Zu Analyse und Entwurf evolutionärer Algorithmen

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Universität Dortmund

am Fachbereich Informatik

von

Stefan Droste

Dortmund

2000

Tag der mündlichen Prüfung: 29. November 2000

Dekan:	Prof. Dr. Bernd Reusch
Gutachter:	Prof. Dr. Ingo Wegener
	Prof. Dr. Heinrich Müller

Hiermit möchte ich mich bei Ingo Wegener für seine sehr gute Betreuung und Zusammenarbeit bedanken. Ich habe nicht nur vieles über die Analyse evolutionärer Algorithmen, sondern wohl noch mehr über wissenschaftliches Arbeiten bei ihm gelernt.

Thomas Jansen danke ich für viele Diskussionen und die Möglichkeit, gemeinsam offene Probleme und neue Lösungsansätze jederzeit auch ganz ins Blaue hinein anzugehen. Bei Dirk Wiesmann bedanke ich mich für manches Gespräch, das mir andere, neue Sichtweisen auf evolutionäre Algorithmen offengelegt hat.

Beate Bollig bin ich für viele Hinweise und aufmunternde Worte, wenn es einmal nur schlecht voranging, dankbar. Meinen Eltern danke ich für ihre Unterstützung und ihr Verständnis.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Arbeit werden Ergebnisse zu Analyse und Entwurf evolutionärer Algorithmen vorgestellt. Deshalb werden in diesem ersten Kapitel die Merkmale evolutionärer Algorithmen besprochen, die in den folgenden Kapiteln behandelten Fragestellungen motiviert, die wichtigsten Klassen evolutionärer Algorithmen vorgestellt und es wird zusammengefasst, wie sich die Resultate dieser Arbeit in Veröffentlichungen wiederfinden.

1.1 Motivation und Themenübersicht

Unter *evolutionären Algorithmen* (*EA*) verstehen wir randomisierte Heuristiken, die Suchprobleme näherungsweise durch vereinfachende algorithmische Umsetzung von Prinzipien der natürlichen Evolution zu lösen versuchen. Somit geben evolutionäre Algorithmen in der Regel weder eine Garantie bzgl. der benötigten Rechenzeit noch der Güte der ausgegebenen Lösung. Ein Suchproblem besteht darin, zu einer *Zielfunktion* ein Element aus deren Definitionsbereich zu finden, dessen Funktionswert möglichst gut ist. Darunter verstehen wir im Folgenden, wenn nicht ausdrücklich anders vermerkt, einen möglichst großen Funktionswert, weshalb der Zielfunktionswert eines Elements auch als seine *Fitness* bezeichnet wird.

Der Aufbau eines evolutionären Algorithmus lässt sich dann grob wie folgt beschreiben: in jedem Schritt verwaltet er eine Menge fester Größe von Suchpunkten, die so genannte *Population*, wobei jeder einzelne Suchpunkt auch als *Individuum* bezeichnet wird. Aus den Punkten der Population neue Punkte zu erzeugen, ist Aufgabe von *Mutation* und *Rekombination*. Dabei steht hinter der Mutation die Idee, jeweils nur ein einzelnes Individuum zufällig zu verändern, ohne dass andere Individuen dabei berücksichtigt werden. Durch Rekombination wird hingegen aus mehreren, meist zwei Individuen zufällig ein neues gebildet, das von diesen möglichst gute Eigenschaften übernehmen soll. Durch Mutation und Rekombination werden also neue Individuen (*Kinder* genannt) aus bestehenden Individuen (*Eltern* genannt) erzeugt. Beide Operatoren hängen oftmals stark von Zufallsentscheidungen ab. Jedoch fließt in der Regel weder in Mutation noch Rekombination der Zielfunktionswert der Individuen ein.

Die Zielfunktion beeinflusst nur die *Selektion*. Dieser Operator wählt Individuen der Population aus, sei es zur Auswahl der Eltern für eine Rekombination oder Mutation oder, um aus der Menge von Eltern und Kindern die nächste Population zu wählen, was den Übergang zur nächsten *Generation* darstellt. Dadurch, dass die Selektion Punkte mit höherem Zielfunktionswert mit größerer Wahrscheinlichkeit auswählt, soll erreicht werden, dass nach und nach immer bessere Punkte gefunden werden.

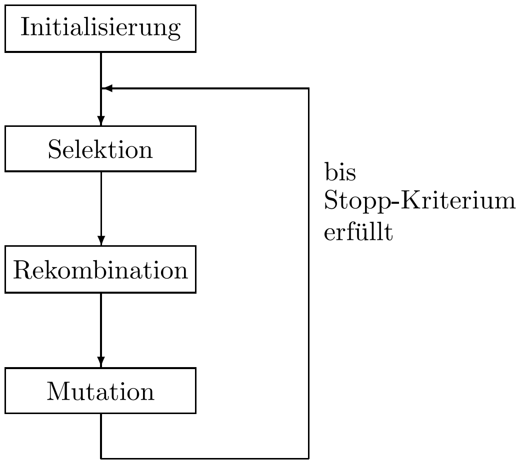

Abbildung 1.1: Schematischer Aufbau eines evolutionären Algorithmus.

Diese Abfolge der *genetischen Operatoren* Selektion, Mutation und Rekombination wird über viele Generationen hinweg wiederholt, bis ein Stopp-Kriterium den Prozess beendet und ein Individuum mit dem größten Zielfunktionswert, welches in einer der Generationen gefunden wurde, ausgegeben wird. Zu Beginn des evolutionären Algorithmus steht natürlich die *Initialisierung* der ersten Population, die häufig durch *uniforme*, d. h. gleichverteilte Auswahl der Individuen aus dem Suchraum erfolgt. Graphisch kann der Aufbau eines evolutionären Algorithmus also ganz grob wie in Abbildung 1.1 dargestellt werden.

Schon diese bewusst kurz gehaltene Übersicht über die wichtigsten Prinzipien evolutionärer Algorithmen zeigt, dass diese eine sehr große Klasse von Suchverfahren bilden. Denn für die algorithmische Umsetzung der genetischen Operatoren gibt es aufgrund ihrer Unspezifiziertheit eine Vielzahl von Wahlmöglichkeiten. Dafür spielt die Wahl der Repräsentation der Individuen zusätzlich eine entscheidende Rolle. Auch besteht keine Einigkeit darüber, welche genetischen Operatoren in einem Algorithmus auftauchen müssen, um ihn evolutionär nennen zu können. Dies hat zur Folge, dass jeder Algorithmus, der mindestens eines der obigen Prinzipien berücksichtigt, als evolutionärer Algorithmus bezeichnet werden kann.

Diese große Zahl von Umsetzungsmöglichkeiten der Prinzipien ist zusammen mit ihrer großen Anschaulichkeit mitverantwortlich für den Erfolg evolutionärer Algorithmen in praktischen Anwendungen. Denn zu einem gegebenen Problem lässt sich sehr schnell ein erster einfacher evolutionärer Algorithmus entwickeln, ohne große Einsicht in die Struktur des Problems haben zu müssen.

Durch die schnelle Erreichbarkeit lauffähiger und oftmals auch gute Resultate erzielender evolutionärer Algorithmen sind die theoretischen Grundlagen in diesem Bereich zum Teil nur schwach entwickelt. Denn oftmals werden neue und immer kompliziertere Varianten evolutionärer Algorithmen entwickelt und getestet, ohne ein Basiswissen zu haben, welche Zielfunktionstypen von welchen evolutionären Algorithmen erfolgreich optimiert werden können und welche Mechanismen dafür entscheidend sind. Demzufolge gibt es auch keine theoretisch abgesicherten Methoden, wie evolutionäre Algorithmen zu entwerfen sind. Häufig wird ein *Trial-and-Error* Verfahren eingesetzt, nur durch eine Reihe von allein empirisch gestützten Entwurfsregeln begleitet.

In Abschnitt 1.2 werden wir die wichtigsten Klassen evolutionärer Algorithmen, die sich durch verschiedene Ausprägungen des Suchraum und darauf abgestimmter genetischer Operatoren unterscheiden, vorstellen. Dies dient einerseits dazu, durch konkrete Beispiele die bisher doch sehr abstrakte Beschreibung evolutionärer Algorithmen mit mehr Leben zu füllen, aber auch, die verschiedenen Ansätze theoretischer Analysen, die sich zum Teil getrennt für die einzelnen Klassen herausgebildet haben, zu beschreiben. Die dort festgestellten Mängel sind Motivation für einen Großteil dieser Arbeit.

Die leichte Umsetzbarkeit evolutionärer Algorithmen mit oftmals guten Erfolgen hat auch zusammen mit dem Grundgedanken, dass die natürliche Evolution ein überlegenes Optimierprinzip darstellt, zu der Behauptung geführt, dass evolutionäre Algorithmen anderen Suchverfahren überlegen sind, ohne diese Behauptung genau zu formalisieren (Goldberg (1989)). Dass gemittelt über alle Zielfunktionen alle Suchverfahren gleich gut sind, was im so genannten NFL („No-Free-Lunch")-Theorem (Wolpert und Macready (1997)) formalisiert wurde, hat demzufolge zu zum Teil großer Verwunderung geführt.

In Kapitel 2 wird die Aufgabenstellung von Suchverfahren zusammengefasst und das NFL-Theorem mit einem kurzen Beweis wiederholt, um seine grundlegende Aussage und einfache Struktur deutlich zu machen. Ohne an der Korrektheit zweifeln zu wollen, werden wir dann seine Voraussetzung, dass alle Zielfunktionen bei der Optimierung gleichrangig sind, durch komplexitätstheoretische Argumente ad absurdum führen. Dies zeigt zusammen mit einem konkreten Beispiel unter realistischeren Annahmen, dass zumindest kleine Unterschiede verschiedener Suchverfahren bei realistischen Voraussetzungen möglich sind. Eine neue allgemeinere Version des NFL-Theorems und der Nachweis, dass es für ein beliebiges festes Suchverfahren zu jeder schnell optimierbaren wenig verschiedene, jedoch nur langsam optimierbare Zielfunktionen gibt, zeigen, dass Untersuchungen über zu große Klassen von Funktionen und Suchverfahren nicht zu aussagekräftigen Resultaten führen können.

Die Idee, die natürliche Evolution als Vorbild zu nutzen, war entscheidend zur Entwicklung evolutionärer Algorithmen in den späten 50er und 60er Jahren und ist für eine erste anschauliche Beschreibung sehr geeignet. Oftmals wird dieser Gedanke auch so weiterentwickelt, dass man hofft, durch möglichst genaue Modellierung von evolutionären Vorgängen der Natur in evolutionären Algorithmen mittels der Ergebnisse dieser Algorithmen Rückschlüsse auf die natürliche Evolution ziehen zu können. Dieser Ansatz, neue Erkenntnisse aus der Biologie in evolutionäre Algorithmen einfließen zu lassen und umgekehrt aus ihnen auf die natürliche Evolution zu schließen, wird hier bewusst nicht verfolgt.

Vielmehr werden wir evolutionäre Algorithmen als eine Teilklasse von randomisierten Suchheuristiken betrachten. Damit sind evolutionäre Algorithmen algorithmische Beschreibungen stochastischer Prozesse. Diese Sichtweise impliziert, dass evolutionäre Algorithmen auch nur vollständig durch mathematische Methoden zu analysieren sind. Dabei wird die Möglichkeit, durch wiederholte Testläufe Daten über das Verhalten des evolutionären Algorithmus zu erhalten, d. h. die experimentelle Analyse, hier nicht als im Prinzip irreführend bezeichnet.

Im Gegenteil, sie wird von uns auch in manchen Teilen durchgeführt, wo sich der betrachtete evolutionäre Algorithmus als für eine mathematische Analyse zu kompliziert erweist. Jedoch kann eine rein experimentelle Analyse nur Gewissheit über die durchgeführten Läufe geben, alle weiteren Läufe können schon aufgrund der stochastischen Natur des evolutionären Algorithmus andere Resultate liefern. Eine weitergehende Gewissheit kann in diesem Fall natürlich mit statistischen Methoden gewonnen werden, womit Sicherheiten bestimmt werden können, mit denen das beobachtete Verhalten auftreten wird. Die theoretische Analyse ohne unbewiesene Annahmen kann jedoch als Einzige sichere Ergebnisse über das generelle Verhalten eines evolutionären Algorithmus liefern.

Ein sehr einfacher, dabei jedoch wichtige Prinzipien verkörpernder evolutionärer Algorithmus, der so genannte (1+1) EA, und Variationen von ihm werden in Kapitel 3 mit mathematischen Methoden analysiert. Die Einschränkung auf einen solch einfachen Algorithmus und genau definierte Klassen von Zielfunktionen begründet sich aus den Ergebnissen aus Kapitel 2, wonach über zu allgemeine Klassen von Zielfunktionen keine sinnvollen Aussagen gemacht werden können. Schon die Analysen dieser „einfachen“ Probleme werden oftmals schwierige Fragestellungen aufwerfen und allgemeiner anwendbare Prinzipien offenbaren.

Dabei wird primär die erwartete Laufzeit, die der (1+1) EA bis zum Erreichen eines Optimums braucht, für verschiedene Zielfunktionen untersucht, wobei oftmals auch die Wahrscheinlichkeit, mit der eine bestimmte Laufzeit benötigt wird, abgeschätzt wird. Diese Größen werden abhängig von der Dimension des Suchraums untersucht, wobei davon unabhängige Konstanten oftmals vernachlässigt werden. Dies spiegelt die Überzeugung wider, dass das qualitative Verhalten eines Algorithmus von der Größenordnung, mit der die Eingabelänge, in diesem Fall die Dimension des Suchraums, in die Laufzeit einfließt, beschrieben wird. Dies ist in der Algorithmenanalyse gängige Praxis (Motwani und Raghavan (1995)). Hauptaugenmerk ist dabei der mathematisch korrekte Nachweis, ob alle Zielfunktionen einer Klasse effizient, d. h. in polynomieller Laufzeit (im Erwartungsfall oder mit hoher Wahrscheinlichkeit) optimiert werden können oder ob sie Funktionen enthalten, die exponentielle Laufzeit haben. Dabei werden zum ersten Mal lange gehegte Vermutungen mathematisch genau bestätigt (für die Klasse der linearen Zielfunktionen) oder als falsch nachgewiesen (für die Klasse der unimodalen Zielfunktionen).

Ein zweiter Schwerpunkt ist der Versuch einer Konkretisierung der Prinzipien evolutionärer Algorithmen, um den Entwurf auf ein Problem abgestimmter evolutionärer Algorithmen zu erleichtern. Der zu Beginn besprochene Aufbau evolutionärer Algorithmen macht durch seine Allgemeinheit eine schnelle Anwendung zwar einfach, doch ist zur Anpassung an komplexe Probleme oft eine aufwendige Veränderung seiner Parameter oder Operatoren notwendig. Um diesen Prozess zu vereinfachen, werden in Kapitel 4 formal genau definierte Richtlinien für gewünschte Auswirkungen der genetischen Operatoren angegeben. In diese muss nach den Ergebnissen von Kapitel 2 die Zielfunktion einfließen, um überdurchschnittliche Leistungen des Algorithmus nicht dem glücklichen Zusammenpassen von Algorithmus und Zielfunktion zu überlassen.

Da die Zielfunktion nur anhand einer Metrik auf dem Suchraum in die genetischen Operatoren einfließt, werden diese Richtlinien erfüllende evolutionäre Algorithmen *Metrik-basierte evolutionäre Algorithmen* (*MBEA*) genannt. Die vorgeschlagenen Richtlinien sind um ihrer leichteren Anwendbarkeit wegen aber noch zu allgemein, als dass ihre Auswirkungen hier theoretisch analysiert werden können. Deshalb werden wir sie nur anhand von Experimenten untersuchen.

Dies geschieht für eine spezielle Ausprägung evolutionärer Algorithmen, der *genetischen Programmierung* (*GP*). Diese hat das entscheidende Charakteristikum, dass die Elemente des Suchraums als Programme interpretiert werden. Hier betrachtete GP-Systeme versuchen, ein unbekanntes Programm nur anhand einer implizit gegebenen vollständigen bzw. unvollständigen Trainingsmenge zu lernen. Dass aus Effizienzgründen die Individuen in der genetischen Programmierung durch Datenstrukturen variabler Länge, wie Bäume, Listen oder Graphen, repräsentiert werden, erschwert eine mathematische Analyse, so dass im Gebiet der genetischen Programmierung theoretische Grundlagen kaum vorhanden sind.

Da die vorgeschlagenen MBEA-Richtlinien gerade bei GP-Systemen oft nicht eingehalten werden, wird in Kapitel 4 ein GP-System nach diesen Richtlinien entworfen und seine Leistung auf Benchmark-Problemen mit Prototypen gängiger GP-Systeme verglichen. Dabei wird dieses GP-System zur Repräsentation so genannte *Ordered Binary Decision Diagrams* (*OBDDs*) verwenden. Diese sind eine effiziente Datenstruktur für boolesche Funktionen und finden, auch weil sie theoretisch gut verstanden sind, z. B. im CAD-Bereich große Verwendung. Anhand der empirischen

Resultate wird sich die Nützlichkeit der MBEA-Richtlinien für die betrachteten Probleme erweisen. Diese geben somit einen möglichen Weg an, wie unabdingbares Problemwissen gewinnbringend in den Entwurf evolutionärer Algorithmen einfließen kann, ohne an Anschaulichkeit zu verlieren.

Ist die Trainingsmenge unvollständig, so steht die Approximation der unbekannten Funktion auf den Elementen, die nicht in der Trainingsmenge sind, im Vordergrund, d. h. die Bestimmung einer gut *generalisierenden* Funktion. Oftmals wird dazu das einfachste Programm gesucht, das die Trainingsdaten korrekt wiedergibt, in der Hoffnung, dass die Umsetzung dieses *Occam's Razor*-Prinzip eine gute Generalisierung liefert. In der genetischen Programmierung wird dieses Verfahren auch häufig eingesetzt, ohne seine Nützlichkeit durch über Experimente hinausgehende Resultate belegt zu haben (siehe z. B. Zhang und Joung (1999)). In Kapitel 5 wird durch eine Übertragung des so genannten Occam's Razor-Theorems von Blumer, Ehrenfeucht, Haussler und Warmuth (1990) zum ersten Mal gezeigt, wie die Generalisierungsgüte der von GP-Systemen gelieferten Lösungen theoretisch garantiert werden kann. Experimente zeigen, dass diese theoretischen Resultate zu nichttrivialen Gütegarantien der von GP-Systemen gelieferten Lösungen führen können.

Mit einer Zusammenfassung der vorgestellten Ergebnisse und einem Ausblick auf sich ergebende Fragestellungen und mögliche Weiterführungen endet diese Arbeit. Im Anhang sind die wichtigsten mathematischen Begriffe, soweit sie für diese Arbeit relevant sind, zusammengestellt.

1.2 Übersicht evolutionärer Algorithmen

In diesem Abschnitt wird der Aufbau evolutionärer Algorithmen präzisiert, indem die zwei wichtigsten Grundformen evolutionärer Algorithmen vorgestellt werden. Für eine wesentlich genauere Übersicht siehe Bäck, Hammel und Schwefel (1997) oder sehr detailliert Bäck, Fogel und Michalewicz (1997). Dabei sind die besprochenen Grundformen in heutigen evolutionären Algorithmen oftmals vermischt. Jedoch stellen sie die wichtigsten Anhaltspunkte dar, an denen sich der Entwurf und die Analyse evolutionärer Algorithmen orientiert. Weiterhin werden auch davon unabhängige Ansätze zur Analyse vorgestellt, um die Ergebnisse zur theoretischen Analyse in dieser Arbeit zu motivieren (siehe Eiben und Rudolph (1999) für einen Überblick der Theorie evolutionärer Algorithmen).

Genetische Algorithmen

Genetische Algorithmen (*GA*) (siehe Holland (1975) und in einer erweiterten Neuauflage Holland (1992) oder Goldberg (1989)) sind eines wichtigsten Paradigmen evolutionärer Algorithmen. In ihnen sind folgende Formen der Repräsentation der Individuen und daran angepasster genetischer Operatoren gebräuchlich:

- **Repräsentation:** Die Individuen sind Bitstrings einer festen Länge n, d. h. Elemente der Menge $\{0,1\}^n$.
- **Selektion:** Zur Selektion der Individuen einer Population, die als Eltern für Mutation bzw. Rekombination benutzt werden, wird vorrangig die *fitness-proportionale* Selektion verwendet. Ist also F die zu maximierende positiv-reellwertige Zielfunktion, so wird aus der Population $P = \{s_1, \ldots, s_N\}$ der Suchpunkte aus dem Suchraum S mit Wahrscheinlichkeit

$$\frac{F(s_i)}{\sum_{j=1}^{N} F(s_j)}$$

das Individuum s_i gewählt.

- **Mutation:** Die Mutation besteht darin, jedes Bit des mutierenden Strings mit einer Wahrscheinlichkeit $p_m(n)$ jeweils unabhängig zu negieren (*bit-weise Mutation*). Der so gebildete String bildet dann das mutierte Individuum. Gebräuchlich für $p_m(n)$ sind Werte in der Größenordnung von $1/n$ (Bäck (1993)).

- **Rekombination:** Beim *k-Punkt Crossover* werden k paarweise verschiedene Indizes $i_1, \ldots, i_k \in \{1, \ldots, n-1\}$ zufällig gewählt, die aufsteigend sortiert als „Trennstellen" dienen, an denen die Eltern x und y aufgeteilt und zu z zusammengefügt werden. Dies bedeutet, dass von links nach rechts für die Indizes bis einschließlich i_1 jeweils $z_i = x_i$ gesetzt wird, dann $z_i = y_i$ bis einschließlich i_2, dann wieder $z_i = x_i$ bis einschließlich i_3, usw. Formal bedeutet dies:

$$\forall i \in \{1, \ldots, n\}: \; z_i := \begin{cases} x_i & \text{, falls } |\{j \in \{1, \ldots, k\} \,|\, i_j \leq i\}| \text{ gerade,} \\ y_i & \text{, falls } |\{j \in \{1, \ldots, k\} \,|\, i_j \leq i\}| \text{ ungerade.} \end{cases}$$

  Beim *uniformen Crossover* wird aus den Eltern x und y ein Nachkomme z gebildet, indem für jeden Index $i \in \{1, \ldots, n\}$ die Stelle z_i mit Wahrscheinlichkeit $1/2$ als x_i oder als y_i gewählt wird. Somit wird z für Indizes, an denen x und y übereinstimmen, den gleichen Wert annehmen und für alle anderen Stellen unabhängig voneinander den Wert des ersten oder des zweiten Elternteils.

  Für beide Rekombinationsarten wird auch häufig die Variante benutzt, dass statt eines Kindes zwei Kinder z^1 und z^2 gebildet werden, wobei sich das zweite durch die komplementären Entscheidungen des ersten ergibt, d. h. für alle $i \in \{1, \ldots, n\}$ gilt jeweils $\{x_i, y_i\} = \{z_i^1, z_i^2\}$.

- **Gesamtablauf:** Zur Erzeugung des nächsten Kindes wird zufällig entschieden, ob dieses per Mutation oder Rekombination geschieht, wobei die Mutation eine oftmals sehr geringe Wahrscheinlichkeit hat oder gar nicht vorkommt. Dann werden ein bzw. zwei Elternteile fitness-proportional ausgewählt, Mutation bzw. Rekombination angewandt und der erzeugte Nachkomme in die Kinderpopulation aufgenommen. Dies wird wiederholt, bis deren Anzahl gleich der der Eltern ist. Daraufhin wird die Elternpopulation durch die der Kinder ersetzt und die Prozedur wiederholt, bis ein Stopp-Kriterium zutrifft. Die Selektion greift bei genetischen Algorithmen also nur zur Auswahl der Eltern ein; die erzeugten Kinder werden die Eltern stets komplett ersetzen, ohne von der Zielfunktion beeinflusst zu werden.

Man sieht an dieser Stelle gut, wie die genetischen Operatoren auf die Darstellungsform abgestimmt sind und die anschaulich an sie gestellten Vorstellungen auf dieser Ebene erfüllen: misst man den Unterschied zweier Bitstrings durch ihren *Hamming-Abstand*, d. h. die Zahl der Stellen, an denen sie sich unterscheiden, so wird ein mutierter String von seinem Elter bei $p_m(n) = 1/n$ nur an durchschnittlich einer Stelle abweichen und mit hoher Wahrscheinlichkeit eine große Nähe zu seinem Elter aufweisen. Ebenso wird bei der Rekombination jedes Kind mit seinen Eltern an den Stellen übereinstimmen, an denen diese es tun. Somit wird ein Kind von keinem seiner Eltern weiter entfernt sein als diese voneinander.

Ob sich die Tatsache, dass die anschaulichen Vorstellungen von Mutation und Rekombination erfüllt sind, auch für den Optimierprozess günstig auswirkt, hängt natürlich von der zu optimierenden Funktion ab. Der grundlegende, schon in Holland (1975) eingeführte Erklärungsversuch der Funktionsweise genetischer Algorithmen ist das so genannte *Schema-Theorem*. Ein *Schema* ist dabei ein nur partiell definierter String $a \in \{0, 1, \star\}^n$. Damit ist die Menge $s(a) \subseteq \{0, 1\}^n$ aller Bitstrings, die zu diesem Schema passen, verbunden:

$$s(a) := \{x \in \{0, 1\}^n \,|\, \forall i \in \{1, \ldots, n\} \text{ mit } a_i \neq \star : x_i = a_i\}.$$

Der Zielfunktionswert eines Schemas a ist gleich dem Durchschnitt der Zielfunktionswerte der zu dem Schema passenden Bitstrings aus $s(a)$ in der Population. Das Schema-Theorem besagt nun, ohne auf die genaue Formel einzugehen (Holland (1992)), dass der erwartete Anteil eines Schemas mit überdurchschnittlich hohem Zielfunktionswert (auf alle Individuen der Population bezogen) beim Übergang von einer Population zur Nachfolgepopulation in genetischen Algorithmen um einen konstanten Faktor gegenüber dem bisherigen Anteil wächst. Dies liegt daran, dass zu einem solchen Schema passende Eltern von der fitness-proportionalen Selektion überdurchschnittlich häufig gewählt werden und ihre Kinder nur dann nicht zu demselben Schema passen, wenn Rekombination bzw. Mutation störend wirken. Für 1-Punkt-Crossover lässt sich diese Störung auf Schemata a, deren definierte Stellen nicht zu weit auseinander stehen, gut genug nach oben abschätzen, ebenso für die bit-weise Mutation, wenn nicht zuviele Stellen in a als 0 oder 1 definiert sind.

Dieses Schema-Theorem wird oftmals fälschlicherweise dahingehend verallgemeinert, dass sich die Zahl der Vertreter guter Schemata exponentiell mit der Zahl der Generationen erhöht. Dies gilt für ein Schema aber nur, wenn seine Fitness in allen betrachteten Generationen um einen konstanten Faktor höher als der Durchschnitt der Fitness aller Individuen der Population ist. Da sich jedoch durch das erhöhte Auftreten eines überdurchschnittlich guten Schemas die durchschnittliche Fitness der Population erhöht, muss ein jetzt überdurchschnittliches Schema dies in ein paar Generationen nicht mehr sein. Eine Verallgemeinerung auf mehrere Generationen ist also ohne zusätzliche Annahmen nicht möglich.

Ein weiterer Kritikpunkt, insbesondere was die Aussagekraft für den Entwurf genetischer Algorithmen betrifft, ist der rein zerstörerische Einfluss von Mutation und Rekombination in die Abschätzung des Schema-Theorems. Denn der Fall, dass durch Mutation und Rekombination neue Schemata mit hohem Zielfunktionswert entstehen können, wird nicht betrachtet. Um den Zuwachs von Vertretern guter Schemata zu maximieren, müsste man nach dieser Abschätzung weder Rekombination noch Mutation verwenden. Wie wenig das Schema-Theorem trotz seines „Fundamentcharakters“ für genetische Algorithmen zu konkreten Berechnungen oder Abschätzungen benutzt wurde, zeigt, dass erst in Menke (1997) ein Fehler in der in Holland (1975) veröffentlichten Form des Schema-Theorems gefunden wurde.

Den in vielen Experimenten beobachteten positiven Effekt der Rekombination erklären soll die *Building-Block Hypothese* (Goldberg (1989)). Danach wird ein genetischer Algorithmus gut arbeiten, wenn durch Crossover aus Vertretern überdurchschnittlich guter Schemata mit hoher Wahrscheinlichkeit Vertreter noch besserer Schemata erzeugt werden. Eine formale Beschreibung der Funktionen, für die dies gilt, fehlt aber. Selbst die so genannten *Royal-Road*-Funktionen (Mitchell, Forrest und Holland (1992)), die speziell so konstruiert wurden, dass sich durch den Austausch guter Schemata noch bessere Individuen bilden, um somit die Korrektheit der Building-Block Hypothese zu stützen, tun dies bei genauerer Untersuchung nicht: ein einfacher rein mutationsbasierter Ansatz ist besser als eine Variante mit Rekombination (siehe Forrest und Mitchell (1993)). Dies zeigt, wie schwierig es ist, solch allgemeine Vermutungen wie die Building-Block-Hypothese auch nur an einem Beispiel zu stützen.

Evolutionsstrategien

Eine zweites Hauptparadigma evolutionärer Algorithmen bilden die *Evolutionsstrategien* (*ES*) (Rechenberg (1994) und Schwefel (1995)). Wie bei genetischen Algorithmen seien die häufigsten Ausprägungen von Repräsentation, Mutation, Rekombination und Selektion kurz zusammengefasst. Da in dieser Arbeit nicht auf Selbst-Adaptation eingegangen wird, werden die entsprechenden Varianten der genetischen

Operatoren dafür hier nicht besprochen, um die Beschreibung möglichst übersichtlich zu lassen:

- **Repräsentation:** Die Individuen sind reelle Vektoren einer festen Länge n, d. h. Elemente der Menge $\mathbb{R}^n$.
- **Selektion:** In Evolutionsstrategien wird in der Regel entweder die so genannte $(\mu + \lambda)$- oder die (μ, λ)-Selektion verwendet. Dabei bezeichnet $\mu \in \mathbb{N}$ die Größe der Elternpopulation, während $\lambda \in \mathbb{N}$ die Zahl der daraus erzeugten Nachkommen ist. Beide Selektionsarten wählen die μ Individuen, die die nächste Population bilden, als diejenigen mit den höchsten Zielfunktionswerten aus einer bestimmten Menge aus. Bei der $(\mu+\lambda)$-Selektion ist diese Menge die Vereinigung der μ Eltern und λ Nachkommen, bei der (μ, λ)-Selektion ist es nur die Menge der λ Nachkommen. Im letzteren Fall ist es also notwendig, dass $\lambda \geq \mu$ ist, wobei die Zielfunktion umso stärker einfließt, je größer das Verhältnis λ/μ ist.
- **Mutation:** Die Mutation eines Individuums $s \in \mathbb{R}^n$ erfolgt durch komponentenweise Addition einer jeweils unabhängig gezogenen normalverteilten Zufallsgröße mit Erwartungswert 0 und Standard-Abweichung σ (Krengel (1991)). Dabei sind der Erwartungswert und die Varianz für alle Individuen gleich; letztere wird oftmals über die Zahl der Generationen kleiner, um eine genaue Annäherung an ein Optimum zu ermöglichen.
- **Rekombination:** Für die Rekombination zweier Individuen $x, y \in \mathbb{R}^n$ zum Nachfolger $z \in \mathbb{R}^n$ wird komponentenweise für jeden Index $i \in \{1, \ldots, n\}$ der Wert von z_i aus dem Intervall $[x_i, y_i]$ gewählt (o. B. d. A. sei $x_i < y_i$). Wird z_i jeweils mit Wahrscheinlichkeit $1/2$ als x_i oder y_i gewählt, erhält man die *diskrete Rekombination*, die eine Übertragung des aus genetischen Algorithmen bekannten uniformen Crossovers auf allgemeinere Suchräume darstellt. Auch oft angewandt wird die *intermediäre Rekombination*, in der für z_i das arithmetische oder geometrische (falls $x, y \in (\mathbb{R}^+)^n$) Mittel von x_i und y_i oder gleichverteilt ein Element aus $[x_i, y_i]$ zufällig gewählt wird.
- **Gesamtablauf:** Wie bei genetischen Algorithmen stellt sich der Ablauf einer Evolutionsstrategie als Folge von Mutations- bzw. Rekombinationsoperationen zur Erzeugung der Nachkommen und der anschließenden Selektion, um aus Eltern und Nachkommen die nächste Elternpopulation zu wählen, dar. Von genetischen Algorithmen unterschiedlich ist auf dieser Ebene nur die Selektion: die Eltern, die rekombiniert und danach mutiert werden, werden gleichverteilt aus der Elternpopulation gewählt. Der Einfluss der Zielfunktion erfolgt durch die $(\mu + \lambda)$- bzw. (μ, λ)-Selektion nach der Generierung der Nachkommen.

In Evolutionsstrategien wird traditionell ein starkes Gewicht auf die Mutation gelegt, d. h. in der Regel hat jedes Kind eine Mutation erfahren und ist nur ab und an ursprünglich durch Rekombination erzeugt worden. Für eine Abgrenzung zu genetischen Algorithmen ist in erster Linie die Repräsentation wichtig: die Darstellung mit reellen Vektoren erlaubt z. B. die Verwendung der intermediären Rekombination. Benutzt man im $\mathbb{R}^n$ den euklidischen Abstand als Maß für die Verschiedenheit zweier Individuen, so werden auch hier Mutation und Rekombination die anschaulichen Vorstellungen erfüllen.

Im Bereich der Evolutionsstrategien hat sich kein so grundlegender Erklärungsversuch wie das Schema-Theorem bei genetischen Algorithmen gebildet. Doch hat sich gerade die Analyse von Evolutionsstrategien neben der Untersuchung der globalen Konvergenz gegen ein Optimum (Rudolph (1997)) auf die *lokaler Maße* konzentriert (Beyer und Rudolph (1997)), d. h. von Kenndaten, die sich schon aus der

Betrachtung weniger Generationen ergeben. Die am häufigsten betrachteten lokalen Maße sind der *erwartete Qualitätsgewinn* und der *erwartete Fortschritt*. Diese geben den Erwartungswert der Zunahme des Zielfunktionswerts bzw. der Distanzverringerung zum Optimum des jeweils besten Individuums beim Übergang von einer Generation in die nächste an. Da auch schon die Abschätzung dieser Maße große Schwierigkeiten macht, beschränken sich diesbezügliche theoretische Untersuchungen auf Testfunktionen wie die Kugelfunktion (Rudolph (1997)) oder die *Ridge*-Funktion (Oyman, Beyer und Schwefel (1998))

Häufig werden für diese Analysen vereinfachende Annahmen gemacht, die durch anschließende Vergleiche der theoretischen Vorhersagen mit empirischen Ergebnissen gerechtfertigt werden. Oftmals ist unklar, welche Folgerungen aus den lokalen Maßen für globale Maße, wie die Zahl von Schritten bis zum Erreichen eines Optimums, gezogen werden können, deren größere Aussagekraft unbestritten ist (siehe z. B. ebenfalls Oyman, Beyer und Schwefel (1998)). Die allgemeine Übertragbarkeit lokaler Maße auf globale Eigenschaften zumindest lässt sich an konkreten Beispielen widerlegen (Jansen und Wegener (2000a)).

Andere Analysemethoden

Es gibt noch weitere Paradigmen evolutionärer Algorithmen mit jeweils eigenständigen Repräsentationsformen und darauf abgestimmten genetischen Operatoren. Die beiden wichtigsten sind das *evolutionäre Programmieren* (*EP*) (Fogel (1995)) und die *genetische Programmierung* (*GP*) (Koza (1992)). Beide Paradigmen haben aber keine eigenständigen Theorieansätze gebildet: zum evolutionären Programmieren gibt es nur wenige, denjenigen der ebenfalls mutationsbasierten Evolutionsstrategien recht ähnlich. In der genetischen Programmierung gibt es aufgrund ihrer Entwicklung aus genetischen Algorithmen Übertragungen des Schema-Theorems und der Building-Block-Hypothese (Poli und Langdon (1998)), die somit deren prinzipielle Schwächen übernehmen. In Abschnitt 4.2 wird die genetische Programmierung im Rahmen einer Anwendung der MBEA-Richtlinien detailliert vorgestellt.

Natürlich haben sich auch Analysemethoden unabhängig von einem der besprochenen Algorithmenparadigmen entwickelt. Eine sehr naheliegende und allgemeine Methode ist es, die Zustandsüberführungsmatrix des dem evolutionären Algorithmus zugrundeliegenden *Markoff-Prozesses* zu analysieren. Dabei heißt eine Folge $(Z_t)_{t\in\mathbb{N}_0}$ von Zufallsvariablen ein Markoff-Prozess, wenn die Verteilung von Z_t nur von Z_{t-1}, aber nicht von $Z_{t-2},\ldots,Z_0$ abhängt. Gibt Z_t die Verteilung über die möglichen Populationen des evolutionären Algorithmus zum Zeitpunkt t an, wird $(Z_t)_{t\in\mathbb{N}_0}$ ein Markoff-Prozess sein, da sich die Population eines evolutionären Algorithmus zum Zeitpunkt t nur aus der Population zum Zeitpunkt $t-1$ ergibt. Diese Modellierung ergibt sogar ein lineares System zur Bestimmung der Verteilung von Z_t, da sich der Verteilungsvektor von Z_t durch ein Matrix-Vektor-Produkt von Zustandsüberführungsmatrix und dem Verteilungsvektor von Z_{t-1} ergibt. Dabei wird der Markoff-Prozess $|S|^N$ Zustände haben, wenn N die Populationsgröße ist. Da schon die Markoff-Prozesse einfacher evolutionärer Algorithmen sehr kompliziert werden können, ist diese Modellierung nur für sehr einfache evolutionäre Algorithmen und Funktionen erfolgreich gewesen (siehe Rudolph (1998)). Die Anwendung mathematischer Analysemethoden auf diese Markoff-Prozesse liefert allgemeine Aussagen über Eigenschaften evolutionärer Algorithmen, die jedoch nur schwer für globale Maße wie die Laufzeit konkretisiert werden können (Vose (1999)).

Gibt Z_t den Anteil der verschiedenen Individuen an der Population zum Zeitpunkt t an, so gibt es „nur“ $|S|$ Zustände, jedoch ist das sich in dieser Modellierung ergebende System im Allgemeinen nicht mehr linear, sondern quadratisch in Z_t, da die Rekombination mindestens zwei Individuen verbindet. Solche quadratischen Systeme können jedoch sehr komplex werden und es ist nicht immer klar, wie genau die dabei meistens vorausgesetzten unendlichen Populationen reale evolutionäre Algorithmen mit endlichen Populationen beschreiben (siehe unten).

Ein in Shapiro und Prügel-Bennett (1995) zusammengefasster Ansatz ist die so genannte *statistical mechanics*-Methode. In diesem wird versucht, nur aus Kenntnis der k-ten Momente für kleine k, d. h. Erwartungswert, Varianz, usw., der Verteilung der Fitnesswerte diese für einen evolutionären Algorithmus und eine feste Zielfunktion für die nächste Generation vorherzusagen. Um die Veränderung dieser Größen zu berechnen, werden eine Reihe von Voraussetzungen gemacht: erstens wird angenommen, dass die Population maximale Entropie hat, so dass alle Individuen mit demselben Zielfunktionswert den gleichen Anteil an der Population besitzen. Dass mit Anteilen gerechnet wird, setzt implizit voraus, dass die Population unendlich groß ist. Um aus der Kenntnis der ersten (meist vier) Momente die Form der Population genauer zu bestimmen, wird zusätzlich angenommen, dass die Zielfunktionswerte nach der Gauss'schen Normalverteilung (Krengel (1991)) verteilt sind. Unter diesen Annahmen können Gleichungen aufgestellt werden, die den Übergang von einer Generation zur nächsten beschreiben.

Diese Annahmen sind anschaulich zumindest näherungsweise erfüllt, wenn die Population sehr groß und durch eine häufige Anwendung von Rekombination gut „vermischt" ist. Die Vorhersagen stimmen oftmals für Testfunktionen gut mit Experimenten überein und ermöglichen Aussagen über optimale Parametereinstellungen. Jedoch sind die Ergebnisse oftmals lokaler Natur, eine Übertragung auf globale Maße wie die Laufzeit ist nicht immer möglich.

Ein anderer Ansatz ist die Bestimmung von Eigenschaften, die Funktionen für bestimmte evolutionäre Algorithmen leicht oder schwierig machen. Nach der Bestimmung genügend vieler solcher Eigenschaften hofft man, neue Funktionen bzgl. dieser Eigenschaften charakterisieren zu können und somit deren Schwierigkeit für einen evolutionären Algorithmus vorhersagen zu können (Mitchell, Forrest und Holland (1992)). Jedoch ist es schwierig, Maße zu finden, die die Schwierigkeit bzgl. einer ganzen Klasse von Algorithmen zur Folge haben (Jansen (2000)).

Viele dieser Ansätze haben die Schwäche, dass sie vereinfachende Annahmen machen, den daraus möglicherweise resultierenden Fehler aber nicht abschätzen, sondern nur durch Experimente als oftmals vernachlässigbar überprüfen. Damit teilen sie natürlich die Schwäche rein empirischer Vorhersagen: Experimente können schon für die betrachteten Funktionen und Algorithmen nur statistische Sicherheit geben, Verallgemeinerungen sind nicht theoretisch abgesichert möglich. Die Notwendigkeit vereinfachender Annahmen liegt oftmals darin begründet, dass theoretische Ergebnisse für eine große Klasse von Zielfunktionen und evolutionären Algorithmen erzielt werden sollen. Dass dies jedoch schwierig ist, zeigen Untersuchungen über die Komplexität der von evolutionären Algorithmen durchgeführten Berechnungen.

Allgemeine komplexitätstheoretische Überlegungen

Ein wichtiges Ziel theoretischer Untersuchung von Algorithmen ist es stets, den maximalen Ressourcenverbrauch zu bestimmen, den diese zur Lösung von Problemen einer bestimmten Klasse brauchen. Eine relativ mächtige Methode, dies nachzuweisen, ist es zu zeigen, dass NP-vollständige Probleme in polynomieller Zeit auf ein Problem der betrachteten Klasse reduziert werden können. Unter der Annahme, dass $\mathrm{P} \neq \mathrm{NP}$ ist, folgt dann, dass es Probleme in der Klasse geben muss, für die der Algorithmus exponentielle Rechenzeit benötigt (Garey und Johnson (1979)).

Ein ähnlicher Ansatz, die Berechnungskraft von evolutionären Algorithmen zu bestimmen, wurde von Arora, Rabani und Vazirani (1994) durchgeführt. Dabei spielt die Rekombination eine wichtige Rolle. Denn betrachtet man das Verhalten eines evolutionären Algorithmus im Grenzwert für eine unendlich große Population, so kann die aktuelle Population jeweils durch einen Wahrscheinlichkeitsvektor dargestellt werden, der an seiner i-ten Position den Anteil des i-ten Suchpunkts an der Population angibt. Verwendet der evolutionäre Algorithmus Rekombination, so wird die Zustandsüberführungsfunktion, die den Wahrscheinlichkeitsvektor der t-ten in den der $(t+1)$-ten Generation überführt, quadratisch und meistens symmetrisch (d. h. die Wahrscheinlichkeit, aus zwei Eltern a und b zwei Kinder c und d zu erzeugen, ist genauso gross, wie aus c und d die Punkte a und b zu erzeugen) in den Elementen des Wahrscheinlichkeitsvektors sein. Allgemeine symmetrische quadratische dynamische Systeme (QDS) zu simulieren, ist, so haben Arora, Rabani und Vazirani (1994) gezeigt, PSPACE-vollständig. Unter der Annahme P $\neq$ PSPACE folgt also, dass nicht alle symmetrischen QDS in polynomieller Zeit simuliert werden können.

Für die sich dann stellende Frage, ob die Klasse der evolutionären Algorithmen entsprechenden symmetrischen QDS leichter zu simulieren ist, wurden schon von Rabinovich, Sinclair und Wigderson (1992) positive Hinweise gefunden. Sie konnten zeigen, dass symmetrische QDS unter wenigen Annahmen gegen eine relativ gut zu charakterisierende stationäre Verteilung konvergieren. Trotz ihrer Berechnungskraft gibt es also Ansätze, QDS zu analysieren.

