Theoretische Analyse evolutionärer Algorithmen unter dem Aspekt der Optimierung in diskreten Suchräumen

Thomas Jansen

Fachbereich Informatik, Lehrstuhl 2
Universität Dortmund
http://ls2-www.cs.uni-dortmund.de/~jansen

Evolutionäre Algorithmen (EA) sind allgemeine, randomisierte Suchverfahren, die unter anderem zur Optimierung eingesetzt werden können. Ein hier verfolgter Ansatz, der zu einem besseren Verständnis führen soll, ist die theoretische Analyse. Dabei werden vor allem einfache EA auf konkreten, typischen Zielfunktionen bezüglich ihrer Effizienz untersucht. Die Analyse beginnt nach einer Diskussion der Grenzen und Möglichkeiten evolutionärer Algorithmen mit dem vielleicht einfachsten EA. Wir sprechen wesentliche Ergebnisse für den so genannten (1+1) EA an und erklären, wie man davon ausgehend das weite Feld evolutionärer Algorithmen erschließen kann mittels Analyse von Variationen des (1+1) EA. Ein Höhepunkt ist die Analyse eines speziellen EA mit Crossover, für den man an einem Beispiel nachweisen kann, dass er mutations-basierte Algorithmen bei weitem schlägt.

1 Einleitung

Evolutionäre Algorithmen sind randomisierte Suchheuristiken, die zu vielen unterschiedlichen Zwecken eingesetzt werden können. Eine typische Anwendung ist die Optimierung von pseudo-booleschen Funktionen $f\colon \{0,1\}^n \to \mathbb{R}$, die einem Bitstring der Länge n eine reelle Zahl zuordnen. Man ist daran interessiert, einen Bitstring zu finden, dem eine möglichst große Zahl zugeordnet ist, man will f also maximieren. Der Fall, dass f minimiert werden soll, ist aber nicht wesentlich verschieden. Grundsätzlich geht man davon aus, dass die Optimierung in einem „Black-Box-Szenarium“ erfolgt: Man stellt sich die Zielfunktion f verborgen in einem schwarzen Kasten vor, Zugang zu f hat man nur durch eine Art Orakel. Man legt einen Bitstring $x \in \{0,1\}^n$ vor und erhält zuverlässig den zugehörigen Funktionswert $f(x) \in \mathbb{R}$ als Antwort. Das wiederholt man so lange, bis erstmals ein Bitstring mit maximalem Funktionswert gefunden wird. Wir nehmen hier vereinfachend an, dass man die Information, erfolgreich ein Maximum gefunden zu haben, vom Orakel erhält und machen uns hier keine Gedanken darüber, wann man die Suche abbrechen sollte, wenn man das, wie es natürlich in der Praxis ist, nicht erfährt.
Evolutionäre Algorithmen orientieren sich am Vorbild der natürlichen Evolution. Sie wird als ein verbessernder Prozess aufgefasst und man hat die Hoffnung, durch Nachahmung zu „verbessernden Algorithmen“ zu kommen. Dafür wird der Zielfunktion f die Rolle der Umwelt zugeschrieben, die unter Einbeziehung zufälliger Elemente über das Überleben von Individuen, die einfach Bitstrings, also Punkte im Suchraum, sind, entscheidet. Der Algorithmus verläuft in immer gleichen Runden, die man Generationen nennt. In jeder Generation werden einige Individuen der aktuellen Population als Eltern ausgewählt. Dann werden mittels Rekombination (Crossover) und Mutation Kinder erzeugt. Da man

die Populationsgröße gerne konstant halten möchte, wird die neue Population, die als Ausgangspunkt für die nächste Generation dient, aus der alten Population, den Eltern und den Kindern gewählt. Dabei spielt die „Fitness“, also der Funktionswert $f(x)$, eine Rolle.
Wir betrachten EA hier losgelöst von ihrer biologischen Motivation als randomisierte Optimierverfahren, die Gegenstand einer in der Informatik üblichen Analyse sind. Die durchaus interessante Frage, was man mit diesen Algorithmen über Evolution lernen kann, verweisen wir an die Biologie.
Trotz ihrer recht frühen Entwicklung in den 60er Jahren und ihrer wachsenden Popularität seit den 80er Jahren des 20. Jahrhunderts ist die Theorie evolutionärer Algorithmen immer noch unbefriedigend. Die Dissertation [Jan00] geht in diese Richtung. Wie bei vielen wissenschaftlichen Arbeiten hat aber am Ende die Anzahl offener Fragen eher zugenommen. Dennoch wird hier insgesamt ein hoffentlich interessanter Erkenntnisgewinn vermittelt.
Im nächsten Abschnitt wird kurz auf grundsätzliche Probleme des schon angesprochenen Black-Box-Szenariums eingegangen. Dann lernen wir den (1+1) EA kennen (Abschnitt 3), den vielleicht einfachsten EA überhaupt, der eine Population aus nur einem Individuum und ausschließlich Mutation sowie eine deterministische Selektion verwendet. Wir fassen die wesentlichen Ergebnisse für diesen Algorithmus zusammen. In Abschnitt 4 diskutieren wir einige Varianten des (1+1) EA, die durch Modifikation der Mutation und Selektion entstehen. Danach beschäftigen wir uns mit der Frage, was man durch Einsatz von Crossover gewinnen kann, und zeigen auf einer Familie von Beispielfunktionen konkret, warum ein EA mit Crossover signifikant schneller als der (1+1) EA ist. Schließlich fassen wir das Gesagte kurz zusammen und geben Hinweise für mögliche künftige Forschung.

