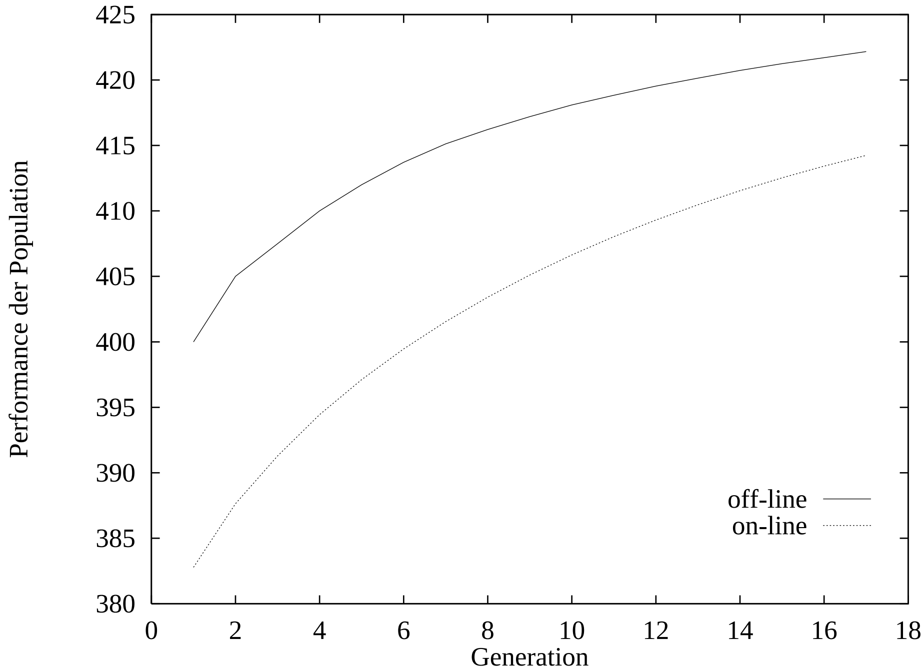
MetaGen-Ersetzungschema REPLACE-M. Die Berechnung der Funktion REPLACE-M$(\mathcal{H}, \mathcal{H}', \mathcal{H}'')$ wird wie folgt realisiert. $\mathcal{H}$, $\mathcal{H}'$ und $\mathcal{H}''$ sind als indizierte Multimengen mit der gleichen Kardinalität λ. Dabei bezeichnet ${}^{\mathcal{H}}x_i$ das i-te Element einer Multimenge $\mathcal{H}$. $\mathcal{H}_r$ ist der zu berechnende Funktionswert. Für alle $1 \leq i \leq N$ gilt:

$${}^{\mathcal{H}_r}x_i = \begin{cases} {}^{\mathcal{H}''}x_i & \text{falls } f_{obj}({}^{\mathcal{H}''}x_i) \geq f_{obj}({}^{\mathcal{H}}x_i) \text{ und } f_{obj}({}^{\mathcal{H}''}x_i) \geq f_{obj}({}^{\mathcal{H}'}x_i), \\ {}^{\mathcal{H}'}x_i & \text{falls } f_{obj}({}^{\mathcal{H}'}x_i) > f_{obj}({}^{\mathcal{H}}x_i) \text{ und } f_{obj}({}^{\mathcal{H}'}x_i) > f_{obj}({}^{\mathcal{H}''}x_i), \\ {}^{\mathcal{H}}x_i & \text{sonst.} \end{cases}$$

Der beschriebene Algorithmus MGA wurde auf die Formeln der Formelklasse $(forced, 100)$ angewandt. Dabei wurde lediglich eine Erfolgsrate von 55 Prozent erreicht. Das entspricht bei weitem nicht den Erwartungen. Trotzdem ist eine bemerkenswerte Eigenschaft erkennbar. Die mittlere Anzahl von benötigten Generationen beträgt nur 39, wenn man nur die erfolgreichen



(a) Verlauf der Individuen-Bewertungen über den Generationen



(b) Verlauf der Performance über den Generationen

Abbildung 5.1: Eine Anwendung des genetischen Algorithmus MGA mit $\lambda = 200$ auf die Formel $(forced, 100, 10)$

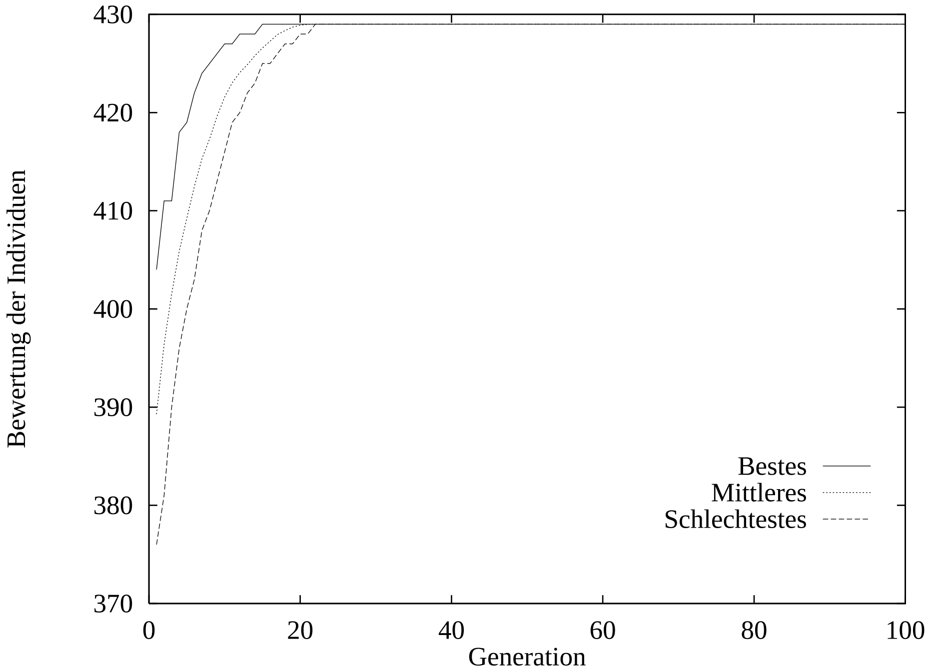


Abbildung 5.2: Ein Beispiel für erfolglose Suche bei der Anwendung des genetischen Algorithmus MGA mit $\lambda = 200$ auf die Formel $(forced, 100, 10)$

Läufe in die Rechnung einbezieht. Das heißt, MGA arbeitet außerordentlich effizient. In den Abbildungen 5.1 und 5.2 sind typische Anwendungen von MGA dargestellt.

In weiteren Experimenten soll nun geklärt werden, worin die Ursachen für die starke Konvergenz von MGA bestehen, in dem Populationsmodell oder in den verwendeten genetischen Operatoren. Zum Vergleich wurden die folgenden beiden Algorithmen verwendet:

- *MGA-UX* als *MGA* mit *Uniform crossover* und
- *MGA-UM* als *MGA* mit gleichverteilter Mutation.

Die zugehörigen Testergebnisse sind im ersten Teil von Tabelle 5.1 dargestellt. Dabei stellt sich heraus, daß nur MetaGen-Crossover eine positive Wirkung in Hinblick auf die Verbesserung der Erfolgsrate hat. Die MetaGen-Mutation trägt eher zu einer Verschlechterung bei.

Es stellt sich die Frage, ob sich nicht ein anderer problemspezifischer Mutationsoperator finden läßt, der insbesondere gute Explorationseigenschaften

```
MGA-UM()
t ← 0
H ← INITIALIZE()
while ¬DONE(t, H)
do t ← t + 1
     H' ← T_MX(H)
     H'' ← T_UM(H)
     H ← REPLACE-M(H, H', H'')
return H
```

Algorithmus 5.2: Der MetaGen-Algorithmus *MGA-UM* mit gleichverteilter Mutation

```
MGA-UX()
t ← 0
H ← INITIALIZE()
while ¬DONE(t, H)
do t ← t + 1
     H' ← T_UX(H)
     H'' ← T_MM(H)
     H ← REPLACE-M(H, H', H'')
return H
```

Algorithmus 5.3: Der MetaGen-Algorithmus *MGA-UX* mit *Uniform crossover* besitzt. Die naheliegendste und einfachste Variante ist sicherlich, eine beliebige Variable auszuwählen, die in einer noch nicht erfüllten Klausel vorkommt, und ihre Belegung umzukehren . Dieses Mutationsverfahren wird im weiteren *Unsatisfied clause*-Mutation genannt.