Dies gilt auch für QDS, die sich direkter aus evolutionären Algorithmen herleiten, wie Rabani, Rabinovich und Sinclair (1995) gezeigt haben. Sie geben einerseits durch die Analyse der entsprechenden QDS untere und obere Schranken für die Konvergenzzeit der Wahrscheinlichkeitsverteilungen von rein rekombinationsbasierten Algorithmen gegen die stationäre Verteilung. Andererseits zeigen sie, dass die unrealistische Annahme einer unendlich großen Population fallen gelassen werden kann, da eine in t und der Dimension des Suchraums polynomielle Populationsgröße genügt, damit die Verteilung des rekombinationsbasierten Algorithmus in der t-ten Generation die des entsprechenden QDS auf einen beliebigen konstanten Wert annähert. Somit können zumindest rein rekombinationsbasierten Algorithmen zugrundeliegende QDS effizient approximiert werden.

Insgesamt zeigen diese Ergebnisse, dass die Analyse evolutionärer Algorithmen wesentlich komplexer wird, wenn diese einen Rekombinationsoperator enthalten, weshalb diesbezügliche genaue theoretische Untersuchungen selten sind. Eine Ausnahme stellt Rabinovich und Wigderson (1991) dar, wo unter Annahme einer unendlich großen Population die erwartete Zeit, bis die durchschnittliche Fitness der Population eines genetischen Algorithmus mit uniformem Crossover einen Schwellenwert erreicht, abgeschätzt wird.

Schlussfolgerungen

Die Erklärungen der Arbeitsweise evolutionärer Algorithmen basieren oftmals auf aus der Anschauung entstandenen Vermutungen, die nur zum Teil empirisch belegt sind. Um aber gesicherte Erkenntnisse zu haben, ist eine theoretische Analyse unerlässlich, in der der Fehler getroffener Annahmen abgeschätzt ist. Dies dient auch der leichteren Einsetzbarkeit evolutionärer Algorithmen, da erst diese Analyseergebnisse Sicherheit geben, welche Algorithmentypen für welche Arten von Zielfunktionen sinnvoll sind.

Die theoretischen Untersuchungen evolutionärer Algorithmen in Kapitel 3 werden deshalb vereinfachende Annahmen nur machen, wenn der resultierende Fehler abgeschätzt werden kann. Aufgrund der im vorstehenden Abschnitt beschriebenen Komplexität evolutionärer Algorithmen mit Rekombination werden wir uns auf rein mutationsbasierte evolutionäre Algorithmen einschränken. Dabei werden wir primär den so genannten (1+1) EA und Variationen auf genau definierten Klassen von Zielfunktionen untersuchen.

Die so erreichten Analyseergebnisse werden nur in Teilaspekten Auswirkungen auf den Entwurf evolutionärer Algorithmen in der Praxis haben, da diese oftmals wesentlich komplexer aufgebaut sind als die besprochenen Grundformen. Zwar haben sich für den Entwurf einige heuristische Regeln herausgebildet, doch sind diese weitestgehend unverbunden. Gerade wenn es um den Entwurf von evolutionären Algorithmen für Probleme geht, deren natürliche Repräsentation nicht eine der Standard-Wahlen wie die Mengen $\{0,1\}^n$ oder $\mathbb{R}^n$ ist, muss das Problem entweder erst in diese umkodiert werden, was oftmals schlechte Resultate zur Folge hat, oder an die Repräsentation angepasste genetische Operatoren müssen neu entwickelt werden. Um letzteres zu erleichtern, wird in Kapitel 4 eine Reihe von zusammenhängenden Richtlinien vorgeschlagen, an denen sich der Entwurf von evolutionären Algorithmen orientieren kann.

Zuvor soll jedoch in Kapitel 2 eine Klärung der Grenzen und Möglichkeiten allgemeiner Suchverfahren, wie es auch evolutionäre Algorithmen sind, stehen.

1.3 Überblick über Veröffentlichungen

Die in dieser Arbeit vorgestellten Resultate finden sich in folgenden Veröffentlichungen wieder:

- Die Ergebnisse aus Kapitel 2 zu den Möglichkeiten allgemeiner Suchverfahren sind zuerst teilweise in Droste, Jansen und Wegener (1999) erschienen. Eine ausführlichere, diese erweiternde Arbeit stellt dann Droste, Jansen und Wegener (2000d) dar.

- Die Analyseergebnisse zum (1+1) EA aus Kapitel 3 sind in Droste, Jansen und Wegener (1998a), Droste, Jansen und Wegener (1998b) und Droste, Jansen und Wegener (1998c) erschienen. Eine diese Ergebnisse zum Teil umfassende und wesentlich erweiterte Version stellt Droste, Jansen und Wegener (2000c) dar. Die vergleichende Analyse der statischen und dynamischen Varianten Static EA und Dynamic EA des (1+1) EA bildet einen Teil von Droste, Jansen und Wegener (2000a). Die natürliche Funktion MaxCount und die diesbezüglichen Analysen werden in Droste, Jansen und Wegener (2000b) erläutert.

- Die MBEA-Richtlinien aus Kapitel 4 sind in Droste und Wiesmann (2000a) zum ersten Mal vorgestellt, eine ausführlichere Version stellt Droste und Wiesmann (2000b) dar. Die dort benutzten GP-Systeme basieren zum Teil auf einem in Droste (1997) vorgestellten System.

- Die in Kapitel 5 beschriebene Übertragung des Occam's Razor-Theorems ist in Droste (1998) wiederzufinden.

Kapitel 2

Zu Grenzen und Möglichkeiten allgemeiner Suchverfahren

Evolutionäre Algorithmen sind eine Teilklasse allgemeiner Suchverfahren. Deshalb wird in diesem Kapitel geklärt, welche Art von Algorithmen unter allgemeinen Suchverfahren verstanden wird und wo die Grenzen und Möglichkeiten dieser Algorithmen liegen.

Allgemeine Suchverfahren sind nicht auf eine spezielle Menge von Zielfunktionen abgestimmt. Deshalb ist es sinnvoll, ihre Leistung im Mittel über alle möglichen Zielfunktionen zu betrachten. Das „No-Free-Lunch" (NFL)-Theorem (Wolpert und Macready (1997)) besagt dann, dass im Durchschnitt über alle Zielfunktionen zwischen zwei endlichen Mengen alle allgemeinen Suchverfahren dieselbe Qualität haben. Die darin implizit enthaltene Voraussetzung, dass alle Funktionen gleich relevant sind, werden wir durch komplexitätstheoretische Argumente als unrealistisch kennzeichnen. Darüber hinaus wird anhand eines Beispiels gezeigt, dass in realistischen Szenarien verschiedene Suchverfahren verschiedene Qualität haben können.

Jedoch können wir das NFL-Theorem auch unter der allgemeineren Voraussetzung beweisen, dass die betrachtete Funktionenmenge nur abgeschlossen unter Permutationen ist. Weiterhin werden wir in einem „Almost-No-Free-Lunch" (ANFL)-Theorem zeigen, dass es zu jeder Funktion, die ein allgemeines Suchverfahren effizient optimiert, Funktionen nicht wesentlich größerer Komplexität gibt, die es nicht effizient optimiert. Diese beiden Resultate zeigen deutlich die Grenzen allgemeiner Suchverfahren auf. Deshalb ist eine Analyse von Algorithmen auf zu großen oder zu wenig Struktur enthaltenden Klassen von Funktionen aussichtslos.

2.1 Allgemeine Suchverfahren

Ein zentrales Problem in vielen Bereichen ist die Optimierung: aus einer endlichen Menge S von Lösungen soll ein Element herausgesucht werden, das ein vorgegebenes Kriterium *optimiert*, d. h. mindestens so gut wie alle anderen Lösungen ist. Dieses Kriterium wird durch eine Funktion $f : S \mapsto W$ repräsentiert, wobei W eine geordnete Menge ist. Somit ist $f(s)$ die Güte der Lösung s und das Optimierproblem besteht darin, einen Punkt $s \in S$ mit o. B. d. A. maximalem Funktionswert zu finden. Soll die Lösung mehrere Kriterien gleichzeitig optimieren, führt dies zu nur halb-geordneten Mengen W. Die Problematik der Behandlung solcher Funktionen wird hier nicht betrachtet.

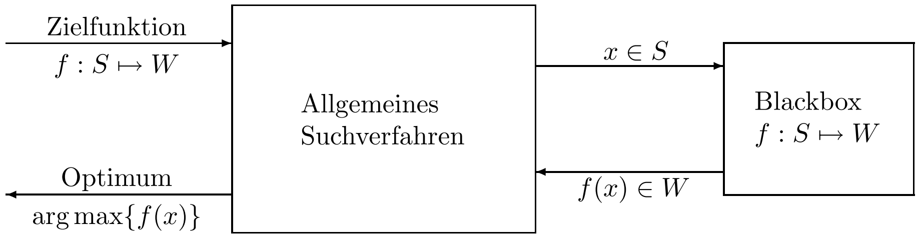

Abbildung 2.1: Ein allgemeines Suchverfahren im Black-Box Szenario.

Zur Untersuchung solcher Optimierprobleme aus Sicht der Informatik ist es notwendig, dass man nicht nur eine zu optimierende Funktion f betrachtet, da in diesem *One-Shot* Szenario ein optimaler Algorithmus ohne jede Berechnung ein optimales $s \in S$ ausgibt. Dieser Algorithmus hat konstanten Ressourcenbedarf, auch wenn es natürlich schwer sein kann, ihn zu finden. Um für die Informatik ein Problem darzustellen, muss das Optimierproblem über eine Menge von Funktionen $f : S \mapsto W$ betrachtet werden.

Suchverfahren sind dann solche Algorithmen, die zu einer Eingabe $f : S \mapsto W$ einen optimalen Punkt $s \in S$ ausgeben, d. h. ein Optimierproblem lösen. Beim Entwurf von Suchverfahren kann man zwei Vorgehensweisen unterscheiden: bei *speziellen* Suchverfahren werden möglichst viele Informationen über die zu optimierenden Funktionen berücksichtigt, um einen Algorithmus mit möglichst geringem Ressourcenverbrauch zu finden. Dies setzt voraus, dass die Klasse der zu optimierenden Funktionen bekannt ist und eine erkennbare Struktur hat. Ein Beispiel hierfür sind Sortieralgorithmen, bei denen die zu optimierenden Funktionen Sortierprobleme repräsentieren (dazu sei z. B. S die Menge aller Permutationen der zu sortierenden Elemente $\{a_1, \ldots, a_n\}$ und der Funktionswert einer Permutation gleich der Anzahl der Paare in ihr, die nicht aufsteigend angeordnet sind). Der Entwurf solcher Algorithmen ist Thema des Forschungsgebiets der *effizienten Algorithmen*.

Dem Entwurf auf die zu optimierenden Funktionen angepasster spezieller Suchverfahren steht der Entwurf *allgemeiner* Suchverfahren gegenüber: diese sind nicht auf eine Klasse zu optimierender Funktionen zugeschnitten, sondern folgen allgemeinen Prinzipien, die ihre Effizienz für möglichst viele Optimierungsprobleme zur Folge haben sollen. Diese Prinzipien enthalten implizit oftmals schwierig zu formalisierende Vorstellungen, wie „wichtige“ zu optimierende Funktionen aussehen, auf denen das allgemeine Suchverfahren arbeiten soll. Evolutionäre Algorithmen sind ein Beispiel hierfür, bei denen den Grundprinzipien der natürlichen Evolution gefolgt wird. Daneben sind z. B. auch Tabu-Search (Glover und Laguna (1998)), der Metropolis-Algorithmus (Metropolis, Rosenbluth, Rosenbluth, Teller und Teller (1953)) und Simulated Annealing (van Laarhoven und Aarts (1987)) zu nennen, wobei letzteres sich beispielsweise an einem physikalischen Prozess zur Erzeugung möglichst reiner Kristalle orientiert.

Die Verwendung solcher allgemeiner Suchverfahren ist dann sinnvoll, wenn die Struktur der zu optimierenden Funktion so kompliziert, dass ein Entwurf eines speziellen Suchverfahrens zu aufwendig scheint, oder schlichtweg nicht bekannt ist, da die Funktion z. B. nur durch ein Experiment auszuwerten ist. Natürlich kann und sollte aus den Erfahrungen mit dieser Zielfunktion das Suchverfahren verbessert werden, doch stellt es dann kein allgemeines Suchverfahren mehr dar.

Deshalb und da allgemeine Suchverfahren für beliebige Funktionen $f : S \mapsto W$ ein Optimum liefern sollen, wird im Folgenden stets angenommen, dass sie auf die Funktion f nur durch Funktionsauswertungen zugreifen können. In diesem so genannten *Black-Box* Szenario hat ein Suchverfahren also keine Möglichkeit, darüber hinaus auf die Struktur der Funktion zuzugreifen (siehe Abbildung 2.1).

Im Allgemeinen werden in Abhängigkeit von den betrachteten Funktionen nicht alle Suchverfahren für ihre Optimierung gleich gut geeignet sein. Schränkt man z. B. die Menge der Funktionen so ein, dass diese Sortierprobleme darstellen, so werden gute Sortieralgorithmen besser sein als allgemeine Suchverfahren, da in ihren Entwurf viel Wissen über das Sortierproblem eingeflossen ist. Andererseits sollen allgemeine Suchverfahren auch auf Zielfunktionen, die kein Sortierproblem darstellen, ein Optimum liefern, während dies von Sortieralgorithmen nicht erwartet wird.

Zwei spezielle Suchverfahren können also nur verglichen werden, wenn sie für dieselbe Klasse von Funktionen entworfen sind, und dann sinnvollerweise nur auf Funktionen dieser Klasse. Für den Vergleich allgemeiner Suchverfahren kann man jedoch keine Teilklasse von Funktionen von vorneherein als besonders geeignet ansehen.

Um den Unterschied bei einem Vergleich genau ausdrücken zu können, braucht man ein Maß für die Effizienz der Suchverfahren. Die gebräuchlichen Maße zur Bestimmung der Effizienz eines Algorithmus, wie seine Laufzeit und sein Speicherplatzverbrauch, setzen voraus, dass zumindest abgeschätzt werden kann, wie aufwendig in diesen Maßen die Auswertung von f für eine Eingabe ist. Da im Black-Box Szenario kein Wissen über die Funktion f vorhanden ist, kann der Ressourcenverbrauch nur ohne die Funktionsauswertungen von f bestimmt werden. Da aber oftmals diese besonders aufwendig sind, wird von uns als Kostenmaß eines allgemeinen Suchverfahrens die Anzahl seiner verschiedenen Funktionsauswertungen von f, bis ein Optimum von f gefunden wird, benutzt. Für randomisierte allgemeine Suchverfahren ist diese Größe natürlich zufallsbeeinflusst, weshalb dann die erwartete Anzahl betrachtet wird.

Da wir uns im Folgenden auf die Kosten eines allgemeinen Suchverfahrens konzentrieren, kann dieses auch mit der Folge der ausgewerteten Punkte identifiziert werden. Indem wir o. B. d. A. voraussetzen, dass bereits durchgeführte Funktionsauswertungen abgespeichert werden, wird jedes Suchverfahren nur verschiedene Punkte auswerten. Da ohne Beschränkung der Klasse der Funktionen, die optimiert werden sollen, nie ausgeschlossen werden kann, dass ein noch nicht ausgewerteter Punkt das Optimum ist, wird jede Folge der ausgewerteten Punkte aber auch alle Elemente enthalten. Dies führt zu folgender Definition:

Definition 2.1.1 *Seien S und W endliche Mengen, wobei W geordnet ist. Ein* deterministisches allgemeines Suchverfahren *ist ein deterministischer Algorithmus A, der bei Eingabe einer Funktion $f : S \mapsto W$ die Folge $S_f^A = (s_1, \ldots, s_{|S|})$ der paarweise verschiedenen von ihm ausgewerteten Punkte aus S ausgibt. Ein* randomisiertes allgemeines Suchverfahren *entspricht demgemäß einer Wahrscheinlichkeitsverteilung über alle deterministischen allgemeinen Suchverfahren.*

Da wir voraussetzen, dass der beste bislang gefundene Punkt in jedem Suchverfahren stets separat abgespeichert wird, kann ein Suchverfahren stoppen, wenn es zum ersten Mal ein Optimum gefunden hat. Dies ist aber in der Regel nur von theoretischem Nutzen, da ohne Wissen über die Funktion ein Optimum nur erkannt werden kann, wenn sein Funktionswert gleich dem maximalen Element in W ist. Für Funktionen, für die dies nicht gilt, kann nie garantiert werden, ein Optimum gefunden zu haben, wenn nicht alle Punkte in S ausgewertet wurden. Indem die Zahl der Funktionsauswertungen, bis zum ersten Mal ein Optimum erreicht ist, als Kosten eines Suchverfahrens definiert wird, wird der Aufwand bei Benutzung eines optimalen Stopp-Kriteriums gemessen:

Definition 2.1.2 *Die Kosten $c_A(f)$ eines deterministischen allgemeinen Suchverfahrens A zur Optimierung einer Funktion $f : S \mapsto W$ sind gleich*

$$\min\{t \in \{1, \ldots, |S|\} \mid f((S_f^A)_t) = \max\{f(s) \mid s \in S\}\}.$$

Die Kosten $c_A(f)$ *eines randomisierten allgemeinen Suchverfahrens* A *auf einer Funktion* $f : S \mapsto W$ *sind gleich dem Erwartungswert der Kosten der sich dadurch ergebenden deterministischen allgemeinen Suchverfahren.*

Die Kosten $c_A(F)$ *eines allgemeinen Suchverfahrens* A *auf einer Menge* F *von Funktionen von* S *nach* W *sind gleich dem arithmetischen Mittel der Kosten auf den einzelnen Funktionen aus* F

$$c_A(F) := \frac{1}{|F|} \sum_{f \in F} c_A(f).$$

Natürlich ist es wichtig, zu einer Klasse von zu optimierenden Funktionen allgemeine Suchverfahren zu finden, deren Kosten auf dieser Klasse möglichst gering sind. Gerade im Bereich der evolutionären Algorithmen wird oft behauptet, dass manche Algorithmen besser als andere sind, ohne das Kostenmaß oder die Klasse von Funktionen, die betrachtet werden, näher zu spezifizieren (siehe Goldberg (1989)).

In Hart und Belew (1991) wurde schon festgestellt, dass unter der Annahme RP $\neq$ NP (Papadimitriou (1994)) kein evolutionärer Algorithmus alle Zielfunktionen $f : \{0,1\}^n \mapsto \{0,1\}$ effizient maximieren kann. Diese aus komplexitätstheoretischer Sicht triviale Feststellung bezieht sich auf die *Worst-Case*-Laufzeit von Algorithmen. Für das in Definition 2.1.2 angegebene *Average-Case*-Kostenmaß der Zahl der verschiedenen Funktionsauswertungen wurde in Wolpert und Macready (1997) mit dem NFL-Theorem gezeigt, dass im Durchschnitt über alle Funktionen zwischen zwei Mengen S und W alle allgemeinen Suchverfahren dieselben Kosten haben. Dies beruht auf der einfachen Tatsache, dass kein Suchverfahren aus bisher ausgewerteten Punkten des Suchraums S Rückschlüsse auf die vorliegende Funktion ziehen kann, wenn jede Funktion möglich ist.

Im folgenden soll das NFL-Theorem mit einem einfacheren Beweis als in Wolpert und Macready (1997) vorgestellt werden, um so seine grundlegende Natur zu verdeutlichen.

2.2 Das NFL-Theorem

Die Aussage, dass im Mittel über alle Funktionen $F_{S,W} = \{f : S \mapsto W\}$ alle allgemeinen Suchverfahren A die gleichen Kosten $c_A(F_{S,W})$ haben, folgt sehr anschaulich, wenn wir uns zunächst auf deterministische Suchverfahren A und die sie darstellenden Folgen von Suchpunkten S_f^A beschränken. Denn es ist leicht einzusehen, dass es für zwei beliebige deterministische Suchverfahren A_1 und A_2 zu jeder Funktion $f \in F_{S,W}$ eine Funktion $f' \in F_{S,W}$ gibt, so dass A_1 bei der Optimierung von f dieselbe Folge von Funktionswerten erhält wie dies A_2 bei der Optimierung von f' tut. Damit sind natürlich auch die Kosten von A_1 bzw. A_2 bei Optimierung von f bzw. f' gleich. Da A_1 auf verschiedenen Funktionen verschiedene Folgen von Funktionswerten erhält, lässt sich zu jeder Funktion, die A_1 mit Kosten $k \in \{1, \ldots, |S|\}$ optimiert, eindeutig genau eine Funktion finden, die A_2 mit Kosten k optimiert.

Unser Beweis des NFL-Theorems wird genau dieser Idee folgen. Dazu sei noch folgende Definition vorangestellt:

Definition 2.2.1 *Sei* $f : S \mapsto W$ *die zu maximierende Funktion und* A *ein deterministisches allgemeines Suchverfahren. Dann sei* $W_f^A = (w_1, \ldots, w_{|S|})$ *die Folge der Funktionswerte, die* A *auf* f *erhält.*

Natürlich gilt dann, dass $f((S_f^A)_i) = (W_f^A)_i$ für alle $i \in \{1, \ldots, |S|\}$ ist. Somit lässt sich beweisen:

Lemma 2.2.2 *Seien A_1 und A_2 zwei deterministische allgemeine Suchverfahren. Dann gibt es eine bijektive Abbildung $g : F_{S,W} \mapsto F_{S,W}$, so dass für alle $f \in F_{S,W}$ die Folgen $W_f^{A_1}$ und $W_{g(f)}^{A_2}$ gleich sind.*

Beweis: Wir werden die Funktion g im Folgenden konstruktiv beschreiben. Dazu sei $s_1^1 \in S$ bzw. $s_1^2 \in S$ der erste Punkt, dessen Funktionswert A_1 bzw. A_2 auswertet. Dann wird $g(f)(s_1^2) := f(s_1^1)$ gesetzt. Damit stehen die Punkte $s_2^1 \in S$ bzw. $s_2^2 \in S$ fest, die A_1 bzw. A_2 als nächste auswerten.

Allgemein seien für $t \in \{1, \ldots, |S|\}$ die Punkte $s_t^1 \in S$ bzw. $s_t^2 \in S$ bekannt, die A_1 bzw. A_2 im t-ten Schritt auswerten werden. Dann wird $g(f)(s_t^2) := f(s_t^1)$ gesetzt. Da nun die Punkte $s_{t+1}^1 \in S$ bzw. $s_{t+1}^2 \in S$ feststehen, die als nächste von A_1 bzw. A_2 ausgewertet werden, kann die Konstruktion induktiv bis $t = |S|$ fortgesetzt werden.

Weil die Vereinigung aller $\{s_t^2\}$ für $t \in \{1, \ldots, |S|\}$ gleich S ist, ist die Funktion $g(f)$ eindeutig definiert. Nach Konstruktion ist klar, dass die Folgen der Funktionswerte von A_1 auf f und A_2 auf $g(f)$ gleich sind. Da zwei unterschiedliche Funktionen f und f' unterschiedlichen Folgen $W_f^{A_1}$ und $W_{f'}^{A_1}$ entsprechen, ist die Funktion g injektiv und somit bijektiv. □

Die Funktion g ist natürlich von A_1 und A_2 abhängig. Um eine Überfrachtung der Notation zu vermeiden und da die Suchverfahren aus dem Zusammenhang ersichtlich sind, werden wir dies nicht weiter vermerken. Klar ist, dass A_1 auf f dieselben Kosten wie A_2 auf $g(f)$ hat. Somit sind auch die durchschnittlichen Kosten von A_1 und A_2 über die Menge $F_{S,W}$ gleich:

$$c_{A_1}(F_{S,W}) = \frac{1}{|F_{S,W}|} \cdot \sum_{f \in F_{S,W}} c_{A_1}(f) = \frac{1}{|F_{S,W}|} \cdot \sum_{f \in F_{S,W}} c_{A_2}(g(f)) = c_{A_2}(F_{S,W}).$$

Randomisierte allgemeine Suchverfahren A können stets so aufgefasst werden, dass sie zuerst die zufälligen Entscheidungen treffen, in einem Bitstring $x \in \{0,1\}^m$ (da die Zahl deterministischer Suchverfahren endlich ist, ist auch m eine endliche Zahl) abspeichern und dann deterministisch in Abhängigkeit von den erhaltenen Funktionswerten und x arbeiten. Bezeichnen wir mit $A(x)$ das deterministische allgemeine Suchverfahren, das sich ergibt, wenn A auf den Zufallsentscheidungen x arbeitet, und mit $\mathrm{P}(x) \in [0,1]$ deren Wahrscheinlichkeit, so gilt für beliebige Mengen $F \subseteq F_{S,W}$:

$$c_A(F) = \frac{1}{|F|} \cdot \sum_{f \in F} \sum_{x \in \{0,1\}^m} \mathrm{P}(x) \cdot c_{A(x)}(f) = \sum_{x \in \{0,1\}^m} \mathrm{P}(x) \cdot \frac{1}{|F|} \cdot \sum_{f \in F} c_{A(x)}(f).$$

Da alle deterministischen Suchverfahren über $F_{S,W}$ gemittelt die gleichen Kosten haben, folgt dies also auch für alle randomisierten Suchverfahren:

Korollar 2.2.3 (NFL-Theorem) *Seien A_1 und A_2 zwei beliebige randomisierte oder deterministische allgemeine Suchverfahren, so sind die durchschnittlichen Kosten $c_{A_1}(F_{S,W})$ von A_1 und $c_{A_2}(F_{S,W})$ von A_2 gleich.*

In Anbetracht des NFL-Theorems macht es also keinen Sinn, ein über alle Funktionen bestes Suchverfahren entwerfen zu wollen. Für jede Funktion, auf dem ein Suchverfahren besser als der Durchschnitt aller Suchverfahren arbeitet, gibt es Funktionen, auf denen es schlechter arbeitet. Diese Tatsache hat zu teilweise intensiven Diskussionen geführt (Culberson (1998)), da praktische Erfahrungen scheinbar dagegen sprechen: ist ein einfaches *Hillclimber*-Verfahren, das z. B. im $S = \{0,1\}^n$ stets einen benachbarten Punkt mit dem höchsten Funktionswert auswählt, bei einer Maximierungsaufgabe einem *Hilldescender*-Verfahren, das stets einen Nachbarn mit kleinstem Funktionswert wählt, nicht naturgemäß überlegen?

Dieser scheinbare Widerspruch erklärt sich dadurch, dass die Durchschnittsbildung über *alle* Funktionen zwischen den Mengen S und W, auch wenn sie, wie besprochen, für allgemeine Suchverfahren natürlich zu sein scheint, doch vollkommen unrealistisch ist. Denn einfache komplexitätstheoretische Überlegungen und Beispiele zeigen, dass schon für relativ kleine Mengen S und W nur ein Bruchteil aller Funktionen von S nach W bei der Optimierung vorkommen kann. Darauf basierend werden im nächsten Abschnitt dem *Szenario* des NFL-Theorems, d. h. der Situation, in der alle Voraussetzungen zu seiner Gültigkeit erfüllt sind, realistischere Szenarien gegenübergestellt.

2.3 Realistische Szenarien für allgemeine Suchverfahren

Die Motivation, die Menge aller Funktionen von S nach W bei der Betrachtung von allgemeinen Suchverfahren nicht einzuschränken, ist, dass jegliche Einschränkung unnatürlich zu sein scheint, da allgemeine Suchverfahren, ihrem Namen folgend, für keine Teilmenge von Funktionen speziell angepasst sind. Insofern scheint das NFL-Szenario das bestmögliche Szenario zu sein, um allgemeine Suchverfahren zu vergleichen:

Szenario 2.3.1 (NFL-Szenario) *Alle Funktionen $f : S \mapsto W$ haben die gleiche Wahrscheinlichkeit, bei der Black-Box Optimierung aufzutreten.*

Doch dieses Szenario wird seinem Anspruch, ein möglichst gutes Modell der realen Situation, in der allgemeine Suchverfahren eingesetzt werden, zu sein, nicht gerecht, wie folgende Überlegung beispielhaft zeigt: ist $S = \{0,1\}^{50}$ und $W = \{0,1\}$, so gibt es $2^{2^{50}}$ Funktionen von S nach W. Mit einer Speicherkapazität von S Bits sind höchstens 2^S verschiedene Funktionen darstellbar. Somit ist mit $2^{50} - m$ Bits ein Anteil von 2^{-m} aller Funktionen von S nach W darstellbar. Wenn man eine zwar willkürliche, aber zur Zeit realistische Obergrenze von einem Gigabyte (1024^3 Bytes) Speicher zur Darstellung der Zielfunktion annimmt, so ist damit also nur ein Anteil von $2^{-2^{50}+2^{33}} \approx 10^{-3.38 \cdot 10^{15}}$ aller Funktionen darstellbar, da

$$1024^3 \cdot 8 = 2^{33} = 2^{50} - \left(2^{50} - 2^{33}\right)$$

ist. Schon an diesem Beispiel mit nur moderat großer Menge S und der kleinstdenkbaren Anzahl von möglichen Funktionswerten sieht man, dass der Anteil behandelbarer Funktionen nur einen astronomisch geringen Bruchteil an allen Funktionen darstellt. In der Realität werden also nur Funktionen zu optimieren sein, deren Beschreibungslänge stark eingeschränkt ist.

Natürlich kann man einwenden, dass je nach Wahl der Darstellung (z. B. der benutzten Programmiersprache) die Menge der mit beschränktem Speicher darstellbaren Funktionen variiert. Diesem Einwand kann jedoch mit der Theorie der Kolmogoroff-Komplexität (Li und Vitányi (1993)) begegnet werden: diese misst die Komplexität eines Bitstrings s durch die Länge des kürzesten Programms, das auf einer festgelegten universellen Turing-Maschine diese Folge s erzeugt. Dieses Maß scheint nun von der verwendeten universellen Turing-Maschine abhängig zu sein, doch besagt das zentrale *Invarianz-Theorem*, dass der Übergang zu einer anderen universellen Turing-Maschine die Kolmogoroff-Komplexität nur um einen additiven konstanten Summanden ändern kann. Fixiert man also eine bestimmte universelle Turing-Maschine, so ist die resultierende Kolmogoroff-Komplexität bis auf einen konstanten Summanden gleich der Länge jedes Programms, das diesen Bitstring erzeugt.

Selbstverständlich kann man die Kolmogoroff-Komplexität auch von Funktionen $f : S \mapsto W$ definieren, indem man eine Funktion durch die Folge ihrer Funktionswerte darstellt und diese in einen Bitstring überführt. Da sich Programme zur Funktionsauswertung von f und solche, die die ganze Folge von Funktionswerten berechnen, mit nur geringem Aufwand ineinander überführen lassen, kann kein Programm zur Funktionsauswertung eine um einen mehr als linear großen Summanden größere minimale Beschreibungslänge als die Kolmogoroff-Komplexität der Folge der Funktionswerte haben. Beschränkt man letztere nach oben, so wird auch die Länge des kürzesten Programms zur Funktionsauswertung nach oben beschränkt. Ein wesentlich realistischeres Szenario als das NFL-Szenario 2.3.1 ist also

Szenario 2.3.2 *Alle Funktionen $f : S \mapsto W$, deren Kolmogoroff-Komplexität durch $K \in \mathbb{N}$ nach oben beschränkt ist, haben die gleiche Wahrscheinlichkeit, bei der Black-Box Optimierung aufzutreten.*

Das Szenario 2.3.2 ist insofern sehr allgemein, als die Darstellung einer Funktion in realistischer Größe nur eine notwendige, aber keine hinreichende Bedingung zur realistischen Black-Box Optimierung darstellt. Denn was nützt eine ausreichend kompakte Darstellung, wenn eine einzige Auswertung der Funktion auf dem schnellsten verfügbaren Rechner Jahre dauert?

Die weitverbreitetste Möglichkeit, formal zu definieren, wann eine Funktion f für ihre Auswertung nicht zu viel Zeit benötigt, ist vorauszusetzen, dass die Funktion in P ist, sich also in polynomieller Zeit bzgl. der Eingabelänge von einer deterministischen Turing-Maschine berechnen lässt. Diese Einschränkung sollte nicht zu viele interessante Zielfunktionen ausschließen, da sich selbst Zielfunktionen vieler NP-harter Optimierungsprobleme in polynomieller Zeit berechnen lassen.

Jedoch setzt diese voraus, dass sich $f : S \mapsto W$ sinnvoll in eine Familie von Funktionen $(f_n)_{n\in\mathbb{N}} : S_n \mapsto W$ einbetten lässt, wobei n die Dimension des Suchraums ist. Um jedoch auch Funktionen zuzulassen, die dies nicht erfüllen, beschränken wir im Folgenden die Laufzeit nur durch eine feste obere Schranke:

Szenario 2.3.3 *Alle Funktionen $f : S \mapsto W$, deren Laufzeit durch $T \in \mathbb{N}$ nach oben beschränkt ist, haben die gleiche Wahrscheinlichkeit, bei der Black-Box Optimierung aufzutreten.*

Der Begriff „Laufzeit“ ist hier nicht weiter definiert, eine Möglichkeit wäre es, die Laufzeit als die Zahl der Schritte des schnellsten Turing-Maschinen-Programms zur Berechnung von f zu wählen. Nach der erweiterten Churchschen These ist diese Größe beim Übergang zu anderen sinnvollen Maschinenmodellen polynomiell, wenn dies auch für die Laufzeit bzgl. der Turing-Maschine gilt.

Der Übergang zu Mengen S und W fester Größe stellt den Übergang von uniformen zu nicht-uniformen Komplexitätsmaßen dar. Das gebräuchlichste Komplexitätsmaß im nicht-uniformen Berechnungsmodell ist die Größe eines kleinsten booleschen Schaltkreises mit AND-, OR- und NOT-Bausteinen (Papadimitriou (1994)) für die Funktion $f : S \mapsto W$ (wobei S und W geeignet in boolesche Mengen eingebettet sind). Somit lässt sich das folgende Szenario gut auf Situationen anwenden, in denen die zu optimierende Funktion nicht durch ein Programm, sondern durch Hardware ausgewertet wird:

Szenario 2.3.4 *Alle Funktionen $f : S \mapsto W$, deren Schaltkreisgröße durch $G \in \mathbb{N}$ nach oben beschränkt ist, haben die gleiche Wahrscheinlichkeit, bei der Black-Box Optimierung aufzutreten.*

Mit für die jeweilige Situation geeignet gewählten Werte K, T und G sehen wir also, dass die obigen Szenarien 2.3.2, 2.3.3 und 2.3.4 die Situation bei der Optimierung realistischer modellieren als dies das NFL-Szenario tut. Denn Funktionen, deren Kolmogoroff-Komplexität, benötigte Laufzeit oder minimale Schaltkreisgröße zu groß sind, sind in der Praxis nicht zu optimieren.

Natürlich stellt sich nun die Frage, ob in diesen Szenarien auch ein NFL-Theorem gilt, d. h. ob alle allgemeinen Suchverfahren gemittelt z. B. über alle Funktionen mit Kolmogoroff-Komplexität höchstens K die gleichen Kosten haben. Es scheint sehr schwierig zu sein, das Wissen über die beschränkte Komplexität einer Funktion nach den obigen Maßen in Aussagen über die durchschnittlichen Kosten eines Algorithmus, der diese Funktion optimiert, umzusetzen.

Anschaulich scheint es eher möglich, dass in komplexitätsbeschränkten Szenarien kein NFL-Theorem gilt: denn das NFL-Theorem stützt sich darauf, dass die Menge $F_{S,W}$ keine Struktur enthält. Eine beschränkte Komplexität impliziert aber, dass sich die $|S|$ Ausgabewerte viel kompakter beschreiben lassen. Dies könnte der Funktionenmenge eine Struktur auferlegen, die vielleicht von dem einen Suchverfahren besser ausgenutzt wird als von einem anderen. Dass dies für eine sehr kleine Funktionsmenge gilt, wird im folgenden Abschnitt nachgewiesen.

2.4 Ein „Free Appetizer“ in einem realistischen Szenario

Die Beschränkung der Komplexität einer Funktion zu nutzen, um darüber hinausgehende Eigenschaften zu erkennen, ist im Allgemeinen ein sehr schwieriges Problem, wie viele ungelöste Probleme aus der Komplexitätstheorie zeigen (Papadimitriou (1994)). Deshalb werden in diesem Abschnitt zum Vergleich der Kosten bestimmter Suchverfahren die betrachteten Mengen S und W sehr klein gewählt, so dass $F_{S,W}$ mit Rechnerhilfe komplett durchmustert werden kann. Zwar ergibt dies nur ein unrealistisch kleines Beispiel, doch wird sich zeigen, dass die betrachteten Suchverfahren je nach Komplexität der zugelassenen Funktionen unterschiedliche durchschnittliche Kosten besitzen. Somit muss in komplexitätsbeschränkten Szenarien kein NFL-Theorem gelten.

Im Folgenden wird $S = \{0,1\}^3$ und $W = \{0,1\}^2$ gewählt, so dass $F_{S,W}$ genau $4^8 = 65536$ Funktionen enthält. Diese Menge ist so klein, dass mit Rechnerhilfe für jede der im Nachfolgenden betrachteten Klassen die Zugehörigkeit jeder Funktion schnell entschieden werden kann. Somit können für die anschließend definierten Suchverfahren die durchschnittlichen Kosten genau bestimmt werden. Da zur Berechnung mit der Funktionsbibliothek LEDA (Mehlhorn und Näher (1999)) gearbeitet wird, die die Benutzung beliebig genauer rationaler Zahlen ermöglicht, können keine Rundungsfehler auftreten. Die Resultate werden erst am Ende der Rechnung gerundet, wobei darauf geachtet wird, dass etwaige Rundungsfehler so klein sind, dass sie die Anordnung zweier Zahlen zueinander nicht beeinflussen.

Die benutzten, in ihrer Komplexität beschränkten Klassen von Funktionen aus $\{f : \{0,1\}^3 \mapsto \{0,1\}^2\}$ sind die Folgenden:

- Die Klasse C, bestehend aus allen Funktionen, die einen {AND,OR,NOT}-Schaltkreis mit höchstens drei Bausteinen besitzen, wobei die Eingänge nicht mitgezählt werden.

- Die Klassen F_i, $i \in \{0, \ldots, 8\}$, bestehend aus allen Funktionen, die sich durch ein *Shared Binary Decision Diagram* (*SBDD*) (siehe Bryant (1986) und auch Abschnitt 4.3) mit höchstens i inneren Knoten darstellen lassen.

Dabei wird die SBDD-Größe als Komplexitätsmaß gewählt, da SBDDs eine der gebräuchlichsten Darstellungsformen für boolesche Funktionen darstellen. Für ihre Definition sei auf Abschnitt 4.3 verwiesen, es sei hier nur der Klarheit halber vermerkt, dass zur Bestimmung der SBDD-Größe das Minimum über alle Variablenordnungen gebildet wird. Eine Funktion in der Klasse F_3 z. B. lässt sich also mit mindestens einer Variablenordnung durch ein SBDD mit höchstens drei Knoten darstellen. Die Klasse C wurde in Übereinstimmung mit Szenario 2.3.4 gewählt.

Als Suchverfahren werden wir im Folgenden vier einfache adaptive allgemeine Suchverfahren und die Klasse aller nicht-adaptiven Suchverfahren betrachten. *Nicht-adaptiv* bedeutet in diesem Zusammenhang, dass die Folge der ausgewerteten Punkte unabhängig von den Ergebnissen der Funktionsauswertungen ist. Somit kann ähnlich zu Definition 2.1.1 ein nicht-adaptives Suchverfahren mit einer vorgegebenen Aufzählung des Suchraums $S = \{0,1\}^3$ identifiziert werden, die sogar von der zu optimierenden Funktion unabhängig ist. Deshalb gibt es genau $8! = 40320$ verschiedene nicht-adaptive Suchverfahren auf Funktionen $f : \{0,1\}^3 \mapsto \{0,1\}^2$.

Die adaptiven Algorithmen, die wir betrachten, sind:

Algorithmus 2.4.1 (Evolutionärer Algorithmus $EA_{\geq}$)

1. *Wähle $x \in \{0,1\}^3$ gleichverteilt zufällig.*
2. *Erzeuge $y \in \{0,1\}^3$, indem alle Bits von x unabhängig voneinander mit Wahrscheinlichkeit $1/3$ negiert werden.*
3. *Falls $f(y) \geq f(x)$ ist, setze $x := y$.*
4. *Gehe zu Schritt 2.*

Algorithmus 2.4.2 (Evolutionärer Algorithmus $EA_{>}$)

1. *Wähle $x \in \{0,1\}^3$ gleichverteilt zufällig.*
2. *Erzeuge $y \in \{0,1\}^3$, indem alle Bits von x unabhängig voneinander mit Wahrscheinlichkeit $1/3$ negiert werden.*
3. *Falls $f(y) > f(x)$ ist, setze $x := y$.*
4. *Gehe zu Schritt 2.*

Dieses sind zwei sehr einfache evolutionäre Algorithmen, wobei wir den ersteren in Kapitel 3 als (1+1) EA in seiner allgemeinen Form noch sehr intensiv untersuchen werden. Die folgenden zwei Algorithmen sind streng genommen keine allgemeinen Suchverfahren, da sie die Menge $F \subseteq \{f : \{0,1\}^3 \mapsto \{0,1\}^2\}$ der als Black-Box in Frage kommenden Funktionen kennen. Dass beide Algorithmen versuchen, diese Struktur zu einer schnellen Optimierung auszunutzen, sollte natürlich bessere Ergebnisse zur Folge haben als die der evolutionären Algorithmen 2.4.1 und 2.4.2. Jedoch bedeutet es auch, dass ein Vergleich der Ergebnisse nicht zu der Behauptung führen darf, dass einer der folgenden Algorithmen den evolutionären Algorithmen überlegen ist, da ihre Voraussetzungen unterschiedlich sind:

Algorithmus 2.4.3 (MaxOpt)

1. *Setze $G := F$.*
2. *Für jedes $x \in \{0,1\}^3$, dessen Funktionswert noch nicht bestimmt wurde, setze*
$$m_x := |\{g \in G \mid x \text{ ist maximal für } g\}|.$$
3. *Bestimme $f(x)$ für ein $x \in \{0,1\}^3$ mit maximalem m_x-Wert.*
4. *Lösche alle Funktionen $g \in G$, für die $g(x)$ maximal oder $g(x) \neq f(x)$ ist.*
5. *Gehe zu Schritt 2.*

Algorithmus 2.4.4 (MinMax)

1. *Setze $G := F$.*
2. *Für jedes $x \in \{0,1\}^3$, dessen Funktionswert noch nicht bestimmt wurde, und jedes $y \in \{0,1\}^2$ setze*

$$m_{x,y} := |G \setminus \{g \in G \,|\, g(x) \text{ ist maximal oder } g(x) \neq y\}|.$$

3. *Für jedes $x \in \{0,1\}^3$ sei $m_x := \max\{m_{x,y} \,|\, y \in \{0,1\}^2\}$.*
4. *Bestimme $f(x)$ für ein $x \in \{0,1\}^3$ mit minimalem m_x-Wert.*
5. *Lösche alle Funktionen $g \in G$, für die $g(x)$ maximal oder $g(x) \neq f(x)$ ist.*
6. *Gehe zu Schritt 2.*

Während also der Algorithmus MaxOpt stets einen Suchpunkt auswertet, der für die meisten noch möglichen Funktionen maximal ist, wählt MinMax stets einen Suchpunkt, so dass die Größe der größten Menge von noch möglichen Zielfunktionen nach der Funktionsauswertung minimal ist. Beide heuristischen Methoden versuchen also, das Wissen über die Menge der zu optimierenden Funktionen zu einer möglichst schnellen Optimierung zu nutzen.

Für diese vier Algorithmen 2.4.1, 2.4.2, 2.4.3 und 2.4.4 sind in Tabelle 2.1 die durchschnittlichen Kosten $c_A(F)$ über die Mengen $F \in \{F_0, \ldots, F_8, C\}$ vermerkt. Zusätzlich sind die geringsten (NA_{min}), durchschnittlichen (NA_{dur}) und größten (NA_{max}) Kosten über alle nicht-adaptiven Suchverfahren vermerkt. Die Rundung diese Werte auf vier Nachkommastellen ist erst jeweils am Ende jeder Rechnung erfolgt, weshalb in der gesamten Tabelle gilt, dass das relative Größenverhältnis der angegebenen Zahlen das der durchschnittlichen Kosten korrekt widerspiegelt.