2 Möglichkeiten und Grenzen evolutionärer Algorithmen

Evolutionäre Algorithmen werden häufig zur Black-Box-Optimierung eingesetzt und man stellt sich vor, dass über die zu optimierende Funktion f nichts bekannt ist. Man kann sich das konkret so vorstellen, dass f irgendeine Funktion $f\colon A \to B$ ist, wobei A und B endliche Mengen sind und B total geordnet ist. Weil wir möglichst wenige Anfragen an das Orakel stellen wollen, legen wir ihm niemals einen Bitstring $x \in A$ öfter als einmal vor. Unter diesen Umständen kann man einen zunächst ganz erstaunlichen Satz beweisen.

Satz 1 (NFL-Theorem, Wolpert und Macready (1997) [WM97]). *Seien A und B endliche Mengen, B total geordnet. Im Mittel über alle Funktionen $f\colon A \to B$ brauchen alle (randomisierten und deterministischen) Algorithmen, die keinen Punkt $x \in A$ mehrfach abfragen, gleich viele Abfragen zur Optimierung.*

Der Beweis ist einfach. Am Anfang ist jede Funktion mit gleicher Wahrscheinlichkeit Zielfunktion. Nach Abfrage des ersten Punktes x kommen nur noch die Funktionen in Frage, die bei x den erfragten Funktionswert $f(x)$ haben. Weil man aber über alle Funktionen spricht, sind das aus Symmetriegründen unabhängig von der Wahl von x gleich viele.
Man darf das NFL-Theorem nicht überinterpretieren. Es stimmt: Gemittelt über alle Funktionen kann es keine besseren oder schlechteren Optimieralgorithmen geben. Aber nie-

mand will alle Funktionen optimieren. In der Praxis gilt für jede zu optimierende Funktion auf jeden Fall, dass die Berechnung des Funktionswertes an einer Stelle, eine Orakelantwort also, mit akzeptablem Aufwand möglich ist. Das trifft nur für einen verschwindenden Bruchteil der Funktionen zu, gleich ob man „akzeptabler Aufwand“ durch beschränkte Berechnungszeit, beschränkten Berechnungsplatz oder beschränkte Kolmogoroff-Komplexität formal fasst. Man kann an kleinen Beispielen exakt nachrechnen, dass unter solchen in der Praxis notwendigen Beschränkungen kein NFL-Theorem gilt. Wir brauchen uns also vom NFL-Theorem nicht abschrecken lassen: Zwar bekommt man nichts geschenkt (NFL steht für No Free Lunch), aber in der Praxis können evolutionäre Algorithmen anderen Verfahren überlegen sein und sind es auch oft.

3 Der (1+1) EA

Für theoretische Analyse ist es oft hilfreich, wenn das Studienobjekt möglichst einfach ist. Es wird darum gerne so weit vereinfacht, wie das möglich ist, ohne den betrachteten Gegenstand uninteressant zu machen. Wir beschäftigen uns hier mit einem sehr einfachen, kleinen EA, der zwar die übliche Mutation und eine übliche Selektion (Plus-Selektion von Evolutionsstrategien) benutzt, aber auf eine echte Population und Crossover verzichtet.

Algorithmus 1 ((1+1) EA)

1. Wähle $x \in \{0,1\}^n$ zufällig gleichverteilt.
2. $y := x$
3. Führe unabhängig für jedes Bit in y ein Zufallsexperiment durch, ändere seinen Wert mit Wahrscheinlichkeit $1/n$.
4. Falls $f(y) \geq f(x)$ gilt, setze $x := y$.
5. Weiter bei 2.