Unsatisified clause-Mutation T_{CM}. Zur Berechnung von $T_{CM}(\mathcal{H})$ werden die folgenden Schritte für jedes $x \in \mathcal{H}$ ausgeführt. Wähle einen Index i mit der Bedingung

$$\|\mathcal{C}_u(\Phi, V_i, x)\| > 0 \text{ und } 1 \leq i \leq N$$

zufällig und gleichverteilt aus, und invertiere das i-te Bit x_i.

Zur Bewertung des entworfenen Mutationsoperators wurden die folgenden Algorithmen implementiert und untersucht:

- *MGA-UM* als *MGA* mit *Unsatisfied clause*-Mutation sowie
- *MGA-UX-CM* als *MGA* mit *Uniform crossover* und *Unsatisfied clause*-Mutation.

```
MGA-CM()
t ← 0
H ← INITIALIZE()
while ¬DONE(t, H)
do t ← t + 1
     H' ← T_MX(H)
     H'' ← T_CM(H)
     H ← REPLACE-M(H, H', H'')
return H
```

Algorithmus 5.4: Der MetaGen-Algorithmus *MGA-CM* mit *Unsatisified clause*-Mutation

Tatsächlich konnte durch Verwendung von *Unsatisfied clause*-Mutation eine erhebliche Verbesserung erreicht werden. *MGA-UM* erreichte mit einer Erfolgsrate von 84 Prozent und einer mittleren Anzahl von 16 Generationen

```
MGA-UX-CM()
t ← 0
H ← INITIALIZE()
while ¬DONE(t, H)
do t ← t + 1
     H' ← T_UX(H)
     H'' ← T_CM(H)
     H ← REPLACE-M(H, H', H'')
return H
```

Algorithmus 5.5: Der MetaGen-Algorithmus *MGA-UX-CM* mit *Uniform crossover* und *Unsatisified clause*-Mutation

$\lambda = 200$, $p_c = 0.95$ bei UX, $p_m = 0.009$ bei UM			
Algorithmus	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
MGA	200	55.0	39
MGA-UX	200	58.5	92
MGA-UM	200	72.0	62
MGA-CM	200	84.7	16
MGA-UX-CM	200	89.5	34

Tabelle 5.1: Testergebnisse zum Vergleich des *MetaGen*-Algorithmus mit seinen modifizierten Varianten

das bisher beste Ergebnis. Die Testergebnisse sind in Tabelle 5.1 zusammengefaßt. Angesichts der schnellen Konvergenz des Verfahrens bietet es sich an, die effektive Erfolgsrate durch kontrolliertes Neustarten des Algorithmus noch weiter zu erhöhen.

An dieser Stelle sollen die Ergebnisse von MetaGen und seiner modifizierten Varianten noch einmal im Zusammenhang betrachtet werden. MetaGen ist ein evolutionärer Algorithmus, der speziell zur Lösung des $3CNF$-Problems entwickelt wurde. Durch die Konstruktion von problemspezifischen Operatoren für Crossover und Mutation sollte einerseits eine hohe Erfolgsrate und andererseits eine effiziente Suche gewährleistet werden. Dieses Ziel wurde nur teilweise erreicht (siehe die Ergebnisse von MGA in Tabelle 5.1). Die Population konvergiert tatsächlich schnell in Richtung einer vermuteten Lösung, jedoch ist die Erfolgsrate alles andere als befriedigend. Der Algorithmus MGA besitzt gute Exploitationseigenschaften, aber eine schlechte Exploration. Es stellt sich die Frage, wie man diesen Nachteil beseitigen kann. Die Explorationseigenschaften lassen sich verbessern, indem man den Anteil von „*Random walk*" vergrößert (siehe $GWSAT$-Algorithmus). Das kann geschehen entweder durch die Benutzung des klassischen probabilistischen Bit-Mutationsoperators UM oder des problemspezifischen Mutationsoperators CM. Diese Vermutung läßt sich experimentell bestätigen (siehe die Ergebnisse von MGA-UM und MGA-CM in Tabelle 5.1). Man kann ferner erkennen, daß durch die gezielten Veränderung der Chromosomen durch problemspezifische Operatoren in jedem Fall eine schnelle Konvergenz erreicht wird.

In den bisher vorgestellten genetischen und evolutionären Algorithmen wurde die Erzeugung neuer Generationen abgebrochen, wenn entweder ein Modell gefunden wurde oder die maximale Anzahl von Generationen t_{max} erreicht war, siehe Definition der Funktion DONE in Abschnitt 4.3. Diese Form der Beendigung des Algorithmus wird *statisches Abbruchkriterium* genannt. Entsprechend sind auch sogenannte *dynamische Abbruchkriterien* denkbar, wobei anhand bestimmter Größen entschieden wird, ob ein weiteres Vorantreiben des Evolutionsprozesses noch sinnvoll ist oder nicht.

Die Entscheidung zu treffen, den Lauf eines genetischen Algorithmus abzubrechen und einen neuen Versuch zu beginnen, ist im allgemeinen eine schwierige Aufgabe. Je länger der simulierte Evolutionsprozeß andauert, um so einheitlicher wird die Population der Individuen und die Effizienz der Suche nimmt stark ab, siehe Abschnitt 3.2.4. Aus diesem Grund soll ein Zeitpunkt bestimmt werden, ab dem es unwahrscheinlich ist, daß die Bewertung des besten gefundenen Individuums noch weiter verbessert werden kann. Denkbare Varianten für die Bestimmung dieses Zeitpunktes sind unter anderem:

- die Berechnung der Standardabweichnung über den Bewertungen der Individuen nach jeder Generation und Abbruch des Algorithmus, wenn dieser Wert eine bestimmte Schwelle unterschreitet,

- die Berechnung der Hamming-Abstände aller Individuen zur Bestimmung des Konvergenzgrades der Population und Abbruch, wenn 90 Prozent Konvergenz erreicht sind,

- das Verfolgen des Verlaufs der besten bzw. mittleren Bewertungen der Population über den Generationen und Abbruch, wenn über einen bestimmten Zeitraum keine Verbesserung erreicht wurde.

Im Falle des Erfüllbarkeitsproblems ist ein bitweiser Vergleich der Individuen angesichts der Größe der Problemrepräsentation mit einem hohen Rechenaufwand verbunden. Degegen entsteht bei der Umsetzung der dritten Variante praktisch kein Mehraufwand, da die benötigten Größen auch in anderen Teilen des Algorithmus benötigt werden und somit zur Verfügung stehen. Aus diesem Grund wurde die dritte beschriebene Variante für das $3SAT$-Problem benutzt. Dabei wird die mittlere Bewertung der Population über einem sogenannten *Halting window* von h Generationen aufgezeichnet.

Der Evolutionszyklus wird abgebrochen, wenn die mittlere Bewertung der aktuellen Generation nicht größer ist als h Generationen früher. Gute Ergebnisse konnten mit $h = 25$ erzielt werden. Angesichts der schnellen Konvergenz der Algorithmen läßt sich h sicherlich noch auf Werte bis 5 reduzieren. Neben der Größe des *Halting window* muß noch die maximale Anzahl von Versuchen r_{max} definiert werden. Je größer r_{max} gewählt wird, um so größer ist auch die Wahrscheinlichkeit, daß ein Modell gefunden wird.