F	$\|F\|$	$EA_>$	$EA_\geq$	NA_{dur}	NA_{min}	NA_{max}	MaxOpt	MinMax
F_0	4	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000
F_1	34	1,7929	1,7959	1,7059	1,4412	2,0294	1,4412	1,4412
F_2	280	2,2927	2,2944	2,2441	1,9607	2,5214	1,9464	2,0071
F_3	2166	2,5085	2,5089	2,4867	2,2636	2,6957	2,2433	2,2322
F_4	8908	2,8714	2,8708	2,8622	2,6918	3,0071	2,6411	2,6404
F_5	25694	2,9834	2,9829	2,9792	2,8697	3,0594	2,8352	2,8460
F_6	51520	3,0262	3,0261	3,0253	2,9845	3,0493	2,9655	2,9798
F_7	65144	3,0639	3,0639	3,0639	3,0613	3,0650	3,0593	3,0622
F_8	65536	3,0637	3,0637	3,0637	3,0637	3,0637	3,0637	3,0637
C	1290	2,7658	2,7650	2,7471	2,2674	3,5581	2,1395	2,1574

Tabelle 2.1: Durchschnittliche Kosten verschiedener Suchverfahren

Da das gewählte Beispiel sehr klein ist und MaxOpt und MinMax wesentlich mehr Information und Ressourcen benötigen als die anderen Suchverfahren, sollen die Ergebnisse in Tabelle 2.1 nun nicht im Detail untersucht werden. Einzig soll festgehalten werden, dass die durchschnittlichen Kosten der Suchverfahren auf eingeschränkten Funktionsmengen nicht mehr stets gleich sind. Dieses Beispiel zeigt also, dass ein NFL-Theorem in komplexitätsbeschränkten Szenarien im Allgemeinen nicht gilt. Da die Unterschiede der einzelnen Verfahren aber nur sehr gering sind, sprechen wir in Analogie zum „No Free Lunch"-Theorem von einem „Free Appetizer".

Weiterhin soll nur anhand der Mengen F_i bemerkt werden, dass mit zunehmender Komplexität, d. h. wachsendem $i \in \{0, \ldots, 8\}$, die durchschnittlichen Laufzeiten aller Verfahren anwachsen (bis auf den Übergang von F_7 zu F_8). Insofern ist die

Verwendung der minimalen Knotenanzahl eines SBDDs als ein Komplexitätsmaß zumindest für dieses kleine Beispiel aller Funktionen $\{f : \{0,1\}^3 \mapsto \{0,1\}^2\}$ und die betrachteten Algorithmen sinnvoll.

Dieses Resultat scheint Hoffnung zu machen, dass es auch in realistisch großen komplexitätsbeschränkten Szenarien Suchverfahren gibt, die geringere durchschnittliche Kosten haben als andere Verfahren. Diese Hoffnung wird davon genährt, dass eine Einschränkung der Komplexität der betrachteten Funktionen dazu führt, dass die Funktionenmenge eine bestimmte Struktur enthält. Denn hat die Funktion z. B. eine durch einen kleinen Wert K beschränkte Kolmogoroff-Komplexität, so lassen sich die $\log(|W|)^{|S|}$ Bits der Funktionstabelle durch K Bits darstellen, was ohne eine Struktur in der Funktionstabelle nicht möglich wäre. Damit kann es möglich sein, dass das benutzte Suchverfahren gerade auf Funktionen dieser Struktur effizient funktioniert.

In den nächsten zwei Abschnitten werden wir zeigen, dass aber auch in solchen realistischeren Szenarien ein Suchverfahren nicht für alle Funktionen des Szenarios gut geeignet sein kann. Denn zuerst wird gezeigt, dass das NFL-Theorem auch in eingeschränkteren Szenarien als dem NFL-Szenario 2.3.1 gilt. Dann wird nachgewiesen, dass es zu jeder Funktion, die ein Suchverfahren effizient optimiert, auch stets Funktionen gibt, die dieses Suchverfahren mit hoher Wahrscheinlichkeit nicht effizient optimiert.

2.5 Ein allgemeines NFL-Theorem

Das NFL-Theorem (Korollar 2.2.3) beruht einzig auf Lemma 2.2.2. Dieses besagt, dass es zu zwei Suchverfahren und jeder Funktion aus $F_{S,W}$ stets eine zweite Funktion aus $F_{S,W}$ gibt, so dass das erste Suchverfahren auf die erste Funktion angewandt dieselbe Folge von Funktionswerten erhält, wie dies das zweite Verfahren auf der zweiten Funktion tut. Da die Abbildung g, die der ersten Funktion die zweite zuweist, bijektiv ist, folgt leicht, dass die durchschnittlichen Kosten beider Suchverfahren über alle Funktionen gleich sein müssen.

Nun kann man leicht sehen, dass es zur Gültigkeit dieses Lemmas nicht notwendig ist, über die Menge aller Funktionen zwischen S und W zu argumentieren. Denn die Abbildung $g : F_{S,W} \mapsto F_{S,W}$ wird einer Funktion $f \in F_{S,W}$ stets eine Funktion $g(f) \in F_{S,W}$ zuordnen, die sich durch Permutation aus f ergibt. Dies folgt daraus, dass zwei Funktionen f_1 und f_2, die für alle $w \in W$ jeweils dieselbe Anzahl $n_w \in \mathbb{N}_0$ von Elementen aus S auf w abbilden, durch Permutation auseinander hervorgehen: dazu muss bei einer beliebigen Reihenfolge der Elemente aus S einfach nur für alle $w \in W$ und alle $i \in \{1, \ldots, n_w\}$ das i-te von f_1 auf w abgebildete Element auf das i-te von f_2 auf w abgebildete Element abgebildet werden.

Somit gilt das NFL-Theorem schon im Durchschnitt über Funktionsmengen, die mit einer Funktion auch jede sich aus ihr durch Permutation ergebende Funktion enthalten. Dies sei folgendermaßen formalisiert:

Definition 2.5.1 *Sei $F \subseteq \{f : S \mapsto W\}$. Dann heißt F* unter Permutationen abgeschlossen*, wenn für jede Permutation $\pi : S \mapsto S$ auch $f(\pi) : S \mapsto W$, definiert durch $f(\pi)(s) := f(\pi(s))$, in F enthalten ist.*

Also ergibt sich aus Lemma 2.2.2:

Korollar 2.5.2 (Allgemeines NFL-Theorem) *Seien A_1 und A_2 zwei beliebige randomisierte oder deterministische Suchverfahren, so sind die durchschnittlichen Kosten $c_{A_1}(F)$ von A_1 und $c_{A_2}(F)$ von A_2 über die Funktionenmenge $F \subseteq F_{S,W}$ gleich, wenn F unter Permutationen abgeschlossen ist.*

Was bedeutet dies für die Frage, ob es in realistischen komplexitätsbeschränkten Szenarien überdurchschnittliche Suchverfahren gibt? Da die Relation, die zwei Funktionen f_1 und f_2 in Beziehung setzt, wenn sie durch eine Permutation auseinander hervorgehen, eine Äquivalenzrelation ist, zerfällt die Menge aller Funktionen $F_{S,W}$ in disjunkte Klassen von diesbezüglichen Äquivalenzmengen. Auf jeder dieser Mengen und auf allen Mengen, die sich durch Vereinigung einiger dieser Äquivalenzmengen ergeben, haben alle Suchverfahren die gleichen durchschnittlichen Kosten.

Wenn nun eine Menge von Funktionen, deren Komplexität auf eine der in den Szenarien 2.3.2, 2.3.3 oder 2.3.4 beschriebenen Arten beschränkt ist, sich genau als Vereinigung einiger solcher Äquivalenzklassen ergibt, so müssen alle Suchverfahren auf dieser Menge die gleichen Kosten haben. In diesem Fall wäre unsere Vermutung, dass in diesen Szenarien kein NFL-Theorem gilt, widerlegt.

Um dieses nachzuweisen, muss man untersuchen, wieviel komplexer eine Funktion durch Permutation der Funktionswerte werden kann. Hier kann man leicht folgende Extremfälle ausmachen:

- Sei $S = \{0,1\}^n$, $W = \{0,1\}$ und für einen Punkt $a \in \{0,1\}^n$ die Funktion $f_a : \{0,1\}^n \mapsto \{0,1\}$ definiert durch

  $$f_a(x) := \begin{cases} 1 & \text{, falls } x = a, \\ 0 & \text{, falls } x \neq a. \end{cases}$$

  Diese Funktion wird in Kapitel 3 unter dem Namen PEAK noch näher untersucht. Die Menge $F := \{f_a \,|\, a \in \{0,1\}^n\}$ ist unter Permutationen abgeschlossen, jede Funktion aus ihr hat eine sehr geringe Komplexität nach den drei in den Szenarien 2.3.2, 2.3.3 und 2.3.4 besprochenen Komplexitätsmaßen: ein Programm zur Auswertung von f_a vergleicht die Eingabe mit dem Punkt a und gibt bei Gleichheit eine Eins aus, ansonsten eine Null. Somit ist die Kolmogoroff-Komplexität als auch die Zeit zur Auswertung der Funktion gleich $O(n)$. Dies gilt auch für die Schaltkreisgröße von f_a, da sich f_a einfach durch UND-Verknüpfung der positiven (falls $a_i = 1$) bzw. negierten (falls $a_i = 0$) Eingaben ergibt.

- Sei $S = \{0,1\}^n$, W mit $|W| \geq 2^n$ und $f : S \mapsto W$ eine injektive Funktion. Die kleinste Menge F, die f enthält und unter Permutationen abgeschlossen ist, enthält genau $(2^n)!$ Elemente, da jede dieser Funktionen eindeutig mit einer Permutation des $\{0,1\}^n$ identifiziert werden kann. Da nach der Stirling'schen Formel (A.10) $N!$ durch $(N/\exp(1))^N$ nach unten abgeschätzt werden kann, muss es eine Funktion in F mit Kolmogoroff-Komplexität $\Omega(2^n \cdot n)$ geben. Wenn aber z. B. W ebenfalls gleich $\{0,1\}^n$ ist und f die identische Abbildung, so hat diese natürlich nur eine in n linear große Kolmogoroff-Komplexität.

Somit gibt es unter Permutationen abgeschlossene Funktionsklassen, die ganz innerhalb einer komplexitätsbeschränkten Funktionsklasse sind, wenn deren Schranke zumindest linear wächst, aber auch solche, die sowohl sehr einfache als auch sehr komplexe Funktionen enthalten.

Das erste Beispiel zeigt, dass auch die besprochenen realistischeren Szenarien 2.3.2, 2.3.3 und 2.3.4 in ihren jeweils zugelassenen Mengen von Funktionen Teilklassen enthalten, in denen das NFL-Theorem gilt. Jedoch werden sich im Allgemeinen diese Mengen nicht als Vereinigung solcher Mengen darstellen lassen, wie das zweite Beispiel zeigt. Somit lässt sich aus dem verallgemeinerten NFL-Theorem nicht folgern, dass in den realistischeren Szenarien stets ein NFL-Theorem gilt.

Dass bei der Optimierung von Funktionen auch in realistischen Szenarien eine Abgrenzung zwischen gut und schlecht zu optimierenden Funktionen schwer zu ziehen ist, wird auch im folgenden Abschnitt untermauert.

2.6 Ein „Almost No Free Lunch“ Theorem

In diesem Abschnitt wird gezeigt, dass es für jedes allgemeine Suchverfahren und jede Funktion nur wenig komplexere Funktionen gibt, die dieses Suchverfahren mit hoher Wahrscheinlichkeit (wenn das Suchverfahren randomisiert ist) bzw. mit Sicherheit nicht effizient optimiert. Insbesondere lassen sich zu effizient optimierbaren Funktionen solche finden, für die dies nicht gilt und die nur wenig komplexer sind.

Dieses Ergebnis wird sich auf die einfache Tatsache stützen, dass ein deterministisches Suchverfahren nach T Funktionsauswertungen zwei Funktionen nicht unterscheiden kann, die auf den T bisher ausgewerteten Punkten gleich sind. Wenn man somit eine Funktion so ändert, dass der $|S|$-te Punkt in der Reihenfolge der Auswertung des Suchverfahrens zum Optimum gemacht wird, so wird das Suchverfahren auf dieser Funktion natürlich die größtmögliche Laufzeit haben.

Auf randomisierte Suchverfahren kann diese Idee übertragen werden, indem man durch Anwendung des „Pigeonhole Principle“ zeigt, dass es eine Teilmenge des Suchraums S geben muss, deren Punkte mit sehr hoher Wahrscheinlichkeit nicht in den ersten T Schritten besucht werden. Wird das neue Optimum dann in diese Teilmenge gelegt, wird die Funktion mit hoher Wahrscheinlichkeit erst nach mehr als T Schritten optimiert.

Um diese Ideen zu formalisieren, gehen wir wieder davon aus, dass jedes randomisierte Suchverfahren A zuerst hinreichend viele Zufallsentscheidungen trifft, diese in dem Bitstring $x \in \{0,1\}^m$ speichert und dann abhängig von x wie ein deterministisches Suchverfahren $A(x)$ arbeitet. Um Aussagen treffen zu können, welche Punkte mit hoher Wahrscheinlichkeit von A ausgewertet werden, sei folgende Definition vorangestellt:

Definition 2.6.1 *Sei A ein randomisiertes allgemeines Suchverfahren, das Funktionen $f : S \mapsto W$ optimiert. Für $t \in \{1, \dots, |S|\}$ und $s \in S$ sei $p_A(t, s)$ die Wahrscheinlichkeit, dass A im t-ten Schritt den Punkt s auswertet.*

Sei $\mathrm{P}(x) \in [0, 1]$ die Wahrscheinlichkeit der Zufallsentscheidung $x \in \{0,1\}^n$. Dann können wir folgendes Lemma beweisen:

Lemma 2.6.2 *Sei A ein randomisiertes allgemeines Suchverfahren, das Funktionen $f : S \mapsto W$ optimiert, und $K \in \mathbb{N}$. Zu jeder Funktion f, jedem $T \in \{1, \dots, |S|\}$ und jeder Partition von S in K Mengen $S_1, \dots, S_K \subseteq S$ gibt es ein $k \in \{1, \dots, K\}$, so dass die Wahrscheinlichkeit, dass A in den ersten T Schritten einen Punkt aus S_k auswertet, höchstens gleich T/K ist.*

Beweis: Die abzuschätzende Wahrscheinlichkeit ist

$$\sum_{s \in S_k} \sum_{t=1}^{T} p_A(t, s).$$

Wenn $p_{A(x)}(t, s)$ die $\{0, 1\}$-wertige Indikatorvariable ist, die anzeigt, ob $A(x)$ in der t-ten Funktionsauswertung den Punkt $s \in S$ auswertet, so gilt:

$$\begin{aligned}
\sum_{s \in S} \sum_{t=1}^{T} p_A(t, s) &= \sum_{t=1}^{T} \sum_{s \in S} \sum_{x \in \{0,1\}^m} \mathrm{P}(x) \cdot p_{A(x)}(s, t) \\
&= \sum_{t=1}^{T} \sum_{x \in \{0,1\}^m} \mathrm{P}(x) \cdot \sum_{s \in S} p_{A(x)}(s, t) \\
&= \sum_{t=1}^{T} \sum_{x \in \{0,1\}^m} \mathrm{P}(x) \cdot 1 = T.
\end{aligned}$$

Da die Mengen $S_1, \ldots, S_K$ eine Partition von S bilden, muss es deshalb nach dem „Pigeonhole Principle" mindestens eine Menge S_k geben, so dass die Wahrscheinlichkeit, dass A in den ersten T Schritten einen Punkt aus S_k auswertet, höchstens T/K ist. $\square$

Auch wenn dieses Lemma den Kern der „Almost No Free Lunch"-Aussage enthält, dass es auch in komplexitätsbeschränkten Szenarien neben einfachen auch immer schwierige Funktionen geben muss, wird diese erst mit einer beispielhaften Anwendung auf Funktionen $f : \{0,1\}^n \mapsto \{0, \ldots, N-1\}$ deutlicher. Denn wählt man z. B. $T = (2^n)^{1/3}$ und $K = (2^n)^{2/3}$, so folgt aus Lemma 2.6.2, dass es in jeder Partition von $\{0,1\}^n$ in $(2^n)^{2/3}$ Teilmengen eine davon geben muss, so dass die Wahrscheinlichkeit, dass A in einem der ersten $T = (2^n)^{1/3}$ Schritte einen Punkt dieser Teilmenge auswertet, höchstens gleich $(2^n)^{1/3}/(2^n)^{2/3} = 2^{-n/3}$ ist. Erhöht man den Funktionswert eines dieser Punkte zum neuen Maximalwert von f (wofür der Wertebereich von f vergrößert werden muss, falls der alte Maximalwert von f gleich dem Maximum von W ist), so wird A mit exponentiell hoher Wahrscheinlichkeit diese neue Funktion nicht mit $2^{n/3}$ Auswertungen optimieren:

Korollar 2.6.3 („Almost No Free Lunch" Theorem) *Sei A ein allgemeines Suchverfahren, das Funktionen $f : \{0,1\}^n \mapsto \{0, \ldots, N-1\}$ optimiert. Zu jeder Funktion $f : \{0,1\}^n \mapsto \{0, \ldots, N-1\}$ gibt es $2^{n/3}$ Funktionen $f' : \{0,1\}^n \mapsto \{0, \ldots, N\}$, die jeweils mit f auf nur einer Eingabe nicht übereinstimmen und von A mit einer Wahrscheinlichkeit von mindestens $1 - 2^{-n/3}$ nicht in $2^{n/3}$ Schritten optimiert werden.*

Dies gilt insbesondere auch für Funktionen f, die von A effizient optimiert werden. Da f' nur auf einem Punkt nicht mit f übereinstimmt, ist die Komplexität von f' nach allen in den Szenarien 2.3.2, 2.3.3 und 2.3.4 angeführten Maßen nur höchstens jeweils um einen linearen Summanden größer als die von f. Somit zeigt sich, dass es auch in den Funktionsmengen der realistischeren Szenarien neben effizient zu optimierenden Funktionen stets sehr schwierige Funktionen gibt, unabhängig vom betrachteten Suchverfahren.

Aus Korollar 2.6.3 folgt nur die Existenz von $2^{n/3}$ Funktionen, die von A mit exponentiell hoher Wahrscheinlichkeit nicht in $2^{n/3}$ Schritten optimiert werden. Da $N \geq 2$ ist, ist dies gemessen an der doppelt exponentiell wachsenden Mindestzahl 2^{2^n} aller Funktionen nur ein exponentiell kleiner Anteil. Somit könnte man einwenden, dass nur über einen unbedeutend geringen Anteil an Funktionen in Korollar 2.6.3 eine Aussage getroffen wird. Jedoch kann die Funktion f auf den $2^{n/3}$ Punkten, die von A mit Wahrscheinlichkeit $1 - 2^{-n/3}$ nicht in den ersten $2^{n/3}$ Schritten ausgewertet werden, beliebige Funktionswerte aus $\{0, \ldots, N\}$ besitzen, solange der Wert N mindestens einmal angenommen wird. Somit gilt:

Korollar 2.6.4 (Allgemeines „Almost No Free Lunch" Theorem) *Sei A ein allgemeines Suchverfahren zur Optimierung von Funktionen $f : \{0,1\}^n \mapsto \{0, \ldots, N-1\}$. Zu jeder Funktion $f : \{0,1\}^n \mapsto \{0, \ldots, N-1\}$ gibt es $N^{2^{n/3}} - (N-1)^{2^{n/3}}$ Funktionen $f' : \{0,1\}^n \mapsto \{0, \ldots, N\}$, die von A mit einer Wahrscheinlichkeit von mindestens $1 - 2^{-n/3}$ nicht in $2^{n/3}$ Schritten optimiert werden.*

Viele der Funktionen f' werden jedoch gegenüber f eine wesentlich erhöhte Komplexität haben. Dies gilt z. B. wenn die zugrundeliegende Menge S_k der Partition des $\{0,1\}^n$ eine sehr hohe Kolmogoroff-Komplexität hat. Da Lemma 2.6.2 die Wahl der Partition des $\{0,1\}^n$ nicht vorschreibt, kann man diese so wählen, dass die entstehende Kolmogoroff-Komplexität möglichst gering ist. Beispielsweise kann man als Partition des $\{0,1\}^n$ für $k \in \{0,1\}^{2n/3}$ die Mengen

$$S_k := \{x \in \{0,1\}^n \mid \forall i \in \{1, \ldots, 2n/3\} : x_i = k_i\}$$

wählen. Dann gibt es nach Lemma 2.6.2 für jedes allgemeine Suchverfahren A ein $k \in \{0,1\}^{2n/3}$, so dass A in den ersten $2^{n/3}$ Schritten mit einer Wahrscheinlichkeit von mindestens $1-2^{-n/3}$ keinen Punkt aus S_k auswertet. Somit kann man eine Funktion $f : \{0,1\}^n \mapsto \{0,\dots,N-1\}$ so verändern, dass die neue Funktion f' nur auf S_k nicht mit f übereinstimmt und eingeschränkt auf S_k gleich einer Funktion f'' ist. Dann ergibt sich die Kolmogoroff-Komplexität von f' als Summe der Komplexitäten von f und f'' zusammen mit einem Term linearer Größe $O(n)$ zum Abspeichern von k und Zusatzinformationen. Deshalb lassen sich exponentiell viele Funktionen konstruieren, die A mit exponentiell hoher Wahrscheinlichkeit nicht in $2^{n/3}$ Schritten optimiert und wie f eine z. B. lineare Kolmogoroff-Komplexität haben.

2.7 Konsequenzen

Das NFL-Theorem spiegelt die Tatsache wider, dass ein Suchverfahren stets auf die Struktur der zu optimierenden Funktion angewiesen ist, um diese effizient zu optimieren. Die ursprüngliche Fassung von Wolpert und Macready (1997) macht deutlich, dass zwei Suchverfahren stets nur auf einer eingeschränkten Funktionenmenge sinnvoll verglichen werden können. Sinnvolle Aussagen über die Effizienz von Verfahren, die die Funktionenmenge nicht einschränken, sind nicht zu erreichen.

Jedoch ist das im NFL-Theorem implizit vorausgesetzte Szenario, dass alle Zielfunktionen gleich wahrscheinlich sind, unrealistisch, wie leicht nachvollziehbare komplexitätstheoretische Argumente zeigen. Szenarien, in denen die Komplexität der Funktionen beschränkt sind, sind realistischer, doch scheinen sich diese Voraussetzungen nicht ausnutzen zu lassen, um theoretisch die Existenz überlegener Suchverfahren nachzuweisen. Eine vollständige Aufzählung eines kleinen Beispiels zeigt zumindest, dass in diesen Szenarien kleine Unterschiede existieren können.

Die Verallgemeinerung des NFL-Theorems zeigt, dass auch in den Funktionsmengen der realistischen Szenarien Teilmengen enthalten sind, in denen ein NFL-Theorem gilt. Da sie sich aber im Allgemeinen nicht als Vereinigung solcher Teilmengen ausdrücken lassen, wird in ihnen in der Regel kein NFL-Theorem gelten (was ja schon das angegebene Beispiel zeigt). Weiterhin zeigt das ANFL-Theorem, dass für jedes Suchverfahren stets auch bzgl. ihrer Kosten extrem unterschiedliche Funktionen existieren, die von den realistischeren Szenarien nicht getrennt werden.

Somit sind die von uns angeführten komplexitätsbeschränkten Szenarien zwar realistischer als das allgemeine NFL-Szenario, doch ist es sinnlos, Suchverfahren in diesen Szenarien zu untersuchen, um zu bestimmen, ob alle Funktionen einer Klasse effizient von einem bestimmten Suchverfahren optimiert werden können oder nicht. Aus diesem Grunde wird die Analyse von Algorithmen im Allgemeinen in Szenarien durchgeführt, bei denen die Semantik bzw. die Syntax der Zielfunktion vorgegeben ist:

Szenario 2.7.1 *Alle Funktionen, die zu einem bestimmten Problemtyp (z. B. Sortierproblem, Travelling Salesman Person-Problem, usw.) gehören, haben die gleiche Wahrscheinlichkeit, bei der Optimierung aufzutreten.*

Szenario 2.7.2 *Alle Funktionen, die eine bestimmte syntaktische Eigenschaft (z. B. Linearität, Unimodalität, usw.) besitzen, haben die gleiche Wahrscheinlichkeit, bei der Optimierung aufzutreten.*

Zwar würde man in beide Szenarien keine allgemeinen, sondern spezielle Suchverfahren einsetzen, doch ist die Motivation, allgemeine Suchverfahren auf diesen Mengen zu untersuchen, nicht zu zeigen, dass sie besser als auf diese Szenarien abgestimmte spezielle Suchverfahren sind. Vielmehr wollen wir untersuchen, wie sich verschiedene allgemeine Suchverfahren auf solchen Klassen verhalten. Da man oftmals für solche Klassen weiß, wieviele Ressourcen zu ihrer Optimierung notwendig sind, kann man die Leistung der allgemeinen Suchverfahren nicht nur relativ zueinander vergleichen, sondern sogar absolut. Weiterhin kann es wichtig sein, eine Funktion festzuhalten und zwei Suchverfahren zu vergleichen, von denen das eine effizient optimiert, das andere jedoch nicht. Denn die so trennende Funktion zeigt Grenzen des einen Verfahrens auf, die für das andere nicht gelten. Somit hoffen wir neben der Verbesserung der Analysemethoden einen vertieften Einblick in die Möglichkeiten und Grenzen der untersuchten allgemeinen Suchverfahren zu erhalten.

Kapitel 3

Zur theoretischen Analyse evolutionärer Algorithmen

In diesem Kapitel werden Resultate der theoretischen Analyse evolutionärer Algorithmen vorgestellt. Diese beziehen sich auf den so genannten (1+1) EA, dessen Population nur ein Individuum enthält und der nur Mutation und Selektion benutzt, und Variationen hiervon. Zunächst wird dieser mitsamt des zu untersuchenden Kriteriums, der Laufzeit, formal definiert und es werden einige einfache grundlegende Eigenschaften des Algorithmus bewiesen. Darauf folgt eine Übersicht bestehender Resultate zum (1+1) EA zur Motivation der beschriebenen neuen Ergebnisse.

Diese beginnen mit einfachen oberen und unteren Schranken für die erwartete Laufzeit des (1+1) EA auf relativ großen Klassen von Funktionen, wobei sich diese Grenzen anhand einiger konkreter Funktionen zum Teil als scharf erweisen werden. Anschließend wird die Mutationswahrscheinlichkeit auf $1/n$ eingeschränkt und zuerst für einfache lineare Funktionen wie OneMax und BinVal die erwartete Laufzeit des (1+1) EA als $\Theta(n \log(n))$ nachgewiesen, was danach für alle linearen Funktionen gezeigt wird. Betrachtet man Polynome höheren Grades, so wird die erwartete Laufzeit schon bei Polynomen zweiten Grades exponentiell. Dies wird schon dadurch nahegelegt, dass es NP-vollständige Probleme gibt, die sich als Polynome zweiten Grades ausdrücken lassen, und wird hier anhand der Funktion Distance explizit nachgewiesen. Auch für die Klasse der unimodalen Funktionen wird anhand der Funktion LongPath gezeigt, dass sie Funktionen mit exponentieller Laufzeit enthält. Funktionen, die eine erwartete Laufzeit von $O(n^k \cdot \log(n) + n)$, aber nicht von $O(n^{k-1} \cdot \log(n) + n)$ ($k \in \{1, \ldots, n\}$) haben, werden wir anschließend konstruieren.

Darauf schließt sich die Betrachtung einer statischen bzw. dynamischen Variante des (1+1) EA mit exemplarischen Beweisen an, dass eine dynamische Parameteränderung im Vergleich zu jeder statischen Festsetzung exponentielle erwartete Laufzeiten verhindern kann. Da die erste Variante ein Metropolis-Algorithmus und die zweite ein Simulated Annealing Algorithmus ist, sind die vorgestellten einfachen Beweise auch ein Schritt zur Lösung des Problems, eine natürliche Funktion zu finden, für die ein Simulated Annealing Algorithmus besser ist als jeder Metropolis-Algorithmus (Jerrum und Sinclair (1997)).

Vielen verwendeten Funktionen ist gemein, dass sie keinem bekannten Problem entspringen, sondern explizit so konstruiert sind, dass der betrachtete Algorithmus auf ihnen exponentielle erwartete Laufzeit hat. Demgegenüber entspringt die danach vorgestellte Funktion MaxCount einer Instanz des bekannten NP-harten MAXSAT-Optimierungsproblems; auch für sie kann gezeigt werden, dass der (1+1) EA zu ihrer Optimierung exponentielle erwartete Laufzeit braucht und dass dies sogar für eine größere Klasse von evolutionären Algorithmen gilt.

3.1 Der (1+1) EA

Der (1+1) EA dient der Maximierung einer pseudobooleschen Zielfunktion $f : \{0,1\}^n \mapsto \mathbb{R}$, wobei die Dimension $n \in \mathbb{N}$ des Suchraums eine beliebige, aber feste Zahl ist. Dass als Wertebereich der Zielfunktion die Menge der reellen Zahlen gewählt wird, ist nicht zwingend; jede andere total geordnete Menge ließe sich ebenso gut benutzen. Denn wir werden sehen, dass der (1+1) EA unabhängig von den absoluten Werten, die die Zielfunktion annimmt, arbeitet, solange die Anordnung der Funktionswerte zueinander gleich ist; deshalb kann man durch eine geeignete Umskalierung sogar erreichen, dass als Zielfunktionswerte nur Elemente aus $\{1, \dots, 2^n\}$ angenommen werden.

Doch genug der Vorrede, kommen wir direkt zu einer Definition des (1+1) EA:

Algorithmus 3.1.1 ((1+1) EA) *Sei $f : \{0,1\}^n \mapsto \mathbb{R}$ die zu maximierende Zielfunktion und $p_m(n) \in]0, 1/2]$ die* Mutationsstärke.

1. *Setze $t := 0$.*
2. *Wähle x_t gleichverteilt zufällig aus $\{0,1\}^n$.*
3. *Setze $x'_t := x_t$.*
4. *Negiere alle Bits in x'_t unabhängig voneinander mit Wahrscheinlichkeit $p_m(n)$.*
5. *Falls $f(x'_t) \geq f(x_t)$, setze $x_{t+1} := x'_t$, ansonsten $x_{t+1} := x_t$.*
6. *Setze $t := t + 1$.*
7. *Gehe zu Schritt 3.*

Zuerst sollen in diesem Algorithmus die im ersten Kapitel ausgemachten Hauptbestandteile eines evolutionären Algorithmus bestimmt werden:

- Die Population des (1+1) EA besteht jeweils nur aus einem einzigen Individuum, einem Bitstring der Länge n.
- Die Population wird in Schritt 2 initialisiert. Da ohne Kenntnis der Zielfunktion kein Grund vorliegt, einen Punkt vorzuziehen, wird hier gleichverteilt zufällig ein Bitstring aus $\{0,1\}^n$ gewählt.
- Aus dem Elter x_t wird in den Schritten 3 und 4 das Kind x'_t durch Mutation erzeugt, indem jede Stelle von x_t mit Wahrscheinlichkeit $p_m(n)$ unabhängig von allen anderen negiert wird. Dies ist die aus genetischen Algorithmen bekannte bit-weise Mutation. Somit unterscheidet sich x'_t im Erwartungswert an $p_m(n) \cdot n$ Stellen von x_t, d. h. $p_m(n)$ bestimmt, wie stark das Kind x'_t von seinem Elter x_t abweicht, weshalb dieser Wert *Mutationsstärke* genannt wird (im Gegensatz zur *Mutationswahrscheinlichkeit*, worunter bei evolutionären Algorithmen mit „echten“ Populationen die Wahrscheinlichkeit verstanden wird, dass die Mutation anstatt der bzw. zusätzlich zur Rekombination zur Bildung der Nachkommen verwendet wird).

  Wir setzen $p_m(n) \in]0, 1/2]$ voraus. Denn mit $p_m(n) = 0$ wäre der aktuelle String x_t stets gleich dem Initialstring x_0, da im Mutationsschritt nie ein Bit negiert wird. Wäre $p_m(n) > 1/2$, so würden im Erwartungswert mehr als $n/2$ Bits in einem Mutationsschritt negiert werden. Wenn man den Hamming-Abstand als Maß zur Beurteilung von Nachbarschaft in der Menge $\{0,1\}^n$ heranzieht, so läge x_{t+1} im Erwartungsfall näher zum Komplement $\overline{x_t}$ von x_t als zu x_t selber. Ohne Kenntnis über die Zielfunktion wird man aber eine gewisse Lokalität voraussetzen, d. h. annehmen, dass sich in der Umgebung von

Punkten des Suchraums mit hohem Zielfunktionswert weitere solche Punkte befinden. In diesem Fall macht eine verstärkte Suche in der Umgebung von bereits gefundenen guten Suchpunkten, d. h. ein Wert $p_m(n) \leq 1/2$, Sinn.

An dieser Stelle soll nicht behauptet werden, dass für jede Zielfunktion eine Mutationsstärke $p_m(n) \leq 1/2$ existiert, die den (1+1) EA im Vergleich zu allen Mutationsstärken größer als 1/2 verbessert. Jedoch werden Zielfunktionen, die die angesprochene Lokalitätseigenschaft haben, als wichtiger, da natürlicher als die anderen angesehen, so dass es sinnvoll ist, den Algorithmus auf diese Klasse von Zielfunktionen abzustimmen. Außerdem entspricht eine Mutationsstärke, die größer als 1/2 ist, nicht mehr der anschaulichen Vorstellung, dass die Mutation eines evolutionären Algorithmus ein dem Elter ähnliches Kind erzeugt.

- In Schritt 5 wird aus dem Elter und dem Kind das Individuum der neuen Population selektiert, indem dasjenige mit höherer Fitness gewählt wird, wobei bei Gleichheit der Nachkomme x'_t bevorzugt wird. Letzteres soll Neuerungen, selbst wenn sie den Funktionswert gleichlassen, fördern, wodurch erhofft wird, von großen Mengen, deren Elemente für den evolutionären Algorithmus benachbart sind und gleichen Funktionswert haben (so genannte *Plateaus*), leichter zu anderen, besseren Punkten zu kommen. Die Selektion folgt also der von Evolutionsstrategien bekannten $(\mu + \lambda)$-Strategie.

In dem (1+1) EA kommen also ausschließlich die genetischen Operatoren Mutation und Selektion vor. Die Wahl des Namens ergibt sich einerseits aus der Verwendung der $(\mu + \lambda)$-Selektionstrategie, wobei aufgrund der Populationsgröße $\mu = \lambda = 1$ ist. Da der Selektionsmechanismus von den Evolutionsstrategien übernommen ist, die Benutzung von Bitstrings zur Repräsentation der Individuen aber traditionell im Bereich der genetischen Algorithmen beobachtet werden kann, wird die Bezeichnung „evolutionärer Algorithmus“ gewählt, um klar zu machen, dass der (1+1) EA von keinem der Paradigmen evolutionärer Algorithmen vollständig abgedeckt wird.

Wenn man den (1+1) EA einsetzt, um zu einer gegebenen Zielfunktion $f : \{0,1\}^n \mapsto \mathbb{R}$ einen Punkt $x \in \{0,1\}^n$ mit möglichst großem Zielfunktionswert zu finden, so wird man in Schritt 7 natürlich eine Bedingung angeben müssen, wann der Algorithmus beendet wird. Dafür gibt es u. a. folgende Möglichkeiten:

- Kennt man den maximalen Wert $MAX \in \mathbb{R}$ der Funktion f und will man ein globales Optimum von f, d. h. ein $x \in \{0,1\}^n$ mit $f(x) = MAX$ bestimmen, so sollte man abbrechen, wenn $f(x_t) = MAX$ ist.

- Kennt man eine Schranke $K \in \mathbb{R}$, so dass man einen Punkt sucht, dessen Zielfunktionswert mindestens gleich dieser Schranke ist, so sollte man abbrechen, wenn $f(x_t) \geq K$ ist.

- Hat man über die Funktion keinerlei weitere Informationen, was ja im Black-Box Szenario angenommen wird, so gibt es mindestens zwei oft genutzte Strategien: entweder man bricht ab, wenn sich nach einer bestimmten Anzahl von Generationen die Zielfunktionswerte der jeweils besten Individuen in einer Generation nicht mindestens um einen bestimmten Schwellenwert verbessert haben, oder man bricht nach einer vorgegebenen Anzahl $G \in \mathbb{N}$ von Generationen ab.

Letztere Abbruchstrategie wird im Weiteren untersucht. Für ihren Erfolg ist die Wahrscheinlichkeit entscheidend, mit der nach einer bestimmten Zahl von Generationen ein „gutes“, wenn nicht optimales Individuum gefunden wurde. Die Untersuchung dieser Wahrscheinlichkeit erfordert natürlich eine formale Definition, wann der Algorithmus das Gewünschte getan hat. Da die Qualität eines „guten“ Suchpunkts von der spezifischen Anwendung abhängt und sich, wenn überhaupt, nur mithilfe eines zusätzlichen Parameters, der z. B. den Grad der Abweichung vom optimalen Wert angibt, ausdrücken lässt, wird das von uns untersuchte Ziel stets das exakte Erreichen eines optimalen Suchpunkts sein. Auf dieser Vereinbarung aufbauend wird die Formalisierung wie folgt fortgeführt:

Die algorithmische Beschreibung des (1+1) EA definiert implizit eine zufällige Folge von Elementen $x_t \in \{0,1\}^n$ für $t \in \mathbb{N}_0$, indem die Folge der Elternindividuen x_t betrachtet wird. Dies ähnelt Definition 2.2.1, doch können hier Elemente auch mehrfach in der Folge vorkommen. Sei $X_t^f : \Omega_t \mapsto \{0,1\}^n$ ($t \in \mathbb{N}_0$) die Zufallsvariable (A.2), die die Verteilung des Bitstrings $x_t \in \{0,1\}^n$ im (1+1) EA bei Zielfunktion f in der t-ten Generation widerspiegelt. Dabei wird der zugrundeliegende Wahrscheinlichkeitsraum (Ω_t, P_t) die $n \cdot (t+1)$ bis dahin getroffenen binären Zufallsentscheidungen des (1+1) EA repräsentieren, d. h. Ω_t könnte als $\{0,1\}^{n\cdot(t+1)}$ mit

$$\forall \omega \in \Omega_t : \mathrm{P}_t(\omega) = \left(\frac{1}{2}\right)^n \cdot \prod_{i=n+1}^{n\cdot(t+1)} p_m(n)^{\omega_i} \cdot (1-p_m(n))^{1-\omega_i}$$

gewählt werden. Denn die ersten n in der Initialisierung durchgeführten binären Zufallsentscheidungen sind gleichverteilt, während alle weiteren jeweils Wahrscheinlichkeit $p_m(n)$ bzw. $1 - p_m(n)$ haben. Hierbei wird $\omega_i = 1$ für $i > n$ als Zufallsentscheidung für das Negieren eines Bits während der Mutation interpretiert.

Selbstverständlich kann diese Wahrscheinlichkeitsverteilung auch durch wiederholtes Multiplizieren des anfänglichen Verteilungsvektors $\vec{X}_0^f$ (mit $(\vec{X}_0^f)_x = 1/2^n$ für alle $x \in \{0,1\}^n$) mit der Zustandsüberführungsmatrix M^f berechnet werden, da diese *homogen* ist, d. h. sich nicht über die Zeit ändert. Sie hat 2^n Zeilen und Spalten und enthält, wenn diese mit den Elementen des $\{0,1\}^n$ indiziert werden, die Einträge

$$(M^f)_{x,y} := \begin{cases} 0 & \text{, falls } f(y) < f(x), \\ \sum\limits_{\substack{y \in \{0,1\}^n \\ f(y)<f(x) \vee y=x}} p_m(n)^{H(x,y)} \cdot (1-p_m(n))^{n-H(x,y)} & \text{, falls } y = x, \\ p_m(n)^{H(x,y)} \cdot (1-p_m(n))^{n-H(x,y)} & \text{, sonst.} \end{cases}$$

Dabei bezeichnet $H(x,y)$ den Hamming-Abstand zwischen x und y, d. h. die Zahl der Indizes i mit $x_i \neq y_i$. Somit ergibt sich der Verteilungsvektor $\vec{X}_t^f = (\mathrm{P}(X_t^f = (0,\dots,0)), \dots, \mathrm{P}(X_t^f = (1,\dots,1)))$ als

$$\vec{X}_t^f = \vec{X}_{t-1}^f \cdot M^f = \vec{X}_0^f \cdot (M^f)^t.$$

Wenn diese (oder eine andere passende) Formalisierung des stochastischen Prozesses des (1+1) EA für die Zielfunktion f bekannt ist, so lässt sich für jede beliebige, aber feste Wahl von $t \in \mathbb{N}$, $n \in \mathbb{N}$ und $f : \{0,1\}^n \mapsto \mathbb{R}$ die Verteilung der Zufallsvariablen X_t^f numerisch berechnen. Diese Methode liefert jedoch keinerlei weitergehende Informationen über den Ablauf des (1+1) EA und lässt sich auch nicht auf andere Wahlen von t, n oder f verallgemeinern.

Für die Benutzung einer Implementation des (1+1) EA, die nach G Generationen das aktuelle Element x_G ausgibt, ist auch vorrangig nur die Wahrscheinlichkeit, dass x_G ein Optimum ist, von Interesse. Deshalb wird im Weiteren nicht die komplette Verteilung der Zufallsvariablen X_t^f betrachtet, sondern der erste Zeitpunkt, an dem x_t gleich einem Punkt ist, an dem f seinen maximalen Funktionswert annimmt:

Definition 3.1.2 *Sei* $f : \{0,1\}^n \mapsto \mathbb{R}$ *eine Fitnessfunktion. Dann sei die* Laufzeit $T^f_{p_m(n)}$ *des* (1+1) EA *mit Mutationswahrscheinlichkeit* $p_m(n)$ *auf* f *der erste Zeitpunkt, an dem er eine Maximalstelle von* f *erreicht hat:*

$$T^f_{p_m(n)} := \min\{t \in \mathbb{N}_0 \,|\, f(x_t) = \max\{f(x) \,|\, x \in \{0,1\}^n\}\}.$$

Natürlich ist $T^f_{p_m(n)}$ eine $\mathbb{N}_0$-wertige Zufallsvariable und die Kenntnis ihrer Verteilung ist wichtig, wenn die Implementation des (1+1) EA nach einer bestimmten Anzahl G von Generationen endet. Denn man kann mittels der Verteilung von $T^f_{p_m(n)}$ die Wahrscheinlichkeit bestimmen, mit der zu diesem Zeitpunkt ein Optimum gefunden wurde. Umgekehrt kann man mittels der Verteilung natürlich auch die Zahl der Generationen bestimmen, nach der mit einer vorgegebenen Wahrscheinlichkeit das Optimum gefunden wurde, d. h. Analyseergebnisse können somit auch den Entwurf ganz unmittelbar beeinflussen.

Dass diese Beschränkung auf die Laufzeit sinnvoll ist, wird durch die folgenden Ergebnisse gestützt, die Änderungen an Funktionen klassifizieren, die die Wirkungsweise des (1+1) EA nur unwesentlich beeinflussen sollten. Denn es wird sich herausstellen, dass die Laufzeit gegenüber diesen Veränderungen invariant ist.

Die erste unwesentliche Änderung ist eine *Umskalierung* der Zielfunktion, d. h. der Übergang zu einer neuen Zielfunktion, so dass die Anordnung der Zielfunktionswerte zweier Elemente des Suchraums stets erhalten bleibt. Eine solche Änderung sollte das Verhalten eines Suchverfahrens auf dieser Funktion nicht wesentlich beeinflussen. Dass dies für die Laufzeit des (1+1) EA auch gilt, liegt in der Selektion begründet, dem einzigen von der Zielfunktion f beeinflussten Schritt des (1+1) EA, in die nur das Vorzeichen von $f(x'_t) - f(x_t)$, aber nicht sein Betrag einfließt. Es ist also nur wichtig, ob der neue Punkt einen mindestens so hohen Funktionswert wie der alte hat, der Abstand zwischen ihnen aber nicht. Demzufolge gilt:

Lemma 3.1.3 *Seien* $f_1, f_2 : \{0,1\}^n \mapsto \mathbb{R}$. *Wenn für alle* $x, y \in \{0,1\}^n$ *gilt, dass* $f_1(x) < f_1(y)$ *äquivalent zu* $f_2(x) < f_2(y)$ *ist, so gilt für alle Mutationsstärken* $p_m(n) \in]0, 1/2]$, *dass* $T^{f_1}_{p_m(n)}$ *und* $T^{f_2}_{p_m(n)}$ *dieselbe Verteilung haben.*

Weiterhin spielt eine Reihenfolgeänderung der Bits des Arguments der Zielfunktion keine Rolle, da alle Stellen von Initialisierung, Mutation und Selektion gleich behandelt werden:

Lemma 3.1.4 *Sei* $\pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$ *eine Permutation und* $f : \{0,1\}^n \mapsto \mathbb{R}$. *Dann gilt für die Funktion* $f' : \{0,1\}^n \mapsto \mathbb{R}$, *definiert als*

$$\forall x \in \{0,1\}^n : f'(x_1, \ldots, x_n) := f(x_{\pi(1)}, \ldots, x_{\pi(n)}),$$

und alle Mutationsstärken $p_m(n) \in]0, 1/2]$, *dass* $T^f_{p_m(n)}$ *und* $T^{f'}_{p_m(n)}$ *dieselbe Verteilung haben.*

Dass beide behandelten Umformungen einer Funktion keine Auswirkungen auf die Laufzeit des (1+1) EA haben, entspricht der Vorstellung, dass ein sinnvolles Leistungsmaß durch diese Umformungen nicht beeinflusst werden sollte. Insofern erfüllt die Laufzeit diese beiden Minimalanforderungen.