Die Wahrscheinlichkeit $1/n$, mit der ein Bit seinen Wert von 0 auf 1 oder umgekehrt ändert, nennt man Mutationswahrscheinlichkeit. Andere Werte als $1/n$ sind natürlich denkbar, allerdings ist $1/n$ die unter Experten häufigste Empfehlung; wir werden noch sehen, warum das so ist. Die Anzahl der Orakelanfragen, bis erstmalig x ein Optimum ist, nennen wir T und $\mathrm{E}(T)$ die erwartete Laufzeit. Manchmal ist $\mathrm{E}(T)$ allerdings irreführend: Es gibt Funktionen, bei denen zwar mit sehr großer Wahrscheinlichkeit ein Optimum schnell gefunden wird, aber trotzdem $\mathrm{E}(T)$ sehr groß ist. Das kann daran liegen, dass der erste Bitstring x mit verschwindender Wahrscheinlichkeit so ungünstig gewählt wird, dass das Erreichen eines Optimums dann extrem lange dauert. Wir bleiben trotzdem bei $\mathrm{E}(T)$ und betrachten nur nötigenfalls zusätzlich die Wahrscheinlichkeit $\mathrm{Prob}(T \leq s)$, in nur s Schritten fertig zu werden. Man kann für den (1+1) EA viele Ergebnisse, darunter auch durchaus unerwartete, zeigen. Wir zählen die wichtigsten Erkenntnisse auf.

- Für jede Funktion $f\colon \{0,1\}^n \to \mathbb{R}$ gilt $\mathrm{E}(T) \leq n^n$.
- Eine Funktion $f\colon \{0,1\}^n \to \mathbb{R}$ heißt *linear*, wenn es reelle Gewichte $w_0, \ldots, w_n$

gibt, so dass $f(x) = w_0 + \sum_{i=1}^{n} w_i x_i$ gilt. Für lineare Funktionen ist $\mathrm{E}(T) = \mathrm{O}(n \log n)$.

- Ein Bitstring $x \in \{0,1\}^n$ heißt *lokales Optimum* von $f\colon \{0,1\}^n \to \mathbb{R}$, wenn für alle Bitstrings $y \in \{0,1\}^n$, die sich von x in genau einem Bit unterscheiden, $f(y) \leq f(x)$ gilt. Hat f nur ein lokales Optimum, nennt man f *unimodal*. Offenbar gilt bei unimodalen Funktionen für jeden Bitstring x, dass x das gesuchte globale Maximum ist oder einen leicht zu findenden Nachbarn mit echt größerem Funktionswert hat. Darum könnte man glauben, dass unimodale Funktionen leicht zu optimieren sind. Man kann aber eine unimodale Funktion angeben mit $\mathrm{E}(T) = \Theta\left(n^{3/2} \cdot 2^{\sqrt{n}}\right)$.

- Man kann jede Funktion $f\colon \{0,1\}^n \to \mathbb{R}$ auch als Polynom über den Variablen $x_1, x_2, \ldots, x_n$ darstellen. Polynome vom Grad 0 sind konstante Funktionen und offenbar trivial zu optimieren. Polynome vom Grad 1 sind lineare Funktionen und wie erwähnt in Zeit $\mathrm{O}(n \log n)$ zu optimieren. Es ist aus der der Komplexitätstheorie bekannt, dass die Optimierung von Polynomen vom Grad 2 NP-hart ist. Mann kann aber sogar ein Polynom vom Grad 2 konkret angeben, für dass $\mathrm{E}(T) = \Theta\left(n^n\right)$ gilt.

- Man kann Funktionen $f\colon \{0,1\}^n \to \mathbb{R}$ nach der erwarteten Laufzeit klassifizieren: Sei $F_i := \left\{f\colon \{0,1\}^n \to \mathbb{R} \mid \mathrm{E}(T) = \mathrm{O}\left(n^i\right)\right\}$ mit $i \in \{0,1,\ldots,n\}$. Man kann beweisen, dass die F_i eine Hierarchie bilden, also $F_0 \subsetneq F_1 \subsetneq F_2 \subsetneq \cdots \subsetneq F_n$ gilt.