Das beschriebene dynamische Abbruchkriterium wurde für den Algorithmus $MGA\text{-}CM$ implementiert und mit verschiedenen Klassen von *forced*-Formeln getestet. Die Ergebnisse sind in Tabelle 5.2 zusammengefaßt. Schon mit relativ kleinen Werten von r_{max} lassen sich Erfolgsraten von 90 Prozent und mehr erreichen. Bei Formeln mit 200 und mehr Variablen sollte, wenn möglich, mit größeren Populationen ($\lambda > 200$) gearbeitet werden.

MGA-CM mit $\lambda = 200$, $h = 25$ und $r_{max} = 20$				
Testformel-Klasse	Anzahl Testformeln	Erfolgsrate in Prozent p^+_{model}	Erfolgsrate für 1. Lauf in Prozent	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
$(forced, 100)$	200	99.5	84.7	16
$(forced, 200)$	200	94.0	57.0	30
$(forced, 300)$	100	90.0	33.0	41

Tabelle 5.2: Testergebnisse zum Verhalten von *MGA-CM* mit dynamischer Reinitialisierung für verschiedene Klassen von *forced*-Formeln

5.2 Die Anwendung problemspezifischer Operatoren

Im vorangegangenen Abschnitt wurden ausgehend vom MetaGen-Algorithmus problemspezifische genetische Operatoren entworfen und untersucht. Dabei konnte gezeigt werden, daß mit Hilfe von MetaGen-Crossover und *Unsatisfied clause*-Mutation ein erheblicher Geschwindigkeitsgewinn erreichbar ist. Diese schnelle Konvergenz hat aber auch eine nachteilige Wirkung auf die Breite der Suche und damit auf die Erfolgsrate.

Daraufhin soll nun untersucht werden, ob mit den bekannten Populationsmodellen, Elitismus bzw. *Crowding population*, eine weitere Verbesserung der Erfolgsrate möglich ist. Sämtliche dafür benötigten genetischen Operatoren und Ersetzungsverfahren sind bereits definiert und müssen nur noch miteinander kombiniert werden. Folgende evolutionäre Algorithmen wurden untersucht:

- *EGA-MX-CM* als Elitismus-Population mit MetaGen-Crossover und *Unsatisfied clause*-Mutation,
- *EGA-UX-CM* als Elitismus-Population mit *Uniform crossover* und *Unsatisfied clause*-Mutation,
- *CGA-MX-CM* als *Crowding*-Population mit MetaGen-Crossover und *Unsatisfied clause*-Mutation sowie
- *CGA-CM* als *Crowding*-Population mit *Uniform crossover* und *Unsatisfied clause*-Mutation.

$\lambda = 200$, $GG = 0.9$, $p_c = 0.55$ bei UX			
Algorithmus	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
EGA-MX-CM	200	94.5	33
EGA-UX-CM	200	83.0	108

Tabelle 5.3: Testergebnisse zum Verhalten des genetischen Algorithmus mit Elitismus bei Verwendung problemspezifischer Mutations- und Crossover-Operatoren

EGA-MX-CM()
$t \leftarrow 0$
$\mathcal{H} \leftarrow$ Initialize()
while $\neg$Done$(t, \mathcal{H})$
do $t \leftarrow t + 1$
    $\mathcal{H}' \leftarrow T_{SUS}(\mathcal{H})$
    $\mathcal{H}' \leftarrow T_{MX}(\mathcal{H}')$
    $\mathcal{H}' \leftarrow T_{CM}(\mathcal{H}')$
    $\mathcal{H} \leftarrow$ Replace-E$(\mathcal{H}, \mathcal{H}')$
return $\mathcal{H}$

Algorithmus 5.6: Der evolutionäre Algorithmus *EGA-MX-CM* mit Elitismus, MetaGen-Crossover und *Unsatisified clause*-Mutation

Die Ergebnisse der empirischen Untersuchungen sind in den Tabellen 5.3 und 5.4 zusammengefaßt. Die Vermutung hat sich bestätigt, daß mit den klassischen Populationsmodellen die Erfolgsrate weiter vergrößert werden konnte. Das beste Ergebnis von 94 Prozent wurde mit *EGA-MX-CM* und der Formelklasse $(forced, 100)$ erzielt. Diese Verbesserung wird mit einer doppelt so großen Anzahl benötigter Generationen gegenüber *MGA-CM* bezahlt. Auch die *CGA*-Varianten zeigen erheblich bessere Ergebnisse als die reinen genetischen Algorithmen.

Bei der Anwendung von evolutionären Algorithmen zur Lösung des

```
EGA-UX-CM()
t ← 0
H ← INITIALIZE()
while ¬DONE(t, H)
do t ← t + 1
    H' ← T_SUS(H)
    H' ← T_UX(H')
    H' ← T_CM(H')
    H ← REPLACE-E(H, H')
return H
```

Algorithmus 5.7: Der evolutionäre Algorithmus *EGA-UX-CM* mit Elitismus, *Uniform crossover* und *Unsatisified clause*-Mutation

```
CGA-CM()
t ← 0
H ← INITIALIZE()
while ¬DONE(t, H)
do t ← t + 1
    H' ← T_SUS(H)
    H' ← T_UX(H')
    H' ← T_CM(H')
    H ← REPLACE-C(H, H')
return H
```

Algorithmus 5.8: Der evolutionäre Algorithmus *CGA-CM* mit *Crowding factor*-Modell und *Unsatisified clause*-Mutation

CGA-MX-CM()
$t \leftarrow 0$
$\mathcal{H} \leftarrow$ INITIALIZE()
while $\neg$DONE$(t, \mathcal{H})$
do $t \leftarrow t + 1$
  $\mathcal{H}' \leftarrow T_{SUS}(\mathcal{H})$
  $\mathcal{H}' \leftarrow T_{MX}(\mathcal{H}')$
  $\mathcal{H}' \leftarrow T_{CM}(\mathcal{H}')$
  $\mathcal{H} \leftarrow$ REPLACE-C$(\mathcal{H}, \mathcal{H}')$
return $\mathcal{H}$

Algorithmus 5.9: Der evolutionäre Algorithmus *CGA-MX-CM* mit *Crowding factor*-Modell, MetaGen-Crossover und *Unsatisified clause*-Mutation

$\lambda = 200$, $CF = 2$, $p_c = 0.55$ bei UX

Algorithmus	*Generation gap* GG	Anzahl Testformeln $(forced, 100)$	Erfolgsrate in Prozent p^+_{model}	Mittlere Anzahl Generationen bei Erfolg t^+_{avg}
CGA-MX-CM	0.3	200	81.5	56
	0.7	200	83.5	38
	1.0	200	88.5	23
CGA-CM	0.8	200	80.0	59
	0.9	200	83.0	54
	1.0	200	77.5	53

Tabelle 5.4: Testergebnisse zum Verhalten des *crowding population*-Modells mit problemspezifischen Mutations- und Crossover-Operatoren

Erfüllbarkeitsproblems ist die Maximierung der Erfolgsrate das primäre Ziel. Die Konvergenzgeschwindigkeit ist dabei nur von sekundärem Interesse. Grundsätzlich sind bei der Untersuchung der problemspezifischen und der klassischen genetischen Verfahren zwei Eigenschaften erkennenbar:

1. Die Verwendung problemspezifischer Operatoren ermöglicht eine effiziente Suche, die Population konvergiert schnell in Richtung der vermuteten Lösung.
2. Durch die Verwendung der klassischen genetischen Operatoren, die sich durch die Möglichkeit von „*Random walk*“ bzw. durch einen geringeren Selektionsdruck auszeichnen, läßt sich die Vielfalt der Population über eine längere Zeit erhalten, und die Suche erfolgt breiter. Eine höher Erfolgsrate ist die Folge.

Durch die Kombination beider Eigenschaften (zum Beispiel im Algorithmus *EGA-MX-CM*) lassen sich die besten Ergebnisse erzielen.