Wiederum kann man einwenden, dass für praktische Anwendungen nicht die exakte, sondern nur die näherungsweise Optimierung im Vordergrund steht, so dass die Untersuchung der Anzahl der Schritte, bis ein Suchpunkt mit einem um höchstens ε Prozent niedrigeren Funktionswert als das Optimum gefunden wurde, sinnvoller wäre. Dennoch wird im Folgenden der Wert $T^f_{p_m(n)}$ untersucht, da nach Lemma 3.1.3 jede Zielfunktion so verändert werden kann, dass der Funktionswert der Optima um einen beliebigen Prozentsatz größer ist als der Wert der zweitbesten Punkte, indem man die Werte der Optima einfach erhöht. Wenn ε unterhalb dieses Prozentsatzes liegt, sind beide Laufzeiten gleich, so dass eine getrennte Untersuchung sinnlos ist. Mit derselben Konsequenz könnte man die Funktionswerte aller Punkte, die um höchstens ε Prozent niedriger als das Optimum sind, auf den des Optimums setzen. Auch wenn durch diese Veränderungen der Funktionswerte in der Regel die neue Funktion nicht mehr zu derselben Klasse von Funktionen wie die alte gehört (z. B. bei linearen oder quadratischen Funktionen), so lassen sich doch in allen Funktionsklassen, die wir im Folgenden betrachten, Beispiele finden, bei denen die Optima zumindest um einen von n unabhängigen konstanten Faktor besser sind als der zweitbeste Wert. Deshalb wird im Weiteren nur die Laufzeit bis zum genauen Erreichen eines Optimums betrachtet.

Wenn sich die genaue Untersuchung der Verteilung der Laufzeit $T^f_{p_m(n)}$ noch als zu schwierig erweist, werden wir stattdessen nur den Erwartungswert $\mathrm{E}(T^f_{p_m(n)})$ der Laufzeit untersuchen. Auch wenn aus dem Erwartungswert nicht die Verteilung bestimmt werden kann, so lässt sich diese mithilfe der Markoff-Ungleichung (A.8) abschätzen. Denn jede obere Abschätzung $t \in \mathbb{R}^+$ für den Erwartungswert $\mathrm{E}(T^f_{p_m(n)})$ liefert für beliebiges $c \in \mathbb{R}^+$ eine untere Schranke von $1 - 1/c$ für die Wahrscheinlichkeit, dass nach spätestens $c \cdot t$ Schritten das Optimum gefunden wurde:

$$\mathrm{P}\left(T^f_{p_m(n)} \leq c \cdot t\right) \geq 1 - \mathrm{P}\left(T^f_{p_m(n)} \geq c \cdot \mathrm{E}(T^f_{p_m(n)})\right) \geq 1 - \frac{1}{c}.$$

Diese Schranke ist jedoch oft recht ungenau, weshalb wir im Folgenden häufig versuchen, die Wahrscheinlichkeit, mit der eine Laufzeit mindestens bzw. höchstens benötigt wird, direkter abzuschätzen.

Für festes $n \in \mathbb{N}$ und zwei fest gewählte Funktionen $f_1, f_2 : \{0,1\}^n \mapsto \mathbb{R}$ lassen sich die erwarteten Laufzeiten des (1+1) EA durch vollständige Berechnung der Wahrscheinlichkeitsverteilungen von $T^{f_1}_{p_m(n)}$ und $T^{f_2}_{p_m(n)}$ bestimmen. Damit kann zwar festgestellt werden, ob f_1 für den (1+1) EA „schwieriger“ als f_2 ist, wenn man die erwartete Laufzeit als geeignetes Maß akzeptiert. Doch dies lässt keine Verallgemeinerungen auf andere Funktionen zu. Insbesondere nicht einmal auf Funktionen, die denselben funktionalen Zusammenhang wie f_1 bzw. f_2 darstellen, jedoch Argumente anderer Länge benutzen.

Um jedoch die Möglichkeiten und Grenzen eines Algorithmus zu erkennen, sind Untersuchungen auf ganzen Folgen von Funktionen $(f_n)_{n\in\mathbb{N}} : \{0,1\}^n \mapsto \mathbb{R}$ sinnvoll, wenn diesen ein einheitlicher funktionaler Zusammenhang zugrunde liegt. Wird nun die erwartete Laufzeit $\mathrm{E}(T^{f_n}_{p_m(n)})$ betrachtet, so ist dies eine Funktion in n. Somit sind Aussagen über alle Funktionen der Folge und damit weitergehende Vergleiche möglich. Der Schwerpunkt unserer theoretischen Untersuchungen wird also auf der asymptotischen Analyse solcher Funktionenfolgen liegen. Die von n unabhängigen Konstanten werden dabei oft vernachlässigt, in der Überzeugung, dass die wahre Komplexität einer Funktionenfolge durch die Größenordnung des Wachstums in n ausgedrückt wird. Dies ist das in der Komplexitätstheorie (Papadimitriou (1994)) und Algorithmenanalyse (Motwani und Raghavan (1995)) gängige Verfahren.

Wenn in den nächsten Abschnitten Resultate über die erwarteten Laufzeiten des (1+1) EA zitiert bzw. vorgestellt werden, so wird, um die Notation einfach zu halten, auf die ausdrückliche Angabe der Länge n verzichtet. Ebenso ist mit „Funktion“, wenn nicht ausdrücklich anders vermerkt, stets eine ganze Funktionenfolge gemeint, deren Elementen ein gemeinsamer Aufbau zugrundeliegt.

Im nächsten Abschnitt werden wir nun der Frage nachgehen, für welche Funktionen bzw. sogar Klassen von Funktionen Resultate zur erwarteten Laufzeit des (1+1) EA bekannt sind, welche Analysemethoden dafür benutzt wurden und wo offene Fragen existieren.

3.2 Bekannte Resultate zur Analyse des (1+1) EA

Der (1+1) EA wurde schon öfter als ein sehr einfacher, aber doch wichtige Prinzipien verkörpernder evolutionärer Algorithmus untersucht. So wurde in Mühlenbein (1992) der (1+1) EA für die Zielfunktion OneMax (Definition 3.4.1, eine einfache lineare Funktion, die die Anzahl der Einsen in ihrem Argument misst) untersucht und die erwartete Laufzeit bei Mutationsstärke $p_m(n) = k/n$ approximativ als $\exp(k) \cdot (n/k) \cdot \ln(n/2)$ bestimmt. Approximativ bedeutet in diesem Zusammenhang, dass während der Rechnung einige Vereinfachungen durchgeführt wurden, ohne den resultierenden Fehler zu begrenzen oder die Art der Abschätzung (nach oben bzw. nach unten) zu bestimmen. Die dabei benutzte Vorgehensweise lässt sich jedoch leicht dazu nutzen, die erwartete Laufzeit mathematisch exakt als $\Theta(n \log(n))$ nachzuweisen (siehe Lemma 3.4.2). Zugleich wird in Mühlenbein (1992) die Vermutung aufgestellt, dass diese Laufzeit nicht nur für die Optimierung von OneMax gilt, sondern für jede unimodale Funktion, ohne diesen Begriff zu definieren. Wir werden in Theorem 3.4.24 für eine kanonische Definition von Unimodalität zeigen, dass es unimodale Funktionen mit exponentieller erwarteter Laufzeit gibt.

In Bäck (1992) wird der (1+1) EA ebenfalls auf der Funktion OneMax untersucht, jedoch in Hinblick auf eine die Erfolgswahrscheinlichkeit in jedem Schritt optimierende Mutationsstärke. Die sich dabei durch numerische Analyse einiger Werte von n ergebenden Resultate deuten darauf hin, dass gegen Ende des Optimierprozesses, wenn nur noch wenige Bits falsch gesetzt sind, eine Mutationsstärke von $1/n$ optimal zu sein scheint. Dies wird in Bäck (1993) für die Zielfunktionen OneMax und BinVal (Definition 3.4.5) fortgesetzt, wobei die hier durchgeführten numerischen Analysen ebenfalls einen Wert von $p_m(n) = 1/n$ gegen Ende des Suchprozesses empfehlen. Aus diesen Untersuchungen heraus hat sich $p_m(n) = 1/n$ als die Standardwahl der Mutationsstärke ergeben, weshalb auch für unsere Analysen vorrangig dieser Wert benutzt wird.

Intensiv wird der (1+1) EA in Rudolph (1997) untersucht. Neben Untersuchungen globaler Konvergenzeigenschaften wird die erwartete Laufzeit des (1+1) EA für verschiedene Funktionsklassen analysiert. Für OneMax wird die erwartete Laufzeit als $O(n \log(n))$ nachgewiesen, für lineare Zielfunktionen wird dieselbe Schranke für die Variante des (1+1) EA nachgewiesen, die in jeder Mutation nur genau ein zufälliges Bit negiert. Die erwartete Laufzeit des ursprünglichen (1+1) EA auf linearen Funktionen wird nicht bestimmt. Für LeadingOnes (Definition 3.4.16) wird gezeigt, dass die erwartete Laufzeit des (1+1) EA gleich $O(n^2)$ ist, eine untere Schranke von $\Omega(n^2)$ wird nicht nachgewiesen. Wir werden dies in Lemma 3.4.18 nachreichen und somit die erwartete Laufzeit des (1+1) EA als $\Theta(n^2)$ bestimmen. Die in Mühlenbein (1992) angestellte Vermutung, dass alle unimodalen Funktionen vom (1+1) EA in erwarteter Zeit $O(n \log(n))$ optimiert werden, kann von Rudolph (1997) zwar nicht widerlegt werden, doch präsentiert er experimentelle Daten zur unimodalen Funktion LongPath_2 (Definition 3.4.20), die eine Laufzeit von $\Omega(n^3)$ nahelegen. Ein Beweis erfolgt jedoch nur für die obere Schranke $O(n^3)$, womit die Vermutung in Horn, Goldberg und Deb (1994), dass die erwartete Laufzeit exponentiell ist, widerlegt wird. Mit Theorem 3.4.24 werden wir zeigen, dass eine ähnlich aufgebaute unimodale Funktion von dem (1+1) EA nur in exponentieller erwarteter Zeit optimiert werden kann. Insgesamt finden sich in Rudolph (1997) die bislang weitreichendsten Untersuchungen zum (1+1) EA; viele der dort angesprochenen Probleme werden von uns weiterverfolgt.

In Salomon (1996) ist zwar nicht der (1+1) EA, sondern ein allgemeinerer genetischer Algorithmus Gegenstand der Untersuchung, doch sind die dort angestellten approximativen Berechnungen trotzdem von Interesse. Denn dort wird die generelle Vermutung geäußert, dass evolutionäre Algorithmen, die auf Bitstrings mit einer bitweisen Mutation mit Mutationsstärke $1/n$ arbeiten, zur Optimierung von linearen

Funktionen erwartete Zeit $O(n \log(n))$ benötigen. Die dort angestellten Berechnungen können nicht als Beweis dienen, da vereinfachend nur der Fall betrachtet wird, dass genau ein Bit in der Mutation negiert wird. Obwohl dies natürlich die erwartete Zahl von mutierenden Bits ist, ist die Wahrscheinlichkeit, dass eine konstant große Anzahl $k \geq 2$ von Bits mutiert, konstant groß. Denn da

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \cdots \cdot (n-k+1)}{k!} \geq \frac{(n-k+1)^k}{k!} = \Theta(n^k)$$

ist, beträgt die Wahrscheinlichkeit einer Mutation von k Bits mindestens

$$\binom{n}{k} \cdot \left(\frac{1}{n}\right)^k \cdot \left(1 - \frac{1}{n}\right)^{n-k} \geq c \cdot n^k \cdot \left(\frac{1}{n}\right)^k \cdot \exp(-1) = c \cdot \exp(-1).$$

Dabei ist $c > 0$ eine von n unabhängige Konstante und zur letzten Abschätzung wurde die noch häufig verwendete Ungleichung (A.12) benutzt. Die Vernachlässigung von Mutationen von mehr als einem Bit kann also schon allein aus diesem Grund nicht ohne eine nähere Fehlerabschätzung durchgeführt werden (auch wenn der Wert der Konstanten c, wie wir noch sehen werden, exponentiell in k sinkt). Dass die angestellte Vermutung aber richtig ist, wird in Theorem 3.4.9 mit einem mathematisch exakten Beweis gezeigt.

In vielen anderen Untersuchungen dienen der (1+1) EA und seine Varianten als Vergleichsverfahren, mit denen sich komplizierter aufgebaute Algorithmen messen müssen. So fungiert in Forrest und Mitchell (1993) die Variante des (1+1) EA, die in jedem Schritt genau ein zufällig gewähltes Bit mutiert, als Messlatte für die Leistung eines genetischen Algorithmus mit 1-Punkt Crossover, um auf der Royal-Road-Funktion die Building-Block-Hypothese experimentell zu belegen.

Der (1+1) EA ist also, obwohl häufig betrachtet, theoretisch nicht gut verstanden. Seine Laufzeit wurde nur für einfache Funktionen wie OneMax, BinVal oder LongPath theoretisch abgeschätzt. Aufgrund seiner Einfachheit bietet er jedoch die ideale Möglichkeit, viele Vermutungen über evolutionäre Algorithmen zu überprüfen und allgemeiner verwendbare Beweisverfahren zu deren Untersuchung zu erproben.

3.3 Einfache untere und obere Schranken für den (1+1) EA

In diesem Abschnitt werden für große Klassen von Zielfunktionen die erwarteten Laufzeiten des (1+1) EA für beliebige Mutationswahrscheinlichkeiten $p_m(n) \in]0, 1/2]$ abgeschätzt. Damit werden grundlegende Methoden, mit denen spätere kompliziertere Analysen durchgeführt werden, eingeführt und eine erste Einschätzung der Laufzeiten des (1+1) EA ermöglicht. Der Fall, dass $p_m(n) = 1/2$ ist und der (1+1) EA damit zur rein zufälligen Suche entartet, wird kurz behandelt und als ineffizient erkannt.

Zuerst soll untersucht werden, wie groß die erwartete Laufzeit des (1+1) EA maximal werden kann. Hier gilt folgende obere Schranke:

Lemma 3.3.1 *Sei $f : \{0,1\}^n \mapsto \mathbb{R}$ eine zu maximierende Zielfunktion. Dann gilt für die Laufzeit $T^f_{p_m(n)}$ des* (1+1) EA *mit Mutationswahrscheinlichkeit $p_m(n)$ auf f, dass $\mathrm{E}(T^f_{p_m(n)}) \leq (1/p_m(n))^n$ ist.*

Beweis: Wenn x^* eine der Maximalstellen von f ist, so gilt für alle $x \in \{0,1\}^n$, dass die Wahrscheinlichkeit, im Mutationsschritt 4 von Algorithmus 3.1.1 den aktuellen String x_t zu x^* zu mutieren, gleich $p_m(n)^{H(x_t,x^*)} \cdot (1 - p_m(n))^{n-H(x_t,x^*)}$ ist.

Da $1 - p_m(n) \geq p_m(n)$ ist, ist diese Wahrscheinlichkeit mindestens gleich $p_m(n)^n$. Man kann demnach für jedes t und jedes x_t die Wahrscheinlichkeit, im nächsten Schritt zu der ausgezeichneten Maximalstelle x^* zu mutieren, durch $p_m(n)^n$ nach unten abschätzen und erhält demnach mit der erwarteten Wartezeit auf ein solches Ereignis eine obere Schranke für $\mathrm{E}(T^f_{p_m(n)})$.

Da der Zeitpunkt, an dem ein Ereignis mit Wahrscheinlichkeit p zum ersten Mal bei unabhängigen Wiederholungen auftritt, geometrisch mit Parameter p verteilt ist und deshalb den Erwartungswert $1/p$ hat (A.5), ist $(1/p_m(n))^n$ eine obere Schranke für $\mathrm{E}(T^f_{p_m(n)})$. □

Lemma 3.3.1 stellt eine allgemeine obere Schranke für die erwartete Laufzeit des (1+1) EA dar, die mit wachsendem $p_m(n)$ kleiner wird, da sie allein die erwartete Wartezeit auf eine Mutation zu einem Optimum abschätzt. Es zieht dabei aber nicht in Betracht, dass in einem „normalen" Lauf des (1+1) EA die Zielfunktionswerte $f(x_t)$ stetig anwachsen. Wenn durch diese Steigerung der Funktionswerte die Wahrscheinlichkeit, sich dem Optimum weiter anzunähern, groß bleibt, so wird die erwartete Laufzeit klein sein.

Diese Überlegungen formal genau beschreiben zu können, ist Aufgabe der folgenden Definition:

Definition 3.3.2 *Sei $f : \{0,1\}^n \mapsto \{0,1\}$ eine zu maximierende Zielfunktion. Eine Partition $Z_1, \ldots, Z_N \subseteq \{0,1\}^n$ $(N \geq 2)$ des $\{0,1\}^n$ bildet eine* Ebeneneinteilung *bzgl. f, wenn für alle $i, j \in \{1, \ldots, N\}$, $x \in Z_i$ und $y \in Z_j$ stets aus $i < j$ folgt, dass $f(x) < f(y)$ ist, und Z_N nur die Optima von f enthält. Die Teilmengen $Z_1, \ldots, Z_N$ heißen dann* Ebenen *von f.*

Entscheidend ist, dass der (1+1) EA keine Verschlechterungen zulässt. Dies impliziert, dass, wenn jemals der aktuelle Bitstring x_t aus der Ebene Z_i kommt, kein danach erreichter Bitstring in einer niedrigeren Ebene $Z_{i'}$, d. h. mit $i' < i$, liegen kann. Sei nun $p_i \in]0,1]$ $(i \in \{1, \ldots, N-1\})$ das Minimum über alle $x \in Z_i$ der Wahrscheinlichkeiten, dass der (1+1) EA von x aus zu einem Punkt einer höheren Ebene $Z_{i'}$ mit $i' > i$ wechselt. Betrachten wir nun den Markoff-Prozess mit N Zuständen, der vom i-ten Zustand $(i \in \{1, \ldots, N-1\})$ mit Wahrscheinlichkeit p_i in den $(i+1)$-ten Zustand wechselt und ansonsten in dem i-ten Zustand bleibt. Dann ist seine erwartete Zeit, vom i-ten in den N-ten Zustand zu kommen, eine obere Schranke für die erwartete Zeit des (1+1) EA, von einem $x \in Z_i$ zu einem Optimum zu kommen.

Dies folgt leicht induktiv: für $i = n-1$ ist dies klar, da p_{n-1} eine untere Schranke der Wahrscheinlichkeit, von $x \in Z_{n-1}$ zu einem Optimum zu wechseln, ist. Geht man induktiv von $i+1$ zu i, so wird der neue Prozess in erwarteter Zeit $1/p_i$ den $(i+1)$-ten Zustand erreichen. Dieser Wert ist eine obere Schranke für die erwartete Dauer, bis der (1+1) EA von einem $x \in Z_i$ zu einem Punkt einer höheren Ebene gewechselt ist. Da die erwarteten Zeiten des neuen Prozesses, vom i-ten Zustand zu einem Optimum zu kommen, mit steigendem i nicht steigen können, folgt mit der Induktionsvoraussetzung die Behauptung.

Um diese Überlegung auf den (1+1) EA anzuwenden, kann noch die Wahrscheinlichkeit $|Z_i|/2^n$, in Ebene Z_i zu initialisieren, berücksichtigt werden. Ist eine Abschätzung der Größen der Ebenen zu aufwendig oder nicht vielversprechend, liefert $\sum_{i=1}^{N-1} 1/p_i$, die erwartete Zeit, um vom ersten Zustand in den N-ten Zustand zu kommen, eine obere Schranke für die Laufzeit des (1+1) EA:

Lemma 3.3.3 *Seien $f : \{0,1\}^n \mapsto \mathbb{R}$ die zu maximierende Funktion und die Mengen $Z_1, \ldots, Z_N \subseteq \{0,1\}^n$ eine Ebeneneinteilung des $\{0,1\}^n$ bzgl. f. Ist für alle $i \in \{1, \ldots, N-1\}$ und $x \in Z_i$ die Wahrscheinlichkeit des* (1+1) EA *mit Mutationsstärke $p_m(n)$ auf f, von x in einen Punkt $y \in Z_{i+1} \cup \cdots \cup Z_N$ zu wechseln,* *mindestens gleich* $p_i \in]0,1]$, *so gilt*

$$\mathrm{E}(T^f_{p_m(n)}) \leq \sum_{i=1}^{N-1} 1/p_i.$$

Dieses Lemma stellt eine einfache und bei geschickter Ebeneneinteilung oftmals recht genaue Methode dar, die erwartete Laufzeit des (1+1) EA nach oben abzuschätzen. Eine Anwendung, die eine asymptotisch optimale obere Schranke ergibt, stellt Lemma 3.4.2 für die Zielfunktion OneMax im nächsten Abschnitt dar.

Eine für OneMax optimale allgemeine untere Schranke der erwarteten Laufzeit erhält man durch folgende Überlegung für Funktionen mit nur einem Optimum: hat der Punkt, in dem der (1+1) EA initialisiert, vom Optimum der Funktion einen Hamming-Abstand von mindestens d, so ist eine zum Erreichen des Optimums notwendige Bedingung, dass in Schritt 4 des (1+1) EA über alle durchgeführten Generationen zumindest diese d Bits ausgewählt wurden, d. h. *mutieren wollten.* Dies ergibt in Anlehnung an die Analyse des *Coupon Collector's Problem* aus Motwani und Raghavan (1995) folgende untere Schranke:

Lemma 3.3.4 *Besitzt die Zielfunktion* $f : \{0,1\}^n \mapsto \{0,1\}$ *genau ein Optimum, so ist für jedes* $c \in \mathbb{R}$ *und jede Konstante* $c_1 \in]0,1/2[$:

$$\mathrm{P}\left(T^f_{p_m(n)} > \frac{1-p_m(n)}{p_m(n)} \cdot (\ln(n) - c)\right) \geq (1-\exp(-\Omega(n)))\cdot(1-\exp(-c_1 \cdot \exp(c)))\,.$$

Beweis: Nach Lemma 3.1.3 können wir annehmen, dass $(1,\dots,1)$ das einzige Optimum von f ist. Zuerst berechnen wir die Wahrscheinlichkeit, dass der (1+1) EA in einem Punkt mit höchstens $(1/2+\varepsilon)\cdot n$ Einsen initialisiert, wobei $\varepsilon > 0$ eine Konstante ist. Da sich die Zahl Z der Einsen im Initialstring als Summe von n gleichverteilten $\{0,1\}$-wertigen Zufallsvariablen ergibt, folgt aus der Tschernoff-Ungleichung (A.9)

$$\mathrm{P}\left(Z > \left(\frac{1}{2}+\varepsilon\right)\cdot n\right) = \mathrm{P}\left(Z > (1+2\cdot\varepsilon)\cdot\frac{n}{2}\right) \leq \exp\left(-\frac{2\cdot\varepsilon^2\cdot n}{3}\right).$$

Somit ist die Wahrscheinlichkeit für jede Konstante $c_1 \in]0,1/2[$ exponentiell hoch, dass nach der Initialisierung mindestens $c_1 \cdot n$ Bits falsch, d. h. gleich Null gesetzt sind. Gibt die Zufallsvariable T^* den ersten Zeitpunkt an, zu dem alle $c_1 \cdot n$ ausgezeichneten Nullen mutieren wollten, so ist also

$$(1-\exp(-\Omega(n)))\cdot \mathrm{P}(T^* > t)$$

eine untere Schranke für $\mathrm{P}(T^f_{p_m(n)} > t)$.

Dass eine ausgezeichnete Stelle in einem Schritt nicht mutieren will, hat Wahrscheinlichkeit $1-p_m(n)$; dass sie in t Schritten nicht mutieren will, hat Wahrscheinlichkeit $(1-p_m(n))^t$; dass sie in t Schritten mindestens einmal mutieren will, hat demgemäß Wahrscheinlichkeit $1-(1-p_m(n))^t$; dass dies für $c_1 \cdot n$ ausgezeichnete Stellen gilt, hat Wahrscheinlichkeit $(1-(1-p_m(n))^t)^{c_1\cdot n}$. Also ist die Wahrscheinlichkeit $\mathrm{P}(T^* > t)$ für $t = \frac{1-p_m(n)}{p_m(n)} \cdot (\ln(n)-c))$ gleich

$$\begin{aligned}
& 1-\left(1-(1-p_m(n))^t\right)^{c_1\cdot n} \\
= \;& 1-\left(1-(1-p_m(n))^{(p_m(n)^{-1}-1)\cdot(\ln(n)-c)}\right)^{c_1\cdot n} \\
\geq \;& 1-\left(1-\frac{\exp(c)}{n}\right)^{n\cdot\exp(c)^{-1}\cdot c_1\cdot\exp(c)} \geq 1-\exp(-c_1\cdot\exp(c)).
\end{aligned}$$

□

Da für $c = \ln(n)/2$ die Wahrscheinlichkeit, mehr als $(1-p_m(n))\cdot\ln(n)/(2\cdot p_m(n))$ Schritte zu benötigen, somit exponentiell hoch ist, folgt:

Korollar 3.3.5 *Besitzt die Zielfunktion $f : \{0,1\}^n \mapsto \{0,1\}$ genau ein Optimum, so gilt:*

$$\mathrm{E}\left(T^f_{p_m(n)}\right) = \Omega\left(\frac{1-p_m(n)}{p_m(n)} \cdot \log(n)\right).$$

Die unteren Schranken von Lemma 3.3.4 bzw. Korollar 3.3.5 liefern für $p_m(n) = 1/n^k$ die Größenordnung

$$\Omega\left(\log(n) \cdot n^k \cdot \left(1 - \frac{1}{n^k}\right)\right) = \Omega\left(\log(n) \cdot n^k\right).$$

Damit ist klar, dass die Mutationsstärke nicht zu klein gewählt werden darf, da sonst die Laufzeit mit exponentiell hoher Wahrscheinlichkeit stark anwächst.

Dabei ist Lemma 3.3.4 aussagekräftiger als das daraus folgende Korollar 3.3.5. Denn einerseits lässt sich ersteres nicht aus letzterem folgern, andererseits kann auch selbst bei einer exponentiell hohen unteren Schranke für den Erwartungswert der Laufzeit die betrachtete Zielfunktion ggf. durch eine leichte Modifikation des (1+1) EA mit hoher Wahrscheinlichkeit effizient, d. h. in polynomieller Laufzeit, optimiert werden.

In diesem Fall könnte nämlich eine Multistartvariante des (1+1) EA benutzt werden:

Definition 3.3.6 *Die* Multistartvariante *eines Suchverfahrens mit den Parametern $T(n)$ und $L(n)$ besteht darin, das Suchverfahren nach $\lceil T(n) \rceil$ Schritten abzubrechen und dann mit einem neuen Lauf zu beginnen, wobei eines der Elemente mit höchstem Zielfunktionswert über alle Läufe gespeichert wird. Dies wird $\lceil L(n) \rceil$-mal wiederholt. Danach wird das Suchverfahren ohne Schrittbegrenzung laufen gelassen.*

Zur Berechnung der Wahrscheinlichkeit, ein Optimum durch eine Multistartvariante gefunden zu haben, ist folgende Abschätzung zentral:

Lemma 3.3.7 *Ist die Wahrscheinlichkeit, dass ein Suchverfahren in $T(n)$ Schritten ein Optimum findet, mindestens gleich $\varepsilon(n) > 0$, so ist die Wahrscheinlichkeit, dass seine Multistartvariante mit Parametern $T(n)$ und $L(n)$ kein Optimum findet, höchstens* $\exp(-L(n) \cdot \varepsilon(n))$.

Beweis: Da die einzelnen Läufe unabhängig voneinander stattfinden, ist die Wahrscheinlichkeit, dass nach Ende der $L(n)$ Wiederholungen kein Optimum gefunden wird, höchstens

$$(1 - \varepsilon(n))^{L(n)} = \left(1 - \frac{1}{\varepsilon(n)^{-1}}\right)^{\varepsilon(n)^{-1} \cdot L(n) \cdot \varepsilon(n)} \leq \exp(-L(n) \cdot \varepsilon(n)).$$

□

Wenn der (1+1) EA also auf einer Funktion mit Wahrscheinlichkeit $\varepsilon(n)$ Laufzeit $T(n)$ hat, so erhält man durch $n/\varepsilon(n)$-fache Wiederholung von Phasen der Länge $T(n)$ einen Algorithmus, der in Zeit $n/\varepsilon(n) \cdot T(n)$ mit einer Wahrscheinlichkeit von mindestens $1 - \exp(-n)$ das Optimum findet. Die Tatsache, dass der (1+1) EA exponentielle erwartete Laufzeit für eine Funktion hat, bedeutet also nicht, dass diese Funktion nicht nach einer leichten Änderung des Algorithmus mit an Sicherheit grenzender Wahrscheinlichkeit in polynomieller Zeit optimiert werden kann, wenn nach polynomiell vielen Schritten $T(n)$ mit Wahrscheinlichkeit $1/poly(n)$ ein Optimum gefunden wird. Denn ist in obiger Überlegung $\varepsilon(n) = 1/p(n)$ für ein Polynom $p(n)$, so hat die Multistartvariante mit Wahrscheinlichkeit $1 - \exp(-n)$ nach $n \cdot p(n) \cdot T(n)$, also polynomiell vielen Schritten, ein Optimum gefunden.

Wenn man zudem eine obere Schranke von $\exp(n^k)$ mit konstantem $k \in \mathbb{N}$ für die erwartete Laufzeit des (1+1) EA kennt, kann man durch häufigere Wiederholung auch erreichen, dass die Wahrscheinlichkeit, vor dem letzten Aufruf des (1+1) EA kein Optimum erreicht zu haben, so gering ist, dass die erwartete Laufzeit polynomiell ist. Da für $L(n) = p(n) \cdot n^k$ Wiederholungen die Wahrscheinlichkeit, kein Optimum gefunden zu haben, maximal gleich $\exp(-n^k)$ ist, ist der Anteil des letzten Laufs der Multistartvariante am gesamten Erwartungswert konstant groß, so dass die erwartete Laufzeit der Multistartvariante polynomiell beschränkt ist.

Diese Erkenntnisse seien zusammengefasst:

Korollar 3.3.8 *Sei die Wahrscheinlichkeit, dass der* (1+1) EA *mit Mutationsstärke* $p_m(n) \in]0, 1/2]$ *nach* $p_1(n)$ *Schritten ein Optimum der Zielfunktion* f *gefunden hat, gleich* $1/p_2(n)$*, wobei* $p_1(n)$ *und* $p_2(n)$ *polynomiell beschränkt sind. Dann hat die Multistartvariante des* (1+1) EA *mit* $T(n) = p_1(n)$ *und* $L(n) = n \cdot p_2(n)$ *nach* $L(n) \cdot T(n)$ *Schritten mit einer Wahrscheinlichkeit von mindestens* $1 - \exp(-n)$ *ein Optimum gefunden. Ist zusätzlich die erwartete Laufzeit* $\mathrm{E}(T^f_{p_m(n)}) = O(\exp(n^k))$ *für eine Konstante* $k \in \mathbb{N}$*, so hat die Multistartvariante des* (1+1) EA *mit* $T(n) = p_1(n)$ *und* $L(n) = p_2(n) \cdot n^k$ *polynomielle erwartete Laufzeit.*

Die vorgestellten oberen und unteren Schranken sind für beliebige Mutationsstärken $p_m(n)$ und auf große Klassen von Zielfunktionen anwendbar und könnten deshalb sehr ungenau sein. Um zu zeigen, dass dem für die obere Schranke nicht so ist, soll nun kurz der (1+1) EA mit Mutationsstärke $p_m(n) = 1/2$ besprochen werden, mit der dieser zur rein zufälligen Suche entartet.

Aus Lemma 3.3.1 folgt, dass die erwartete Laufzeit des (1+1) EA für eine beliebige Zielfunktion höchstens gleich 2^n ist. Diese Schranke wird aber auch angenommen, wenn die Zielfunktion nur ein Optimum besitzt:

Lemma 3.3.9 *Sei* $f : \{0,1\}^n \mapsto \mathbb{R}$ *eine zu maximierende Zielfunktion mit genau* k *Optima. Dann gilt* $\mathrm{E}(T^f_{1/2}) = 2^n/k$*.*

Beweis: Bei Mutationsstärke 1/2 ist für alle $t \in \mathbb{N}$ und $y, z \in \{0,1\}^n$ die Wahrscheinlichkeit $\mathrm{P}(x'_t = y \,|\, x_t = z)$ gleich $(1/2)^n$, d. h. die Folge der Kinder x'_t ist rein zufällig, auch wenn natürlich für den Selektionsschritt die Zielfunktion eine Rolle spielt.

Also ist für jedes $t \in \mathbb{N}$ die Wahrscheinlichkeit, dass x'_t eines der Optima ist, gleich $k/2^n$. Da die Wartezeit auf dieses Ereignis geometrisch mit Parameter $k/2^n$ verteilt ist, ist die erwartete Wartezeit gleich $2^n/k$. □

Da es also eine die Voraussetzungen von Lemma 3.3.1 erfüllende Funktion f gibt, für die $\mathrm{E}(T^f_{p_m(n)}) = (1/p_m(n))^n$ ist, kann es eine kleinere obere Schranke nur für eine eingeschränktere Menge von Funktionen oder einen kleineren Bereich von Mutationsstärken geben. Da die aus Lemma 3.3.1 resultierenden oberen Schranken, wie wir noch sehen werden, für viele Funktionen sehr ungenau sind, werden wir zur Berechnung genauerer oberer Schranken andere Methoden einsetzen.

Lemma 3.3.9 zeigt, dass für Funktionen mit höchstens $2^{\varepsilon \cdot n}$ Optima (mit von n unabhängigem $\varepsilon \in [0, 1]$) die erwartete Laufzeit des (1+1) EA mit Mutationsstärke 1/2 genau $2^{(1-\varepsilon) \cdot n}$ ist. Nun könnte man einwenden, dass dieses einer effizienten Optimierung einer Funktion mit einer Multistartvariante des (1+1) EA mit Mutationsstärke 1/2 nicht im Wege stehen muss, wenn dieser mit relativ hoher Wahrscheinlichkeit in polynomieller Laufzeit ein Optimum findet.

Wie sieht es aber für den (1+1) EA mit Mutationsstärke 1/2 aus, hat er mit relativ hoher Wahrscheinlichkeit eine polynomielle Laufzeit oder braucht er fast immer exponentielle Laufzeit? Diese Frage beantwortet das folgende einfache Lemma zugunsten der zweiten Möglichkeit:

Lemma 3.3.10 *Die Wahrscheinlichkeit, dass der* (1+1) EA *mit Mutationsstärke* 1/2 *nach* $T \in \mathbb{N}$ *Schritten kein Optimum einer Zielfunktion mit* k *Optima gefunden hat, ist mindestens gleich* $1 - \frac{T \cdot k}{2^n}$.

Beweis: Die Wahrscheinlichkeit, dass x_t eines der k Optima ist, ist für alle $t \in \mathbb{N}$ gleich $k/2^n$. Also lässt sich die Wahrscheinlichkeit, nach T Schritten eines der Optima gefunden zu haben, nach oben durch $T \cdot k/2^n$ abschätzen. □

Der (1+1) EA mit Mutationsstärke 1/2 wird nach polynomiell vielen Schritten also mit exponentiell schnell gegen Eins konvergierender Wahrscheinlichkeit kein Optimum einer beliebigen Zielfunktion mit polynomiell vielen Optima finden. Um mit einer Multistartvariante mit zumindest konstanter Wahrscheinlichkeit Erfolg zu haben, müsste man den (1+1) EA also mit polynomiell großer Laufzeitschranke $T(n)$ exponentiell oft wiederholen, was natürlich zu exponentieller Laufzeit führt. Damit zeigt sich, dass die Mutationsstärke 1/2 für den (1+1) EA nicht sinnvoll ist, da dieser dann für jede nicht zu einfache Zielfunktion, d. h. mit polynomiell vielen Optima, exponentielle erwartete Laufzeit hat und auch durch eine Multistartvariante nicht entscheidend beschleunigt werden kann.

Im folgenden Abschnitt werden wir uns deshalb auf die Betrachtung der Mutationsstärke $p_m(n) = 1/n$ einschränken.

3.4 Der (1+1) EA mit Mutationsstärke $p_m(n) = 1/n$

Die bisherigen Resultate geben eine grobe Orientierung, wie Aussagen über die erwartete Laufzeit des (1+1) EA bewiesen werden und in welchen Größenordnungen sich seine Laufzeiten bewegen können. Dabei klafft zwischen der oberen (Lemma 3.3.1) und der unteren Schranke (Lemma 3.3.4 bzw. Korollar 3.3.5) eine große Lücke, was auch zu erwarten ist, da diese über sehr großen Klassen von Zielfunktionen und Mutationsstärken Aussagen treffen. Deshalb werden in diesem Abschnitt zweierlei Einschränkungen getroffen: einerseits wird die Mutationsstärke $p_m(n)$ auf den empfohlenen (Bäck (1993)) und am weitest verbreiteten Wert $1/n$ beschränkt, andererseits werden die Laufzeiten nur auf einzelnen Funktionen bzw. relativ kleinen Klassen von Funktionen analysiert.

3.4.1 Sehr einfache und sehr schwierige Funktionen

Für $p_m(n) = 1/n$ (weil wir uns in diesem Abschnitt auf diese Mutationsstärke beschränken, wird sie im Weiteren für gewöhnlich nicht mehr explizit angegeben) ist der Unterschied zwischen der unteren Schranke von $\Omega(n \log(n))$ von Korollar 3.3.5 und der oberen Schranke n^n von Lemma 3.3.1 sehr groß. Deshalb ist es von Interesse herauszufinden, ob diese beiden Schranken zu verbessern sind, oder, falls dies nicht der Fall ist, Funktionen bzw. ganze Klassen von Funktionen mit erwarteter Laufzeit $\Theta(n \log(n))$ bzw. $\Theta(n^n)$ zu finden.

Die erste Funktion, die wir dazu betrachten wollen, ist die Funktion OneMax, definiert gemäß:

Definition 3.4.1 OneMax : $\{0,1\}^n \mapsto \mathbb{R}$ *ist definiert als*

$$\text{OneMax}(x) := \sum_{i=1}^{n} x_i.$$

Das einzige Optimum dieser Funktion ist $(1, \ldots, 1)$, so dass sich nach Lemma 3.3.1 und Korollar 3.3.5 die erwartete Laufzeit zwischen $\Omega(n \log(n))$ und $O(n^n)$

bewegen kann. Weiterhin ist die Funktion OneMax *symmetrisch*, d. h. ihr Funktionswert ist nur von der Anzahl $||x||_1$ der Einsen, aber nicht von deren Position im Argument x abhängig.

Anschaulich gesprochen ist diese Funktion sehr einfach, da sie nur „richtige Hinweise" ermöglicht. Dabei verstehen wir unter einem richtigen Hinweis eine Mutation eines Punkts $x \in \{0,1\}^n$, die den Funktionswert erhöht und den Hamming-Abstand zum Optimum verringert. Natürlich verringert bei OneMax jede Mutation, die den Funktionswert, d. h. die Anzahl der Einsen, erhöht, den Hamming-Abstand zum Optimum $(1,\ldots,1)$. Da der (1+1) EA eine Verbesserung stets akzeptiert, sollte eine Funktion, die viele richtige Hinweise gibt, von ihm schnell zu optimieren sein.

Das folgende Lemma zeigt uns, wie sich diese anschauliche Argumentation auch in einen formalen Beweis umsetzen lässt, dass die erwartete Laufzeit des (1+1) EA gleich $\Theta(n\log(n))$ ist (siehe Mühlenbein (1992)):

Lemma 3.4.2 *Die erwartete Laufeit des* (1+1) EA *auf der Funktion* OneMax *beträgt* $\Theta(n\log(n))$.

Beweis: Da Korollar 3.3.5 die untere Schranke $\Omega(n\log(n))$ zeigt, reicht der Nachweis der oberen Schranke $O(n\log(n))$ aus. Dazu wenden wir Lemma 3.3.3 an, indem wir den Raum $\{0,1\}^n$ in die Ebenen Z_i ($i \in \{0,\ldots,n\}$) einteilen, wobei Z_i jeweils alle Elemente $x \in \{0,1\}^n$ mit genau i Einsen, d. h. $\textsc{OneMax}(x) = i$ enthält. Somit kann der (1+1) EA von einem Zustand in Z_i nur zu Zuständen aus $Z_i \cup \cdots \cup Z_n$ wechseln. Die Wahrscheinlichkeit von einem Zustand aus Z_i zu einem aus Z_{i+1} zu wechseln, beträgt mindestens

$$p_i := \binom{n-i}{1} \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{n-i}{n} \cdot \exp(-1).$$

Somit lässt sich Lemma 3.3.3 anwenden und liefert folgende obere Schranke für die erwartete Laufzeit des (1+1) EA:

$$\mathrm{E}(T_{1/n}^{\textsc{OneMax}}) \leq \sum_{i=0}^{n-1} \frac{n}{n-i} \cdot \exp(1) \leq \exp(1) \cdot n \cdot \sum_{i=1}^{n} \frac{1}{i}.$$

Da die harmonische Reihe von unten gegen $\ln(n) + \gamma$ konvergiert (A.13), folgt die Behauptung. □

Somit ist die untere Schranke $\Omega(n\log(n))$ von Korollar 3.3.5 auch für $p_m(n) = 1/n$ scharf. Wie steht es aber mit der oberen Schranke von n^n, die nach Lemma 3.3.1 für den (1+1) EA mit $p_m(n) = 1/n$ gilt? Um zu zeigen, dass auch sie von optimaler Größenordnung ist, müssen wir eine Funktion finden, zu deren Optimierung der (1+1) EA erwartete Laufzeit $\Omega(n^n)$ braucht. Nach obiger Argumentation müsste eine Funktion für den (1+1) EA schwierig sein, wenn sie nur falsche Hinweise gibt, d. h. wenn jede erfolgreiche Mutation den Hamming-Abstand zum Optimum vergrößert. Dies ist in dieser Ausschließlichkeit natürlich nicht möglich, da die direkte Mutation zum Optimum den Hamming-Abstand zu demselben nicht vergrößern kann. Doch zumindest sollte die Funktion sehr viele falsche Hinweise geben.

Eine solche Funktion zu konstruieren, fällt leicht, wenn man sich die Funktion OneMax anschaut. Hier werden, wie erwähnt, nur richtige Hinweise gegeben. Wenn man nun das bisherige Optimum $(1,\ldots,1)$ zu einem lokalen Optimum macht, indem man einem anderen Punkt einen höheren Funktionswert gibt, werden viele Hinweise falsch, die nicht zu dem neuen Optimum führen. Um die Zahl und Wahrscheinlichkeit der falschen Hinweise zu maximieren, wählen wir als neues Optimum den Punkt $(0,\ldots,0)$. Aus dieser Überlegung heraus kommt man zu der Funktion Trap (Ackley (1987)):

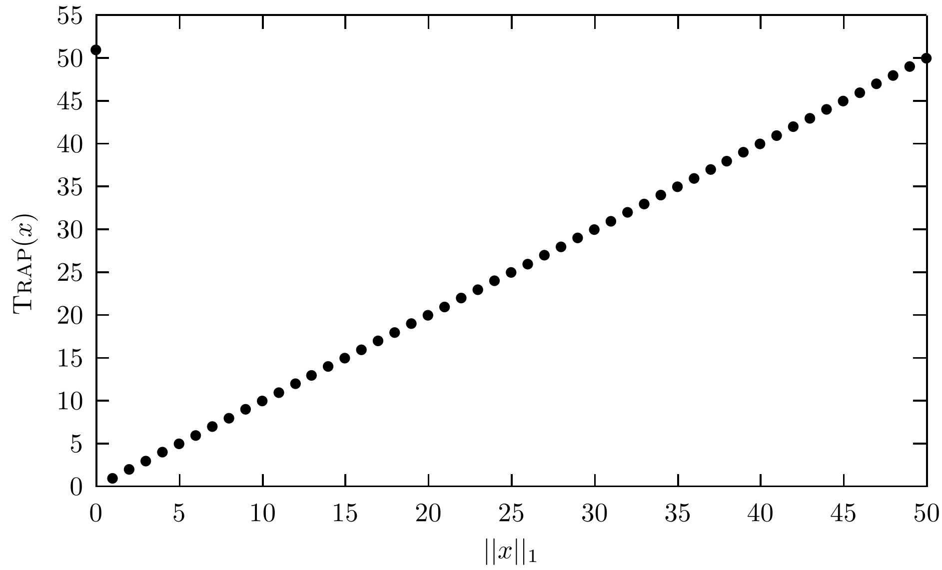

Abbildung 3.1: Die Funktion TRAP für $n = 50$.