4 Variationen des (1+1) EA

Der (1+1) EA verändert bei einer Mutation den Wert eines jeden Bits mit Wahrscheinlichkeit $1/n$. Da der Bitstring x aus n Bits besteht, wird im Durchschnitt ein Bit verändert. Es ist nahe liegend zu fragen, was sich ändert, wenn man stattdessen bei einer Mutation immer genau ein Bit ändert, wobei man rein zufällig bestimmt, welches Bit das ist.
Wir wissen, was ein lokales Optimum einer Funktion $f\colon \{0,1\}^n \to \mathbb{R}$ ist. Gilt, dass jedes lokale Optimum auch globales Optimum ist, so nennen wir f *schwach unimodal*. Es ist offensichtlich, dass ein (1+1) EA mit 1-Bit-Mutationen nur schwach unimodale Funktionen garantiert optimieren kann. Für andere Funktionen besteht die Möglichkeit, ein lokales Optimum, das nicht gleichzeitig globales Optimum ist, nicht mehr verlassen zu können. Dann macht es keinen Sinn, nach der erwarteten Laufzeit $\mathrm{E}(T)$ zu fragen. Wir wollen also nur noch schwach unimodale Funktionen betrachten. Für solche Funktionen, über die wir im Abschnitt 3 gesprochen haben, stellt man leicht fest, dass der (1+1) EA mit 1-Bit-Mutationen nicht wesentlich schneller ist als der „normale“ (1+1) EA (Algorithmus 1), meist aber auch nicht wesentlich langsamer. Was allerdings deutlich einfacher wird, sind Beweise zur erwarteten Laufzeit. Man kann sich also einfachere Analyse durch eingeschränkte Fähigkeiten erkaufen.
Auch die Selektion des (1+1) EA kann man verändern. Anstatt immer nur Verbesserungen zu akzeptieren ($f(y) \geq f(x)$), kann man mit gewisser (nicht zu großer) Wahr-

scheinlichkeit Übergänge zu Bitstrings mit kleinerem Funktionswert machen. Wir wählen $\min\{1, \alpha(t)^{f(y)-f(x)}\}$ als Übergangswahrscheinlichkeit, wobei $\alpha\colon \mathbb{N} \to [1;\infty[$ eine von der Anzahl der Schritte abhängige Funktion ist. Kombinieren wir diese Selektion mit der einfacheren 1-Bit-Mutation, so ist der entstandene Algorithmus unter dem Namen Simulated Annealing bekannt [KGV83]. Wählt man $\alpha(t) = \alpha$ konstant, so spricht man vom Metropolis-Algorithmus [MRR+53]. Mit dem von uns erarbeiteten Methodenarsenal ist es möglich, zwei Funktionen anzugeben, so dass beide Funktionen für passend gewählte Funktionen $\alpha\colon \mathbb{N} \to [1;\infty[$ in polynomieller erwarteter Zeit optimiert werden, während der Metropolis-Algorithmus selbst bei optimal gewähltem α im Durchschnitt exponentiell lange braucht. Dabei ist α für eine der beiden Funktionen monoton wachsend, für die andere monoton fallend. Das besondere an diesen Funktionen sind die einfach und verständlichen Definitionen und Beweise, die wir hier aber nicht wiedergeben.

Wir kehren stattdessen zurück zum (1+1) EA, wie wir ihn ursprünglich kennen gelernt haben und hinterfragen die Wahl der Mutationswahrscheinlichkeit $1/n$. Wir wissen durch das NFL-Theorem, dass diese Wahl nicht immer optimal sein kann. Sie ist aber zumindest für lineare Funktionen gut getroffen. Wenn wir bei solchen Funktionen von dieser Wahl weit nach oben oder unten abweichen, wenn wir konkret für eine beliebige Funktion $\alpha\colon \mathbb{N} \to \mathbb{R}$ mit $\lim_{n\to\infty} \alpha(n) \to \infty$ die Mutationswahrscheinlichkeit mit $\alpha(n)/n$ deutlich größer oder mit $1/(\alpha(n) \cdot n)$ deutlich kleiner machen, so erhöht sich die erwartete Laufzeit auf $\mathrm{E}(T) = \Omega(\alpha(n) \cdot n \log n)$. Das ist auch einzusehen: Wählen wir die Mutationswahrscheinlichkeit klein, so gibt es viele Mutationen, bei denen gar kein Bit seinen Wert ändert. Solche Schritte sind verschwendet. Wählen wir die Mutationswahrscheinlichkeit groß, so ändern oft viele Bits gleichzeitig ihren Wert, was jedenfalls dann fast immer zu Verschlechterungen führt, wenn schon fast alle Bits ihren richtigen Wert haben. Allerdings bringt uns diese Beobachtung gleich zu einer Vorstellung davon, wie eine Zielfunktion f aussehen könnte, bei der größere Mutationswahrscheinlichkeiten doch hilfreich sind: Es muss selbst (oder gerade) am Ende der Optimierung günstig sein, viele Bits gleichzeitig zu verändern. Tatsächlich kann man eine solche Funktion einfach angeben.