5.3 Zusammenfassung der Ergebnisse

In diesem Kapitel wurde die Wirkung problemspezifischer Operatoren auf das Verhalten evolutionärer *SAT*-Algorithmen untersucht. Im Mittelpunkt stand dabei die erreichte Erfolgsrate beim Finden der Modelle und die Konvergenzgeschwindigkeit der Verfahren. Zuerst wurden verschiedene Varianten des MetaGen-Algorithmus und danach auch klassische Populationsmodelle untersucht. Alle entworfenen evolutionären Algorithmen lassen sich als Instanz des generischen Modells *GenEA* darstellen, siehe Abbildung 5.3.

Der MetaGen-Algorithmus *MGA*, wie er von Young et al. [48] vorgestellt wurde, zeigte nur enttäuschende Ergebnisse, hauptsächlich bedingt durch die ungünstige Wahl des Mutationsoperators. Durch die Verwendung der *Unsatisfied clause*-Mutation konnte dieser Nachteil beseitigt werden. Eine weitere Verbesserung der Erfolgsrate konnte durch die Verwendung einer dynamischen Abbruchbedingung bzw. durch die Verwendung von Elitismus-Populationen erreicht werden, siehe Abbildung 5.4.

Betrachtet man nur die *forced*-Formelklassen, dann können sich die entworfenen evolutionären *SAT*-Algorithmen hinsichtlich der Erfolgsrate sehr wohl mit den *GSAT*- bzw. Simulated annealing-Varianten messen, siehe Abbildung 5.5. Dieses Bild sieht aber schon entscheidend anders aus, wenn man zu den *unforced*-Formelklassen übergeht. Auch der beste evolutionäre Algorithmus, das heißt, der mit der größten Erfolgsrate bei *forced*-Formeln, scheitert

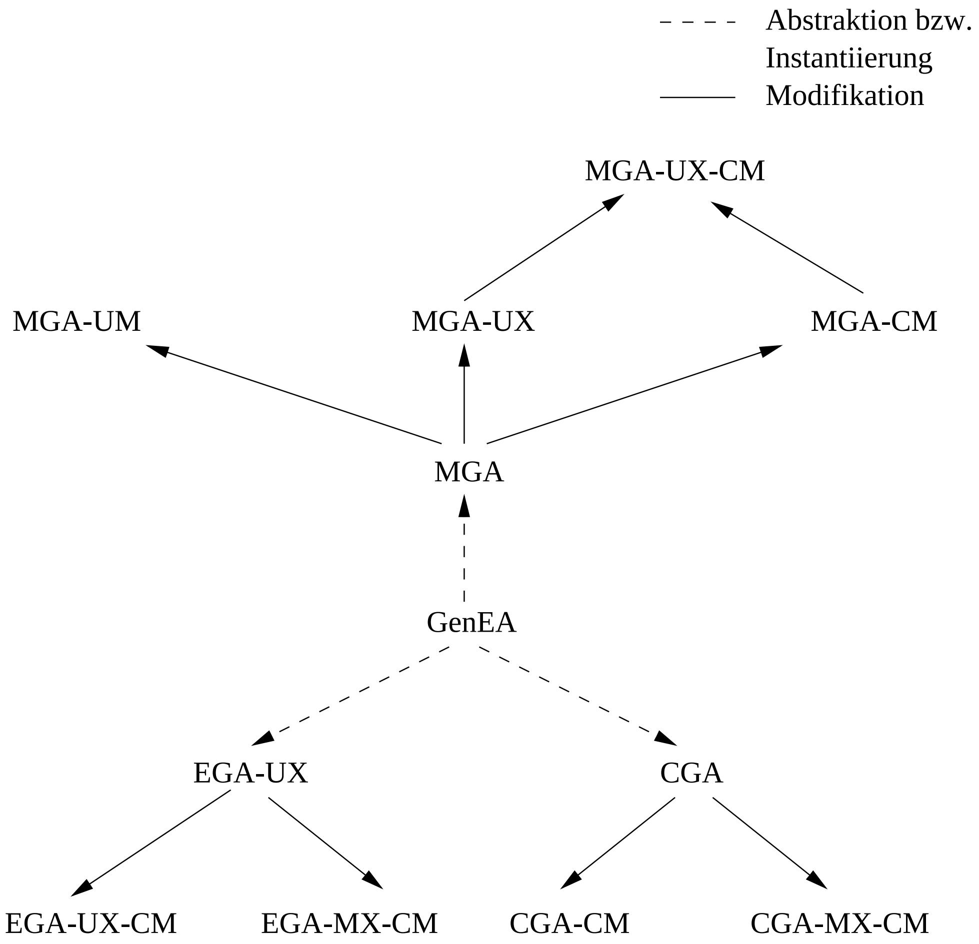


Abbildung 5.3: Die evolutionären Algorithmen für das $3SAT$-Problem

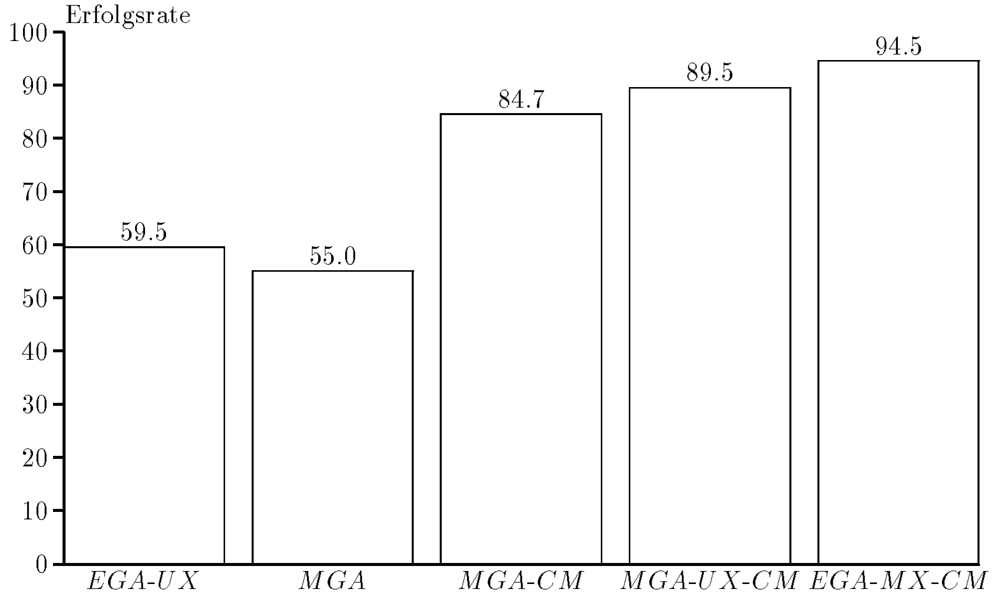


Abbildung 5.4: Der Vergleich von evolutionären Algorithmen bezüglich der Erfolgsrate beim Finden von Modellen für die Formelklasse $(forced, 100)$

bei *unforced*-Formeln völlig. Sowohl unter Verwendung des statischen als auch des dynamischen Abbruchkriteriums wurde für keine einzige *unforced*-Formel ein Modell gefunden. Der Algorithmus konvergierte jedesmal in eine suboptimale Lösung. In Abbildung 5.6 ist eine solche Testreihe dargestellt. Pro Formel wurden jeweils fünf Versuche mit dynamischem Abbruchkriterium durchgeführt.

Das völlig unterschiedliche Verhalten der evolutionären Algorithmen bei den beiden Formelklassen ist außerordentlich bemerkenswert.