Definition 3.4.3 *Die Funktion* TRAP : $\{0,1\}^n \mapsto \mathbb{R}$ *ist definiert gemäß:*

$$\text{TRAP}(x) := \sum_{i=1}^{n} x_i + (n+1) \cdot \prod_{i=1}^{n} (1 - x_i).$$

In Abbildung 3.1 ist die Funktion TRAP für $n = 50$ veranschaulicht. Da TRAP eine so große Ähnlichkeit mit ONEMAX hat, lässt sich aufbauend auf Lemma 3.4.2 recht einfach die Laufzeit des (1+1) EA auf TRAP bestimmen:

Lemma 3.4.4 *Die Wahrscheinlichkeit, dass der* (1+1) EA *auf* TRAP *nach polynomiell vielen Schritten das Optimum gefunden hat, ist exponentiell klein. Weiterhin gibt es für jede Konstante $c < 1$ ein konstant großes $\varepsilon > 0$, so dass die Laufzeit des* (1+1) EA *auf* TRAP *mit einer Wahrscheinlichkeit von mindestens c mindestens gleich $\varepsilon \cdot n^n$ ist.*

Beweis: Wir gehen zu Beginn davon aus, dass der (1+1) EA in einem Punkt mit mindestens $n/4$ Einsen initialisiert. Nach der Tschernoff-Ungleichung (A.9) hat dies eine Wahrscheinlichkeit von mindestens $1 - \exp(-n/16)$, was exponentiell schnell gegen Eins konvergiert. Unter dieser Annahme kann das globale Optimum $(0, \ldots, 0)$ nur erreicht werden, wenn in einem Schritt mindestens $n/4$ Einsen gleichzeitig mutieren. Eine solche Mutation hat eine Wahrscheinlichkeit von höchstens $(1/n)^{n/4}$. Betrachten wir die ersten n^2 Schritte des (1+1) EA, so ist die Wahrscheinlichkeit, dass innerhalb dieses Zeitraums eine solche Mutation geschieht, also höchstens gleich $n^2/n^{n/4}$, d. h. exponentiell klein. Nehmen wir deshalb an, dass dies nicht passiert. Da dies auch für ein beliebiges Polynom statt n^2 gilt, folgt die erste Behauptung

Unter diesen zwei Annahmen verhält sich der (1+1) EA auf TRAP wie auf ONEMAX, d. h. die erwartete Zeit bis zum Erreichen des lokalen Optimums $(1, \ldots, 1)$ ist $\Theta(n \log(n))$. Also beträgt nach der Markoff-Ungleichung (A.8) die Wahrscheinlichkeit, den Punkt $(1, \ldots, 1)$ innerhalb der ersten n^2 Schritte nicht zu erreichen, höchstens $\log(n)/n$. Von $(1, \ldots, 1)$ aus ist die einzige akzeptierte Mutation jedoch die zum globalen Optimum $(0, \ldots, 0)$, worauf im Erwartungsfall genau n^n Schritte gewartet werden muss.

Die Wahrscheinlichkeit, auf diese Mutation mindestens $\varepsilon \cdot n^n$ ($\varepsilon \in [0,1]$) Schritte zu warten, beträgt

$$\left(1-n^{-n}\right)^{\varepsilon \cdot n^n - 1} \geq \exp(-\varepsilon).$$

Zu gegebenem c kann man also eine Konstante $\varepsilon' > 0$ und ein $n_0 \in \mathbb{N}$ finden, so dass die Summe der Wahrscheinlichkeit, weniger als $\varepsilon' \cdot n^n$ Schritte für diesen letzten Schritt zu brauchen, und der Wahrscheinlichkeiten der oben getroffenen Annahmen für alle $n \geq n_0$ kleiner als c ist. Da sich für die konstant vielen $n < n_0$ jeweils ein $\varepsilon_n > 0$ finden lässt, so dass die Laufzeit mit einer Wahrscheinlichkeit von mindestens c mindestens $\varepsilon_n \cdot n^n$ ist, gilt die Aussage des Lemmas mit $\varepsilon := \min\{\varepsilon', \varepsilon_1, \ldots, \varepsilon_{n_0 - 1}\}$. □

Aus Lemma 3.4.4 folgt, dass die erwartete Laufzeit des (1+1) EA auf TRAP gleich $\Theta(n^n)$ ist. Also ist auch Lemma 3.3.1 für $p_m(n) = 1/n$ optimal. Die Grenzen $\Omega(n \log(n))$ (wenn wir uns auf Funktionen mit einem Optimum beschränken, siehe Korollar 3.3.5) und $O(n^n)$ für $p_m(n) = 1/n$ sind somit beide scharf. Weiterhin gibt es einfach zu beschreibende Funktionen, für die der (1+1) EA diese extremen Laufzeiten annimmt. Diese bestätigen die Anschauung, dass richtige Hinweise, d. h. erfolgreiche Mutationen, die den Hamming-Abstand zum Optimum verringern, für eine schnelle Optimierung nützlich sind.

Natürlich sind wir daran interessiert, die erwarteten Laufzeiten genauer einschränken zu können, was aber nur geht, wenn wir uns auf bestimmte Klassen von Funktionen einschränken. Im Folgenden werden wir dies für lineare Funktionen, Funktionen vom Grad Zwei und unimodale Funktionen tun.

3.4.2 Analysen für lineare Funktionen

ONEMAX ist eine sehr einfache Funktion, da sie anschaulich gesprochen nur richtige Hinweise gibt. Wir wollen nun untersuchen, welche Eigenschaften von ONEMAX hinreichend sind, dass der (1+1) EA sie in der für Funktionen mit genau einem Optimum minimalen erwarteten Laufzeit $\Theta(n \log(n))$ minimiert.

Bevor wir zeigen, dass die Linearität einer Funktion ein hinreichendes Kriterium ist, werden wir zuerst eine andere lineare Funktion betrachten. Diese wird insofern ein Gegenstück zu ONEMAX bilden, als dass bei ONEMAX alle Bits das gleiche Gewicht haben, bei ihr jedoch ein Bit x_i stets „wichtiger" als alle Bits x_j mit $j > i$ zusammen ist:

Definition 3.4.5 *Die Funktion* BINVAL : $\{0,1\}^n \mapsto \mathbb{R}$ *ist definiert gemäß*

$$\text{BINVAL}(x) := \sum_{i=1}^{n} x_i \cdot 2^{n-i}.$$

Die Funktion BINVAL gibt also die Zahl zurück, die ihr Argument binär kodiert. Bei ihr kann der Hamming-Abstand zum Optimum bei Verbesserung des Funktionswerts sinken, das Extrembeispiel hierfür ist der Übergang von $(0, 1, \ldots, 1)$ mit Funktionswert $2^{n-1}-1$ zu $(1, 0, \ldots, 0)$ mit Funktionswert 2^{n-1}, wobei der Hamming-Abstand von 1 auf $n-1$ steigt. Es sind also falsche Hinweise möglich, die zudem noch sehr stark vom Optimum wegführen können. Während bei ONEMAX eine Mutation genau dann übernommen wird, wenn der Hamming-Abstand zum Optimum $(1, \ldots, 1)$ nicht abnimmt, geschieht dies bei BINVAL genau dann, wenn das linkeste Bit, das mutieren will, eine Null ist.

Eine Ebeneneinteilung gemäß Definition 3.3.2 bzw. Lemma 3.3.3 ist für BINVAL schwierig: eine Zusammenfassung aller Bitstrings mit Hamming-Gewicht i zu einer Ebene ist nicht möglich, da dann keine Anordnung der Ebenen möglich ist, so dass der (1+1) EA von einer Ebene nicht zu tieferen Ebenen wechseln kann. Eine andere

Möglichkeit ist, alle Punkte x mit $x_1 = \cdots = x_i = 1$ und $x_{i+1} = 0$ zu einer Ebene Z_i ($i \in \{0, \ldots, n-1\}$) zusammenzufassen, wobei Z_n nur aus dem Optimum $(1, \ldots, 1)$ besteht. Dies hat den Vorteil, dass der (1+1) EA stets von Z_i nur zu höheren Ebenen $Z_{i'}$ mit $i' \geq i$ mutieren kann. Die Wahrscheinlichkeit, von einem Punkt aus Z_i zu einem Punkt einer höheren Ebene zu mutieren, ist genau $(1-1/n)^i \cdot 1/n$. Also liefert diese Beweismethode die obere Schranke

$$\sum_{i=0}^{n-1} n \cdot \left(\frac{1}{1-1/n}\right)^i > n^2.$$

Eine bessere obere Schranke lässt sich mit dieser Methode bei der vorgegebenen Ebeneneinteilung nicht erreichen. Obwohl es also viele anschaulich einsichtige Argumente gibt, warum BinVal schwieriger als OneMax ist, werden wir im Folgenden zeigen, dass der (1+1) EA auf BinVal eine erwartete Laufzeit von $\Theta(n \log(n))$ hat. Die Analyse ist jedoch ungleich schwieriger als für OneMax:

Lemma 3.4.6 *Die erwartete Laufzeit des* (1+1) EA *auf der Funktion* BinVal *beträgt* $\Theta(n \log(n))$.

Beweis: Wiederum muss nur die obere Schranke $O(n \log(n))$ bewiesen werden, da Korollar 3.3.5 die untere Schranke garantiert.

Der Beweis beruht auf folgender Idee: die Menge $\{0,1\}^n$ aller Punkte wird in die drei Teilmengen Z_1 (alle Punkte mit höchstens $n/2-1$ führenden Einsen, wobei n o. B. d. A. gerade ist), Z_2 (alle Punkte mit mindestens $n/2$ und höchstens $n-1$ führenden Einsen) und Z_3 (das Optimum $(1, \ldots, 1)$) aufgeteilt. Dann werden die erwarteten Zeiten, von Z_1 nach Z_2 und von Z_2 nach Z_3 zu kommen, abgeschätzt, doch werden dazu nicht wie in Lemma 3.3.3 die Übergangswahrscheinlichkeiten zwischen diesen Mengen abgeschätzt.

Wenn T_1 bzw. T_2 die Zeit sei, um beginnend in Z_1 bzw. Z_2 zum ersten Mal einen Punkt aus Z_2 bzw. Z_3 zu erreichen, so ist $\mathrm{E}(T^{\mathrm{BinVal}})$ maximal gleich $\mathrm{E}(T_1) + \mathrm{E}(T_2)$, weil nach Erreichen eines Punkts aus Z_i niemals eine Teilmenge mit niedrigerem Index erreicht werden kann. Deshalb können die Erwartungswerte von T_1 und T_2 getrennt betrachtet werden:

1. Um $\mathrm{E}(T_1)$ nach oben abzuschätzen, werden die $\mathbb{N}_0$-wertigen Zufallsvariablen X_i ($i \in \{0, \ldots, n/2\}$) eingeführt, die jeweils den ersten Zeitpunkt angeben, an dem die linke Hälfte des aktuellen Suchpunkts x_t, d. h. $(x_t)_1$ bis $(x_t)_{n/2}$, mindestens i Einsen enthält. Da $X_0 = 0$ ist, folgt

$$T_1 = X_{n/2} = (X_{n/2} - X_{n/2-1}) + (X_{n/2-1} - X_{n/2-2}) + \cdots + (X_1 - X_0),$$

wobei $X_i - X_{i-1}$ gerade die Zahl der Schritte ist, um die Zahl von $i-1$ Einsen in der linken Hälfte auf mindestens i zu steigern. Um die Erwartungswerte dieser Größen zu bestimmen, wird zwischen *erfolgreichen* und *erfolglosen* Mutationen unterschieden. Erfolgreiche Mutationen seien hierbei solche, die mindestens ein Bit der linken Hälfte mutieren und den Funktionswert erhöhen. Erfolglose Mutationen können zwar auch im Selektionsschritt zu einer Veränderung des aktuellen Elements führen, dies jedoch nur in der rechten Hälfte des Strings, die wir hier ausklammern. Da es ausschließlich erfolgreiche oder erfolglose Mutationen gibt, lässt sich $X_i - X_{i-1}$ auch als $Y_i + Z_i$ ausdrücken, wenn Y_i die Zahl der erfolgreichen und Z_i die Zahl der erfolglosen Mutationen ist, um die Zahl der Einsen in der linken Hälfte bei $i-1$ startend auf mindestens i zu steigern.

Eine obere Abschätzung von $\mathrm{E}(Y_i + Z_i)$ für alle $i \in \{0, \ldots, n/2-1\}$ führt also zu einer oberen Abschätzung von $\mathrm{E}(X_i - X_{i-1})$ und somit von $\mathrm{E}(T_1)$. Nehmen wir an, dass für alle $i \in \{0, \ldots, n/2-1\}$ gilt $\mathrm{E}(Y_i) \leq 2$, d. h., dass im Erwartungswert nach spätestens zwei erfolgreichen Schritten die Anzahl der Einsen in der linken Hälfte zugenommen hat. Wenn wir weiterhin noch eine untere Schranke $p_i \in [0,1]$ für die Wahrscheinlichkeit einer erfolgreichen Mutation eines Punkts mit weniger als $i \in \{1, \ldots, n/2-1\}$ Einsen gefunden haben, so lässt sich hiermit $\mathrm{E}(Y_i + Z_i)$ mit der Methode der *bedingten Erwartungswerte* abschätzen:

$$\begin{aligned}
\mathrm{E}(Y_i + Z_i) &= \sum_{k=1}^{\infty} \mathrm{P}(Y_i = k) \cdot \mathrm{E}(Y_i + Z_i \,|\, Y_i = k) \\
&\leq \sum_{k=1}^{\infty} \mathrm{P}(Y_i = k) \cdot \frac{k}{p_i} \\
&= \frac{\mathrm{E}(Y_i)}{p_i} \leq \frac{2}{p_i},
\end{aligned}$$

wobei man sich nur zunutze macht, dass die erwartete Zahl von Mutationen bis zur k-ten erfolgreichen Mutation höchstens gleich k/p_i ist.

Eine hinreichende Bedingung für eine erfolgreiche Mutation in der Phase bis zum Erreichen von mindestens i Bits in der linken Hälfte ist, dass alle der höchstens $i-1$ Einsen der linken Hälfte nicht mutieren wollen und genau eine der mindestens $n/2-(i-1)$ Nullen der linken Hälfte mutieren will. Also ist

$$\binom{n/2-(i-1)}{1} \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n/2-1} \geq \left(\frac{n}{2} - i + 1\right) \cdot \frac{\exp(-1/2)}{n} =: p_i$$

eine untere Schranke der Wahrscheinlichkeit einer erfolgreichen Mutation in der Phase bis zum Erreichen von i Einsen in der linken Hälfte.

Damit lässt sich der Erwartungswert von T_1 folgendermaßen abschätzen:

$$\begin{aligned}
\sum_{i=1}^{n/2} \mathrm{E}(Y_i + Z_i) &\leq \sum_{i=1}^{n/2} 2 \cdot \left(\left(\frac{n}{2} - i + 1\right) \cdot \frac{\exp(-1/2)}{n}\right)^{-1} \\
&= 2 \cdot \exp(1/2) \cdot n \cdot \sum_{i=1}^{n/2} \frac{1}{i} \leq 2 \cdot \exp(1/2) \cdot n \cdot (\ln(n) + \gamma).
\end{aligned}$$

Der in diesen Überlegungen angestellte, aber noch nicht bewiesene Schritt ist die obere Abschätzung des Erwartungswerts der Zahl erfolgreicher Mutationen, bis die Zahl der Einsen von $i-1$ auf mindestens i zugenommen hat. Dazu sei t der erste Zeitpunkt, an dem x_t genau $i-1$ Einsen enthält, und $x_t, x_{t+1}, x_{t+2}, \ldots$ die Folge verschiedener Bitstrings des (1+1) EA, die jeweils durch erfolgreiche Mutationen entstehen. Die Zufallsvariable $D_j := ||x_j||_1^* - ||x_{j-1}||_1^*$ gebe die Zunahme der Anzahl der Einsen in der linken Hälfte beim Übergang von x_{j-1} zu x_j (dabei sei $||x||_1^*$ die Anzahl der Einsen in der linken Hälfte von x) an. Wir sind interessiert an der kleinsten Zeit t', bis $D_{t+1} + \cdots + D_{t+t'} \geq 1$ ist, da diese genau die Verteilung von Y_i besitzt.

Für alle $j \in \{t+1, \ldots, t+t'\}$ gilt, dass der String x_{j-1} nur maximal $i-1$ Einsen in der linken Hälfte enthält und x_j durch eine erfolgreiche Mutation aus einem String hervorgegangen ist, der zumindest in der linken Hälfte mit x_{j-1} übereinstimmt. Eine Mutation ist genau dann erfolgreich, wenn das linkeste mutierende Bit von x_{j-1} eine Null ist und in der linken Hälfte liegt. Wenn dessen Position l sei, so ist also $l \leq n/2$. Da an Stelle l eine Eins hinzugekommen ist und links davon kein Bit mutiert, können nur Bits zusätzlich mutieren, die rechts davon liegen, d. h. einen Index aus $\{l+1, \dots, n/2\}$ haben. Wenn $k \in \{0, \dots, n/2-l\}$ die Anzahl der Einsen von x_{j-1} aus diesem Bereich ist, so ist die Zahl $B_{j,1}$ bzw. $B_{j,0}$ der mutierenden Einsen bzw. Nullen in der linken Hälfte von x_{j-1} binomial-verteilt mit Parametern $1/n$ und k bzw. $n/2-l-k$ (A.6). Da die Anzahl der Einsen in diesem Bereich um $B_{j,1} - B_{j,0}$ abnimmt, ist $D_j = 1 - B_{j,1} + B_{j,0}$. Wie können wir nun $B_{j,1}$ und $B_{j,0}$ abschätzen, um eine untere Schranke für den Erwartungswert von D_j zu erhalten?

Im schlimmsten Fall für uns wären alle Bits von x_{j-1} außer an der Stelle l gleich Eins, da dann die Wahrscheinlichkeit für mindestens m mutierende Einsen maximal ist. Formal bedeutet dies, dass für alle $m \in \{0, \dots, n/2-l\}$ gilt

$$\mathrm{P}(B_j^* \geq m) \geq \mathrm{P}(B_{j,1} - B_{j,0} \geq m),$$

wenn B_j^* eine binomial-verteilte Zufallsvariable mit Parametern $1/n$ und $n/2-1$ ist (A.6). Also gilt $\mathrm{P}(1 - B_j^* \leq m) \geq \mathrm{P}(1 - B_{j,1} + B_{j,0} \leq m)$ für alle $m \in \{1 - n/2 + l, \dots, 1\}$. Wenn man $1 - B_j^*$ als D_j^* und mit T^* den ersten Zeitpunkt $t' \geq 1$ bezeichnet, an dem $D_{t+1}^* + \cdots + D_{t+t'}^* \geq 1$ ist, so gilt also $\mathrm{P}(T^* \geq t) \geq \mathrm{P}(Y_i \geq t)$ für alle $t \geq 1$. Damit folgt

$$\begin{aligned} \mathrm{E}(Y_i) &= \sum_{t=0}^{\infty} t \cdot \mathrm{P}(Y_i = t) = \sum_{t=0}^{\infty} \mathrm{P}(Y_i \geq t) \\ &\leq \sum_{t=0}^{\infty} \mathrm{P}(T^* \geq t) = \mathrm{E}(T^*), \end{aligned}$$

d. h. zur Abschätzung des Erwartungswerts von Y_i nach oben reicht es aus, den von T^* nach oben abzuschätzen. Was haben wir mit dem Übergang von Y_i zu T^* gewonnen?

Da D_j^* den Anstieg der Zahl der Einsen für den Fall modelliert, dass alle Bits der linken Hälfte rechts von der Stelle l (der linkesten mutierenden Null) gleich Eins sind, kann D_j^* keinen Wert größer als Eins annehmen und ihr Wertebereich ist $\{1 - n/2 + l, \dots, 1\}$. Da $\mathrm{E}(B_j^*) = (n/2-1)/n < 1/2$ ist, ist der Erwartungswert von D_j^* größer als $1/2$. Zur Bestimmung von $\mathrm{E}(T^*)$ wenden wir wieder die Methode der bedingten Erwartungswerte an, wobei wir nach dem Wert d von D_{t+1}^*, der Zahl hinzugekommener Einsen bei der ersten erfolgreichen Mutation, unterscheiden. Da alle Zufallsvariablen D_{t+1}^* bis $D_{t+t'}^*$ identisch verteilt sind, ist die Zeit, vom Startpunkt $1-d$ zum ersten Mal zum Punkt 1 zu kommen, gleich $(1-d) \cdot \mathrm{E}(T^*)$. Also ergibt sich

$$\begin{aligned} \mathrm{E}(T^*) &= \sum_{d=-n/2+2}^{1} \mathrm{P}(D_{t+1}^* = d) \cdot \mathrm{E}(T^* \mid D_{t+1}^* = d) \\ &= \sum_{d=-n/2+2}^{1} \mathrm{P}(D_{t+1}^* = d) \cdot (1 + (1-d) \cdot \mathrm{E}(T^*)) \\ &= 1 + \sum_{d=-n/2+2}^{1} \mathrm{P}(D_{t+1}^* = d) \cdot (1-d) \cdot \mathrm{E}(T^*) \\ &= 1 + \mathrm{E}(T^*) - \mathrm{E}(D_{t+1}^*) \cdot \mathrm{E}(T^*), \end{aligned}$$

d. h. $\mathrm{E}(T^*)$ ist $1/\mathrm{E}(D_{t+1}^*)$, eine in allgemeinerer Form als *Wald'sche Identität* bekannte Gleichung (Feller (1971)). Da $\mathrm{E}(D_{t+1}^*) > 1/2$ ist, ist $\mathrm{E}(T^*) < 2$, was gerade die uns noch fehlende Abschätzung ist.

2. Um die Abschätzung von $\mathrm{E}(T)$ fertigzustellen, muss noch der erwartete Wert von T_2, der Anzahl der Schritte, bis auch die rechte Hälfte nur aus Einsen besteht, nach oben abgeschätzt werden. Dazu kann die Abschätzung von $\mathrm{E}(T_1^*)$ fast exakt übernommen werden, nur die untere Abschätzung p_i für die Wahrscheinlichkeit einer erfolgreichen Mutation muss geändert werden: denn nun reicht es nicht aus, wenn bis auf ein Bit alle Bits der betrachteten Hälfte nicht mutieren, da auch alle Bits der anderen (der linken) Hälfte nicht mutieren dürfen. Damit bekommt man die untere Schranke

$$\binom{n/2-(i-1)}{1} \cdot \frac{1}{n} \cdot \left(1-\frac{1}{n}\right)^{n-1} \geq \left(\frac{n}{2}-i+1\right) \cdot \frac{\exp(-1)}{n}$$

für die Wahrscheinlichkeit einer erfolgreichen Mutation, falls die Zahl der Einsen kleiner als i ist, weshalb man letztendlich die obere Schranke

$$\mathrm{E}(T_2) \leq 2 \cdot \exp(1) \cdot n \cdot (\ln(n) + \gamma)$$

bekommt und somit die Gesamtlaufzeit abschätzen kann als:

$$\mathrm{E}(T^{\textsc{BinVal}}) \leq 2 \cdot (\exp(1) + \exp(1/2)) \cdot n \cdot (\ln(n) + \gamma).$$

□

Damit ist klar, dass auch Funktionen, bei denen der Hamming-Abstand zum Optimum mit wachsendem Funktionswert steigen kann, die also falsche Hinweise geben, für den (1+1) EA leicht sein können und zwar in dem Sinne, dass die erwartete Laufzeit des (1+1) EA zu ihrer Optimierung unter allen Funktionen mit einem Optimum in der Größenordnung minimal ist.

Nun kann man bei der Untersuchung des (1+1) EA willkürlich weitere Funktionen suchen, die erwartete Laufzeit $\Theta(n \log(n))$ haben, oder versuchen, die Resultate über OneMax und BinVal zu verallgemeinern. Letzteres soll hier nun getan werden und zwar aus der Beobachtung heraus, dass OneMax und BinVal im folgendem Sinne die extremsten linearen Funktionen sind: während bei OneMax alle Koeffizienten gleiche Größen haben, ist bei BinVal jeder Koeffizient jeweils größer als die Summe aller nachfolgenden. Wegen der in den nächsten Abschnitten folgenden Betrachtungen von Polynomen zweiten Grades seien die folgenden Definitionen und Beobachtungen der weiteren Untersuchung des (1+1) EA vorangestellt:

Lemma 3.4.7 *Sei $f : \{0,1\}^N \mapsto \mathbb{R}$. Dann hat f eine eindeutige* Polynom-Darstellung, *d. h. Koeffizienten $c_S \in \mathbb{R}$ für alle $S \subseteq \{1, \ldots, n\}$, so dass*

$$\forall x \in \{0,1\}^n : f(x) = \sum_{S \subseteq \{1,\ldots,n\}} c_S \cdot \prod_{i \in S} x_i.$$

Beweis: Zuerst sei die Eindeutigkeit nachgewiesen: angenommen, es gäbe zwei Repräsentationen mit verschiedenen Mengen von Koeffizienten c_S und c'_S für eine Funktion f. Dann sei S eine Teilmenge von $\{1, \ldots, n\}$ mit $c_S \neq c'_S$ und $c_T = c'_T$ für alle $T \subset S$. Sei $x^S \in \{0,1\}^n$ der Bitstring, der genau an den mit Elementen aus S indizierten Stellen gleich Eins ist. Dann gilt

$$f(x^S) = \sum_{T \subseteq S} c_T \neq \sum_{T \subseteq S} c'_T = f(x^S),$$

was ein offensichtlicher Widerspruch ist.

Eine Polynom-Darstellung existiert stets, weil das folgende Konstruktionsprinzip für alle $f : \{0,1\}^n \mapsto \mathbb{R}$ die Koeffizienten c_S erzeugt. Sei $S_1, \ldots, S_{2^n}$ eine Folge von Teilmengen von $\{1, \ldots, n\}$, so dass für alle $i \in \{1, \ldots, 2^n\}$ gilt, dass alle

Teilmengen von S_i in der Teilfolge $S_1, \ldots, S_i$ enthalten sind. Dazu wäre es z. B. ausreichend, wenn zuerst die leere Menge, dann alle einelementigen Mengen, dann alle zweielementigen Mengen, usw., aufgezählt werden, wobei die genauere Reihenfolge innerhalb dieser Anfangsstücke unerheblich ist. Dann kann man, bei $j = 1$ beginnend, die Koeffizienten c_{S_j} nach und nach mit steigendem j als

$$c_{S_j} := f(x^{S_j}) - \sum_{S \subset S_j} c_S$$

bestimmen. Also gibt es zu jeder Funktion f eindeutige Koeffizienten c_S. □

Anhand der Polynomdarstellung lässt sich ablesen, von wievielen anderen Bits der Einfluss eines einzelnen Bits auf den Funktionswert abhängig ist. Dieser kann anhand des Grades der Funktion gemessen werden:

Definition 3.4.8 *Sei* $f : \{0,1\}^n \mapsto \mathbb{R}$ *und* $c_S \in \mathbb{R}$ *für* $S \subseteq \{1, \ldots, n\}$ *die Koeffizienten der Polynomdarstellung von* f*. Dann ist*

$$d(f) := \max\{k \in \{0, \ldots, n\} \mid \exists S \subseteq \{1, \ldots, n\} : |S| = k \wedge c_S \neq 0\}$$

der Grad *von* f*.*

Wenn der Grad von f gleich Eins ist, so ist der Einfluss, den der Wert eines Bits x_i auf f hat, unabhängig von den Werten der anderen Bits; man bezeichnet f in diesem Fall als *lineare* Funktion. Nach den Definitionen 3.4.1 und 3.4.5 ist es klar, dass sowohl OneMax als auch BinVal maximal Grad Eins haben. Da sie nicht konstant sind und nur konstante Funktionen Grad Null haben, folgt, dass beide lineare Funktionen sind.

Lineare Funktionen bilden für uns eine sehr einfache Klasse von Funktionen, da es zur Optimierung ausreicht, wenn man die Bits in einer beliebigen Reihenfolge einzeln durchgeht, den Wert des gerade betrachteten Bits auf Null bzw. Eins setzt und den Wert, der einen besseren Funktionswert ergeben hat, in die endgültige Belegung übernimmt. Dieses Verfahren optimiert eine lineare Zielfunktion in Laufzeit $\Theta(n)$. Natürlich benutzt es als Vorwissen, dass die zu optimierende Funktion linear ist; auf eine nicht-lineare Funktion angewandt, wird es in der Regel kein Optimum finden.

Von dem (1+1) EA, der auf keinerlei Weise auf die Optimierung linearer Zielfunktion abgestimmt scheint, sollten wir deshalb keine garantierte erwartete Laufzeit $O(n)$ zur Optimierung linearer Funktionen erwarten. Allein die zufällige Auswahl der zu mutierenden Bits impliziert, dass schon einzeln mutierte Bits durchaus noch einmal mutiert werden können, wobei sie dabei sogar vom richtigen auf den falschen Wert zurückmutiert werden können.

Nun sind, wie schon angesprochen, die Funktionen OneMax und BinVal insofern Extrembeispiele linearer Funktionen, als dass bei OneMax alle Koeffizienten $c_i := c_{\{i\}}$ denselben Wert haben, wohingegen bei BinVal der Koeffizient c_i jeweils größer als die Summe aller nachfolgenden c_j mit $j > i$ ist. Dabei war der Nachweis der Laufzeit von BinVal wesentlich schwieriger als für OneMax, da bei ersterer der Hamming-Abstand zum Optimum mit steigendem Funktionswert zunehmen kann. Dies gilt natürlich für lineare Funktionen ebenso. Dementsprechend wird der Beweis, dass alle linearen Funktionen in erwarteter Laufzeit $\Theta(n \log(n))$ vom (1+1) EA optimiert werden, in seinem Aufbau mehr dem von Lemma 3.4.6 ähneln, jedoch komplizierter sein, da die zusätzlichen Freiheiten der Koeffizienten berücksichtigt werden müssen:

Theorem 3.4.9 *Sei* $f : \{0,1\}^n \mapsto \mathbb{R}$ *eine zu maximierende lineare Zielfunktion. Dann ist die erwartete Laufzeit des* (1+1) EA *auf* f *gleich* $O(n \log(n))$*.*

Beweis: Um die Notation des Beweises zu vereinfachen, machen wir folgende Annahmen, die die Anwendbarkeit des Folgenden nicht einschränken. Erstens seien alle Koeffizienten c_i positiv; wenn dies für die ursprüngliche Funktion f nicht erfüllt ist, so vertauschen wir an den Stellen i mit $c_i < 0$ die Rollen von Null und Eins. Zweitens seien die Koeffizienten absteigend angeordnet, d. h. $c_1 \geq \cdots \geq c_n > 0$, was nach Lemma 3.1.4 durch Vertauschung der Stellen von f erreicht werden kann. Beide Annahmen lassen sich also für jede lineare Zielfunktion erreichen, ohne die erwartete Laufzeit des (1+1) EA zu ändern.

Da die Koeffizienten der linearen Zielfunktion bis auf diese Reihenfolge nicht bekannt sind, benutzen wir zum Messen des Fortschritts des (1+1) EA auf f die folgende Funktion $\textsc{Val} : \{0,1\}^n \mapsto \mathbb{R}$:

$$\textsc{Val}(x) := 2 \cdot \sum_{i=1}^{n/2} x_i + \sum_{i=n/2+1}^{n} x_i.$$

Die Funktion $\textsc{Val}$ gewichtet die ersten $n/2$ Bits genau doppelt so stark wie die restlichen $n/2$ Bits der hinteren, d. h. rechten Hälfte, da diese nach obiger Reihenfolgebedingung ein größeres Gewicht haben (wie beim Beweis von Lemma 3.4.6 sei n o. B. d. A. gerade). Klar ist, dass der Maximalwert von $\textsc{Val}$ gleich $3n/2$ ist und genau bei $(1, \ldots, 1)$, dem Optimum von f, angenommen wird. Also nimmt $\textsc{Val}$ genau dann sein Optimum bei x an, wenn dies auch für f gilt.

Die Funktion $\textsc{Val}$ wird nun folgendermaßen benutzt: der (1+1) EA wird in seinem Ablauf auf f analysiert, wobei ähnlich zum Beweis von Lemma 3.4.6 eine obere Schranke für die Anzahl der erfolgreichen Mutationen bestimmt wird, die notwendig sind, um den Funktionswert um mindestens Eins zu erhöhen. Nur, und das ist eine entscheidende Änderung, wird für diesen und nur für diesen Zweck nicht der Funktionswert von f betrachtet, sondern der von $\textsc{Val}$. Die Funktion $\textsc{Val}$ dient also allein der Messung des Fortschritts des (1+1) EA, welcher aber weiterhin auf f arbeitet. Deshalb nennen wir eine Mutation von x_t zu x'_t genau dann *erfolgreich*, wenn $x_t \neq x'_t$ und $f(x_t) \leq f(x'_t)$ ist. Hervorzuheben ist also, dass die Funktion $\textsc{Val}$ nur zum Messen der erwarteten Laufzeit des (1+1) EA auf f benutzt wird; sie dient nur unserem Beweis, beeinflusst den (1+1) EA aber nicht.

Ist i der Wert von $\textsc{Val}$, so suchen wir eine konstante obere Schranke c^* für die Anzahl erfolgreicher Mutationen, bis der Wert von $\textsc{Val}$ auf $i+1$ gestiegen ist. Da für alle Bitstrings in dieser Phase der Wert von $\textsc{Val}$ maximal gleich i ist, ist die Anzahl der Nullen in jedem dieser Bitstrings größer als $3n/4 - i/2$, d. h. die Anzahl der Einsen ist maximal gleich $n/4 + i/2$. Denn wäre sie größer, so ergäbe sich in beiden der folgenden erschöpfenden Fälle ein Widerspruch:

1. Ist die Anzahl der Einsen höchstens $n/2$, so bildet diese eine untere Schranke für den Wert von $\textsc{Val}$. Da die Anzahl der Einsen größer als $n/4 + i/2$ und der Wert von $\textsc{Val}$ höchstens gleich i ist, folgt $i > n/4 + i/2$, was gleichbedeutend zu $i > n/2$ ist. Da jedoch $n/2$ eine obere Schranke und $n/4 + i/2$ eine untere Schranke für die Zahl der Einsen ist, folgt $i \leq n/2$, was ein Widerspruch ist.

2. Ist die Anzahl der Einsen größer als $n/2$, so muss es Einsen in der linken Hälfte geben. Also muss der Wert von $\textsc{Val}$ größer als $n/2 + 2 \cdot (n/4 + i/2 - n/2) = i$ sein, im Widerspruch zur Voraussetzung.

Da eine Mutation, die nur eine einzige Null mutiert, auf jeden Fall erfolgreich ist, ist die Wahrscheinlichkeit für eine erfolgreiche Mutation mindestens gleich

$$\binom{3n/4 - i/2}{1} \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \geq \left(\frac{3n}{4} - \frac{i}{2}\right) \cdot \exp(-1) \cdot \frac{1}{n}.$$

Demzufolge ist die erwartete Zeit bis zum Erreichen des Optimums von f unter der Annahme, dass sich nach erwartet höchstens $\mathrm{E}(T^*)$ erfolgreichen Mutationen der Funktionswert von VAL um Eins vergrößert, kleiner als

$$\sum_{i=0}^{3n/2-1} \mathrm{E}(T^*) \cdot \frac{\exp(1) \cdot n}{3n/4 - i/2} \leq \frac{2 \cdot \mathrm{E}(T^*)}{\exp(-1)} \cdot n \cdot \sum_{i=1}^{3n/4} \frac{1}{i - 1/2} \leq \frac{4 \cdot \mathrm{E}(T^*)}{\exp(-1)} \cdot n \cdot (\ln(n) + \gamma).$$

Was also im Folgenden „nur“ zu zeigen ist, ist, dass im Erwartungswert konstant viele erfolgreiche Mutationen des (1+1) EA auf f ausreichen, um den Funktionswert von VAL um Eins zu erhöhen. Sei also $x \in \{0,1\}^n$ nicht das Optimum, x' der aus x durch eine erfolgreiche Mutation hervorgegangene Bitstring und die Zufallsvariable $D(x)$ gleich $\mathrm{VAL}(x') - \mathrm{VAL}(x)$. Selbst wenn wir eine obere Schranke für den Erwartungswert von $D(x)$ kennen, können wir die Wald'sche Gleichung nicht verwenden, um damit eine untere Schranke für die erwartete Anzahl von Mutationen, bis der Wert von VAL zunimmt, zu berechnen, da $D(x)$ auch Werte größer als Eins annehmen kann. Für die Wald'sche Gleichung (siehe den Beweis von Lemma 3.4.6) ist es aber notwendig, dass die Variable D (dort D^*_{t+1} genannt) maximal den Wert Eins annimmt.

Deshalb definieren wir $D^*(x)$ als $\min\{1, \mathrm{VAL}(x') - \mathrm{VAL}(x)\}$. Damit ist klar, dass D^* nur ganzzahlige Werte aus $\{1 - 2(n/2) - (n/2 - 1), \ldots, 1\}$ annehmen kann und stochastisch kleiner als $D(x)$ ist, d. h. für alle $r \in \{2 - 3n/2, \ldots, 1\}$ gilt $\mathrm{P}(D^*(x) \leq r) \geq \mathrm{P}(D(x) \leq r)$. Aus letzterem folgt, dass der Erwartungswert von $D^*(x)$ nicht größer als der von $D(x)$ sein kann. Wenn T^* also die Zufallsvariable ist, die angibt, nach wievielen unabhängigen Wiederholungen von $D^*(x)$ deren Summe mindestens gleich Eins ist, so ist diese für unsere obige Abschätzung geeignet.

Wenn $S \subseteq \{1, \ldots, n\}$ die Menge der Indizes der Bits ist, die in dieser Mutation zwischen x und x' von Null zu Eins mutieren, so kann S nicht leer sein. Abhängig davon, ob es einen Index aus S gibt, der höchstens $n/2$ ist, werden wir den Erwartungswert der Zufallsvariablen $D^*(x)$ unterschiedlich abschätzen. Wenn wir zeigen können, dass dieser durch eine positive Konstante d^* nach unten beschränkt ist, so wird daraus mit einer Variation der Wald'schen Gleichung folgen, dass die erwartete Zahl erfolgreicher Mutationen, bis der Wert von VAL um Eins gestiegen ist, höchstens gleich $1/d^*$ ist, was für unsere obige Überlegung notwendig ist.

1. Die Menge S enthält einen Index $p \leq n/2$. Sei T die Menge der Indizes $i \in \{1, \ldots, n\}$ mit $x_i = 1$ und sei $\mathcal{U}$ die Familie aller Teilmengen $T' \subseteq T$, so dass eine Mutation aller x_i mit $i \in T' \cup S$ erfolgreich ist. Diese Teilmengen sind natürlich von S abhängig. Da wir S in allen folgenden Betrachtungen als beliebig, aber fest gewählt haben, wird unsere Notation dies nicht ausdrücklich widerspiegeln.

   Dann bezeichnen wir mit $j_2 \in \{1, \ldots, n/2 - 1\}$ die Zahl der Positionen $i \leq n/2$ mit $\{i\} \in \mathcal{U}$ und mit $j_1 \in \{1, \ldots, n/2\}$ die Zahl der Positionen $\{i\} \in \mathcal{U}$ mit $i > n/2$. Alle anderen Bits mit Index aus T haben einen so großen Koeffizienten, dass sie nicht von Eins zu Null mutieren können, ohne den Funktionswert von f zu verkleinern, da nur die Bits aus S von Null zu Eins werden. Da wir annehmen, dass die Mutation erfolgreich ist, können also neben den mutierten Bits mit Indizes aus S nur diese $j_2 + j_1$ Bits mutiert sein.

   Die erwartete Anzahl davon mutierender Bits ist $(j_2 + j_1)/n$, die den Wert von $D^*(x)$ um erwartet $-(2j_2 + j_1)/n$ senken. Da die Bedingung, dass die Mutation erfolgreich ist, diese Anzahl nur senken kann (mutieren zuviele Bits von Eins auf Null, wird die Mutation nicht mehr erfolgreich sein), ist dies eine obere Schranke für die erwartete Zahl der von Eins zu Null mutierenden Bits.

   Zusammen kämen wir damit auf eine untere Schranke von $2 - (2j_2 + j_1)/n$ für $E(D^*(x))$, wobei wir aber vergessen haben, dass $D^*(x)$ maximal gleich Eins ist, d. h. der Wert von $D^*(x)$ um Eins gesenkt wird, falls keine Eins zu Null mutiert. Dafür kommen natürlich nur die $j_2 + j_1$ ausgezeichneten Einsen in Frage, da in einer erfolgreichen Mutation alle anderen Einsen nicht mutieren können. Also brauchen wir eine obere Schranke für die Wahrscheinlichkeit, dass keine dieser $j_2 + j_1$ Stellen mutiert (Ereignis A) unter der Bedingung, dass die Mutation erfolgreich ist (Ereignis B).

Alle diese Ereignisse werden unter der Voraussetzung betrachtet, dass genau die Bits mit Indizes aus S von Null auf Eins mutieren und keine anderen als die $j_2 + j_1$ betrachteten Einsen mutieren. Da die Wahrscheinlichkeit dieses Ereignisses in der folgenden Betrachtung der bedingten Wahrscheinlichkeit $\mathrm{P}(A|B) = \mathrm{P}(A \cap B)/\mathrm{P}(B)$ in Nenner und Zähler auftritt, spielt sie keine Rolle, was auch anschaulich einsichtig ist.

Die Wahrscheinlichkeit von A ist gleich $(1-1/n)^{j_2+j_1}$. Die Wahrscheinlichkeit von B ist mindestens gleich

$$\left(\left(1-\frac{1}{n}\right)^{j_2+j_1} + \frac{j_2+j_1}{n}\cdot\left(1-\frac{1}{n}\right)^{j_2+j_1-1}\right).$$

Da unter der besprochenen Voraussetzung aus der Tatsache, dass keine der $j_1 + j_2$ Einsen zu Null wird, folgt, dass die Mutation erfolgreich ist, ist $\mathrm{P}(A \cap B) = \mathrm{P}(A)$. Also gilt

$$\mathrm{P}(A|B) \le \frac{1-1/n}{1-1/n+(j_2+j_1)/n} \le \frac{1}{1+(j_2+j_1)/n},$$

wobei die letzte Ungleichung aus

$$\forall a,b,c \in \mathbb{R}^+ : c < \min\{a,b\} : \left(\frac{a-c}{b-c} \le \frac{a}{b} \iff 1-\frac{c}{a} \le 1-\frac{c}{b} \iff a \le b\right)$$

folgt. Also gilt für den Erwartungswert von $D^*(x)$:

$$\mathrm{E}(D^*(x)) \ge 2 - \frac{2j_2+j_1}{n} - \frac{1}{1+(j_2+j_1)/n}.$$

Um diesen Ausdruck nach unten durch eine positive Konstante abzuschätzen, unterscheiden wir nach dem Wert von $j_2 + j_1$:

- Ist $j_2 + j_1 \le n/4$, so ist $2j_2 + j_1 \le n/2$ und deshalb $\mathrm{E}(D^*(x)) \ge 1/2$.
- Ist $j_2 + j_1 \in \{n/4+1, \dots, n/2\}$, so ist $2j_2 + j_1 \le n$ und $j_2 + j_1 > n/4$, was zur Folge hat, dass $\mathrm{E}(D^*(x))$ größer als $1/5$ ist.
- Ist $j_2 + j_1 > n/2$, so erhalten wir, da die untere Schranke in j_2 schneller als in j_1 sinkt, mit $j_2 = n/2$ die $\mathrm{E}(D^*(x))$ nach unten beschränkende Funktion

  $$1 - \frac{j_1}{n} - \frac{1}{3/2 + j_1/n}.$$

  Ist für ein $\varepsilon > 0$ konstanter Größe $j_1 \le (1-\varepsilon)\cdot n/3$, so ist $\mathrm{E}(D^*(x))$ mindestens gleich $\varepsilon/3$. Ist $j_1 > (1-\varepsilon)\cdot n/3$, so ist unsere Argumentation ein wenig komplizierter: sei $v = \binom{j_1+j_2}{2} > n/4 \cdot (n/2-1)$ die Anzahl der Paare der $j_1 + j_2$ betrachteten Positionen und $v' \le v$ die Zahl der Paare, die mutieren können, ohne dass die Mutation unerfolgreich wird. Damit können wir die untere Schranke für die Wahrscheinlichkeit $\mathrm{P}(B)$ einer erfolgreichen Mutation um $v' \cdot 1/n^2 \cdot (1-1/n)^{j_2+j_1-2}$ erhöhen.