Definition 1. *Sei $n = 2^k > 16$ mit $k \in \mathbb{N}$. Wir definieren eine Unterteilung des Suchraums $\{0,1\}^n$ in fünf disjunkte Teilmengen $P_0, P_1, \ldots, P_4$. Dabei bezeichnen wir mit $\|x\|_1$ die Anzahl der Bit mit Wert 1 in einem Bitstring x.*

$$
\begin{aligned}
P_1 &:= \{x \in \{0,1\}^n \mid n/4 < \|x\|_1 < 3n/4\} \\
P_2 &:= \{x \in \{0,1\}^n \mid \|x\|_1 = n/4\} \\
P_3 &:= \left\{x \in \{0,1\}^n \mid \exists i \in \{0,1,\ldots,(n/4)-1\}\colon x = 1^i 0^{n-i}\right\} \\
P_4 &:= \left\{x \in \{0,1\}^n \mid (\|x\|_1 = \log n) \wedge \left(\sum_{i=1}^{3\log n} x_i = 0\right)\right\} \\
P_0 &:= \{0,1\}^n \setminus (P_1 \cup P_2 \cup P_3 \cup P_4)
\end{aligned}
$$

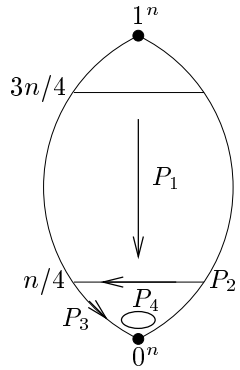

Abbildung 1: Veranschaulinchung der Funktion PATHTOJUMP

Die Funktion PATHTOJUMP$\colon \{0,1\}^n \to \mathbb{R}$ *ist für alle* $x \in \{0,1\}^n$ *definiert durch:*

$$\textsc{PathToJump}(x) := \begin{cases} n - \|x\|_1 & \textit{falls } x \in P_1 \\ (3/4)n + \sum_{i=1}^{n/4} x_i & \textit{falls } x \in P_2 \\ 2n - i & \textit{falls } x \in P_3 \textit{ und } x = 1^i 0^{n-i} \\ 2n + 1 & \textit{falls } x \in P_4 \\ \min\{\|x\|_1, n - \|x\|_1\} & \textit{falls } x \in P_0 \end{cases}$$

Wir veranschaulichen die recht technische Definition mit Zeichnung 1, die den booleschen Verband, also alle Bitstrings der Länge n symbolisiert. Wir stellen uns die Bitstrings ebenen weise so angeordnet vor, dass Bitstrings mit gleicher Anzahl von Bits mit Wert 1 eine Ebene bilden, ganz unten der Nullstring, ganz oben der Einsstring.
Die Pfeile zeigen in Richtung wachsender Funktionswerte. Die Grenzen der Menge P_1, also $n/4$ und $3n/4$ Bits mit Wert Eins sind eingezeichnet, die Lage der Menge P_4 ist durch das Oval angedeutet. Man kann sich nun vielleicht vorstellen, dass bei einer relativ kleinen Mutationswahrscheinlichkeit wie $1/n$ der Nullstring 0^n recht gut gefunden werden kann, der große Sprung in ein Optimum (also zu P_4) aber schwierig ist. Tatsächlich kann man formal beweisen, dass die erwartete Laufzeit nur dann polynomiell ist, wenn die Mutationswahrscheinlichkeit p_m in der Größenordnung von $(\log n)/n$ liegt, also $p_m = \Theta\left((\log n)/n\right)$ gilt. Für substantiell kleinere Mutationswahrscheinlichkeit dauert der Sprung zu P_4 zu lange, mit substantiell größeren Mutationswahrscheinlichkeiten findet man erst gar nicht in den richtigen Teil des Suchraums.
Es ist also so, dass es von der Wahl der richtigen Mutationswahrscheinlichkeit abhängt, ob der (1+1) EA eine Funktion effizient optimieren kann. Leider kann man einer Zielfunktion nicht so ohne weiteres ansehen, welche Wahl der Mutationswahrscheinlichkeit günstig ist. Nimmt man das Black-Box-Szenarium ernst, so ist die Zielfunktion ja „den Blicken entzogen“, man darf also nicht annehmen, Hinweise zur Wahl der Mutationswahrscheinlichkeit zu bekommen. Das lässt den Wunsch entstehen, die Mutationswahrscheinlichkeit

durch den Algorithmus selber bestimmen zu lassen. Man kann die Wahl der Mutationswahrscheinlichkeit einfach mit der Zeit variieren oder auch von den bisher betrachteten Punkten und ihren Funktionswerten abhängig machen. Für den einfacheren ersten Fall schlagen wir einen einfachen an den (1+1) EA angelehnten Algorithmus vor, den wir dynamischen (1+1) EA nennen.