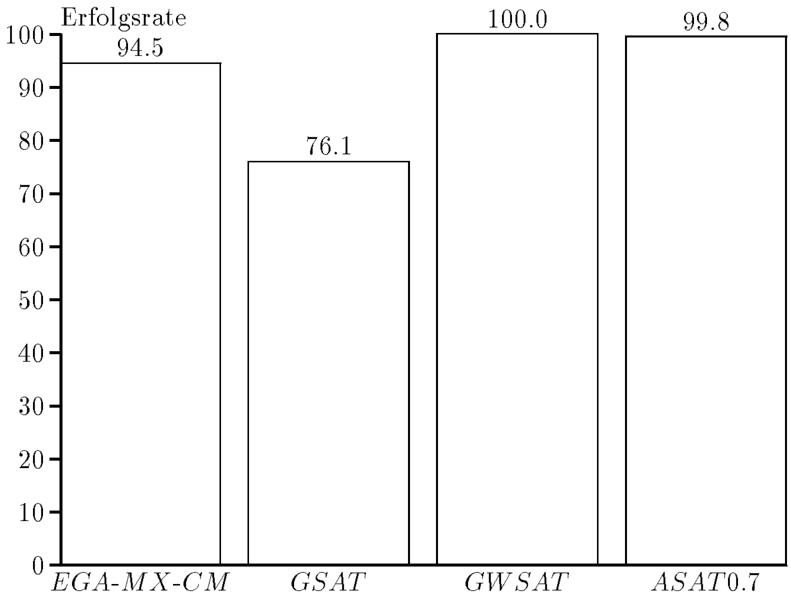


Abbildung 5.5: Der Vergleich von SAT-Algorithmen bezüglich der Erfolgsrate beim Finden von Modellen für die Formelklasse $(forced, 100)$. Die Angaben für die $GSAT$-Varianten beziehen sich dabei auf [18]. $ASAT0.7$ bezeichnet eine Simulated annealing-Variante.

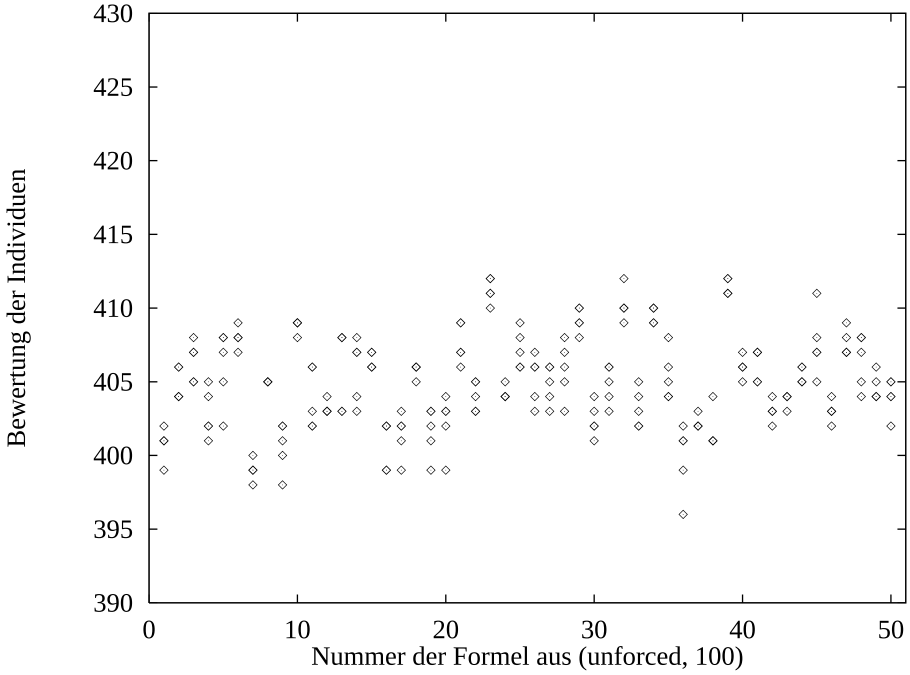


Abbildung 5.6: Die Bewertungen der besten Individuen in fünf unabhängigen Versuchen von *EGA-UX-CM* mit $\lambda = 200$, $GG = 0.9$, $p_c = 0.55$ und $h = 40$

Kapitel 6

Diskussion

Ausgehend von den Ergebnissen über das Verhalten der entworfenen genetischen und evolutionären Algorithmen für das $3SAT$-Problem sollen die folgenden Fragen diskutiert werden:

- Wie ist das Versagen der evolutionären Algorithmen bei den *unforced*-Formeln zu erklären?
- Welche Einschätzungen kann man über die Anwendbarkeit evolutionärer Verfahren für das $3SAT$-Problem treffen?
- Sind evolutionäre Verfahren für geclusterte SAT-Probleme geeignet?
- Welche Möglichkeiten für eine weitere Verbesserung der Algorithmen sind denkbar?
- Welche Möglichkeiten bestehen zur Parallelisierung evolutionärer Algorithmen?
- Welche Vor- und Nachteile besitzen evolutionäre Verfahren gegenüber $GSAT$ und Simulated Annealing?

Die entworfenen genetischen und evolutionären Algorithmen zeigen bei *unforced*-Formeln ein völlig anderes Verhalten als bei *forced*-Formeln. Es konnte kein einziges Modell einer *unforced*-Formel gefunden werden. Offensichtlich konvergieren die Algorithmen fast immer in lokale Maxima der Bewertungsfunktion. Obwohl man auch bei anderen unvollständigen SAT-Algorithmen diese Tendenz erkennen kann, ist sie jedoch lange nicht so ausgeprägt wie bei den evolutionären Verfahren. $GSAT$ erreicht bei Verwendung derselben Testformeln bei $(forced, 100)$ eine Erfolgsrate von 76 Prozent, gegenüber 23

$\lambda = 200$ und $t_{max} = 1$			
Formelklasse	Anzahl Klauseln m	Mittlere Anzahl erfüllter Klauseln m^+	$m - m^+$
$(forced, 100)$	430	395	35
$(forced, 200)$	860	780	80
$(forced, 300)$	1290	1162	128
$(unforced, 100)$	430	358	72
$(unforced, 200)$	860	727	103
$(unforced, 300)$	1290	1111	179

Tabelle 6.1: Die unterschiedliche Schwierigkeit von *forced*- und *unforced*-Formeln. Es wurden pro Formel 200 zufällige Belegungen erzeugt.

Prozent bei $(unforced, 100)$ [18]. Diese Ergebnisse für $GSAT$ lassen sich auch so interpretieren, daß für Formeln mit 100 Variablen 76 Prozent der Maxima bei den *forced*-Formeln auch globale Maxima sind, wohingegen das auf nur 23 Prozent der Maxima bei den *unforced*-Formeln zutrifft. Die Anzahl der lokalen Extrema ist also bei den *unforced*-Formeln bedeutend größer.

Die unterschiedliche Schwierigkeit der beiden Formelklassen läßt sich auch in einem einfachen Experiment verdeutlichen. Für jede einzelne Testformel wurden zufällig 200 Belegungen erzeugt, was einer Population von 200 Individuen entspricht, und für diese Belegungen die mittlere Anzahl von erfüllten Klauseln bestimmt. Die Ergebnisse des Experiments sind in Tabelle 6.1 für die einzelnen Formelklassen dargestellt. Man kann daraus erkennen, daß für *unforced*-Formeln nicht nur mehr lokale Extrema existieren, sondern auch eine völlig andere Qualität in der Struktur der Formel vorliegt. Die innere Struktur einer Formel wird verursacht durch das Erzeugungsverfahren. Man kann erkennen, daß für zufällig erzeugte Belegungen die absolute Anzahl von erfüllten Klauseln bei *forced*-Formeln deutlich größer ist als bei *unforced*-Formeln. Bezogen auf einen genetischen Algorithmus, der eine erfüllende Belegung für eine Formel mit 430 Klauseln und 100 Variablen suchen soll, heißt das, er muß im Falle einer *unforced*-Formel durchschnittlich 35 Klauseln mehr erfüllen als bei einer *forced*-Formel.