  Falls $v' \ge v/2$ ist, so ist diese Erhöhung aufgrund der unteren Schranke für v von $\Omega(n^2)$ mindestens gleich einer Konstanten $\delta > 0$. Setzt man j_1 auf den größtmöglichen Wert $n/2$, so ergäbe sich mit der alten Abschätzung für $\mathrm{P}(A|B)$ genau Null als untere Schranke von $\mathrm{E}(D^*(x))$; mit der neuen Schranke jedoch die positive Konstante $1/2 - (2+\delta)^{-1}$.

Betrachten wir nun den Fall, dass v', die Anzahl der Paare aus den betrachteten $j_2 + j_1$ Bits, die mutieren dürfen, kleiner als $v/2$ ist. Da nun $\Omega(n^2)$ Paare nicht mutieren dürfen, ist die Wahrscheinlichkeit, dass eines dieser Paare mutieren will, mindestens gleich einer Konstanten $\gamma > 0$. Also kann der Anteil $-(2j_2+j_1)/n$ der mutierenden Einsen an $\mathrm{E}(D^*(x))$ um γ gesenkt werden, d. h. $\mathrm{E}(D^*(x)) \geq \gamma$.

Falls also aus der linken Hälfte ein Bit von Null zu Eins mutiert, so ist der erwartete Zugewinn pro erfolgreicher Mutation mindestens gleich einer positiven Konstante.

2. Die Menge S enthält nur Indizes $p > n/2$. Nun müssen wir genauer argumentieren, da nur noch ein Zuwachs von VAL um Eins durch jede der mutierten Nullen in S gesichert ist. Deshalb unterscheiden wir nach $s = |S|$, der Anzahl der mutierten Nullen.

   Die in der erfolgreichen Mutation zu Nullen mutierenden Einsen werden nach ihrer Position unterschieden: j_2 bezeichnet dabei die Anzahl der in der linken Hälfte mutierenden Einsen (die jeweils mit dem Wert -2 in VAL einfließen) und j_1 die der rechten Hälfte (die jeweils mit -1 einfließen). Um den erwarteten Zugewinn von $D^*(x)$ nach unten abzuschätzen, müssen wir dann die Wahrscheinlichkeiten geeignet abschätzen, mit denen verschiedene Wertekombinationen von j_2 und j_1 angenommen werden.

   Dies wird aber nicht stets unter der Bedingung geschehen, dass die Mutation erfolgreich ist, wie es eigentlich geschehen müsste. Grund dafür ist, dass alle im Folgenden betrachteten Fälle implizieren, dass die Mutation erfolgreich ist, weshalb die bedingten Wahrscheinlichkeiten multiplizert mit der einer erfolgreichen Mutation die betrachteten Wahrscheinlichkeiten ergeben. Also sind die bedingten Wahrscheinlichkeiten mindestens so groß wie die der im weiteren betrachteten Ereignisse. Wenn wir also im Folgenden zeigen, dass der erwartete Zugewinn für die verschiedenen Werte von s mindestens konstant ist, so gilt dies auch für die bedingten Erwartungswerte.

   Zur leichteren Abschätzung des Zugewinns $D^*(x)$ werden die Binomial-Verteilungen der Zahlen j_2 und j_1 der mutierenden Einsen durch Binomial-Verteilungen mit Parametern $1/n$ und $n/2$ und diese wiederum durch die Poisson-Verteilung mit Parameter $1/2$ nach oben abgeschätzt (A.7):

$$\sum_{k=0}^{n/2} \left| \binom{n/2}{k} \cdot \left(\frac{1}{n}\right)^k \cdot \left(1 - \frac{1}{n}\right)^{n/2-k} - \frac{1}{\exp(1/2) \cdot k! \cdot 2^k} \right| \leq \frac{1}{n}.$$

   Dadurch, dass die Anzahl der Einsen im mutierenden Bitstring insgesamt durch n nach oben abgeschätzt wird, wird die erwartete Anzahl der mutierenden Einsen erhöht. Da diese aber negativen Einfluss auf $D^*(x)$ haben, ist dies für unsere untere Abschätzung von $D^*(x)$ zulässig. Weil die Binomial-Verteilung gegen die Poisson-Verteilung konvergiert, gibt es für jede Konstante $\varepsilon > 0$ ein $n_0 \in \mathbb{N}$, so dass für alle $n \geq n_0$ der Unterschied zu den beiden Binomial-Verteilungen mit Parametern $n/2$ und $1/n$ jeweils kleiner als $\varepsilon/3$ ist. Da wir im Folgenden mithilfe der Poisson-Verteilung zeigen, dass der erwartete Zugewinn $\mathrm{E}(D^*(x))$ stets gleich einer positiven Konstanten $\varepsilon > 0$ ist, gilt dies ab diesem n_0 dann auch für die ursprünglichen Binomial-Verteilungen mit $\varepsilon/3$.

   Kommen wir jetzt zu den einzelnen Fällen:

(a) Sei $s = 1$. Da nur eine Null der rechten Hälfte zu Eins wird, darf in der linken Hälfte maximal eine Eins mutieren, in der rechten Hälfte dürfen aber beliebig viele Einsen mutieren:

i. Falls $j_2 = 0$ und $j_1 = 0$, so ist der Zugewinn gleich Eins. Da die Wahrscheinlichkeit dafür gleich

$$\frac{1}{\exp(1/2) \cdot 0! \cdot 2^0} \cdot \frac{1}{\exp(1/2) \cdot 0! \cdot 2^0} = \frac{1}{\exp(1)}$$

ist, ist der erwartete Zugewinn in diesem Fall gleich $\exp(-1)$.

ii. Ist $j_2 = 0$ und $j_1 \geq 1$, so ist der erwartete Zugewinn gleich

$$\sum_{j_1=1}^{n/2-1} (1 - j_1) \cdot \frac{1}{\exp(1) \cdot j_1! \cdot 2^{j_1}} \geq -\frac{1}{\exp(1) \cdot 4}.$$

iii. Ist $j_2 = 1$ und $j_1 = 0$, so ist der erwartete Zugewinn gleich

$$-\frac{1}{\exp(1) \cdot 2}.$$

Da alle anderen Fälle nicht zu erfolgreichen Mutationen führen können, ist der erwartete Zugewinn für den Fall $s = 1$ mindestens gleich $\exp(-1) \cdot (1 - 1/4 - 1/2) = \exp(-1) \cdot 1/4$.

(b) Sei $s = 2$. Dann gibt es die folgenden Möglichkeiten für Werte von j_2 und j_1, die zu erfolgreichen Mutationen führen können:

i. Falls $j_2 = 0$ und $j_1 = 0$ ist, so ist der erwartete Zugewinn gleich $\exp(-1)$.

ii. Falls $j_2 = 0$ und $j_1 = 1$ ist, so ist der erwartete Zugewinn gleich $1/(\exp(1) \cdot 2)$.

iii. Falls $j_2 = 0$ und $j_1 \geq 2$ ist, so ist der erwartete Zugewinn gleich

$$\sum_{j_1=2}^{n/2-2} (2 - j_1) \cdot \frac{1}{\exp(1) \cdot j_1! \cdot 2^{j_1}} \geq -\frac{1}{\exp(1) \cdot 24}.$$

iv. Falls $j_2 = 1$ und $j_1 \geq 0$, so ist der erwartete Zugewinn gleich

$$\sum_{j_1=0}^{n/2-2} (-j_1) \cdot \frac{1}{\exp(1) \cdot j_1! \cdot 2^{j_1+1}} \geq -\frac{1}{\exp(1) \cdot 2}.$$

v. Falls $j_2 = 2$ und $j_1 = 0$, so ist der erwartete Zugewinn gleich $-2/(\exp(1) \cdot 2! \cdot 2^2) = -1/(\exp(1) \cdot 4)$.

Da alle anderen Fälle zu erfolglosen Mutationen führen, ist der erwartete Zugewinn für $s = 2$ mindestens gleich

$$\exp(-1) \cdot \left(1 + \frac{1}{2} - \frac{1}{24} - \frac{1}{2} - \frac{1}{4}\right) = \exp(-1) \cdot \frac{17}{24}.$$

(c) Sei $s = 3$. Dann können folgende Fälle zu erfolgreichen Mutationen führen:

i. Falls $j_2 = 0$ und $j_1 = 0$, so ist der erwartete Zugewinn gleich $\exp(-1)$.

ii. Falls $j_2 = 0$ und $j_1 = 1$, so ist der erwartete Zugewinn gleich $1/(\exp(1) \cdot 2)$.

iii. Falls $j_2 = 0$ und $j_1 = 2$, so ist der erwartete Zugewinn gleich $1/(\exp(1) \cdot 8)$.

iv. Falls $j_2 = 0$ und $j_1 \geq 3$, so ist der erwartete Zugewinn gleich

$$\sum_{j_1=3}^{n/2-3} (3 - j_1) \cdot \frac{1}{\exp(1) \cdot j_1! \cdot 2^{j_1}} \geq -\frac{1}{\exp(1) \cdot 192}.$$

v. Falls $j_2 = 1$ und $j_1 = 0$, so ist der erwartete Zugewinn gleich $1/(\exp(1) \cdot 2)$.

vi. Falls $j_2 = 1$ und $j_1 \geq 1$, so ist der erwartete Zugewinn gleich

$$\sum_{j_1=1}^{n/2-3} (1 - j_1) \cdot \frac{1}{\exp(1) \cdot j_1! \cdot 2^{1+j_1}} \geq -\frac{1}{\exp(1) \cdot 8}.$$

vii. Falls $j_2 = 2$ und $j_1 \geq 0$, so ist der erwartete Zugewinn gleich

$$\sum_{j_1=0}^{n/2-3} (-1 - j_1) \cdot \frac{1}{\exp(1) \cdot 2 \cdot j_1! \cdot 2^{2+j_1}} \geq -\frac{1}{\exp(1) \cdot 4}.$$

viii. Falls $j_2 = 3$ und $j_1 = 0$, so ist der erwartete Zugewinn gleich $-1/(\exp(1) \cdot 16)$.

Also ist im Fall $s = 3$ der erwartete Zugewinn mindestens gleich

$$\exp(-1) \cdot \left(1 + \frac{1}{2} + \frac{1}{8} - \frac{1}{192} + \frac{1}{2} - \frac{1}{8} - \frac{1}{4} - \frac{1}{16}\right) = \exp(-1) \cdot 323/192.$$

(d) Sei $s \geq 4$. In diesem Fall lassen wir alle erfolglosen Mutationen zu, was die erwartete Zahl der von Eins zu Null mutierenden Bits nur steigern kann. Wenn wir also schon hier zeigen können, dass der erwartete Wert von $D^*(x)$ größer als eine Konstante $\varepsilon > 0$ ist, so gilt dies bei Beschränkung auf erfolgreiche Mutationen erst recht. Wir werden nach dem Wert des Paares (j_2, j_1) eine vollständige Fallunterscheidung vornehmen, wobei nicht erwähnte Fälle stets den Wert von $D^*(x)$ nicht senken können. Wenn alle betrachteten Fälle in ihrer Summe also den Wert von $D^*(x)$ um mindestens ein $\varepsilon > 0$ steigern, so gilt dies auch für den erwarteten Zugewinn von $D^*(x)$.

i. Falls $j_2 = 0$ und $j_1 = 0$, ist der erwartete Zugewinn gleich $\exp(-1)$.

ii. Falls $j_2 = 0$ und $j_1 \geq 5$ (für $j_1 < 5$ ist der Zugewinn mindestens gleich Null), so ist der erwartete Zugewinn mindestens gleich

$$\sum_{j_1=5}^{n/2-4} (4 - j_1) \cdot \frac{1}{\exp(1) \cdot j_1! \cdot 2^{j_1}} \geq -\frac{1}{\exp(1) \cdot 1920}.$$

iii. Falls $j_2 = 1$ und $j_1 \geq 3$ (für $j_1 < 3$ ist der Zugewinn mindestens gleich Null), so ist der erwartete Zugewinn mindestens gleich

$$\sum_{j_1=3}^{n/2-4} (2 - j_1) \cdot \frac{1}{\exp(1) \cdot j_1! \cdot 2^{j_1+1}} \geq -\frac{1}{\exp(1) \cdot 48}.$$

iv. Falls $j_2 \geq 2$, so ist der erwartete Zugewinn mindestens gleich

$$\begin{aligned}
&\sum_{j_2=2}^{n/2} \sum_{j_1=0}^{n/2-4} (4-2j_2-j_1) \cdot \frac{1}{\exp(1) \cdot j_2! \cdot j_1! \cdot 2^{j_2+j_1}} \\
= &\sum_{j_2=2}^{n/2} \left(\frac{4-2j_2}{\exp(1) \cdot j_2! \cdot 2^{j_2}} + \sum_{j_1=1}^{n/2-4} \frac{4-2j_2-j_1}{\exp(1) \cdot j_2! \cdot j_1! \cdot 2^{j_2+j_1}} \right) \\
\geq &\sum_{j_2=2}^{n/2} \left(\frac{4-2j_2}{\exp(1) \cdot j_2! \cdot 2^{j_2}} + \frac{3-2j_2}{\exp(1) \cdot j_2! \cdot 2^{j_2}} \right) \\
\geq &-\frac{1}{\exp(1) \cdot 8} - \frac{5}{\exp(1) \cdot 48} - \frac{9}{\exp(1) \cdot 192} = -\frac{53}{\exp(1) \cdot 192}.
\end{aligned}$$

Da $1 - 1/1920 - 1/48 - 53/192 = 134/192 > 0$ ist, ist auch für $s \geq 4$ der erwartete Zugewinn mindestens gleich einer Konstanten $\varepsilon > 0$.

Also nimmt $\text{VAL}(x)$ in jeder erfolgreichen Mutation mindestens um eine positive Konstante $d^* \in]0,1]$ zu. Die Größe, die wir damit im Folgenden abschätzen wollen, ist die erwartete Anzahl $\mathrm{E}(T^*(x))$ von Schritten, bis der Wert von VAL größer ist als $\text{VAL}(x)$, wenn bei x gestartet wird. Da diese Größe von x abhängt, benötigen wir als eine leichte Verallgemeinerung der Wald'schen Gleichung, dass auch in diesem Fall $\mathrm{E}(T^*(x)) \leq 1/d^*$ ist. Dies beweisen wir durch Induktion über $\text{VAL}(x)$:

Ist $x = (0, \ldots, 0)$, d. h. $\text{VAL}(x) = 0$, so steigt VAL mit der ersten erfolgreichen Mutation, der Induktionsanfang ist also bewiesen. Nehmen wir nun an, dass die Ungleichung $\mathrm{E}(T^*(x)) \leq 1/d^*$ für alle $x \in \{0,1\}^n$ mit $\text{VAL}(x) \leq k$ bewiesen ist. Sei dann ein $x \in \{0,1\}^n$ betrachtet, das unter allen Bitstrings $x' \in \{0,1\}^n$ mit $\text{VAL}(x') = k+1$ maximalen $\mathrm{E}(T^*(x'))$-Wert hat. Wir schätzen den Erwartungswert $\mathrm{E}(T^*(x))$ nun mittels der Zunahme von VAL in der ersten erfolgreichen Mutation, d. h. $D^*(x)$, ab. Dabei benutzen wir nach Induktionsvoraussetzung, dass die erwartete Zeit, um $\text{VAL}(x) + 1$ zu erreichen, höchstens gleich $1 + \mathrm{E}(T^*(x)) - d/d^*$ ist, wenn in der ersten erfolgreichen Mutation $D^*(x) = d \leq 0$ ist. Also ist $\mathrm{E}(T^*(x))$ höchstens

$$\begin{aligned}
&1 + \sum_{d \leq 0} \mathrm{P}(D^*(x) = d) \cdot \left(\mathrm{E}(T^*(x)) - \frac{d}{d^*} \right) \\
= \; &1 + \mathrm{P}(D^*(x) \leq 0) \cdot \mathrm{E}(T^*(x)) + \frac{\mathrm{P}(D^*(x) = 1)}{d^*} - \sum_{d \leq 1} \frac{\mathrm{P}(D^*(x) = d) \cdot d}{d^*} \\
= \; &1 + \mathrm{E}(T^*(x)) - \mathrm{E}(T^*(x)) \cdot \mathrm{P}(D^*(x) = 1) + \frac{\mathrm{P}(D^*(x) = 1}{d^*} - \frac{\mathrm{E}(D^*(x))}{d^*}.
\end{aligned}$$

Äquivalent dazu ist, dass folgendes gilt

$$\begin{aligned}
\mathrm{E}(T^*(x)) \cdot \mathrm{P}(D^*(x) = 1) \;&\leq\; 1 + \frac{\mathrm{P}(D^*(x) = 1)}{d^*} - \frac{\mathrm{E}(D^*(x))}{d^*} \\
&\leq\; 1 + \frac{\mathrm{P}(D^*(x) = 1)}{d^*} - \frac{d^*}{d^*} = \frac{\mathrm{P}(D^*(x) = 1)}{d^*}.
\end{aligned}$$

Also gilt die Behauptung für alle $x \in \{0,1\}^n$. Mit unseren Vorbemerkungen ist der Beweis damit beendet. □

Eine untere Schranke der erwarteten Laufzeit von $\Omega(n \log(n))$ für lineare Funktionen f folgt analog zum Beweis von Lemma 3.3.4, wenn es mindestens $\Omega(n)$ viele Koeffizienten in der Polynomdarstellung von f gibt, die ungleich Null sind. Denn dann wird nach der Tschernoff-Ungleichung (A.9) ein konstanter Anteil der Bits, die zu diesen Koeffizienten gehören, bei der Initialisierung falsch gesetzt. Die Wartezeit, bis alle diese $\Omega(n)$ Bits mindestens einmal mutieren wollen, ist $\Omega(n \log(n))$. Somit folgt:

Korollar 3.4.10 *Sei* $f : \{0,1\}^n \mapsto \mathbb{R}$ *eine zu maximierende lineare Zielfunktion, deren Polynomdarstellung* $\Omega(n)$ *viele von Null verschiedene Koeffizienten enthält. Dann ist die erwartete Laufzeit des* (1+1) EA *auf* f *gleich* $\Theta(n \log(n))$.

Da ein Bit x_i, das in einer Funktion $f : \{0,1\}^n \mapsto \mathbb{R}$ keinen Einfluss hat, d. h. für das $c_S = 0$ für alle Teilmengen $S \subseteq \{1, \ldots, n\}$ mit $i \in S$ ist, aus der Betrachtung der Funktion gestrichen werden kann, werden wir nur noch Funktionen betrachten, bei denen jedes Bit Einfluss hat. Unter dieser Annahme drückt Korollar 3.4.10 aus, dass jede lineare Funktion vom (1+1) EA in Zeit $\Theta(n \log(n))$ optimiert wird.

Natürlich lassen sich lineare Funktionen einfach in Zeit $\Theta(n)$ optimieren, indem man z. B. mit dem String $(0, \ldots, 0)$ startet und dann jedes einzelne Bit kippt und anhand der Veränderung des Funktionswerts bestimmt, ob dieses Bit einen negativen oder positiven Koeffizienten hat. Insofern scheint eine erwartete Laufzeit von $\Theta(n \log(n))$ schlecht zu sein. Doch muss man hier bedenken, dass der (1+1) EA an keiner Stelle explizit auf lineare Funktionen abgestimmt ist, während das oben erwähnte Verfahren nur für Funktionen, bei denen ein Bit unabhängig von der Belegung der anderen stets einen positiven bzw. negativen Einfluss auf den Funktionswert hat, korrekt ist. Unter diesem Blickwinkel ist die erwartete Laufzeit des (1+1) EA, die nur um einen Faktor $\log(n)$ von der bestmöglichen Laufzeit aller Verfahren entfernt ist, sehr gut.

3.4.3 Analysen für Polynome vom Grad Zwei

Nun ist es natürlich interessant herauszufinden, für welche anderen Funktionen der (1+1) EA die für ihn bestmögliche erwartete Laufzeit $\Theta(n \log(n))$ hat bzw. welche Klassen von Funktionen Elemente besitzen, auf denen er eine größere erwartete Laufzeit hat. Für die im Weiteren zu untersuchende Funktionsklasse gibt es mehrere Möglichkeiten. Wir werden als nächstes Polynome vom Grad Zwei untersuchen und zwar aus der Überlegung heraus, dass vielleicht der Grad k der Polynomdarstellung direkt mit der Schwierigkeit für den (1+1) EA in Zusammenhang stehen könnte. Für die Anfangsfälle $k = 0$ mit konstanter Laufzeit 0 und $k = 1$ mit erwarteter Laufzeit $\Theta(n \log(n))$ trifft dies zu. Können wir zeigen, dass jedes Polynom vom Grad Zwei auch in polynomieller Zeit, z. B. $O(n^2 \log(n))$, vom (1+1) EA optimiert werden kann?

Ein solcher Beweis wird uns nicht gelingen, denn man kann explizit eine Funktion zweiten Grades angeben, zu deren Optimierung der (1+1) EA exponentielle erwartete Laufzeit benötigt. Worin liegt es begründet, dass beim Übergang zu Polynomen zweiten Grades dieser Sprung von effizient zu nicht effizient optimierbaren Funktionen für den (1+1) EA erfolgt. Eine anschauliche Begründung ist die folgende:

Wir haben schon bei der Konstruktion von OneMax und BinVal über den positiven Effekt von „richtigen Hinweisen“, d. h. Mutationen, die den Funktionswert vergrößern und den Hamming-Abstand zum Optimum verringern, argumentiert. Nehmen wir bei einer linearen Funktion mit genau einem Optimum nach Lemma 3.1.4 o. B. d. A. an, dass $(1, \ldots, 1)$ das Optimum ist, so wird jede Mutation, die genau eine Null zu einer Eins negiert, ein richtiger Hinweis sein. Somit gibt es bei linearen Funktionen für jeden nicht-optimalen Punkt einen richtigen Hinweis, der Wahrscheinlichkeit $\Omega(1/n)$ hat.

Da es wahrscheinlicher ist, dass ein Bit nicht mutieren will als dass es mutieren will, werden Mutationen mit zunehmender Zahl von mutierenden Bits im-

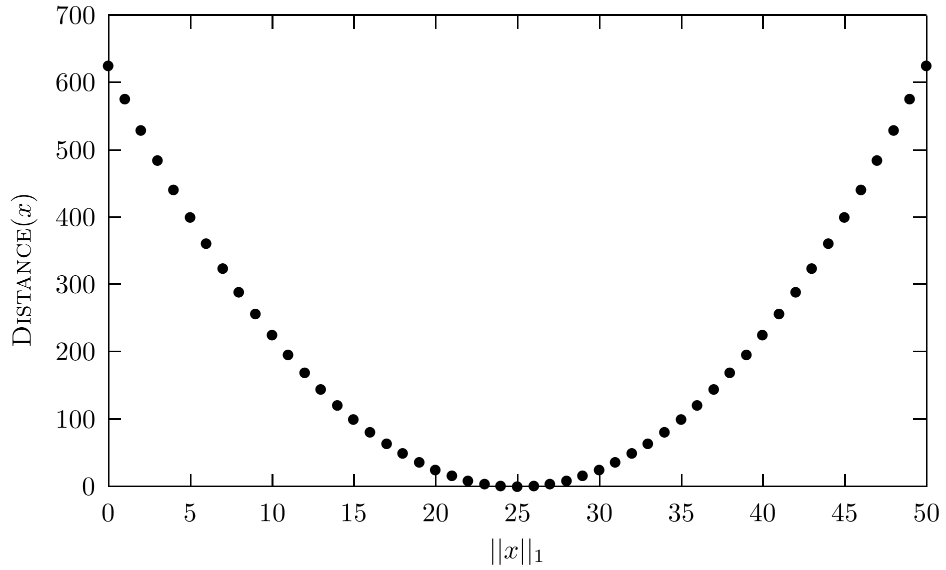

Abbildung 3.2: Die Funktion DISTANCE für $n = 50$.

mer unwahrscheinlicher. Also werden mit hoher Wahrscheinlichkeit nur Punkte „in der Nähe" des aktuellen Punkts x_t, d. h. mit geringem Hamming-Abstand von x_t durch Mutation erreicht. Wenn nun aber für einen mindestens konstanten Anteil der Suchpunkte die meisten benachbarten Suchpunkte, die einen höheren Funktionswert haben, von den Optima wegführen, so wird die Folge der aktuellen Punkte des (1+1) EA mit nicht zu vernachlässigender Wahrscheinlichkeit von dem Optimum wegführen. Wenn die so erreichten lokalen Optima, d. h. global nicht optimale Punkte, deren Nachbarn aber alle schlechter sind, einen mindestens linear großen Hamming-Abstand von allen besseren Punkten haben, so ist die Wartezeit auf eine verbessernde Mutation exponentiell groß.

Dass Polynome zweiten Grades solche Eigenschaften besitzen können, ist am einfachsten mit einem konkreten Beispiel klarzumachen. Dazu dient die folgende Definition der Funktion DISTANCE:

Definition 3.4.11 *Die Funktion* DISTANCE : $\{0,1\}^n \mapsto \mathbb{R}$ *ist definiert als*

$$\text{DISTANCE}(x) := \left(\sum_{i=1}^{n} x_i - \left(\frac{n}{2} + \frac{1}{3} \right) \right)^2 .$$

In Abbildung 3.2 ist die Funktion DISTANCE für $n = 50$ veranschaulicht. Klar ist, dass DISTANCE als Quadrat einer linearen Funktion ein Polynom zweiten Grades ist. Da sie symmetrisch ist, kann eine Funktion DISTANCE* : $\{0,\dots,n\} \mapsto \mathbb{R}$ gemäß

$$\forall s \in \{0,\dots,n\} : \text{DISTANCE}^*(s) := \text{DISTANCE}(x) \text{ mit } ||x||_1 = s$$

eindeutig definiert ist. Die Betrachtung von DISTANCE* macht die Argumentation über DISTANCE einfacher.

Graphisch veranschaulicht entspricht DISTANCE*, wie in Abbildung 3.2 nur noch zu erahnen ist, einer ganz leicht nach links hochgeschobenen Parabel, wodurch die Reihenfolge der Funktionswerte die folgende ist (n sei o. B. d. A. gerade):

$$\text{DISTANCE}^*(n/2) < \text{DISTANCE}^*(n/2+1) < \text{DISTANCE}^*(n/2-1) < \cdots < \text{DISTANCE}^*(n) < \text{DISTANCE}^*(0).$$

Also übernimmt der (1+1) EA, wenn der aktuelle Punkt Hamming-Gewicht $n/2 + d$ hat, nur Mutationen zu Punkten, deren Hamming-Gewicht mindestens $n/2 + d$ oder höchstens $n/2 - d$ ist. Bezeichnet man die rechte Hälfte der Punkte mit mindestens $n/2 + 1$ Einsen als die schlechte Hälfte, so sieht man daran, dass der (1+1) EA nur mit geringer Wahrscheinlichkeit in die linke, d. h. die richtige Hälfte mutieren wird, wenn er erst einmal weit genug in der schlechten Hälfte ist. Da die schlechte Hälfte für sich allein bis auf Skalierung wie die lineare Funktion OneMax aufgebaut ist, wird die Wahrscheinlichkeit recht hoch sein, das lokale Optimum $(1, \ldots, 1)$ erreicht zu haben, ohne zuvor in die linke Hälfte gesprungen zu sein. Einmal dort angekommen ist die Mutation zum globalen Optimum $(0, \ldots, 0)$ die einzig mögliche. Da globales und lokales Optimum komplementär zueinander sind, beträgt die erwartete Wartezeit auf das Erreichen des globalen Optimums dann n^n.

Natürlich lässt sich diese Argumentation auch fast vollkommen symmetrisch für die linke, die gute Hälfte des Suchraums führen, d. h. mit mindestens ebenso hoher Wahrscheinlichkeit wird sich der (1+1) EA nur in der linken Hälfte aufhalten, wo er in Zeit $O(n \log(n))$ das globale Optimum erreicht. Mit großer Wahrscheinlichkeit wird die Laufzeit also polynomiell beschränkt sein.

Die angestellten Überlegungen zu präzisieren und bestätigen, ist Aufgabe des folgenden Theorems:

Theorem 3.4.12 *Die erwartete Laufzeit des* (1+1) EA *auf* Distance *ist* $\Theta(n^n)$. *Für jedes* $\varepsilon > 0$ *ist die Laufzeit mit einer Wahrscheinlichkeit von mindestens* $1/2-\varepsilon$ *jedoch gleich* $O(n \log(n))$.

Beweis: Um zeigen, dass die erwartete Laufzeit $\Theta(n^n)$ ist, werden wir nachweisen, dass der (1+1) EA mit konstanter Wahrscheinlichkeit mit mindestens $n/2 + n^{1/4}$ Einsen initialisiert und daraufhin mit gegen Eins konvergierender Wahrscheinlichkeit das lokale Optimum $(1, \ldots, 1)$ erreicht.

Die Wahrscheinlichkeit, mit mindestens $n/2 + n^{1/4}$ Einsen zu initialisieren, ist gleich

$$\begin{aligned}
& 2^{-n} \cdot \sum_{i=n/2+n^{1/4}}^{n} \binom{n}{i} \\
= \; & 2^{-n} \cdot \left(\sum_{i=n/2}^{n} \binom{n}{i} - \sum_{i=n/2}^{n/2+n^{1/4}-1} \binom{n}{i} \right) \\
\geq \; & 2^{-n} \cdot \left(2^{n-1} - n^{1/4} \cdot \binom{n}{n/2} \right) \\
= \; & 1/2 - 2^{-n} \cdot n^{1/4} \cdot \frac{n!}{((n/2)!)^2} \\
\geq \; & 1/2 - 2^{-n} \cdot n^{1/4} \cdot \frac{(n/\exp(1))^n \cdot \sqrt{2\pi \cdot n} \cdot \exp(1)}{(n/(2\exp(1)))^n \cdot \pi \cdot n} \\
= \; & 1/2 - 2^{-n} \cdot n^{1/4} \cdot \sqrt{\frac{2}{\pi}} \cdot \exp(1) \cdot \frac{2^n}{\sqrt{n}} \\
= \; & 1/2 - O(n^{-1/4}).
\end{aligned}$$

Der (1+1) EA verhält sich auf der schlechten Hälfte von Distance wie auf OneMax, wenn man voraussetzt, dass er niemals zu einem Bitstring mit maximal $n/2$ Einsen mutiert. Da wir im Weiteren davon ausgehen, mit mindestens $n/2+n^{1/4}$ Einsen initialisiert zu haben, muss eine Mutation in die gute Hälfte mindestens $2n^{1/4}$ Bits in einer Mutation negieren, da alle Bitstrings x' mit $n/2-n^{1/4} < ||x'||_1 <$

$n/2+n^{1/4}$ einen echt kleineren Funktionswert als ein Punkt x mit $||x||_1 \geq n/2+n^{1/4}$ haben. Die Wahrscheinlichkeit einer Mutation von mindestens k Bits lässt sich wie folgt nach oben abschätzen:

$$\binom{n}{k} \cdot \left(\frac{1}{n}\right)^k = \frac{n!}{k! \cdot (n-k)!} \cdot \left(\frac{1}{n}\right)^k < \frac{1}{k!}.$$

Nach der Stirling'schen Formel (A.10) gilt:

$$k! \geq \left(\frac{k}{\exp(1)}\right)^k = k^{\Omega(k)} = 2^{\Omega(k \cdot \log(k))}.$$

Also ist die Wahrscheinlichkeit einer Mutation von mindestens $2n^{1/4}$ Bits von der Größenordnung $2^{-\Omega(n^{1/4} \cdot \log(n))}$.

Auf OneMax ist die erwartete Laufzeit nach Lemma 3.4.2 gleich $\Theta(n \log(n))$ und somit für eine Konstante $c > 0$ durch $c \cdot n \log(n)$ nach oben beschränkt. Mit der Markoff-Ungleichung (A.8) folgt, dass der (1+1) EA mit einer Wahrscheinlichkeit von mindestens $1 - 1/c'$ nach $c' \cdot c \cdot n \log(n)$ Schritten das lokale Optimum von Distance erreicht, wenn er mit mindestens $n/2 + n^{1/4}$ Einsen startet und nie in die gute Hälfte mutiert. Dass letzteres in den $c' \cdot c \cdot n \log(n)$ betrachteten Schritten passiert, hat Wahrscheinlichkeit

$$c' \cdot c \cdot n \log(n) \cdot 2^{-\Omega(n^{1/4} \cdot \log(n))} = 2^{-\Omega(n^{1/4} \cdot \log(n))}.$$

Dass ersteres passiert, hat, wie gezeigt, Wahrscheinlichkeit $1/2 + O(n^{-1/4})$. Also ist die Summe der Fehlerwahrscheinlichkeiten gleich $1/2 + 1/c' + o(1)$, was für jedes $\varepsilon > 0$ mit genügend großem c' und n kleiner als $1/2 + \varepsilon$ wird. Damit ist insbesondere die Laufzeit mit konstanter Wahrscheinlichkeit gleich n^n, da vom lokalen Optimum nur durch eine gleichzeitige Mutation aller Bits zu dem globalen Optimum gesprungen werden kann.

Die Beweisführung für diesen Teil hätte auch einfacher erfolgen können, da für den Nachweis der erwarteten Laufzeit eine konstante untere Schranke für die Wahrscheinlichkeit, das lokale Optimum zu erreichen, ausreicht. Jedoch lassen sich die angestellten Betrachtungen auch genau umdrehen, so dass man eine untere Schranke von $1/2 - 1/c' - o(1)$ für die Wahrscheinlichkeit erhält, in Zeit $c' \cdot c \cdot n \log(n)$ das globale Optimum zu erreichen.

Denn aufgrund der Symmetrie beträgt die Wahrscheinlichkeit, mit höchstens $n/2 - n^{1/4}$ Einsen zu initialisieren, $1/2 - O(n^{1/4})$; eine Mutation in die rechte Hälfte muss nun sogar mindestens $2n^{1/4} + 1$ Bits gleichzeitig mutieren, so dass die Wahrscheinlichkeit hierfür auch $2^{-\Omega(n^{1/4} \cdot \log(n))}$ beträgt. Da sich Distance in der guten Hälfte wie −OneMax verhält, folgt ebenfalls, dass in $c' \cdot c \cdot n \log(n)$ Schritten mit einer Wahrscheinlichkeit von mindestens $1 - 1/c'$ das globale Optimum erreicht wird, wenn in der linken Hälfte initialisiert und diese nicht verlassen wird. □

Damit wissen wir, dass Polynome vom Grad Zwei für den (1+1) EA schwierig sein können, wenn man als Kriterium hierfür eine exponentielle erwartete Laufzeit nimmt. Jedoch wird Distance nach Korollar 3.3.8 von einer Multistart-Variante (Definition 3.3.6) mit $T(n) = O(n \log(n))$ und $L(n) = O(n \log(n))$ des (1+1) EA mit exponentiell hoher Wahrscheinlichkeit in polynomieller Zeit optimiert und auch die erwartete Laufzeit dieser Multistart-Variante ist nur polynomiell groß.

Obwohl die Funktion Distance also eine exponentielle erwartete Laufzeit des (1+1) EA impliziert, lässt sie sich von einer Multistartvariante desselben Algorithmus effizient optimieren. Warum denken wir aber, dass es auch eine Funktion vom Grad Zwei geben muss, die auch von jeder Multistartvariante nicht effizient optimiert wird, d. h. die mit exponentiell schnell gegen Eins konvergierender Wahrscheinlichkeit vom (1+1) EA nicht in polynomieller Laufzeit optimiert wird?

Der Grund dafür ist, dass das Problem, zu einer Funktion $f : \{0,1\}^n \mapsto \mathbb{N}$ vom Grad Zwei mit Koeffizienten aus $\mathbb{N}$ und einer Zahl $K \in \mathbb{N}$ zu berechnen, ob es eine Eingabe $x \in \{0,1\}^n$ mit $f(x) \geq K$ gibt, NP-vollständig ist. Dies ergibt sich z. B. durch einfache Transformation des Problems MAX-2-SAT, das NP-vollständig ist (Garey und Johnson (1979)).

Wenn der (1+1) EA also jedes Polynom vom Grad Zwei mit einer Wahrscheinlichkeit von $1/poly(n)$ in polynomieller Laufzeit optimieren würde, so ließe sich daraus per Multistartvariante ein Algorithmus konstruieren, der in polynomieller Laufzeit mit konstanter Wahrscheinlichkeit $c > 1/2$ ein Optimum findet. Gibt der Algorithmus dann die Antwort „Ja", wenn der Funktionswert des besten gefundenen Punkts mindestens K ist, und sonst „Nein" aus, so wird er für eine Funktion, die nie mindestens K ist, stets „Nein" ausgeben, ansonsten aber mit einer Wahrscheinlichkeit größer als c „Ja". Damit ist dieser Algorithmus ein RP („random polynomial")-Algorithmus (Papadimitriou (1994)) für ein NP-vollständiges Problem, d. h. RP wäre gleich NP. Dies wird aber gemeinhin nicht angenommen (Papadimitriou (1994)), da Probleme aus RP in der Praxis effizient lösbar sind, indem man z. B. das entsprechende randomisierte polynomiell zeitbeschränkte Programm $\Theta(n)$-mal wiederholt und akzeptiert, wenn es eine seiner Wiederholungen getan hat. Die Wahrscheinlichkeit, hiermit die falsche Entscheidung getroffen zu haben, ist gleich $O(2^{-n})$, also vernachlässigbar klein, obwohl der Algorithmus polynomielle Laufzeit hat.

Unter der Annahme, dass NP Sprachen enthält, die nicht Elemente von RP sind, muss es für jeden randomisierten Algorithmus eine Folge $(f_n)_{n\in\mathbb{N}}$ von Polynomen zweiten Grades mit Eingaben aus $\{0,1\}^n$ geben, so dass die Laufzeit zur Optimierung von f_n nur mit exponentiell kleiner Wahrscheinlichkeit polynomiell ist. Demzufolge muss es auch eine solche Folge geben, zu deren Optimierung der (1+1) EA mit exponentiell schnell gegen Eins konvergierender Wahrscheinlichkeit super-polynomielle Laufzeit benötigt. Eine solche Funktion wurde kürzlich in Wegener und Witt (2000) gefunden.

3.4.4 Analysen für unimodale Funktionen

Kommen wir zurück zu unserer Ausgangsfrage, welche weiteren Funktionsklassen nur Funktionen enthalten, die vom (1+1) EA in polynomieller erwarteter Zeit optimiert werden. Die anschauliche Begründung bei linearen Funktionen stützte sich darauf, dass in der Umgebung jedes Punkts Nachbarn bzgl. des Hamming-Abstands existieren, die einen höheren Funktionswert und einen geringeren Hamming-Abstand zum globalen Optimum besitzen. Dies muss bei quadratischen Funktionen nicht der Fall sein, wie uns die Funktion DISTANCE beispielhaft gezeigt hat: jeder Punkt $x \in \{0,1\}^n$ mit $||x||_1 > n/2$ hat nur Nachbarn, die mit einem besseren Funktionswert weiter vom globalen Optimum entfernt sind, den (1+1) EA also in die falsche Richtung lenken. Dass die positive Wahrscheinlichkeit, durch eine Mutation von der schlechten Hälfte in die gute Hälfte zu springen, die erwartete Laufzeit nicht polynomiell klein macht, haben wir in Theorem 3.4.12 gezeigt.

Die in unserer Überlegung verwendete Eigenschaft, dass es jeweils einen besseren Nachbarn gibt, der zum Optimum hinführt, erinnert natürlich an die Klasse der unimodalen Funktionen. Damit werden Funktionen $f : \mathbb{R}^n \mapsto \mathbb{R}$ bezeichnet, für die nur im globalen Optimum gilt, dass es in jeder ε-Umgebung ($\varepsilon > 0$) keinen besseren Punkt gibt. Anders ausgedrückt gibt es für alle nicht-optimalen Punkte jeweils Punkte mit beliebig kleinem Abstand, die einen höheren Funktionswert haben.

Wie dies auf Funktionen $f : \{0,1\}^n \mapsto \mathbb{R}$ übertragen werden kann, ist nicht von vorneherein klar. Da in Mühlenbein (1992) keine Definition angegeben ist, ist die dort aufgestellte Behauptung, dass der (1+1) EA alle unimodalen Funktionen in erwarteter Zeit $O(n\log(n))$ optimiert, weder beweis- noch widerlegbar. Wir werden im Folgenden eine naheliegende Definition von Unimodalität für Funktionen $f : \{0,1\}^n \mapsto \mathbb{R}$ vorstellen. Da die nächsten Nachbarn eines Punkts die mit Hamming-Abstand Eins sind, werden wir $f : \{0,1\}^n \mapsto \mathbb{R}$ unimodal nennen, wenn jeder Punkt außer dem einzigen globalen Optimum einen Nachbarn mit Hamming-Abstand Eins hat, der einen echt höheren Funktionswert hat, f also genau ein lokales Optimum besitzt:

Definition 3.4.13 *Sei $f : \{0,1\}^n \mapsto \mathbb{R}$. Ein Punkt $x \in \{0,1\}^n$ heißt* lokales Optimum *von f, wenn für alle Punkte $y \in \{0,1\}^n$ mit $H(x,y) = 1$ gilt, dass $f(y) < f(x)$ ist.*

Definition 3.4.14 *Sei $f : \{0,1\}^n \mapsto \mathbb{R}$. Die Funktion f heißt* unimodal*, wenn sie genau ein lokales Optimum besitzt.*

Das einzige lokale Optimum einer unimodalen Funktion ist zwangsläufig das globale Optimum von f. Eine unimodale Funktion ermöglicht dem (1+1) EA, sich von jedem Punkt aus über eine Folge von 1-Bit-Mutationen zum Optimum zu bewegen. Diese Argumentation führt mithilfe von Lemma 3.3.3 zu folgender oberen Schranke der erwarteten Laufzeit:

Lemma 3.4.15 *Sei $f : \{0,1\}^n \mapsto \mathbb{R}$ eine unimodale Funktion und $n^f \in \{1,\ldots,2^n\}$ die Zahl der verschiedenen Funktionswerte von f. Dann ist die erwartete Laufzeit* $\mathrm{E}(T^f)$ *des* (1+1) EA *auf f höchstens gleich* $\exp(1) \cdot (n^f - 1) \cdot n$.

Beweis: Da die Funktion f unimodal ist, kann man die Menge $\{0,1\}^n$ in die Ebenen $Z_1,\ldots,Z_{n^f} \subseteq \{0,1\}^n$ zerlegen, wobei Z_i ($i \in \{1,\ldots,n^f\}$) alle Punkte des $\{0,1\}^n$ enthält, die von f auf das i-kleinste Element von $\{f(x) \,|\, x \in \{0,1\}^n\}$ abgebildet werden. Dann ist $p = 1/n \cdot (1 - 1/n)^{n-1} \geq \exp(1)/n$ eine untere Schranke der Wahrscheinlichkeit, von einer Ebene Z_i ($i \in \{1,\ldots,n^f - 1\}$) zu einer höheren Ebene zu wechseln, da es stets einen Hamming-Nachbarn mit größerem Funktionswert gibt, wenn der aktuelle Punkt nicht optimal ist. Also liefert Lemma 3.3.3 die gewünschte obere Schranke der erwarteten Laufzeit. □

Nun stellt sich für uns die Frage, ob die Eigenschaft der Unimodalität einer Funktion, die anschaulich nur „einfachen" Funktionen zugeschrieben wird (Mühlenbein (1992)), ausreichend ist, um eine erwartete Laufzeit des (1+1) EA von $O(n \log(n))$ bzw. $poly(n)$ auf allen unimodalen Funktionen zu garantieren. Wir werden diese Frage in zwei Schritten negativ beantworten. Im ersten Schritt wird eine Funktion LeadingOnes angegeben, die eine ähnlich einfach zu definierende Funktion wie OneMax oder BinVal ist und für deren Optimierung der (1+1) EA erwartete Zeit $\Theta(n^2)$ braucht. Selbst wenn dies größer als $O(n \log(n))$ ist, so zeigt es doch nicht, dass unimodale Funktionen nicht effizient vom (1+1) EA optimiert werden können. Dies wird dann anhand der zweiten von uns betrachteten unimodalen Funktion LongPath nachgewiesen, für die der (1+1) EA exponentielle erwartete Laufzeit benötigt.