Algorithmus 2 (Dynamischer (1+1) EA)

1. Wähle $x \in \{0,1\}^n$ zufällig gleichverteilt.
2. Setze $p := 1/n$.
3. $y := x$
4. Führe unabhängig für jedes Bit in y ein Zufallsexperiment durch, ändere seinen Wert mit Wahrscheinlichkeit p.
5. Falls $f(y) \geq f(x)$ gilt, setze $x := y$.
6. Setze $p := 2p$. Falls $p > 1/2$ weiter bei 2.
7. Weiter bei 3.

Der dynamische (1+1) EA probiert also die Mutationswahrscheinlichkeiten $2^i/n$ mit $i \in \{0, 1, \ldots, \lceil \log n \rceil - 1\}$ aus, insgesamt also $\lceil \log n \rceil$ viele. Dabei besteht natürlich die Gefahr, dass nur eine Mutationswahrscheinlichkeit wirklich „passt“ und man so in einem Intervall der Länge $\lceil \log n \rceil$ im wesentlichen nur einen hilfreichen Schritt macht. Tatsächlich kann man beweisen, dass der dynamische (1+1) EA auf vielen typischen (einfachen) Funktionen um einen Faktor $\Theta(\log n)$ langsamer ist als der (1+1) EA. Im Kontext von randomisierten Suchheuristiken wird allerdings ein Faktor in der Größenordnung von $\log n$ häufig noch als akzeptabel angesehen werden. Man kann aber auch nachweisen, dass der dynamische (1+1) EA erhebliche Vorteile haben kann. Es ist möglich eine Funktion anzugeben, bei der der (1+1) EA selbst bei optimal eingestellter Mutationswahrscheinlichkeit mit exponentiell schnell gegen 1 konvergierender Wahrscheinlichkeit nicht in polynomieller Zeit fertig wird, während der dynamische (1+1) EA eine erwartete Laufzeit von nur $O(n^2 \log n)$ hat. Leider hat diese Überlegenheit auch ihren Preis: Es ist ebenso möglich, eine Funktion anzugeben, bei der der dynamische (1+1) EA mit exponentiell schnell gegen 1 konvergierender Wahrscheinlichkeit nicht in polynomieller Zeit fertig wird, während der (1+1) EA mit der üblichen Mutationswahrscheinlichkeit $1/n$ mit exponentiell schnell gegen 1 konvergierender Wahrscheinlichkeit nach nur $O(n^2)$ Schritten fertig ist. Die Beweisideen sind in beiden Fällen ganz ähnlich. Der dynamische (1+1) EA kann genau wie der (1+1) EA mit Mutationswahrscheinlichkeit $1/n$ leicht Pfaden folgen, da er ja die Mutationswahrscheinlichkeit $1/n$ auch immer wieder ausprobiert. Allerdings kann er mehr: Er kann zwischendurch auch immer wieder große Sprünge machen, die für den (1+1) EA sehr unwahrscheinlich sind. Wenn zur Optimierung einer Funktion solche großen Sprünge nötig sind, so ist der dynamische (1+1) EA im Vorteil. Wenn man aber durch einen solchen Sprung ein sehr viel versprechenden Teil des Suchraums erreicht, der sich aber schließlich als schwer zu verlassende „Falle“ erweist, so wird diese größere Flexibilität zum Nachteil. Wesentlich ist, dass wir diese intuitiv einsichtige Erklärung durch Angabe konkreter Beispiele und formale Beweise zu einer Gewissheit machen können.

5 Evolutionärer Algorithmen mit Crossover

Evolutionäre Algorithmen benutzen häufig Crossover als Suchoperator. Das gilt besonders für genetische Algorithmen, bei denen häufig Crossover als der wesentliche Variationsoperator betrachtet wird. Die Analyse rein mutations-basierter Algorithmen ist technisch einfacher als die theoretische Untersuchung von EA, die auch Crossover verwenden. Dennoch werden wir hier anhand einer geeigneten Familie von Beispielfunktionen nachweisen, dass ein geeigneter genetischer Algorithmen den (1+1) EA (und in der Tat sogar im wesentlichen alle mutations-basierten EA) bei weitem schlagen kann. Wir geben wieder eine exakte Definition an und veranschaulichen diese durch Abbildung 2.

Definition 2. *Für $n \in \mathbb{N}$ und $m \in \{1, 2, \ldots, n\}$ ist die Funktion* $\textsc{Jump}_m \colon \{0,1\}^n \to \mathbb{R}$ *durch*

$$\textsc{Jump}_m(x) := \begin{cases} m + \|x\|_1 & \textit{falls } \|x\|_1 \leq n - m \textit{ oder } \|x\|_1 = n \\ n - \|x\|_1 & \textit{sonst} \end{cases}$$

definiert.