Das Scheitern der evolutionären Algorithmen bei bestimmten Formeln läßt sich auch mit Hilfe theoretischer Erkenntnisse interpretieren. Laut Schema-

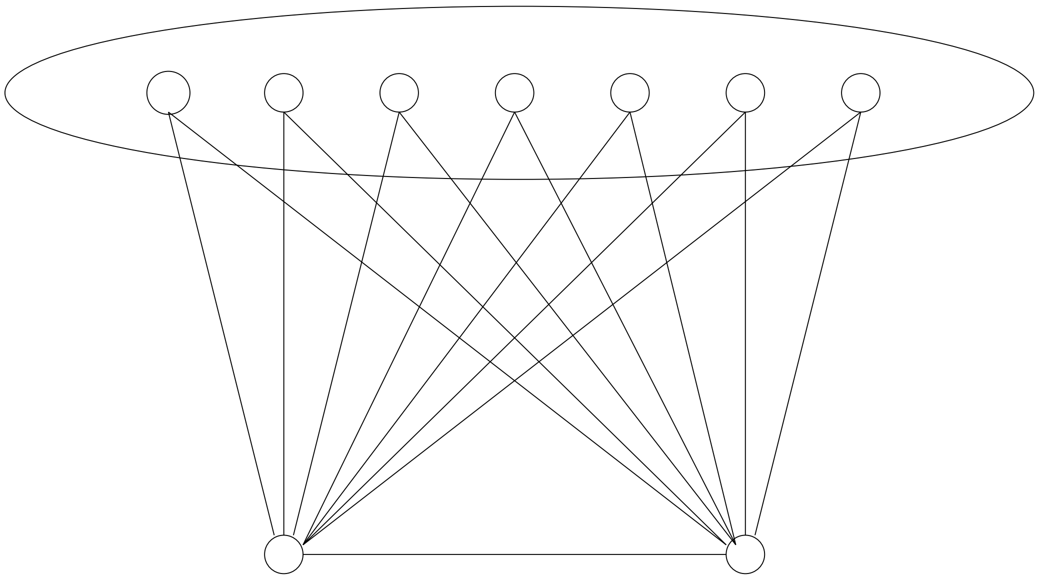


Abbildung 6.1: Ein verfälschter Graph nach Selman et al. [40]

Theorem und *Building block*-Hypothese arbeitet ein genetischer Algorithmus effizient, wenn die optimale Lösung durch Rekombination von Schemata überdurchschnittlicher Bewertung, kurzer definierender Länge und niedriger Ordnung gefunden werden kann. Für das SAT-Problem existieren keine fast optimalen Lösungen. Daraus kann man folgendes ableiten. Wenn ein Chromosom eine große Bewertung besitzt, dann heißt das für das SAT-Problem nicht zwangsläufig, daß seine *Building blocks* auch gut für die Konstruktion der Gesamtlösung geeignet sind. Damit sind grundlegende Voraussetzungen für die Anwendung der *Building block*-Hypothese nicht erfüllt und die effiziente Anwendbarkeit genetischer Algorithmen für das SAT-Problem ist in Frage gestellt. Scheinbar erfüllen *forced*-Formeln die Anforderungen, die an die innere Struktur gestellten werden, und *unforced*-Formeln nicht.

Die vorgestellten evolutionären Algorithmen sind problemspezifischen Lösungsverfahren wie $GSAT$ und $GWSAT$ unterlegen. Insbesondere trifft das auf die Problemklasse der zufällig generierten Formeln zu. Es stellt sich die Frage, ob das auch für andere Klassen von SAT-Problemen der Fall ist. Diese Frage soll an einem ausgewählten Beispiel, den *geclusterten* SAT-Problemen, im folgenden erörtert werden.

Eine mögliche Anwendung des Erfüllbarkeitsproblems ist die Einfärbung von Graphen (*graph coloring*). Dabei hat sich herausgestellt, daß insbesondere $GSAT$ Schwierigkeiten bei einer bestimmten Art von Graphen hat: bei stark asymmetrischen Graphen. Der Effekt der Asymmetrie soll an dem Graphen in Abbildung 6.1 erläutert werden [40]. Derartige Probleme werden auch als *geclusterte SAT-Probleme* bezeichnet, weil einige Klauseln (als Formulierung von Constraints) eng zusammenhängen und mehrere solcher Cluster nur lose verbunden sind.

Man nehme an, der dargestellte Graph soll mit Hilfe eines lokalen Suchverfahrens mit drei Farben markiert werden. Im Ausgangszustand wird jedem Knoten des Graphen eine der drei Farben zufällig zugewiesen. Der Algorithmus wird dann beginnen, die Knoten so zu markieren, daß die Anzahl der Konflikte minimiert wird. Ein Konflikt tritt dann auf, wenn zwei Knoten, die mit einer Kante verbundenen sind, dieselbe Farbe haben. Da die Anzahl der Constraints zwischen der großen Gruppe von Knoten und der einzelnen Knoten im unteren Teil bedeutend größer ist als der einzelne Constraint zwischen den beiden unteren Knoten, wird die Algorithmus schnell in einem Zustand steckenbleiben, in dem die beiden unteren Knoten dieselbe Farbe haben, und jeder Knoten in der größeren Menge jeweils eine andere Farbe hat. Diese Einfärbung erfüllt alle Constraints außer einem. Das heißt, der einzelne Constraint zwischen den beiden unteren Knoten wird einfach von den anderen Knoten überstimmt. Von Selman et al. [40] werden solche Graphen *verfälschte Graphen* genannt.

GSAT ist ein lokales Suchverfahren. Bei der Auswahl der Variable, deren Wahrheitswert im nächsten Schritt verändert werden soll, ist der Variablen-Score das entscheidende Kriterium. Dieser Score beruht auf dem Zählen von erfüllten und unerfüllten Klauseln. Er ist entscheidend abhängig von der Anzahl von Klauseln, in denen eine bestimmte Variable vorkommt. Das lokale Suchverfahren wird immer eine Belegung bevorzogen, die die Mehrzahl von Klauseln erfüllt. Aus diesem Grund hat das vorgestellte *GSAT*-Verfahren erhebliche Schwierigkeiten bei der Lösung derartiger Formeln [1].

Leider bieten auch die in dieser Arbeit vorgestellten evolutionären Algorithmen keinen entscheidenden Vorteil bei der Lösung von geclusterten *SAT*-Problemen. Die Grund dafür ist die verwendete Bewertungsfunktion. Als Bewertungsfunktion wird in allen entworfenen genetischen und evolutionären Algorithmen die Anzahl der durch das Individuum erfüllten Klauseln verwendet. Damit ist die Bewertung und Fitneß eines Individuums ebenfalls stark abhängig von der Anzahl der Klauseln, in denen eine bestimmte Variable vorkommt. Ein evolutionärer Algorithmus mit dieser Bewertungsfunktion wird immer die Indivuuen mit der größten Anzahl erfüllter Klauseln bei der Reproduktion bevorzugen. Das Verhalten des Algorithmus ist aber auch

[1] Der Nachteil kann durch das Verwenden von dynamischen Wichtungen größtenteils behoben werden.

entscheidend von der Wahl des Populationsmodell, der genetischen Operatoren und der zugehörigen Parameter abhängig. Ein Populationsmodell mit einem geringen Selektionsdruck und einem großen Rauschen bei der Selektion wird sicherlich bessere Ergebnisse bei geclusterten SAT-Probleme liefern als entsprechende gegensätzliche Modelle. Eine erfolgversprechendere Variante ist die Verwendung einer anderen Bewertungsfunktion. Eine solche Bewertungsfunktion könnte die erfüllten Klauseln gewichtet zählen, wobei die Wichtungen dynamisch in Abhängigkeit von den noch nicht erfüllten Klauseln vergeben werden.

Angesichts der erreichten Ergebnisse, stellt sich die Frage, ob es sinnvoll ist, genetische bzw. evolutionäre Algorithmen für das SAT-Problem anzuwenden. Diese Frage ist nicht einfach zu beantworten. Wenn man voraussetzt, daß die Formeln überhaupt für evolutionäre Algorithmen effektiv lösbar sind, das heißt, die Formeln eine ähnliche Schwierigkeit wie *forced*-Formeln besitzen, dann sollten Laufzeit- und Implementierungsgesichtspunkte bei der Entscheidung in den Vordergrund treten. Wie am Beispiel von MGA-CM gezeigt wurde, können evolutionäre Algorithmen sehr wohl effizent Modelle aussagenlogischer Formeln finden. Ein weiterer Vorteil der evolutionären Algorithmen sind die guten Parallelisierungsmöglichkeiten. Trotzdem bleibt der Rechenaufwand bei der Verarbeitung einer Population von Individuen immer größer als die Bearbeitung einer einzigen Belegung, wie bei den $GSAT$-Varianten.