Die Funktion LeadingOnes : $\{0,1\}^n \mapsto \mathbb{R}$ weist einem Bitstring $x \in \{0,1\}^n$ die Anzahl der führenden Einsen in x zu (Rudolph (1997)):

Definition 3.4.16 *Die Funktion* LeadingOnes : $\{0,1\}^n \mapsto \mathbb{R}$ *ist definiert gemäß*

$$\textsc{LeadingOnes}(x_1,\ldots,x_n) := \sum_{i=1}^{n} \prod_{j=1}^{i} x_j.$$

Dass diese Funktion unimodal ist, ist direkt einsichtig: jeder Punkt x, der nicht gleich dem globalen Optimum $(1,\ldots,1)$ ist, beinhaltet zwangsläufig eine Stelle $i \in$

$\{1, \ldots, n\}$ mit $x_i = 0$ und $x_1 = \cdots = x_{i-1} = 1$. Durch Ersetzung von x_i durch eine Eins wird der Wert von LeadingOnes gesteigert. Somit gibt es von jedem Punkt $x \in \{0,1\}^n$ einen *Pfad* der maximalen Länge $n -$ LeadingOnes(x) von direkt benachbarten Punkten, die zum globalen Optimum führen. Weil der Begriff eines Pfades im Weiteren noch öfters gebraucht wird, sei er hier formal definiert:

Definition 3.4.17 *Eine Folge $(x_1, \ldots, x_k)$ von Punkten aus der Menge $\{0,1\}^n$ heißt* Pfad *zu einer Funktion $f : \{0,1\}^n \mapsto \mathbb{R}$, wenn gilt*

$$\forall i \in \{1, \ldots, k-1\} : (H(x_i, x_{i+1}) = 1 \textit{ und } f(x_{i+1}) > f(x_i)) .$$

Dabei geht der Pfad von einem Punkt $x \in \{0,1\}^n$ aus, wenn $x_1 = x$ ist.

Informal haben wir schon bei der Diskussion linearer Funktionen als eine Begründung für die erwartete Laufzeit von $\Theta(n \log(n))$ des (1+1) EA benutzt, dass es von jedem Punkt aus einen Pfad zum globalen Optimum gibt. Dabei ist diese Begründung insofern ungenau, als dass es bei linearen Funktionen (deren Gewichte alle ungleich Null sind) von jedem Punkt $x \in \{0,1\}^n$ aus genau $(n - H(x, x^*))!$ verschiedene Pfade zum globalen Optimum x^* gibt. Dass dies einen Unterschied in der erwarteten Laufzeit zu dem Fall nur jeweils eines einzigen Pfads ausmachen kann, wird im Folgenden an der Funktion LeadingOnes bewiesen.

Lemma 3.4.18 *Die erwartete Laufzeit des* (1+1) EA *auf* LeadingOnes *beträgt $\Theta(n^2)$. Es gibt weiterhin Konstanten $c_2 > c_1 > 0$, so dass die Wahrscheinlichkeit, dass die Laufzeit zwischen $c_1 \cdot n^2$ und $c_2 \cdot n^2$ liegt, exponentiell schnell gegen Eins konvergiert.*

Beweis: Für den Beweis der oberen Schranke $\exp(1) \cdot n^2$ der erwarteten Laufzeit lässt sich Lemma 3.4.15 anwenden, da LeadingOnes genau $n + 1$ verschiedene Werte annehmen kann. Weil wir jedoch auch über die Wahrscheinlichkeit, mit der $O(n^2)$ Schritte benötigt werden, eine Aussage treffen wollen, müssen wir genauer argumentieren:

Dafür ist die oben erwähnte Beobachtung entscheidend, dass es für jeden Punkt $x \in \{0,1\}^n$ einen Pfad der Länge höchstens $n -$ LeadingOnes(x) zum globalen Optimum $(1, \ldots, 1)$ gibt. Die Wahrscheinlichkeit einer Mutation zu einem Punkt mit höherem LeadingOnes-Wert beträgt mindestens

$$\frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{1}{\exp(1) \cdot n}.$$

Die erwartete Wartezeit, einen Punkt zu erreichen, für den die obere Schranke der Pfadlänge zum Optimum echt kleiner ist, ist somit höchstens gleich $\exp(1) \cdot n$. Denn alle anderen hier nicht betrachteten Mutationen können den Wert von LeadingOnes natürlich nicht senken. Lässt man den (1+1) EA genau $2\exp(1) \cdot n^2$ Schritte laufen, so ergibt die Tschernoff-Ungleichung (A.9), dass die Wahrscheinlichkeit, dass die Anzahl Z der solchermaßen erfolgreichen Mutationen in diesem Zeitraum kleiner als n ist, höchstens gleich

$$\mathrm{P}(Z < n) = \mathrm{P}\left(Z < \left(1 - \frac{1}{2}\right) \cdot 2n\right) < \exp\left(-\frac{n}{4}\right)$$

ist. Also ist die Laufzeit des (1+1) EA auf LeadingOnes mit exponentiell schnell gegen Eins konvergierender Wahrscheinlichkeit höchstens gleich $2\exp(1) \cdot n^2$. Deshalb kann man c_2 als $2\exp(1)$ wählen.

Für die untere Schranke sei i die Zahl der führenden Einsen des aktuellen Strings $x \in \{0,1\}^n$, d. h. $x_1 = \cdots = x_i = 1$ und $x_{i+1} = 0$. Die nächste erfolgreiche Mutation darf x_1 bis x_i nicht mutieren, muss jedoch das *relevante* Bit x_{i+1} mutieren.

Was mit den restlichen Bits $(x_{i+2}, \ldots, x_n)$ geschieht, ist rein zufällig, da es den Funktionswert nicht senken kann. Da diese Bits zu Beginn gleichverteilt zufällig gesetzt werden und in allen erfolgreichen Mutationen zufällig mit einer für alle Stellen gleichen Wahrscheinlichkeit verändert werden, sind sie auch nach jeder erfolgreichen Mutation gleichverteilt zufällig gesetzt. Für eine schnelle Optimierung ist es förderlich, wenn für ein möglichst großes $k \in \{1, \ldots, n-(i+2)\}$ alle Bits x_{i+2} bis x_{i+2+k} gleich Eins sind. Solche Bits nennen wir *Trittbrettfahrer*. Da die restlichen Bits $x_{i+2}, \ldots, x_n$ gleichverteilt zufällig verteilt sind, ist es unwahrscheinlich, dass es allzu viele Trittbrettfahrer gibt.

Formal werden wir im Folgenden zeigen, dass die Wahrscheinlichkeit, in höchstens $n^2/6$ Schritten das Optimum zu erreichen, exponentiell klein ist. Dazu betrachten wir zwei Ereignisse, deren Wahrscheinlichkeiten als exponentiell klein nachgewiesen werden:

1. Es gibt in $n^2/6$ Schritten mindestens $n/3$ erfolgreiche Schritte.

2. Es gibt in höchstens $n/3$ erfolgreichen Schritten mindestens $2n/3$ Trittbrettfahrer.

Da eine Optimierung in höchstens $n^2/6$ erfolgreichen Schritten impliziert, dass eines dieser beiden Ereignisse eingetreten ist, kann ihre Wahrscheinlichkeit durch die Summe der Wahrscheinlichkeiten dieser Ereignisse nach oben abgeschätzt werden.

Dass es in $n^2/6$ Schritten mindestens $n/3$ erfolgreiche Schritte gibt, ist gleichbedeutend damit, dass eine binomial-verteilte Zufallsvariable mit Parametern $p < 1/n$ und $n^2/6$ (A.6) mindestens den Wert $n/3$ annimmt. Die Wahrscheinlichkeit hierfür kann mittels der Tschernoff-Ungleichung (A.9) durch $\exp(-n/18)$ nach oben abgeschätzt werden, ist also exponentiell klein.

Die Wahrscheinlichkeit des zweiten Ereignisses kann ebenfalls mittels der Tschernoff-Ungleichung (A.9) abgeschätzt werden: dazu sei $\{i_1, \ldots, i_{n/3}\}$ die Menge (bzw. eine Obermenge) der Indizes der relevanten Bits, die in den höchstens $n/3$ erfolgreichen Mutationen negiert wurden. Dann müssen mindestens die Bits mit Indizes aus $\{1, \ldots, n\} \setminus \{i_1, \ldots, i_{n/3}\}$ Trittbrettfahrer, d. h. zum Zeitpunkt einer erfolgreichen Mutation gleich Eins gewesen sein. Da diese jeweils gleichverteilt zufällig gesetzt sind, lässt sich die Wahrscheinlichkeit dieses Ereignisses durch die Wahrscheinlichkeit nach oben abschätzen, dass die Anzahl Z der Einsen eines gleichverteilt zufälligen Bitstrings der Länge n mindestens $2n/3$ ist. Mittels der Tschernoff-Ungleichung (A.9) erhält man dafür die obere Schranke

$$\mathrm{P}\left(Z > \frac{2n}{3}\right) = \mathrm{P}\left(Z > \left(1 + \frac{1}{3}\right) \cdot \frac{n}{2}\right) < \exp\left(-\frac{n}{54}\right).$$

Da auch dieser Wert exponentiell schnell gegen Null konvergiert, konvergiert die Wahrscheinlichkeit, mindestens $n^2/6$ Schritte zu benötigen, exponentiell schnell gegen Eins. Somit ist auch die letzte noch ausstehende Behauptung des Lemmas mit $c_1 = 1/6$ bewiesen. □

Damit ist klar, dass die Existenz eines Pfades der maximalen Länge n zum globalen Optimum von jedem Punkt des Suchraumes aus nicht notwendig eine erwartete Laufzeit von $O(n \log(n))$ impliziert, worauf ja auch schon Lemma 3.4.15 hinweist.

Jedoch liegt es nahe, sich zu fragen, ob es nicht für den (1+1) EA noch schwierigere unimodale Funktionen gibt, deren Pfade zum Optimum eine größere Länge haben. Denn während es bei linearen Funktionen zwangsläufig jeweils einen Pfad mit maximaler Länge n zum Optimum gibt, muss dies bei unimodalen Funktionen nicht der Fall sein. Um eine Funktion zu konstruieren, die von möglichst vielen Punkten aus einen sehr langen Pfad zum Optimum hat, sind die so genannten Langpfade hilfreich (Horn, Goldberg und Deb (1994)):

Definition 3.4.19 *Seien* $n, k \in \mathbb{N}$ *mit* $(n-1)/k \in \mathbb{N}_0$. *Dann ist der* k-Langpfad P_n^k der Dimension n *ein folgendermaßen rekursiv definierter Pfad im* $\{0,1\}^n$*:*

1. *Der* k*-Langpfad* P_1^k *der Dimension 1 ist gleich* $(0,1)$.
2. *Der* k*-Langpfad* P_{n-k}^k *der Dimension* $n-k$ *sei* $(v_1, \ldots, v_l)$. *Dann ist der* k*-Langpfad* P_n^k *der Dimension* n *definiert als*

$$P_n^k := (0^k v_1, \ldots, 0^k v_l, 0^{k-1}1v_l, 0^{k-2}11v_l, \ldots, 01^{k-1}v_l, 1^k v_l, \ldots, 1^k v_1).$$

Dass ein nach dieser Definition erzeugter k-Langpfad einen Pfad bildet, d. h. eine Folge von direkt benachbarten Punkten, ist leicht per Induktion zu sehen. Wie kann ein solcher k-Langpfad der Dimension n nun zu einer unimodalen Funktion „umgebaut" werden? Da wir erreichen wollen, dass der (1+1) EA den Punkten des Langpfades folgt, sollten seine Punkte einen höheren Funktionswert als alle anderen Punkte haben und die Funktionswerte zum Ende des Pfades hin ansteigen. Was ist mit den restlichen Punkten? Damit der (1+1) EA, auch wenn er in einem dieser Punkte initialisiert (was abhängig von der Länge des Langpfades recht wahrscheinlich sein kann), eine hohe erwartete Laufzeit hat, sollte die Funktion so beschaffen sein, dass Mutationen zu Punkten am Anfang des Pfades höhere Erfolgsaussichten haben als andere. Dies kann dadurch erreicht werden, dass Einsen in den ersten k Positionen zu einer höheren Wertverringerung führen als solche in den restlichen $n-k$ Positionen. Formalisiert man diese Ideen, ergibt sich folgende Definition:

Definition 3.4.20 *Seien* $n, k \in \mathbb{N}$ *mit* $(n-1)/k \in \mathbb{N}_0$. *Die Funktion* $\textsc{LongPath}_k : \{0,1\}^n \mapsto \mathbb{R}$ *ist folgendermaßen definiert:*

$$\textsc{LongPath}_k(x) := \begin{cases} n^2 + l & \text{, falls } x \text{ der } l\text{-te Punkt von } P_n^k \text{ ist,} \\ n^2 - n \cdot \sum\limits_{i=1}^{k} x_i - \sum\limits_{i=k+1}^{n} x_i & \text{, falls } x \notin P_n^k \text{ ist.} \end{cases}$$

Es ist klar, dass die so definierte Funktion unimodal ist: für die Punkte $x \in P_n^k$ gilt dies, da P_n^k ein Pfad ist und der nächste Punkt auf dem Pfad einen höheren Funktionswert hat. Liegt $x \in \{0,1\}^n$ nicht auf dem Pfad, so hat jeder Nachbarpunkt mit genau einer Eins weniger einen höheren Funktionswert. Somit kann man über eine Folge von direkten Nachbarn von jedem Punkt außerhalb des Pfades aus nur mit Steigerungen des Funktionswerts den Punkt $(0, \ldots, 0)$ erreichen, der Anfangspunkt des Pfades ist. Also ist $\textsc{LongPath}_k$ eine unimodale Funktion.

Erste Versuche, mittels Langpfaden Funktionen zu konstruieren, die eine exponentiell große Laufzeit des (1+1) EA erzwingen, wurden in Horn, Goldberg und Deb (1994) durchgeführt. Obwohl die empirisch beobachteten Laufzeiten von den Autoren als exponentiell gedeutet wurden, konnte Rudolph (1997) zeigen, dass die erwartete Laufzeit nur $O(n^3)$ ist. Denn in den benutzten Pfaden gibt es „Abkürzungen", d. h. schon durch die Mutation von nur zwei Bits kann man in dem Pfad sehr weit nach vorne kommen, so dass die Vorstellung, der (1+1) EA läuft den Pfad Punkt für Punkt ab und benötigt so exponentielle Laufzeit, nicht zutrifft. Wir werden im Folgenden zeigen, dass mit steigendem Parameter k die Wahrscheinlichkeit solcher Abkürzungen abnimmt, da dazu mindestens k Bits mutieren müssen. Andererseits wird mit steigendem k die Länge des Pfades sinken:

Lemma 3.4.21 *Der* k*-Langpfad* P_n^k *der Dimension* n *hat Länge* $(k+1) \cdot 2^{(n-1)/k} - k + 1$.

Beweis: Wir zeigen dies induktiv über n. Für $n = 1$ ist die Länge des k-Pfades der Dimension n nach Definition gleich Zwei, was auch obiger Ausdruck ergibt.

Betrachten wir einen k-Pfad der Dimension $n > 1$, so setzt sich dieser aus zwei k-Pfaden der Dimension $n-k$ und einem Mittelstück der Länge $k-1$ zusammen. Nach Induktionsvoraussetzung gilt also:

$$\begin{aligned} |P_n^k| &= 2 \cdot |P_{n-k}^k| + k - 1 \\ &= 2 \cdot \Big((k+1) \cdot 2^{(n-k-1)/k} - k + 1 \Big) + k - 1 \\ &= (k+1) \cdot 2^{(n-1)/k} - k + 1. \end{aligned}$$

□

Damit sehen wir, dass für alle Werte $k = O(n^c)$ mit konstantem $c < 1$ die Länge von P_n^k exponentiell in n ist. Wächst k hingegen mit der Größenordnung $\Omega(n/\log(n))$, so ist die Länge des k-Langpfades nur polynomiell groß in n. Für Werte zwischen diesen Extremen ergibt sich eine superpolynomielle, aber subexponentielle Länge.

Um eine möglichst große Länge des k-Langpfades zu erreichen, sollte k also möglichst klein gewählt werden. Dies hätte jedoch nicht zwangsläufig die gewünschte große erwartete Laufzeit des (1+1) EA auf LongPath$_k$ zur Folge, da dann die schon besprochenen Abkürzungen auftreten können:

Lemma 3.4.22 *Seien $n, k \in \mathbb{N}$, wobei $(n-1)/k \in \mathbb{N}_0$ ist, und P_n^k der k-Langpfad der Dimension n. Dann gilt für alle $x, y \in P_n^k$, wobei y der i-te Nachfolger von x in P_n^k ist, dass für $i \leq k-1$ der Hamming-Abstand von x und y gleich i ist und für $i \geq k$ mindestens k.*

Beweis: Wir beweisen die Aussage durch Induktion über n. Ist $n = 1$, so ist der einzig mögliche Wert von i Null oder Eins; in beiden Fällen ist die Aussage wahr.

Ist $n > 1$, so sei x der p-te Punkt auf P_k^n. Wir unterscheiden nun nach den Positionen von x und y, d. h. den Werten von p und $p+i$:

1. Falls $p, p+i \leq |P_{n-k}^k|$, so ist x bzw. y das p-te bzw. das $(p+i)$-te Element von P_{n-k}^k jeweils um k Nullen zu Beginn erweitert. Da diese für den Hamming-Abstand keine Rolle spielen, überträgt sich die Induktionsvoraussetzung auf diesen Fall.

   Aus Symmetriegründen gilt dies auch, falls $p, p+i \geq |P_n^k| - (|P_{n-k}^k| + k - 1)$ sind, d. h. x und y aus der zweiten Kopie von P_{n-k}^k stammen, da sich die Reihenfolge der Punkte genau herumdreht und diese mit Einsen statt Nullen erweitert sind.

2. Falls $p \leq |P_{n-k}^k|$ und $p+i \in \{|P_{n-k}^k|+1, \ldots, |P_{n-k}^k|+k-1\}$, so gilt folgendes: Ist $i \leq k-1$, so ist der Hamming-Abstand von x zum $|P_{n-k}^k|$-ten Punkt des Pfades nach Fall 1 gleich $|P_{n-k}^k| - p$. Da dieser von y einen Hamming-Abstand von $p+i-|P_{n-k}^k|$ hat, sich die ersten beiden Punkte nur auf den letzten $n-k$ und die letzten beiden nur auf den ersten k Positionen unterscheiden, ist der Hamming-Abstand von x und y gleich i.

   Ist $i \geq k$, so unterscheiden wir nach dem Wert von $|P_{n-k}^k| - p$: falls er mindestens k ist, so ist schon der Hamming-Abstand von x zu dem $|P_{n-k}^k|$-ten Punkt des Pfades nach Induktionsvoraussetzung mindestens k. Da sich dieser von y nur auf den ersten k Stellen unterscheidet, auf denen er aber mit x übereinstimmt, muss der Hamming-Abstand von x und y mindestens k sein.

   Ist $|P_{n-k}^k| - p < k$, so ist der Hamming-Abstand von x zu dem $|P_{n-k}^k|$-ten Punkt nach Induktionsvoraussetzung auf den letzten $n-k$ Stellen gleich $|P_{n-k}^k| - p$. Da der Hamming-Abstand des letzteren zu y auf den ersten k Stellen gleich $p+i-|P_{n-k}^k|$ ist, ist der zwischen x und y gleich $i \geq k$.

Aus Symmetriegründen gelten diese Überlegungen auch, falls $p \in \{|P^k_{n-k}| + 1, \ldots, |P^k_{n-k}| + k - 1\}$ und $p + i \geq |P^k_{n-k}| + k$ ist.

3. Falls $p \leq |P^k_{n-k}|$ und $p + i \geq |P^k_{n-k}| + k$, so ist $i \geq k$ und der Hamming-Abstand zwischen x und y ist mindestens k, da die beiden Strings an den ersten k Stellen genau komplementär sind.

□

Somit wissen wir, dass eine Mutation von $i \leq k-1$ Bits nur zu einem Punkt auf dem Pfad führen kann, der auf dem k-Langpfad genau i Stellen weiter vorne steht. Eine Mutation von mindestens k Bits kann auf dem Pfad beliebig weit nach vorne führen. Wenn wir eine Abkürzung als eine Mutation von i Bits definieren, die zu einem Punkt führt, der auf dem Pfad mehr als i Punkte weiter vorne liegt, so müssen zu einer Abkürzung also mindestens k Bits mutieren. Dies hat Wahrscheinlichkeit

$$\binom{n}{k} \cdot \left(\frac{1}{n}\right)^k \leq \frac{1}{k!}.$$

Wenn k konstant in n ist, so wird auch die Wahrscheinlichkeit einer Abkürzung konstant gross sein. Da für Werte von k in der Größenordnung von $\Omega(n/\log(n))$, die Länge des Pfades nur polynomiell in n ist, sollte der Wert von k, den wir wählen, zwischen diesen beiden Größenordnungen liegen. Im Folgenden werden wir zeigen, dass für $k = \sqrt{n-1}$ (wobei wir o. B. d. A. davon ausgehen, dass $n-1$ eine Quadratzahl ist) die erwartete Laufzeit des (1+1) EA auf LongPath_k exponentiell ist.

Exponentielle obere Schranken wurden schon in Rudolph (1997) bewiesen. Auch wenn zum Nachweis der exponentiellen Laufzeit natürlich nur eine untere Schranke notwendig ist, seien diese ausgeführt, um die Argumentation über LongPath_k einzuüben:

Lemma 3.4.23 *Die erwartete Laufzeit des* (1+1) EA *auf* $\text{LongPath}_k : \{0,1\}^n \mapsto \mathbb{R}$ *ist* $O(\min\{n^{k+1}/k, n \cdot |P^k_n|\} + n\log(n))$.

Beweis: Der Laufzeit des (1+1) EA auf LongPath_k wird in zwei Teile aufgeteilt: die Zeit T_1 bis zum ersten Erreichen des Pfades und die Zeit T_2, um von diesem Zeitpunkt an das Optimum $(1, \ldots, 1)$ zu erreichen. Um $\mathrm{E}(T^{\text{LongPath}_k})$ abzuschätzen, werden nun $\mathrm{E}(T_1)$ und $\mathrm{E}(T_2)$ nach oben abgeschätzt.

Initialisiert der (1+1) EA in einem Punkt auf dem Pfad, so ist T_1 natürlich gleich Null. Anderenfalls ist die Beobachtung wichtig, dass sich LongPath_k auf den Punkten, die nicht zum Pfad gehören, wie eine lineare Funktion mit Koeffizienten $-n$ bzw. -1 verhält, deren Optimum $(0, \ldots, 0)$ der erste Punkt des Pfades ist. Somit ist die erwartete Laufzeit $O(n\log(n))$ auf einer linearen Funktion eine obere Schranke für $\mathrm{E}(T_1)$.

Um eine obere Abschätzung für $\mathrm{E}(T_2)$ zu erhalten, können wir Lemma 3.4.15 benutzen, da der (1+1) EA, wenn er einmal den Pfad erreicht hat, diesen nicht mehr verlässt. Somit ist $\mathrm{E}(T_2)$ gleich $O(n \cdot |P^k_n|)$.

Eine zweite Möglichkeit besteht darin, in die Ebeneneinteilung des Pfades gemäß Lemma 3.3.3 nicht nur Ebenen mit jeweils nur einem Punkt aufzunehmen, sondern den Pfad folgendermaßen rekursiv in Ebenen aufzuteilen: die ersten $|P^k_{n-k}| + k - 1$ Punkte des Pfades werden zu Z_1, die letzten $|P^k_{n-k}|$ Punkte werden rekursiv nach demselben Verfahren in die Ebenen $Z_2, \ldots, Z_r$ aufgeteilt. Da dieses Verfahren nach $(n-1)/k$ Teilungen mit den Punkten $((1, \ldots, 1, 0), (1, \ldots, 1, 1))$ endet, die in zwei Ebenen aufgespalten werden, ergibt sich so $r = (n-1)/k + 2$.

Zur Bestimmung einer unteren Schranke p^* der Wahrscheinlichkeit, von einer Ebene in eine höhere zu wechseln, ist die Beobachtung entscheidend, dass ein Punkt aus Z_i durch Mutation von höchstens k Bits zu einem Punkt einer höheren Ebene werden kann. Denn gehört er zu den ersten $|P^k_{n-k}|$ Punkten, so wird er durch Komplementierung der ersten k Bits zu einem Punkt der letzten $|P^k_{n-k}|$ Punkte; gehört er zu den mittleren $k-1$ Punkten, so müssen seine ersten k Bits alle gleich Eins sein, damit er gleich dem $|P^k_{n-k}|$-letzten Punkt wird. Diese Argumentation gilt für den ursprünglichen k-Langpfad P^k_n der Dimension n, für niedrigere Dimensionen $n-k$, $n-2k$, usw. überträgt sie sich aber exakt.

Da die Wahrscheinlichkeit einer Mutation von k vorgegebenen Bits gleich $(1/n)^k \cdot (1-1/n)^{n-k} \geq \exp(-1)/n^k$ ist, ergibt sich somit eine obere Schranke von $O(n^{k+1}/k)$ für $\mathrm{E}(T_2)$. Zusammen mit der oberen Schranke für $\mathrm{E}(T_1)$ ist der Beweis beendet. $\Box$

Für $k = \sqrt{n-1}$ ergibt sich somit eine exponentielle obere Schranke von $O(n^{3/2} \cdot 2^{\sqrt{n}})$. Dies lässt die Möglichkeit einer exponentiellen Laufzeit offen. Der Beweis, dass der (1+1) EA zur Optimierung von $\text{LongPath}_{\sqrt{n-1}}$ exponentielle Zeit braucht, wird mit folgendem Theorem gegeben:

Theorem 3.4.24 *Die erwartete Laufzeit des* (1+1) EA *auf* LongPath_k *für* $k = \sqrt{n-1}$ *ist* $\Theta(n^{3/2} \cdot 2^{\sqrt{n}})$.

Beweis: Die obere Schranke wurde in Lemma 3.4.23 gezeigt. Für die untere Schranke betrachten wir nun nur die Zeit zwischen dem erstmaligen Erreichen des Pfades und dem Erreichen des Optimums. Dazu sei mit $x^* \in \{0,1\}^n$ der Punkt bezeichnet, den der (1+1) EA als ersten Punkt des k-Langpfades P^k_n erreicht.

In einem ersten Schritt zeigen wir, dass mit einer Wahrscheinlichkeit von mindestens $1/4$ der Punkt x^* zu den ersten $|P^k_{n-k}| + k - 1$ Punkten des Pfades gehört. Denn mit einer Wahrscheinlichkeit von mindestens $1/2$ wird in einem Punkt initialisiert, der in den ersten k Stellen mindestens $k/2$ Nullen hat (k sei o. B. d. A. gerade). Jede Mutation von einem solchen Punkt aus, die nicht zu einem Punkt auf dem Langpfad P^k_n führt, wird die Anzahl der Einsen innerhalb der ersten k Stellen nicht steigern. Um also bei einer Initialisierung mit mindestens $k/2$ Nullen in den ersten k Stellen als ersten Punkt x^* des Pfades einen seiner $|P^k_{n-k}|$ letzten Punkte zu erreichen, müssen in einer Mutation mindestens $k/2$ der k vorderen Stellen zu Einsen werden und die restlichen Stellen zu einem Punkt des Pfades P^k_{n-k} mutiert werden. Da die ersten $|P^k_{n-k}|$ Punkte des Pfades in diesen restlichen $n-k$ Stellen wie die letzten $|P^k_{n-k}|$ Punkte des Pfades aufgebaut sind, ist die Wahrscheinlichkeit, dass x^* in den ersten $|P^k_{n-k}|$ Punkten des Pfades liegt, mindestens so hoch wie für die letzten $|P^k_{n-k}|$ Stellen. Also gehört x^* mindestens mit Wahrscheinlichkeit $1/4$ zu den ersten $|P^k_{n-k}| + k - 1$ Punkten des Pfades.

(Da $k = \sqrt{n-1}$ ist, ist die Wahrscheinlichkeit einer Mutation von mindestens $k/2$ Bits nach der Poisson-Verteilung (A.7) ungefähr $1/(\exp(1) \cdot (\sqrt{n-1})!)$. Eine genauere Betrachtung zeigt deshalb, dass mit gegen Eins konvergierender Wahrscheinlichkeit der Pfad erreicht wird, bevor eine solche Mutation eintritt, wenn entsprechend initialisiert wurde. Deshalb ist die Wahrscheinlichkeit, dass x^* zu den ersten $|P^k_{n-k}|+k-1$ Punkten des Pfades gehört, sogar $1/2-o(1)$. Für unsere Zwecke reicht aber obige Abschätzung aus.)

Für den Rest des Beweises ist es ausreichend zu zeigen, dass die erwartete Laufzeit bei Initialisierung in einem der ersten $|P^k_{n-k}| + k - 1$ Punkte des k-Langpfades gleich $\Omega(n \cdot |P^k_{n-k}|)$ ist. Dies wäre erreicht, wenn wir zeigen würden, dass der erwartete Fortschritt, den der (1+1) EA auf dem k-Langpfad bei einer erfolgreichen Mutation macht, von der Größenordnung $O(1/n)$ ist. Zwar können wir dann nicht die Wald'sche Gleichung verwenden (da der Fortschritt ja auch größer als Eins sein kann), doch die Markoff-Ungleichung (A.8) kann hier helfen: denn der Fortschritt, der mindestens zum Erreichen des Optimums notwendig ist, beträgt

$$|P^k_{n-k}| \geq k \cdot 2^{(n-k-1)/k} = \Omega(\sqrt{n} \cdot 2^{\sqrt{n}}).$$

Sei mit $F(t)$ ($t \in \mathbb{N}$) die Zufallsvariable bezeichnet, die den insgesamt erreichten Fortschritt auf dem Pfad nach t Schritten angibt. Da der erwartete Fortschritt einer erfolgreichen Mutation nach Annahme für ein konstantes $c_1 > 0$ höchstens gleich c_1/n ist, ist aufgrund der Linearität des Erwartungswerts für jede Konstante $c_2 > 0$:

$$\mathrm{E}\left(F\left(\frac{c_2}{4 \cdot c_1} \cdot n^{3/2} \cdot 2^{\sqrt{n}}\right)\right) \le \frac{c_2}{4} \cdot \sqrt{n} \cdot 2^{\sqrt{n}}.$$

Somit folgt mit der Markoff-Ungleichung (A.8), dass

$$\mathrm{P}\left(F\left(\frac{c_2}{4 \cdot c_1} \cdot n^{3/2} \cdot 2^{\sqrt{n}}\right) \ge \frac{c_2}{2} \cdot \sqrt{n} \cdot 2^{\sqrt{n}}\right) \le \frac{1}{2}$$

ist. Also ist für ein konstantes $c > 0$ die Wahrscheinlichkeit, dass der (1+1) EA nach $c \cdot n^{3/2} \cdot 2^{\sqrt{n}}$ Schritten kein Optimum erreicht hat, mindestens $1/2$. Daraus folgt die untere Schranke von $\Omega(n^{3/2} \cdot 2^{\sqrt{n}})$ der erwarteten Laufzeit.

Kommen wir zur fehlenden Abschätzung des erwarteten Fortschritts in einem erfolgreichen Schritt. Nach Lemma 3.4.22 führt jeweils genau eine Mutation von $i \le k-1$ Bits genau i Punkte auf dem Langpfad weiter, was eine Wahrscheinlichkeit von $(1/n)^i \cdot (1-1/n)^{n-i} \le 1/n^i$ hat. Mutieren jedoch mindestens k Bits, so kann der Fortschritt nur durch $|P_n^k|$ nach oben abgeschätzt werden. Damit ergibt sich folgende obere Abschätzung für den erwarteten Fortschritt in einem Schritt:

$$\begin{aligned}
& |P_n^k| \cdot \frac{1}{k!} + \sum_{i=1}^{k-1} i \cdot \frac{1}{n^i} \\
< \quad & |P_n^k| \cdot \frac{1}{k!} + \sum_{i=1}^{\infty} \sum_{j=i}^{\infty} \frac{1}{n^j} \\
= \quad & |P_n^k| \cdot \frac{1}{k!} + \sum_{i=1}^{\infty} \left(\frac{1}{1-1/n} - \frac{1-(1/n)^i}{1-1/n}\right) \\
= \quad & |P_n^k| \cdot \frac{1}{k!} + \sum_{i=1}^{\infty} \left(\frac{n}{n-1} \cdot \frac{1}{n^i}\right) \\
= \quad & |P_n^k| \cdot \frac{1}{k!} + \sum_{i=1}^{\infty} \left(\frac{1}{n^{i-1} \cdot (n-1)}\right) \\
= \quad & |P_n^k| \cdot \frac{1}{k!} + \frac{1}{n-1} \cdot \frac{1}{1-1/n} \\
\le \quad & (k+1) \cdot 2^{(n-1)/k} \cdot \frac{1}{k!} + \frac{n}{(n-1)^2}.
\end{aligned}$$

Da für $k = \sqrt{n-1}$ nach der Stirling'schen Formel (A.10) gilt:

$$k! = \left(\sqrt{n-1}\right)! > \left(\frac{\sqrt{n-1}}{\exp(1)}\right)^{\sqrt{n-1}} = 2^{\Omega(\sqrt{n} \cdot \log(n))},$$

kann man den erwarteten Fortschritt folgendermaßen nach oben abschätzen:

$$2^{-\Omega(\sqrt{n} \cdot \log(n))} + \frac{n}{(n-1)^2} = O(1/n).$$

□

Damit haben wir gesehen, dass auch unimodale Funktionen für den (1+1) EA schwierig sein können, so dass Unimodalität keine hinreichende Bedingung für eine polynomielle Laufzeit des (1+1) EA ist.

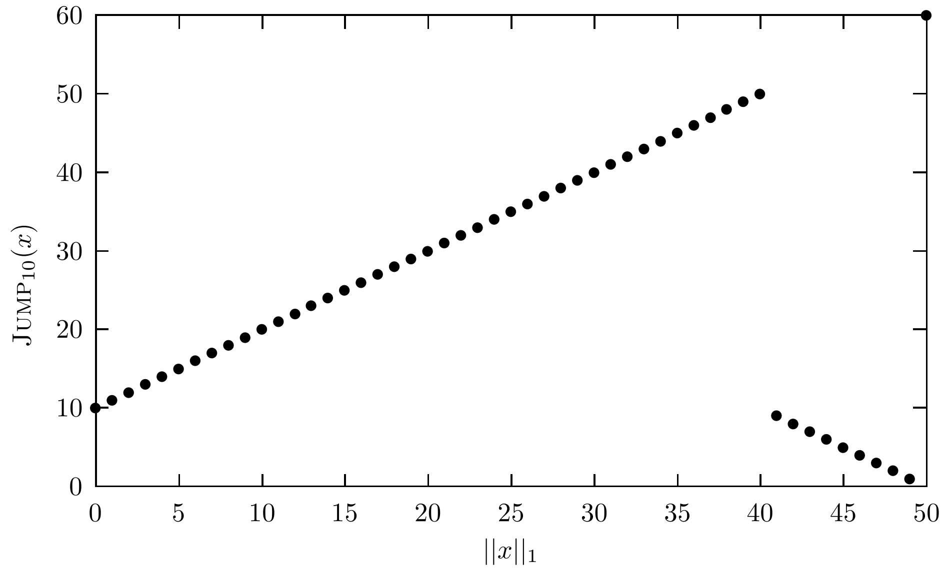

Abbildung 3.3: Die Funktion Jump_m für $n = 50$ und $m = 10$.

3.4.5 Eine Hierarchie von Funktionen

Bei den bisher untersuchten Funktionen waren die sich ergebenden erwarteten Laufzeiten entweder von der Größenordnung Polynome kleinen Grades ($\Theta(n \log(n))$ für OneMax, BinVal und alle linearen Funktionen; $\Theta(n^2)$ für LeadingOnes) oder von exponentieller Größenordnung (Trap, Distance oder $\text{LongPath}_{\sqrt{n-1}}$).

Im Folgenden wollen wir eine ganze Funktionenfolge $\text{Jump}_m : \{0,1\}^n \mapsto \mathbb{R}$ ($m \in \{1, \ldots, n\}$) vorstellen, so dass der (1+1) EA zu ihrer Optimierung erwartete Laufzeit $\Theta(n^m + n \log(n))$ benötigt. (Obwohl schon zuvor erwähnt, soll hier noch einmal ausdrücklich darauf hingewiesen werden, dass es sich somit um eine Folge von Funktionsfolgen handelt, da jede Funktion $\text{Jump}_m : \{0,1\}^n \mapsto \mathbb{R}$ für jeden Wert von $n \in \mathbb{N}$ eine Ausprägung hat.)

Diese Funktionsfolge wird zeigen, dass die Mengen

$$F_m := \{f : \{0,1\}^n \mapsto \mathbb{R} \,|\, \mathrm{E}(T^f) = O(n^m + n \log(n))\}$$

für aufsteigende Werte $m \in \{1, \ldots, n\}$ stets echt größer werden, da $\text{Jump}_m \in M_m$, aber $\text{Jump}_m \notin M_{m-1}$ ist. Somit wissen wir dann, dass diese Klassen eine Hierarchie bilden. Auch wird der Konstruktion der Funktionen Jump_m eine einfache Idee zugrundeliegen, die den (1+1) EA zu einer bestimmten erwarteten Laufzeit zwingt. Die Definition der Funktionen Jump_m ist folgendermaßen:

Definition 3.4.25 *Sei* $m \in \{1, \ldots, n\}$. *Die Funktion* $\text{Jump}_m : \{0,1\}^n \mapsto \mathbb{R}$ *ist definiert als*

$$\text{Jump}_m(x) := \begin{cases} m + \sum_{i=1}^n x_i & \textit{, falls } \sum_{i=1}^n x_i \in \{1, \ldots, n-m, n\}, \\ n - \sum_{i=1}^n x_i & \textit{, sonst.} \end{cases}$$

In Abbildung 3.3 ist die Funktion Jump_m für $n = 50$ und $m = 10$ veranschaulicht. Die Funktionen Jump_m sind symmetrisch, weshalb wir uns bei der Argumentation nur auf die Anzahl der Einsen in x, aber nicht deren konkrete Position beziehen müssen. Alle Jump_m-Funktionen haben ihr Optimum stets beim Einsstring $(1, \ldots, 1)$ mit Funktionswert $m + n$. Schränkt man den Definitionsbereich auf Punkte $x \in \{0,1\}^n$ mit höchstens $n - m$ Einsen ein, so ist der Funktionswert $\text{JUMP}_m(x)$ stets gleich $\text{ONEMAX}(x) + m$. Da eine additive Konstante das Verhalten des (1+1) EA nicht ändert (Lemma 3.1.3), ist die erwartete Laufzeit des (1+1) EA bis zum Erreichen eines Strings mit $n - m$ Einsen gleich $O(n \log(n))$, wenn er nicht zuvor das globale Optimum erreicht.

Da alle Strings, die mehr als $n - m$ Einsen haben, aber nicht gleich dem Optimum sind, einen geringeren Funktionswert als alle anderen Punkte haben und mit abnehmender Zahl von Einsen höheren Funktionswert haben, ist eine langsame Annäherung an das Optimum mittels mehrerer kleiner Mutationen nicht möglich. Deshalb ist eine Mutation aller m Nullen notwendig, um den Sprung von $n - m$ zu n Einsen durchzuführen. Da die Wahrscheinlichkeit hierfür höchstens n^{-m} ist, beträgt die erwartete Wartezeit mindestens n^m.

Dass diese anschauliche Argumentation auch formal bewiesen werden kann, zeigt folgendes Theorem:

Theorem 3.4.26 *Sei $n \in \mathbb{N}$ und $m \in \{1, \ldots, n\}$. Dann ist die erwartete Laufzeit des* (1+1) EA *auf* JUMP_m *gleich* $\Theta(n^m + n \log(n))$.

Beweis: Für $m = 1$ ist JUMP_m gleich $\text{ONEMAX} + 1$. Also folgt in diesem Fall die Aussage aus den Lemmata 3.1.3 und 3.4.2.

Sei $m \in \{2, \ldots, n\}$. Wir teilen den Suchraum $\{0,1\}^n$ gemäß Definition 3.3.2 in die Ebenen A_1, A_2 und A_3 ein, die als

$$\begin{aligned} A_1 &:= \{x \in \{0,1\}^n \mid n - m < \sum_{i=1}^{n} x_i < n\}, \\ A_2 &:= \{x \in \{0,1\}^n \mid \sum_{i=1}^{n} x_i \leq n - m\} \text{ und} \\ A_3 &:= \{(1, \ldots, 1)\} \end{aligned}$$

definiert sind. Dann gilt, dass der (1+1) EA aus einem Punkt in A_i nur zu Bitstrings in A_j mit $j \geq i$ mutieren kann. Jedoch werden wir nicht Lemma 3.3.3 anwenden, um die obere Schranke der erwarteten Laufzeit von $O(n^m + n \log(n))$ zu beweisen.

Wir nehmen zum Beweis der oberen Schranke an, dass wir in einem Punkt in A_1 initialisieren und von da aus A_2 erreichen. Würde der (1+1) EA die Menge A_3 vor A_2 erreichen, so würde die erwartete Laufzeit insgesamt nur sinken. Da JUMP_m eingeschränkt auf A_1 für den (1+1) EA bis auf Mutationen zu $(1, \ldots, 1)$ äquivalent zu $-\text{ONEMAX}$ ist, ist die erwartete Zeit bis zum Erreichen eines Bitstrings von A_2 gleich $O(n \log(n))$ (Lemma 3.4.2).

Um die erwartete Zeit, von A_2 aus A_3 zu erreichen, nach oben abzuschätzen, benutzen wir wieder die Äquivalenz von JUMP_m auf A_2 mit ONEMAX bis auf Mutationen zu $A_1 \cup A_3$. Also ist die erwartete Zeit, bis ein Punkt mit $n - m$ oder n Einsen erreicht wird, gleich $O(n \log(n))$. Wenn ein solches Optimum, ein Punkt mit $n - m$ Einsen, erreicht wurde, ist die Wahrscheinlichkeit einer Mutation zum globalen Optimum $(1, \ldots, 1)$ gleich

$$\left(\frac{1}{n}\right)^m \cdot \left(1 - \frac{1}{n}\right)^{n-m} \geq \frac{1}{\exp(1) \cdot n^m}.$$

Demzufolge ist die erwartete Zeit, bis von einem Punkt mit $n - m$ Einsen aus das Optimum $(1, \ldots, 1)$ erreicht wird, gleich $O(n^m)$. Also ist die erwartete Laufzeit des (1+1) EA auf JUMP_m gleich $O(n^m + n \log(n))$.

Zum Beweis der unteren Schranke betrachten wir zuerst den Fall, dass der (1+1) EA in einem Punkt aus A_2 ist. Da von A_2 aus zu keinem Punkt aus A_1

mutiert werden kann, müssen zum Erreichen des Optimums genau alle Nullen in einem Schritt mutieren. Weil in A_2 die Anzahl der Nullen mindestens m ist, hat eine solche Mutation eine Wahrscheinlichkeit von höchstens

$$\left(\frac{1}{n}\right)^m \cdot \left(1 - \frac{1}{n}\right)^{n-m} \leq \frac{1}{n^m}.$$

Also ist die erwartete Zeit, um von A_2 aus das globale Optimum zu erreichen, mindestens n^m. Nun zeigen wir, dass die Wahrscheinlichkeit des (1+1) EA, einen Punkt von A_2 zu erreichen, mindestens gleich einer Konstanten $\varepsilon > 0$ ist.