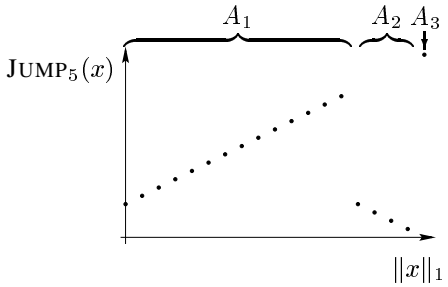

Abbildung 2: Veranschaulichung der Definition 2

Die Funktion $\textsc{Jump}_m$ hängt offenbar nur von der Anzahl der Einsen in x und nicht von x selbst ab. Darum stellen wir zur Veranschaulichung die Funktion in Abhängigkeit von der Anzahl der Einsen dar. Wir erkennen drei Bereiche, die mit A_1, A_2 und A_3 gekennzeichnet sind. In A_1 wächst die Funktion streng monoton mit der Anzahl der Einsen. In A_2 fällt sie hingegen streng monoton mit der Anzahl der Einsen; außerdem hat jeder Punkt $x \in A_2$ kleineren Funktionswert als jeder Punkt $x' \in A_1$. Das globale Optimum allein ist A_3. Weil sich $\textsc{Jump}_m$ in A_1 und A_2 wie eine lineare Funktion verhält, ist klar, dass der (1+1) EA diese Bereiche im Durchschnitt in Zeit $O(n \log n)$ durchschreitet. Der (1+1) EA findet also bei dieser Funktion schnell einen Bitstring x mit genau $n - m$ Einsen, also einen Punkt am rechten Rand von A_1. Bitstrings dieser Art sind lokal optimal. Um von hier aus zum globalen Optimum zu kommen, müssen genau die m Bits mit Wert 0 gleichzeitig mutieren. So eine Mutation hat Wahrscheinlichkeit $(1/n)^m \cdot (1-1/n)^{n-m}$. Darum beträgt die erwartete Zeit bis zum ersten Eintreffen einer solchen Mutation $O(n^m)$ Schritte. Eine sorgfältige Formalisierung dieser Argumentation beweist den folgenden Satz.

Satz 2. *Die erwartete Laufzeit des (1+1) EA auf der Funktion* $\textsc{Jump}_m : \{0,1\}^n \to \mathbb{R}$ *beträgt* $\Theta\left(n^m + n \log n\right)$.

Es gibt verschiedene Crossover-Operatoren, die alle aus zwei Bitstrings $x, y \in \{0,1\}^n$, den „Eltern“, einen neuen Bitstring $z \in \{0,1\}^n$, das „Kind“, erzeugen. Wir konzentrieren uns hier auf das so genannte Uniform Crossover, bei dem für jedes der n Bits in z ein Zufallsexperiment durchgeführt wird und mit Wahrscheinlichkeit jeweils $1/2$ das Bit entweder aus x oder aus y kopiert wird. Wie kann Uniform Crossover bei der Optimierung von $\textsc{Jump}_m$ hilfreich sein? Wir stellen uns den Parameter m jetzt sehr klein im Vergleich zu n vor. Ein EA mit Mutation und Crossover wird das lokale Optimum bei $n - m$ Einsen ebenfalls rasch finden. Dann besteht die Population aus vielen Strings, die genau $n - m$ Einsen enthalten, also sehr viele Einsen und nur wenige Nullen. Da alle diese Strings gleichen Funktionswert haben, erfolgt die Auswahl zum Crossover unter diesen Strings rein zufällig. Wenn sie alle unterschiedlich sind, stehen die Chancen sehr gut, zwei Strings zu wählen, die ihre insgesamt $2m$ Nullen an insgesamt $2m$ verschiedenen Positionen haben. Die Wahrscheinlichkeit, mit Uniform Crossover einen String nur mit Einsen zu erzeugen, beträgt dann $(1/2)^{2m}$. Weil m klein ist, ist die erwartete Wartezeit auch nicht zu groß.
Allerdings besteht kein Grund anzunehmen, dass die Strings der Population in dieser Situation sehr verschieden sind. Sie teilen ihre Vergangenheit, sind aus gemeinsamen Eltern hervorgegangen. Muss muss also zumindest prinzipiell damit rechnen, dass die Strings sich sehr ähnlich sind. Dann bringt Uniform Crossover kaum etwas. Aber das Problem ist lösbar. Wir beschreiben jetzt zunächst formal einen speziellen evolutionären Algorithmus, den Experten „Steady State GA“ nennen. Damit bezeichnet man Algorithmen mit einer Population, Mutation und Crossover, bei denen in jeder Generation wie beim (1+1) EA nur ein Kind erzeugt wird. Diese Algorithmen haben für uns den Vorteil, etwas besser analysierbar zu sein als andere, kompliziertere evolutionäre Algorithmen.