Trotzdem soll an dieser Stelle nicht ausgeschlossen werden, daß man einen evolutionären Algorithmus konstruieren kann, der in der Lage ist, auch *unforced*-Formeln zu lösen. Entsprechende Lösungsansätze können dabei sein:

- die Erhaltung einer inhomogenen Population durch Verwendung von Subpopulationen, zwischen denen Individuen ausgetauscht werden,
- die Konstruktion hybrider Systeme, beispielsweise eine Kombination von vollständigen Suchverfahren und evolutionären Algorithmen,
- die Verwendung einer aufwendigeren Kodierung für die Belegungen in den Individuen sowie
- die Konstruktion weiterer problemspezifischer Operatoren.

Im Anschluß an die Untersuchung der beschriebenen genetischen und evolutionären Algorithmen wurde auch ein Algorithmus implementiert, der das $GSAT$-Verfahren als Mutationsoperator verwendet. Leider lieferte auch diese Variante keine besseren Ergebnisse.

Ein weiterer denkbarer Ansatz zur Konstruktion von evolutionären Algorithmen zur Lösung des SAT-Problems ist die Verwendung einer anderen Bewertungsfunktion. Die verwendete Bewertungsfunktion, die die Anzahl der erfüllten Klauseln als Maß für die Güte eines Individuums benutzt, ist wahrscheinlich nicht für eine effiziente Lösung des Problems geeignet. Vorstellbar wäre zum Beispiel ein Modell, das kontinuierliche Belegungen für die Variablen benutzt.

Die Evolutionsprozesse in der Natur laufen nicht sequentiell ab, sondern parallel. Es gibt immer mehrere Individuen in einer Population und in der Regel fast immer mehrere Populationen einer Art zur gleichen Zeit. Die biologische Evolution ist ein massiv paralleler Prozeß. Im folgenden soll kurz dargestellt werden, welche Möglichkeiten es gibt, evolutionäre Algorithmen zu parallelisieren [38].

Evolutionäre Algorithmen bieten sowohl innerhalb einer Population, also auf der Ebene der Individuen, als auch auf der Ebene der Populationen selbst diverse Möglichkeiten der Parallelisierung. Folgende Teile eines genetischen Algorithmus lassen sich verhältnismäßig problemlos parallelisieren:

- die Erzeugung einer Individuenmenge als Anfangspopulation,
- die Berechnung der Bewertung und Fitneß der Individuen,
- die Rekombination zweier selektierter Individuen sowie
- die Mutation der Individuen.

Dagegen existieren Schwierigkeiten bei der Parallelisierung:

- der Berechnung oder Prüfung der Erfüllung der Abbruchkriterien sowie
- der Auswahl der zu rekombinierenden Individuen (Selektion).

An dieser Stelle sollen $GSAT$, Simulated annealing and die evolutionären Verfahren noch einmal im Zusammenhang betrachtet werden. Die beiden erstgenannten Verfahren wurden bereits ausführlich im Abschnitt 2.4 beschrieben. Die zahlreichen durchgeführten Experimente haben erkennen lassen, daß die entworfenen genetischen und evolutionären Algorithmen den problemspezifischen Lösungsverfahren wie $GSAT$ unterlegen sind. Dennoch ist es von Interesse, die einzelnen Lösungsansätze noch einmal gegenüberzustellen, um Gemeinsamkeiten und Unterschiede zwischen ihnen aufzuzeigen. Ein zusammenfassender Überblick über Grundprinzipien, Vor- und Nachteile der einzelnen Verfahren wird in Tabelle 6.2 gegeben.

GSAT	**Simulated Annealing**	**Evolutionäre Verfahren**
Probabilistisches lokales Suchverfahren zur Lösung von Erfüllbarkeitsproblemen in der Aussagenlogik	Stochastisches lokales Suchverfahren zur Lösung von kombinatorischen Optimierungsproblemen	Stochastisches Optimierungsverfahren, das einen Evolutionsprozeß simuliert
Hillclimbing über der Anzahl der erfüllten Klauseln in einer einzigen Belegung, zufällige Auswahl von gleich guten Möglichkeiten	Lokale Suche des Minimums einer Energiefunktion mit einem bestimmten Grad von probabilistischem Rauschen	Parallele Suche in einer Population von möglichen Lösungen
Hohe Effizienz, hohe Erfolgsrate	Theoretische Konvergenz	Gute Parallelisierbarkeit
Schwierigkeiten bei bestimmten Problemklassen	Schwierigkeiten beim Finden eines geeigneten *Annealing schedule*	Versagen bei *unforced*-Formeln, hoher Rechenaufwand
Beschreibbar als stochastischer Prozeß, vergleichbar mit Simulated annealing bei einer niedrigen Temperatur		Mutation und Selektion beschreibbar als stochastischer Prozeß, Crossover als nicht-lokale Suche

Tabelle 6.2: Ein Vergleich von *GSAT*, Simulated annealing und evolutionären Algorithmen

Die dargestellten Verfahren lassen sich mit Hilfe der Theorie der *stochastischen Prozesse* beschreiben und vergleichen. Stochastische Prozesse werden durch Zufallsgrößen beschrieben, die von wenigstens einem Parameter t abhängen. Ein *Markowscher Prozeß* ist ein stochastischer Prozeß, bei dem sich die Kenntnis der zukünftigen Entwicklung lediglich aus der des gegenwärtigen Standes ergibt. In einer *Markowschen Kette* durchläuft der Parameter t nur diskrete Werte. Eine Markowsche Kette heißt *homogen*, wenn die Übergangswahrscheinlichkeiten zwischen den Prozeßzuständen unabhängig vom jeweiligen Zeitpunkt sind, anderenfalls heißt sie *inhomogen*.

Simulated annealing läßt sich beschreiben als eine inhomogene Markowsche Kette. Die Verringerung der Temperatur durch den *Annealing schedule* hat die Zeit-Inhomogenität zur Folge. *GSAT* wiederum kann betrachtet werde als eine effiziente Variante von Simulated annealing mit einer konstanten niedrigen Temperatur, bei der es sehr unwahrscheinlich ist, Aufwärtsbewegungen in der Energielandschaft auszuführen, Seitwärts- und Abwärtsbewegungen jedoch möglich sind. Das heißt, *GSAT* kann man beschreiben durch eine homogene Markowsche Kette. Der eigentlich interessante Teil ist der Vergleich beider Verfahren mit evolutionären Algorithmen.

Die Beschreibung evolutionärer Verfahren als stochastische Prozesse ist nur bedingt möglich. Deshalb soll die Wirkung der genetischen Operatoren Selektion, Mutation und Crossover einzeln betrachtet werden. Es wurde gezeigt, daß ein genetischer Algorithmus ohne Crossover qualitativ äquivalent zu einem stochastischen Prozeß ist, weil beide exakt durch Zeit-homogene Markowsche Ketten beschrieben werden können: Zustände mit einer hohen Fitneß werden bevorzugt (Selektion) aber verbunden mit stochastischem lokalem Backtracking (Mutation) [33]. Das heißt, ein genetischer Algorithmus, der nur Selektion und Mutation benutzt, ist vergleichbar mit Simulated annealing bei einer konstanten Temperatur und damit auch vergleichbar mit *GSAT*. Die gesamte evolutionäre Suchvorgang (Selektion, Crossover und Mutation) ist ebenfalls beschreibbar als Zeit-homogene Markowsche Kette, jedoch mit einem entscheidenden Unterschied: der nächste Zustand wird erreicht durch ein nicht-lokales Verfahren (Crossover). Wie man erahnen kann, ist eine korrekte, allgemeine theoretische Beschreibung der evolutionären Verfahren eine außerordentlich schwierige Aufgabe. Das trifft aber grundsätzlich auch auf die anderen beiden Verfahren zu: eine exakte Beschreibung der Konvergenzrate bei Simulated annealing oder der effizient lösbaren Problemklassen bei *GSAT* konnte noch nicht geliefert werden. Zur genauen Erklärung der durch Experimente gewonnenen Ergebnisse sind entsprechende theoretische Modelle unumgänglich.

Gewöhnlich sagt man, genetische bzw. evolutionäre Algorithmen sollen dann angewendet werden, wenn keine effizienteren Verfahren zur Lösung des Problems bekannt sind. Ausgehend von den erzielten Ergebnissen kann man die folgenden Schlüsse ziehen:

1. Bevor genetische Algorithmen für das SAT-Problem angewandt werden können, müssen geeignete genetische Operatoren ausgewählt bzw. konstruiert werden.

2. Es konnte nicht gezeigt werden, daß evolutionäre Verfahren beim Umgehen lokaler Extrema anderen unvollständigen SAT-Algorithmen überlegen sind.

3. Die Ausführung genetischer Algorithmen erfordert einen hohen Rechenaufwand, und ihre Anwendung kann nur empfohlen werden, wenn sie für den speziellen Anwendungsfall bessere Ergebnisse liefern.

Literaturverzeichnis

[1] Emile H. Aarts and Jan Korst. *Simulated Annealing and Boltzmann machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* Wiley, Chichester, 1990.

[2] Thomas Bäck. Self-adaptation in genetic algorithms. In *Proceedings of the First European Conference on Artificial Life*, pages 263 – 271, Paris, France, December 1991. MIT Press.

[3] Thomas Bäck. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In Reinhard Männer and Bernhard Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 85 – 94. Elsevier Science Publishers, Amsterdam, Holland, 1992.

[4] Thomas Bäck. Optimal mutation rates in genetic search. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 2 – 9. Morgan Kaufmann, 1993.

[5] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, Fundamentals. *University Computing*, 15(2):58 – 69, 1993.

[6] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 2, Research topics. *University Computing*, 15(4):170 – 181, 1993.

[7] Bob Carter and Kihong Park. How good are genetic algorithms at finding large cliques: an experimental study. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, October 1993.

[8] Michael Chester. *Neural Networks: A Tutorial.* Prentice Hall, Englewood Cliffs, NJ, USA, 1993.

[9] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151 – 158, 1971.

[10] Bennett L. Fox. Uniting probabilistic methods for optimization. In J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, editors, *Proceedings of the 1992 Winter Simulation Conference*, pages 500 – 505, New York, NY, USA, 1992. IEEE.

[11] Matthew L. Ginsberg and David A. McAllestor. GSAT and dynamic backtracking. Technical report, CIRL, University of Oregon, Eugene, OR, USA, 1994.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, USA, 1989.

[13] David E. Goldberg. The existential pleasures of genetic algorithms. Technical report, Department of General Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, USA, November 1994.

[14] Jin-Kao Hao and Raphael Dorne. An empirical comparison of two evolutionary methods for satisfiability problems. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, IEEE World Congress on Computational Intelligence, pages 450 – 455, New York, NY, USA, 1994. IEEE.

[15] Jörg Heitkötter and David Beasley, editors. *The Hitch-Hiker's Guide to Evolutionary Computation: A List of Frequently Asked Questions (FAQ)*. Posted periodically to the newsgroup comp.ai.genetic, March 1995.

[16] John H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[17] John H. Holland. Genetische Algorithmen. *Spektrum der Wissenschaft*, pages 44 – 51, September 1992.

[18] Steffen Hölldobler, Holger Hoos, Antje Strohmaier, and Andreas Weiß. The GSAT/SA family. Technical report, TH Darmstadt, Informatik, Intellektik, Darmstadt, November 1994.

[19] Lester Ingber and Bruce Rosen. Genetic algorithms and very fast simulated reannealing: a comparison. *Mathematical and Computer Modelling*, 16(11):87 – 100, 1992.

[20] Terry Jones. *Evolutionary algorithms, fitness landscapes and search.* PhD thesis, University of New Mexico, Albuquerque, NM, USA, May 1995.

[21] Kenneth A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, University of Michigan, Ann Arbor, MI, USA, 1975.

[22] Kenneth A. De Jong and Jayshree Sarma. Generation gaps revisited. In Darrell Whitley, editor, *Foundations of Genetic Algorithms – 2*, pages 19 – 28. Morgan Kaufmann, Vail, CO, USA, 1992.

[23] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. An evolutionary approach to combinatorial optimization problems. In *Proceedings of the 1994 Computer Science Conference (CSC'94)*, pages 66 – 73, Phoenix, AZ, USA, March 1994. ACM Press.

[24] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[25] Tracy Larrabee and Yumi Tsuji. Evidence for a satisfiability threshold for random 3CNF formulas. Technical report, Baskin Center for Computer Engineering and Information Sciences, University of California, Santa Cruz, CA, USA, November 1992.

[26] Sushil J. Louis and Gregory J. E. Rawlins. Predicting convergence time for genetic algorithms. Technical report, Department of Computer Science, Indiana University, Bloomington, IN, USA, January 1993.

[27] Samir W. Mahfoud. *Niching methods for genetic algorithms.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, May 1995.

[28] William McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, Argonne, IL, USA, September 1994.

[29] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, Berlin, 1992.

[30] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459 – 465, San Jose, CA, USA, July 1992.

[31] Melanie Mitchell, John H. Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing? In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[32] Bernhard Nebel. Komplexitätsanalysen in der Künstlichen Intelligenz. *KI – Künstliche Intelligenz: Forschung, Entwicklung, Erfahrungen*, pages 6 – 14, April 1995.

[33] Kihong Park and Bob Carter. On the effectiveness of genetic search in combinatorial optimization. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, November 1994.

[34] Karl Rüdiger Reischuk. *Einführung in die Komplexitätstheorie*. Teubner, Stuttgart, 1990.

[35] Raúl Rojas. *Theorie der neuronalen Netze: Eine systematische Einführung*. Springer, Berlin, 1993.

[36] Günter Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Trans. on Neural Networks*, 5(1), 1994.

[37] Günter Rudolph and Hans-Paul Schwefel. Evolutionäre Algorithmen: Ein robustes Optimierungskonzept. Technical report, Lehrstuhl für Systemanalyse, Fachbereich Informatik, Universität Dortmund, Dortmund, 1994.

[38] Eberhard Schöneburg, Frank Heinzmann, and Sven Feddersen. *Genetische Algorithmen und Evolutionsstrategien: Eine Einführung in Theorie und Praxis der simulierten Evolution*. Addison-Wesley, Bonn, 1994.

[39] Uwe Schöning. *Logik für Informatiker*. BI-Wissenschaftsverlag, Mannheim, 1992.

[40] Bart Selman and Henry Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the 1993 International Joint Conference on Artificial Intelligence (IJCAI-93)*, 1993.

[41] Bart Selman and Henry Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, Washington, DC, USA, July 1993.

[42] Bart Selman and Henry Kautz. *GSAT User's Guide, version 35.* AT&T Bell Laboratories, Murray Hill, NJ, USA, 1993.

[43] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1993.

[44] Bart Selman, Henry Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, USA, July 1994.

[45] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440 – 446, San Jose, CA, USA, July 1992.

[46] Anthony Yiu-Cheung Tang. *Tolkien: A Toolkit for Genetic-Based Applications.* Department of Computer Science, The Chinese University of Hong Kong, 1994.

[47] Ulrich Wilhelm Thonemann. Finding improved simulated annealing schedules with genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, IEEE World Congress on Computational Intelligence, pages 391 – 395, New York, NY, USA, 1994. IEEE.

[48] R. A. Young and A. Reel. A hybrid genetic algorithm for a logic problem. In L. C. Aiello, editor, *ECAI 90. Proceedings of the 9th European Conference on Artificial Intelligence*, pages 744 – 746, London, UK, 1990. Pitman.