Algorithmus 3 (Steady State GA)

1. Wähle $x_1, \ldots, x_n \in \{0,1\}^n$ zufällig gleichverteilt.
2. Mit Wahrscheinlichkeit p_c weiter bei 4.
3. Wähle x zufällig aus $x_1, x_2, \ldots, x_n$. Erzeuge z aus x durch Mutation. Weiter bei 5.
4. Wähle x, y unabhängig, zufällig aus $x_1, x_2, \ldots, x_n$. Erzeuge z' durch Uniform Crossover aus x und y. Erzeuge z aus z' durch Mutation.
5. Füge z zur Population $x_1, x_2, \ldots, x_n$ hinzu.
6. Wähle zufällig einen Bitstring aus der Population mit minimalem Funktionswert und entferne ihn.
7. Weiter bei 2.

Dieser Steady State GA mit einer Population der Größe n löst das oben angesprochene Problem der eventuell zu geringen Diversität in der Population. Wenn die Crossoverwahrscheinlichkeit p_c recht klein ist, kann man beweisen, dass die Effekte der Mutationen ausreichen, um mit großer Wahrscheinlichkeit zu gewährleisten, dass die Mitglieder der Population so unterschiedlich sind, dass Uniform Crossover am lokalen Optimum mit Wahrscheinlichkeit $(1/2)^{2m+1}$ das globale Optimum erzeugt. Das führt zu folgendem Ergebnis.

Satz 3. *Für $m = O(\log n)$ und jede Wahl der Mutationswahrscheinlichkeit p_c mit $p_c \leq 1/(mn\log n)$ gilt auf* JUMP_m *für den Steady State GA* $E(T) = O\left(n^3m^3 + 2^{2m}/p_c\right)$.

6 Zusammenfassung und Ausblick

Wir haben gezeigt, wie man evolutionäre Algorithmen theoretisch analysieren und dadurch zu interessanten Aussagen über die erwartete Laufzeit kommen kann. Ausgehend von dem sehr einfachen (1+1) EA kann man sich so das weite Feld der EA erschließen. Kennzeichnend für den Ansatz ist der exemplarische Nachweis von wichtigen Aussagen mit Hilfe von künstlichen und konstruierten Zielfunktionen. Diese Beispielfunktionen haben den Vorteil, dass sie wohl strukturiert und gut verständlich sind und dadurch den betrachteten Effekt besonders deutlich und gut einsehbar machen. Sie stehen aber immer in Gefahr, als „nur künstlich“ abgelehnt zu werden mit dem Verdacht, dass evolutionäre Algorithmen sich auf „natürlichen Problemen“ ganz anders verhalten. Es ist darum die vielleicht wichtigste Aufgabe theoretischer Forschung auf dem Gebiet evolutionärer Algorithmen, Ergebnisse über die erwartete Laufzeit auf Problemen von praktischer Relevanz zu beweisen.

Literaturverzeichnis

[Jan00] Jansen, T.: Theoretische Analyse evolutionärer Algorithmen unter dem Aspekt der Optimierung in diskreten Suchräumen. Dissertation, Universität Dortmund, 2000. Online erhältlich unter http://eldorado.uni-dortmund.de/FB4/ls2/forschung/2000/Jansen.

[KGV83] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P.: Optimization by Simulated Annealing. In Science, Bd. 220:(1983), S. 671–680.

[MRR+53] Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; Teller, E.: Equation of state calculation by fast computing machines. In Journal of Chemical Physics, Bd. 21:(1953), S. 1087–1092.

[WM97] Wolpert, D. H.; Macready, W. G.: No Free Lunch Theorem for Optimization. In IEEE Transactions on Evolutionary Computation, Bd. 1 (1):(1997), S. 67–82.

Thomas Jansen geboren am 20. November 1969 in Recklinghausen, Abitur 1989, Diplom in Informatik an der Universität Dortmund 1996, 1990–1996 Stipendiat der Studienstiftung des deutschen Volkes, 1997–2001 wissenschaftlicher Mitarbeiter am Fachbereich Informatik der Universität Dortmund, Promotion am Fachbereich Informatik der Universität Dortmund 2000, 09/2001–08/2002 Postdoc-Stipendiat des DAAD an der George Mason University (Fairfax, VA), verheiratet seit 1993, drei Kinder, wohnhaft in Recklinghausen.