Dazu betrachten wir den Fall, dass in einem Punkt mit höchstens $n/2$ Einsen (n ist o. B. d. A. gerade) initialisiert wird. Wenn $m \leq n/2$ ist, so folgt daraus, dass der (1+1) EA in einem Punkt aus A_2 initialisiert hat. Für diesen Fall haben wir die untere Schranke von $\Omega(n^m)$ gerade nachgewiesen. Sei also $m > n/2$ und nehmen wir an, dass die Anzahl i der Einsen im initial gewählten Punkt zwischen $n-m+1$ und $n/2$ liegt (ansonsten liegt er wiederum in A_2). Wir zeigen, dass der (1+1) EA in den ersten n^2 Schritten mit gegen Eins konvergierender Wahrscheinlichkeit einen Punkt aus A_2 erreicht.

Entscheidend ist hierfür, dass es nur zwei Möglichkeiten gibt, dieses zu vermeiden: entweder wird direkt zu dem globalen Optimum $(1, \ldots, 1)$ mutiert oder der (1+1) EA bleibt in dem Bereich der Strings mit $i \in \{n-m+1, \ldots, n/2\}$ Einsen. Da ersteres eine Wahrscheinlichkeit von höchstens $(1/n)^{n/2}$ hat, geschieht es in den n^2 betrachteten Schritten mit einer Wahrscheinlichkeit von höchstens $n^2/n^{n/2}$, was exponentiell schnell gegen Null konvergiert. Dass der (1+1) EA die Menge A_1 in diesem Zeitraum nicht verlässt, hat ebenfalls gegen Null konvergierende Wahrscheinlichkeit. Denn die erwartete Laufzeit des (1+1) EA auf JUMP_m auf der Menge A_1 bis zum Erreichen eines Punkts $A_2 \cup A_3$ lässt sich durch die erwartete Laufzeit des (1+1) EA auf ONEMAX nach oben abschätzen, ist also gleich $O(n \log(n))$ (Lemma 3.4.2). Somit ist nach der Markoff-Ungleichung (A.8) die Wahrscheinlichkeit, dass A_2 in n^2 Schritten nicht erreicht wird, höchstens gleich $O(n \log(n)/n^2)$, was ebenfalls gegen Null konvergiert.

Also erreicht der (1+1) EA mit gegen Eins konvergierender Wahrscheinlichkeit in den ersten n^2 Schritten einen Punkt der Menge A_2, weshalb seine erwartete Laufzeit gleich $\Theta(n^m + n \log(n))$ ist. □

Die Funktionen JUMP_m zeigen deutlich, dass lokale Optima, die vom globalen Optimum einen Hamming-Abstand von m haben, den (1+1) EA zu der erwarteten Laufzeit $\Theta(n^m)$ „zwingen“ können, wenn die Funktionswerte der restlichen Punkte so beschaffen sind, dass sie zu den lokalen Optima hinführen. Dies spiegelt gut die intuitive Vorstellung wider, dass der (1+1) EA Hinweisen nur über einen kleinen Hamming-Abstand folgt, jedoch nicht „in die Ferne schauen“ kann.

Die Ergebnisse zum (1+1) EA mit Mutationsstärke $1/n$ seien hier kurz zusammengefasst: beginnend mit der einfach aufgebauten linearen Funktion ONEMAX und der bzgl. der Koeffizientengewichte konträren Funktion BINVAL konnten wir zeigen, dass alle linearen Funktionen vom (1+1) EA in erwarteter Zeit $\Theta(n \log(n))$ optimiert werden. Aber schon Funktionen vom Grad Zwei haben nicht mehr diese erwartete Laufzeit, selbst exponentielle Laufzeiten sind möglich, wie wir anhand der Funktion DISTANCE nachgewiesen haben. Auch können unimodale Funktionen, die eine echte Oberklasse der linearen Funktionen bilden, vom (1+1) EA nicht stets in polynomieller erwarteter Laufzeit optimiert werden, wie wir anhand von $\text{LONGPATH}_{\sqrt{n-1}}$ gezeigt haben. Die zuletzt vorgestellte Funktionsklasse JUMP_m zeigt, dass als erwartete Laufzeiten auch Polynome von beliebigem Grad $m \in \{1, \ldots, n\}$ auftreten können, es also möglich ist, eine Hierarchie von immer schwerer werdenden Funktionen anzugeben.

3.5 Variationen des (1+1) EA

Nachdem wir in den letzten Abschnitten den (1+1) EA mit Mutationsstärke $1/n$ ausführlich untersucht haben, sollen nun Variationen des (1+1) EA bzgl. der Mutationsstärke bzw. des Selektionsmechanismus untersucht werden.

3.5.1 Analyse des (1+1) EA mit $p_m(n) \neq 1/n$

Im Abschnitt 3.4 wurde der (1+1) EA nur mit der Mutationsstärke $1/n$ untersucht. Obwohl diese Mutationsstärke natürlich scheint, da durch sie im Erwartungswert je Mutation genau ein Bit geändert wird, und sich auch in empirischen Untersuchungen als vorteilhaft erwiesen hat (Bäck (1993)), ist diese Wahl dennoch vom theoretischen Standpunkt aus bisher noch unbegründet.

Deshalb soll nun zumindest für lineare Funktionen gezeigt werden, dass sich für Mutationsstärken der Größenordnungen $o(1/n)$ bzw. $\omega(\log(n)/n)$ eine echt größere erwartete Laufzeit als $\Theta(n \log(n))$ ergeben kann. Damit ist nur für den Bereich von Mutationsstärken der Größenordnungen aus dem Schnitt von $\Omega(1/n)$ und $O(\log(n)/n)$ nicht gezeigt, dass diese bei der Optimierung linearer Funktionen zu größeren erwarteten Laufzeiten führen. Jedoch werden wir Hinweise finden, dass dies auch für solche Mutationsstärken der Fall ist.

Betrachten wir zuerst den Fall, dass $p_m(n)$ langsamer als $1/n$ wächst, es also eine Funktion $\alpha : \mathbb{N} \mapsto \mathbb{R}$ mit $\lim_{n\to\infty} \alpha(n) = \infty$ gibt, so dass $p_m(n) = 1/(n\alpha(n))$ ist. In diesem Fall sollte die Mutation der zu Beginn mit hoher Wahrscheinlichkeit linear vielen falsch gesetzten Bits länger als mit $p_m(n) = 1/n$ dauern. Genau dies wird ja von der allgemeinen untere Schranke in Korollar 3.3.5 ausgedrückt, wenn wir eine lineare Funktion annehmen, die $\Omega(n)$ Gewichte ungleich Null hat oder sogar nur ein Optimum besitzt. Setzen wir $p_m(n) = 1/(n\alpha(n))$ ein, so ergibt sich als untere Schranke für die erwartete Laufzeit des (1+1) EA

$$\Omega\left(\frac{\log(n) \cdot (1 - 1/(n\alpha(n)))}{1/(n\alpha(n))}\right) = \Omega(\alpha(n) \cdot n \log(n)).$$

Korollar 3.5.1 *Sei* $f : \{0,1\}^n \mapsto \mathbb{R}$ *eine zu maximierende lineare Zielfunktion, die* $\Omega(n)$ *Koeffizienten ungleich Null besitzt, und* $\alpha : \mathbb{N} \mapsto \mathbb{R}$ *eine Funktion mit* $\lim_{n\to\infty} \alpha(n) = \infty$*. Dann ist die erwartete Laufzeit des* (1+1) EA *auf* f *mit* $p_m(n) = 1/(\alpha(n)n)$ *von der Größenordnung* $\Omega(\alpha(n) \cdot n \log(n))$*.*

Wächst die Mutationsstärke $p_m(n)$ stärker als $1/n$, d. h. ist $p_m(n) = \alpha(n)/n$, wobei $\alpha : \mathbb{N} \mapsto \mathbb{R}$ eine mit $n \to \infty$ gegen ∞ konvergierende Funktion ist, so liefert Korollar 3.3.5 eine untere Schranke für die erwartete Laufzeit von $\Omega(n \cdot \log(n)/\alpha(n))$. Dies ist auch anschaulich klar, da sich Korollar 3.3.5 darauf stützt, dass zur erfolgreichen Optimierung alle falsch gesetzten Bits mindestens einmal mutieren müssen, was bei größerer Mutationsstärke natürlich schneller erfolgt.

Eine große Mutationsstärke wird bei linearen Funktionen jedoch gerade dann nachteilig sein, wenn der aktuelle String bis auf wenige Bits gleich dem Optimum ist. In diesem Fall sollte eine erfolgreiche Mutation recht unwahrscheinlich sein, da tendenziell mehr korrekt als falsch gesetzte Bits mutieren als umgekehrt, was in den meisten Fällen eine Verringerung des Funktionswerts bedeutet. Damit diese Situation nicht eintritt, muss das Optimum von einem recht weit entfernten Punkt in einer einzigen erfolgreichen Mutation erreicht werden. Dies erfordert jedoch eine Mutation aller falsch gesetzten Bits, während alle korrekt gesetzten Bits nicht mutieren, was ebenfalls recht unwahrscheinlich ist. Diese anschauliche Argumentation führt zu:

Lemma 3.5.2 *Sei* $\alpha : \mathbb{N} \mapsto \mathbb{R}$ *mit* $\lim_{n\to\infty} \alpha(n) = \infty$. *Die erwartete Laufzeit des* (1+1) EA *für* OneMax *mit* $p_m(n) = \alpha(n)/n$ *ist von der Größenordnung* $\Omega(c^{\alpha(n)})$ *für eine Konstante* $c > 1$.

Beweis: Wir werden im Folgenden zeigen, dass sich die obige Argumentation formal umsetzen lässt: entweder nähert sich der (1+1) EA dem Optimum $(1, \ldots, 1)$ von OneMax sehr nahe an, woraufhin die Erfolgswahrscheinlichkeit sehr gering ist, oder er muss von einem weit entfernten Punkt direkt zum Optimum springen, was ebenfalls sehr unwahrscheinlich ist. Hierbei wird $19 \cdot n/20$ die von uns gewählte Mindestzahl an Einsen eines Punkts sein, der nahe am Optimum liegt.

Nach der Initialisierung wird die Anzahl der Einsen mit exponentiell hoher Wahrscheinlichkeit kleiner als $19 \cdot n/20$ sein, der Punkt also nicht nahe am Optimum $(1, \ldots, 1)$ liegen, da sich mit der Tschernoff-Ungleichung (A.9) leicht ergibt:

$$\mathrm{P}\left(||x_0||_1 \geq \frac{19 \cdot n}{20}\right) = \mathrm{P}\left(||x_0||_1 \geq \left(1 + \frac{9}{10}\right) \cdot \frac{n}{2}\right) \leq \exp\left(-\frac{27}{200} \cdot n\right).$$

Aus der Symmetrie bei der Initialisierung des (1+1) EA folgt, dass die Anzahl der Einsen nach der Initialisierung mit exponentiell großer Wahrscheinlichkeit im Intervall $]n/20, 19 \cdot n/20[$ liegt.

Um im Verlauf des (1+1) EA das Optimum zu erreichen, muss eine der beiden folgenden Möglichkeiten eintreten:

1. Kein Punkt mit mindestens $19 \cdot n/20$ und weniger als n Einsen wird erreicht.
2. Ein Punkt mit mindestens $19 \cdot n/20$ und weniger als n Einsen wird erreicht.

In beiden Fällen wird ein sehr unwahrscheinliches Ereignis eintreten müssen:

1. Wenn kein Punkt mit mindestens $19 \cdot n/20$ Einsen außer dem Optimum erreicht wird, so muss zu einem Zeitpunkt, an dem die Zahl der Einsen kleiner als $19 \cdot n/20$ ist, jede Null mutieren, während keine Eins mutiert, so dass direkt zum Optimum $(1, \ldots, 1)$ von OneMax gesprungen wird. Wenn i die Zahl der Einsen zu diesem Zeitpunkt ist, so ist die Wahrscheinlichkeit einer solchen Mutation gleich

   $$\left(\frac{\alpha(n)}{n}\right)^{n-i} \cdot \left(1 - \frac{\alpha(n)}{n}\right)^{i} \leq \left(\frac{1}{4}\right)^{\min(n-i,i)}.$$

   Für den Fall $n/20 < i < 19 \cdot n/20$, der exponentiell große Wahrscheinlichkeit hat, ist das Minimum von $n - i$ und i größer als $n/20$. Somit kann die Wahrscheinlichkeit einer solchen Mutation durch $(1/4)^{n/20}$ nach oben beschränkt werden. Damit ist die erwartete Wartezeit auf ein solches Ereignis exponentiell groß in n und damit auch in $\alpha(n)$.

2. Wenn ein Punkt mit mindestens $19 \cdot n/20$ Einsen erreicht wurde, so wollen wir zeigen, dass die Wahrscheinlichkeit einer erfolgreichen Mutation sehr gering ist. Eine Mutation ist für die Zielfunktion OneMax genau dann erfolgreich, wenn mindestens so viele Nullen wie Einsen mutieren. Wenn wir demnach für eine Zahl $m(n)$ eine untere Schranke für die Wahrscheinlichkeit zeigen können, dass mindestens $m(n)$ Einsen mutieren, aber weniger als $m(n)$ Nullen, so ist dies ebenfalls eine untere Schranke für die Wahrscheinlichkeit einer erfolglosen Mutation.

   Genau dieser Ansatz wird jetzt mit $m(n) = \alpha(n)/4$ verfolgt, um zu zeigen, dass die Wahrscheinlichkeit einer erfolgreichen Mutation sehr klein ist. Wenn die Zufallsvariable X_j^1 $(j \in \{1, \ldots, i\})$ genau dann gleich Eins ist, wenn die

j-te Eins in dem aktuellen Bitstring mutiert, und sonst gleich Null, so ist $\sum_{j=1}^{i} X_j^1$ gleich der Anzahl der mutierenden Einsen. Da

$$P\left(\sum_{j=1}^{i} X_j^1 \geq \frac{\alpha(n)}{4}\right) \geq P\left(\sum_{j=1}^{19\cdot n/20} X_j^1 \geq \frac{\alpha(n)}{4}\right)$$

ist, können wir uns zur Berechnung einer unteren Schranke auf die ersten $19 \cdot n/20$ Einsen beschränken. Nun gilt mit der Tschernoff-Ungleichung (A.9)

$$\begin{aligned} P\left(\sum_{j=1}^{19\cdot n/20} X_j^1 \geq \frac{\alpha(n)}{4}\right) &\geq 1 - P\left(\sum_{j=1}^{19\cdot n/20} X_j^1 \leq \frac{\alpha(n)}{4}\right) \\ &= 1 - P\left(\sum_{j=1}^{19\cdot n/20} X_j^1 \leq \left(1 - \frac{14}{19}\right) \cdot \frac{19}{20} \cdot \alpha(n)\right) \\ &\geq 1 - \exp\left(-\left(\frac{14}{19}\right)^2 \cdot \frac{19 \cdot \alpha(n)}{20} \cdot \frac{1}{2}\right) \\ &> 1 - 0{,}78^{\alpha(n)}. \end{aligned}$$

Analog werden wir nun eine in $\alpha(n)$ exponentiell große untere Schranke für die Wahrscheinlichkeit bestimmen, dass weniger als $\alpha(n)/4$ Nullen mutieren. Dazu sei die Zufallsvariable X_j^0 ($j \in \{1, \ldots, n-i\}$) genau dann gleich Eins, wenn die j-te Null in dem aktuellen Bitstring mutiert, und sonst gleich Null. Dann folgt aus der Tschernoff-Ungleichung (A.9):

$$\begin{aligned} P\left(\sum_{j=1}^{n-i} X_j^0 < \frac{\alpha(n)}{4}\right) &= 1 - P\left(\sum_{j=1}^{n-i} X_j^0 \geq \frac{\alpha(n)}{4}\right) \\ &\geq 1 - P\left(\sum_{j=1}^{n/20} X_j^0 \geq \frac{\alpha(n)}{4}\right) \\ &> 1 - P\left(\sum_{j=1}^{n/20} X_j^0 \geq 2 \cdot \frac{\alpha(n)}{20}\right) \\ &\geq 1 - \exp\left(-\frac{\alpha(n)}{60}\right) \\ &> 1 - 0{,}99^{\alpha(n)}. \end{aligned}$$

Da die beiden betrachteten Ereignisse unabhängig sind, ist die Wahrscheinlichkeit, dass mindestens $\alpha(n)/4$ Einsen und weniger als $\alpha(n)/4$ Nullen mutieren, größer als

$$\left(1 - 0{,}85^{\alpha(n)}\right) \cdot \left(1 - 0{,}99^{\alpha(n)}\right) > 1 - 2 \cdot 0{,}99^{\alpha(n)}.$$

Somit ist die Wahrscheinlichkeit einer erfolgreichen Mutation, wenn die Anzahl der Einsen mindestens $19 \cdot n/20$ ist, höchstens gleich $2 \cdot 0{,}99^{\alpha(n)}$. Die erwartete Wartezeit ist also mindestens $\Omega((1/0{,}99)^{\alpha(n)})$.

□

Wenn nun $\alpha(n) \geq (1+\varepsilon) \cdot \log(n)$ für ein konstantes $\varepsilon > 0$ ist, so wächst $\Omega(c^{\alpha(n)})$ schneller als $O(n \log(n))$. Somit ergibt sich

Korollar 3.5.3 *Die erwartete Laufzeit des* (1+1) EA *mit* $p_m(n) = \omega(\log(n)/n)$ *auf linearen Funktionen kann* $\omega(n\log(n))$ *betragen.*

Natürlich bleibt damit die Frage offen, ob für Mutationswahrscheinlichkeiten die schneller als $1/n$, aber höchstens so schnell wie $\log(n)/n$ wachsen, die erwartete Laufzeit des (1+1) EA ebenfalls größer als $O(n\log(n))$ ist. Darauf kann der Beweisansatz von Lemma 3.5.2 keine positive Antwort geben, da mit $p_m(n) = \ln(n)/n$ die Wahrscheinlichkeit, dass alle Bits nicht mutieren, gleich

$$\left(1 - \frac{\ln(n)}{n}\right)^n \geq \exp\left(-\frac{n\ln(n)}{n - \ln(n)}\right) = \left(\frac{1}{n}\right)^{n/(n-\ln(n))} = \Omega(n^{-\varepsilon})$$

für beliebiges konstantes $\varepsilon > 1$ ist. Somit lässt sich die Wahrscheinlichkeit einer erfolgreichen Mutation für Mutationsstärken aus dem angesprochenen Bereich bei noch linear vielen Nullen nicht als exponentiell klein charakterisieren.

Für Mutationsstärken $p_m(n) = \omega(1/n)$ wird die erwartete Zahl von Mutationen, die nach Korollar 3.3.5 notwendig sind, damit die durchschnittlich $n/2$ falsch gesetzten Bits überhaupt mutieren, wieder so groß sein, dass man zumindest für einen Spezialfall eine größere untere Schranke zeigen kann:

Lemma 3.5.4 *Sei* $\alpha : \mathbb{N} \mapsto \mathbb{R}$ *mit* $\lim_{n\to\infty} \alpha(n) = \infty$. *Die erwartete Laufzeit des* (1+1) EA *mit* $p_m(n) = \alpha(n)/n$ *auf* BinVal *ist* $\Omega(\alpha(n) \cdot n\log(n))$, *falls er vor dem Optimum den Punkt* $(x_1, \ldots, x_n) \in \{0,1\}^n$ *mit* $x_1 = \cdots = x_{n/2} = 1$ *und* $x_{n/2} = \cdots = x_n = 0$ *erreicht.*

Beweis: Wir betrachten die erwartete Laufzeit, bis von $x = (1, \ldots, 1, 0, \ldots, 0)$ aus das Optimum $(1, \ldots, 1)$ von BinVal erreicht wird. Aufgrund der Definition von BinVal kann eine Mutation nur erfolgreich sein, wenn keines der $n/2$ vorderen Bits mutieren will. Deshalb ist die Wahrscheinlichkeit eines erfolgreichen Schrittes höchstens gleich

$$\left(1 - \frac{\alpha(n)}{n}\right)^{n/2} \leq \exp\left(-\frac{\alpha(n)}{2}\right).$$

Mit Korollar 3.3.5 ergibt sich die erwartete Anzahl von Mutationen, bis alle $n/2$ hinteren Bits von x mutieren wollen, als $\Omega(n\log(n)/\alpha(n))$. Da sich natürlich nur in erfolgreichen Mutationen Bits ändern können und die Ereignisse unabhängig sind, ist die erwartete Laufzeit mindestens gleich

$$\exp\left(\frac{\alpha(n)}{2}\right) \cdot \Omega\left(\frac{n\log(n)}{\alpha(n)}\right) = \Omega(\alpha(n) \cdot n\log(n)).$$

□

Zusammengefasst ergibt sich somit, dass die erwartete Laufzeit des (1+1) EA auf linearen Funktionen bei den meisten von $1/n$ um Größenordnungen verschiedenen Mutationsstärken gleich $\omega(n\log(n))$ ist. Die Mutationsstärke $p_m(n) = 1/n$ scheint zumindest für lineare Funktionen die Bestmögliche zu sein.

3.5.2 Akzeptanz bei Gleichheit

Der (1+1) EA in seiner Urform mit $p_m(n) = 1/n$ hat ein sehr einfaches Selektionsverfahren: der neue Punkt wird genau dann übernommen, wenn sein Zielfunktionswert mindestens so hoch ist wie der des alten Punkts. Während es klar ist, dass eine Verbesserung stets akzeptiert werden sollte, kann man sich die Frage stellen, ob die Akzeptanz bei gleichem Funktionswert immer sinnvoll ist.

Im Folgenden wird gezeigt, dass es eine Zielfunktion gibt, bei der der (1+1) EA eine durch $O(2^n)$ nach oben beschränkte Laufzeit hat, die erwartete Laufzeit seiner

Variante (1+1) EA*, die nur Verbesserungen akzeptiert, jedoch $\Theta(\exp(n\log(n)))$ ist. Auch wenn beide Laufzeiten exponentiell sind, so zeigt doch die Funktion, in welchen Fällen sich das „Akzeptieren bei Gleichheit“ lohnen kann (für weitergehende Vergleiche zwischen dem (1+1) EA und dem (1+1) EA* siehe Jansen und Wegener (2000a)).

Die anschauliche Vorstellung ist, dass dies es dem Algorithmus ermöglicht, auf großen Mengen von benachbarten Punkten mit gleichem Funktionswert, so genannten *Plateaus*, einen rein zufallsgesteuerten Ablauf durchzuführen, der zum Rand des Plateaus und darüber hinaus führen kann. Der Algorithmus hingegen, der nur Verbesserungen akzeptiert, wird auf eine ggf. große Mutation von dem Plateau weg warten müssen, was sehr lange dauern kann.

Die nun betrachtete Funktion PEAK stellt den Extremfall einer Funktion mit Plateaus dar, da sie nur einen einzigen Punkt außerhalb des Plateaus enthält:

Definition 3.5.5 *Die Funktion* PEAK : $\{0,1\}^n \mapsto \mathbb{R}$ *ist definiert als*

$$\text{PEAK}(x) := \prod_{i=1}^{n} x_i.$$

Diese Funktion wird oftmals als Paradebeispiel einer für den (1+1) EA und andere evolutionäre Algorithmen schwierigen Funktion benutzt, da sie keinerlei Hinweise gibt, wo ihr globales Optimum ist. Zwar ist deshalb die erwartete Laufzeit des (1+1) EA exponentiell, wie wir sehen werden, doch ist sie nicht von der Größenordnung $\Omega(n^n)$. Diese wird aber von der erwarteten Laufzeit für die Funktion TRAP angenommen, wie in Lemma 3.4.4 gezeigt. Also wird die Anschauung bestätigt, dass „falsche Hinweise“, wie sie in TRAP von fast allen Punkten zum lokalen und weg vom globalen Optimum führen, noch schlechter sind als keine Hinweise.

Die folgende Schranke der erwarteten Laufzeit des (1+1) EA wurde von Garnier, Kallel und Schoenauer (1999) bewiesen:

Lemma 3.5.6 *Die erwartete Laufzeit des* (1+1) EA *auf der Funktion* PEAK *ist gleich* $\Theta(2^n)$.

Betrachten wir nun die modifizierte Variante (1+1) EA* des (1+1) EA, die, falls der aktuelle Punkt x_t und sein Nachkomme x'_t denselben Zielfunktionswert haben, stets den aktuellen Punkt x_t behält. Dazu ersetzen wir nur Schritt 5 in Algorithmus 3.1.1 durch

5'. Falls $f(x'_t) > f(x_t)$, setze $x_{t+1} := x'_t$, ansonsten $x_{t+1} := x_t$.

Lemma 3.5.7 *Die erwartete Laufzeit des* (1+1) EA* *auf* PEAK *ist* $O(\exp(n \cdot \ln(n/2)))$.

Beweis: Der Ablauf des (1+1) EA* auf PEAK ist besonders einfach zu analysieren, weil es nur zwei Fälle der Initialisierung zu unterscheiden gilt: falls im Optimum $(1,\ldots,1)$ initialisiert wird, ist die Laufzeit gleich Null; falls in einem Punkt mit Hamming-Abstand $k \in \{1,\ldots,n\}$ vom Optimum initialisiert wird, ist die erwartete Laufzeit gleich der erwarteten Zeit, bis nur genau diese k Nullen in einem Schritt mutieren. Somit ist die erwartete Laufzeit gleich

$$\begin{aligned}
& \frac{1}{2^n} \cdot \sum_{k=1}^{n} \binom{n}{k} \cdot n^k \cdot \left(1 - \frac{1}{n}\right)^{-(n-k)} \\
= \;& \frac{1}{2^n} \cdot \left(\left(\sum_{k=0}^{n} \binom{n}{k} \cdot n^k \cdot \left(\frac{n}{n-1}\right)^{n-k} \right) - \left(\frac{n}{n-1}\right)^{n} \right) \\
= \;& \frac{1}{2^n} \cdot \left(\left(n + \frac{n}{n-1} \right)^{n} - \left(\frac{n}{n-1}\right)^{n} \right) \\
= \;& O\left(\left(\frac{n}{2}\right)^{n} \right) = O\left(\exp\left(n \cdot \ln\left(\frac{n}{2}\right)\right)\right). \qquad \square
\end{aligned}$$

Das Akzeptieren von Punkten, die den gleichen Funktionswert wie der derzeitig aktuelle Punkt haben, verringert also die erwartete Laufzeit auf Peak stark. Der hier bewiesene Unterschied liegt aber nur zwischen zwei exponentiell wachsenden Funktionen, von denen die stärker Wachsende im Wesentlichen einen zusätzlichen Faktor $\ln(n/2)$ im Exponenten hat. Da der (1+1) EA für die Optimierung von Peak exponentielle Laufzeit benötigt, ist Peak bewiesenermaßen kein Kandidat, um einen größeren Unterschied zwischen dem (1+1) EA und der Variante (1+1) EA* zu zeigen. Dennoch zeigt dieses Beispiel, dass das Akzeptieren von Punkten mit gleicher Fitness auf großen Plateaus von Nutzen sein kann.

3.6 Zum Vorteil dynamischer Anpassung

Der (1+1) EA und seine betrachteten Variationen haben die Eigenschaft, dass alle Parameter und Strategiewahlen von vorneherein festgelegt sind und sich nicht im Laufe des Algorithmus ändern. Für die Mutation ist dies klar, in der Selektion wird sich mit dem aktuellen Punkt x_t des (1+1) EA im Laufe des Algorithmus zwar die Entscheidung ändern, welche Punkte selektiert werden: ein relativ schlechter Punkt, der zu Beginn übernommen worden wäre, wird zu einem späteren Zeitpunkt, wo ein besserer Punkt bereits gefunden ist, nicht mehr übernommen.

Jedoch wird stets die Strategie, einen neuen Punkt genau dann zu übernehmen, wenn er mindestens denselben Zielfunktionswert wie der alte Punkt hat, benutzt. Insofern wird über den gesamten Lauf dieselbe Strategie bei der Selektion verfolgt. Eine dynamische Variante könnte so aussehen, dass ein veränderlicher Parameter mitsamt der Differenz der Funktionswerte die Wahrscheinlichkeit bestimmt, mit der ein neuer Punkt übernommen wird.

Ein anderer, der Definition des (1+1) EA noch natürlicher entspringender Parameter ist die Mutationsstärke $p_m(n)$. Wie und für welche Funktionen sich eine veränderliche Mutationsstärke auf die effiziente Optimierbarkeit von Funktionen auswirken kann, wurde in Jansen und Wegener (2000b) untersucht.

Hier soll im Folgenden untersucht werden, wie sich eine wie oben beschriebene dynamische Anpassung der Selektion auswirkt. Kann eine Veränderung der Wahrscheinlichkeit, mit der ein Punkt übernommen wird, dafür sorgen, dass Funktionen effizient optimierbar sind, für die dies vorher nicht galt?

Dabei gilt es, den Begriff der Anpassung genau zu definieren. Denn lässt man zu, dass die Anpassung so weit geht, dass z. B. nur Schritte zu Punkten mit geringerem Hamming-Abstand zum Optimum übernommen werden, so könnten unrealistisch viele Funktionen effizient optimierbar sein. Doch setzt dies ein sehr großes Wissen über die zu optimierende Funktion voraus, was dem von uns zugrundegelegten Black-Box Szenario (Abschnitt 2.1) widersprechen würde.

Eine weit einfachere und realistischere Anpassung ist es, die Selektionswahrscheinlichkeit nur abhängig von der Anzahl der bisher durchgeführten bzw. übernommenen Mutationen zu verändern. Eine Einteilung der Möglichkeiten zur Veränderung von Parameterwerten speziell in evolutionären Algorithmen wurde in Bäck (1998) getroffen. Er unterscheidet dabei folgende drei Klassen: in *dynamischen* Methoden wird der Wert nur abhängig von der Anzahl der bisher durchgeführten Schritte verändert. In *adaptiven* Methoden kann zusätzlich der bisherige Erfolg des Algorithmus den Parameterwert beeinflussen, wohingegen bei *selbst-adaptiven* Methoden die Parameterwahl selbst durch den evolutionären Prozess gesteuert wird, wie es z. B. bei Evolutionsstrategien oft der Fall ist.

So unscharf die Abgrenzung zwischen adaptiven und selbst-adaptiven Methoden auch sein mag, so ist es doch klar, dass jede Klasse mehr Möglichkeiten zur Parameterwahl als die zuvor Erwähnten und insbesondere die statische Wahl eines festen Parameters eröffnet. Fraglich ist, ob diese hinzugekommenen Möglichkeiten auch die Mächtigkeit der entsprechenden Algorithmen erhöhen. In diesem Abschnitt wird gezeigt, dass dies für eine dynamische Veränderung im Vergleich zu einer statischen Festlegung des oben beschriebenen Selektionsparameters bewiesen werden kann.

Dazu werden wir den in einer statischen bzw. dynamischen Variante untersuchten Algorithmus zuvor natürlich genau definieren. Da die Veränderung des Selektionsmechanismus bei einer gleichzeitig durchgeführten Vereinfachung der Mutation den Übergang zu Metropolis- bzw. Simulated Annealing-Algorithmen (siehe Metropolis, Rosenbluth, Rosenbluth, Teller und Teller (1953) und van Laarhoven und Aarts (1987)) darstellt, ist unser Problem zur Frage äquivalent, eine Zielfunktion zu finden, für die ein Simulated Annealing-Algorithmus besser als jeder Metropolis-Algorithmus ist. Diese beiden Verfahren sind die wohl bekanntesten Vertreter lokaler Suchverfahren. Deshalb soll das diesen zugrundeliegende Prinzip zuvor vorgestellt werden.

Lokale Suchverfahren zur Maximierung einer Zielfunktion $f : \{0,1\}^n \mapsto \mathbb{R}$ benötigen eine Funktion $N : \{0,1\}^n \mapsto \mathcal{P}(\{0,1\}^n)$, so dass $N(x) \subseteq \{0,1\}^n$ die *Nachbarschaft* des Punkts $x \in \{0,1\}^n$ ist. Dann wird ein lokales Suchverfahren von einem zufällig gewählten Startpunkt aus jeweils zu einem besseren Punkt in der Nachbarschaft des aktuellen Punkts wechseln, bis in der Nachbarschaft nur schlechtere Punkte sind. Effizienz und Erfolgsaussichten dieses Verfahrens hängen natürlich von der Nachbarschaftsfunktion ab. Ist $N(x)$ zu klein, so lässt sich diese Menge zwar schnell durchsuchen, doch können dann oft global sehr schlechte Punkte lokale Optima bzgl. N bilden. Ist $N(x)$ zu groß, wird letzteres nicht eintreten, doch wird die Zeit zum Durchsuchen der Menge $N(x)$ stark anwachsen. Im Grenzfall, dass $N(x)$ stets gleich $\{0,1\}^n$ ist, entartet die lokale Suche zur kompletten Durchmusterung des Suchraums.

Um den Konsequenzen der richtigen Wahl von N etwas auszuweichen, kann man das Verfahren derartig verändern, dass nicht nur Verbesserungen akzeptiert werden, sondern auch Verschlechterungen. Damit man sich von relativ guten Punkten mit nur wenigen besseren Punkten in ihrer Nachbarschaft noch verbessern kann, sollten Verschlechterungen nicht stets akzeptiert werden, sondern nur mit einer gewissen Wahrscheinlichkeit kleiner als Eins.

Diese Ideen sind zum ersten Mal im Metropolis-Algorithmus (Metropolis, Rosenbluth, Rosenbluth, Teller und Teller (1953)) umgesetzt worden. Da dieser dem (1+1) EA sehr ähnlich ist, sei er hier in einer zum Vergleich mit dem (1+1) EA geeigneten Weise formal zusammengefasst:

Algorithmus 3.6.1 *Sei $f : \{0,1\}^n \mapsto \mathbb{R}$ eine zu maximierende Zielfunktion und $T \in \mathbb{R}^+$. Der Metropolis-Algorithmus ist wie folgt definiert:*

1. *Setze $t := 0$.*
2. *Wähle $x_t \in \{0,1\}^n$ gleichverteilt zufällig.*

3. *Wähle* $x'_t \in N(x_t)$ *zufällig und setze* $x_{t+1} := x_t$.

4. *Falls* $f(x'_t) \geq f(x_t)$, *setze* $x_{t+1} := x'_t$.

5. *Falls* $f(x'_t) < f(x_t)$, *setze* $x_{t+1} := x'_t$ *mit Wahrscheinlichkeit* $\exp(-(f(x_t) - f(x'_t))/T)$.

6. *Setze* $t := t + 1$.

7. *Gehe zu Schritt 3.*

Als Nachbarschaft $N(x)$ lässt sich z. B. die Menge aller Bitstrings aus $\{0,1\}^n$ mit Hamming-Abstand Eins von x wählen. Der Parameter T, der auch *Temperatur* genannt wird, steuert, mit welcher Wahrscheinlichkeit ein schlechterer Punkt übernommen wird. Ist z. B. $f(x_t) - f(x'_t) = 1$, so beträgt die Wahrscheinlichkeit, x'_t zu übernehmen, $\exp(-1/T)$. Je größer also T ist, desto größer ist die Wahrscheinlichkeit, dass ein schlechter Punkt übernommen wird.

Im Metropolis-Algorithmus ist die Temperatur fest gewählt, wobei das Problem der Wahl dieses Parameters auftritt. Anschaulich ist es besser, wenn dieser Parameter dynamisch sinkt: denn zu Beginn der Suche sollte lokalen Optima mit einer hohen Temperatur entkommen werden können, die im Verlauf des Algorithmus immer mehr sinkt, damit der Algorithmus gegen das globale Optimum konvergiert, in dessen Nähe er sich dann hoffentlich befindet. Auch wenn nach dem NFL-Theorem eine ansteigende Temperatur ebenso für viele Zielfunktionen günstiger ist, so scheint es doch schwieriger, dafür eine anschauliche Begründung zu finden, was natürlich an der subjektiven Vorstellung von „natürlichen“ Funktionen liegt.

Diese Erweiterung des Metropolis-Algorithmus ist unter dem Namen *Simulated Annealing* (siehe van Laarhoven und Aarts (1987)) bekannt. Die Bezeichnung lehnt sich an den so genannten *Annealing*-Prozess zur Erzeugung möglichst reiner Kristalle an, bei denen ebenfalls mit sehr hoher Temperatur gestartet wird, was eine hohe Veränderlichkeit der bisherigen Struktur ermöglicht, gefolgt von einer langsamen Abkühlung, so dass sich gefundene Strukturen verfestigen. Formal bedeutet dies, dass es eine Temperaturfunktion $T : \mathbb{N}_0 \mapsto \mathbb{R}^+$ gibt, so dass durch Ersetzung der statischen Temperatur T im Metropolis-Algorithmus durch den gerade aktuellen Wert $T(t)$ der Simulated Annealing-Algorithmus wird.

Beide Verfahren, der Metropolis-Algorithmus als auch Simulated Annealing, lassen sich als evolutionäre Algorithmen sehen: die zufällige Auswahl eines Punkts aus $N(x)$ lässt sich als Mutation bezeichnen, da die Punkte aus $N(x)$ dem Punkt x relativ ähnlich sind, und die Auswahl des nächsten Punkts in den Schritten 4 und 5 ist eine sinnvolle Selektion, da Punkte mit größerem Zielfunktionswert stets mit höherer Wahrscheinlichkeit gewählt werden. Da sie jeweils aus einem Elter ein Kind erzeugen und die Selektion zwischen Elter und Kind auswählt, können beide als Varianten des (1+1) EA bezeichnet werden.

Der (1+1) EA, wie wir ihn definiert haben (Algorithmus 3.1.1), benutzt anschaulich eine Temperatur von Null, da niemals Verschlechterungen akzeptiert werden, und eine Nachbarschaftsstruktur $N(x)$, in der jeder Punkt vorkommt, jedoch Punkte mit Hamming-Abstand $d \in \{0, \ldots, n\}$ von x_t eine Wahrscheinlichkeit von $(p_m(n))^d \cdot (1 - p_m(n))^{n-d}$ haben, in Schritt 3 gewählt zu werden. Wir werden uns im Folgenden darauf einschränken, dass $N(x)$ aus der Menge aller direkten Nachbarn von x, d. h. aller Punkte mit Hamming-Abstand Eins besteht. Somit wird x'_t in Schritt 3 aus x_t erzeugt, indem genau ein gleichverteilt zufällig gewähltes Bit von x_t negiert wird. Weiterhin führen wir ein *Selektionsschema* $\alpha : \mathbb{N}_0 \mapsto [1, \infty[$ ein, das als $\alpha(t) := \exp(1/T(t))$ aus dem dynamischen Temperaturschema hervorgeht. Dies dient einzig der leichteren Bezeichnungsweise. Der Klarheit wegen seien die resultierenden Algorithmen zusammengefasst:

Algorithmus 3.6.2 *Sei $f : \{0,1\}^n \mapsto \mathbb{R}$ eine zu maximierende Zielfunktion und $\alpha : \mathbb{N}_0 \mapsto [1, \infty[$. Dann sei folgender Algorithmus definiert:*

1. *Setze $t := 0$.*
2. *Wähle $x_t \in \{0,1\}^n$ gleichverteilt zufällig.*
3. *Setze $x'_t := x_t$, negiere ein gleichverteilt zufällig ausgewähltes Bit von x'_t und setze $x_{t+1} := x_t$.*
4. *Falls $f(x'_t) \geq f(x_t)$, setze $x_{t+1} := x'_t$.*
5. *Falls $f(x'_t) < f(x_t)$, setze $x_{t+1} := x'_t$ mit Wahrscheinlichkeit $\alpha(t)^{f(x'_t)-f(x_t)}$.*
6. *Setze $t := t + 1$.*
7. *Gehe zu Schritt 3.*

Wenn es ein $\alpha \in [1, \infty[$ gibt, so dass $\alpha(t)$ für alle $t \in \mathbb{N}$ konstant gleich α ist, so heißt der Algorithmus STATIC EA*, ansonsten* DYNAMIC EA.

Natürlich ist es von Interesse, ob und wann der Übergang vom STATIC EA zum DYNAMIC EA die effiziente Optimierung von weiteren Zielfunktionen ermöglicht. Auch wenn klar ist, dass es für jede Funktion, die der STATIC EA mit geeignetem T-Wert in polynomieller erwarteter Laufzeit optimiert, eine DYNAMIC EA-Variante gibt, die dieselbe erwartete Laufzeit hat (indem $T(t)$ konstant als T gewählt wird), ist es schwierig zu beweisen, dass die Optimierung mit einer dynamischen Temperatur für eine Zielfunktion effizienter ist als für jede statische Wahl.

Ein derartiger Beweis existiert bisher nur in Sorkin (1991), wo dies für eine speziell konstruierte fraktale Funktion unter Benutzung der Theorie schnell mischender Markoff-Ketten (siehe Sinclair (1993)) gezeigt wird. Weiterhin gibt es ein Resultat von Jerrum und Sorkin (1998), wo gezeigt wird, dass das Graphen-Bisektionsproblem auf bestimmten zufälligen Graphen effizient von einem Metropolis-Algorithmus gelöst werden kann. Beide Ansätze beantworten aber nicht die in Jerrum und Sinclair (1997) aufgestellte Frage, ob es eine natürliche Funktion gibt, für die eine dynamische Temperaturwahl zur effizienten Optimierung notwendig ist.

Auch wenn diese Frage von den im Folgenden vorgestellten Beweisen ebenfalls nicht beantwortet wird, weil die benutzten Zielfunktionen keiner natürlichen Problemstellung entspringen, so sind sie doch vergleichsweise einfach aufgebaut und ermöglichen somit die präsentierten einfachen Beweise, die sich hauptsächlich auf die Markoff-Ungleichung (A.8) stützen. Damit ist die Überzeugung verbunden, dass sich einfachere Beweise leichter auf natürliche Zielfunktionen anwenden lassen, da sie mehr über die Struktur der Algorithmen offen legen.

Da die Zielfunktionen, die wir betrachten werden, symmetrisch sind, können wir von der Position der Einsen abstrahieren. Deshalb werden wir in einem ersten Schritt einfache Eigenschaften des STATIC EA für symmetrische Zielfunktionen beweisen. Wenn die Werte von $\alpha(t)$ geeignet durch eine Konstante nach oben bzw. unten abgeschätzt werden können, so lassen sich diese Resultate auch auf den DYNAMIC EA übertragen.

3.6.1 Einfache Eigenschaften des STATIC EA auf symmetrischen Funktionen

Wir wollen nun grundlegende Eigenschaften über die Laufzeit T^f des STATIC EA auf symmetrischen Funktionen f untersuchen, die zur Verkürzung stets als T bezeichnet wird, wenn die betrachtete Funktion entweder unspezifiziert oder aus dem Zusammenhang klar ist. Da wir symmetrische Zielfunktionen annehmen, spielt nur die Anzahl der Einsen, nicht aber deren Position eine Rolle. Deshalb und da sowohl der Static EA als auch der Dynamic EA alle Positionen eines Suchpunkts gleich behandeln, ist die Zufallsvariable T_i, die die Laufzeit bis zum Erreichen eines Optimums bezeichnet, wenn der Startpunkt ein beliebiger Punkt $x \in \{0,1\}^n$ mit $||x||_1 = i$ ist, eindeutig definiert. Da im Schritt 2 des Static EA die Initialisierung gleichverteilt zufällig stattfindet, lässt sich der Erwartungswert der Laufzeit T als bedingter Erwartungswert

$$\mathrm{E}(T) = \sum_{i=0}^{n} \frac{\binom{n}{i}}{2^n} \cdot \mathrm{E}(T_i)$$

ausdrücken. Somit ermöglicht eine Abschätzung aller $\mathrm{E}(T_i)$ eine Abschätzung des gesamten Erwartungswerts. Diese Eigenschaft gilt natürlich auch für andere evolutionäre Algorithmen mit gleichverteilter Initialisierung, wie z. B. den (1+1) EA.

Im Gegensatz zu diesem kann der Static EA und auch der Dynamic EA von einem Punkt mit i Einsen nur zu Punkten mit $i-1$, i oder $i+1$ Einsen mutieren. Somit lässt sich die Laufzeit von einem Startpunkt mit i Einsen als Summe der Zeiten ausdrücken, von i zu $i+1$, von $i+1$ zu $i+2$, usw., und letztendlich von $n-1$ zu n Einsen zu kommen. Wenn wir als $\mathbb{N}$-wertige Zufallsvariable T_j^+ ($j \in \{0,\ldots,n-1\}$) die Zeit definieren, von j zu $j+1$ Einsen zu kommen, gilt aufgrund der Linearität des Erwartungswerts für alle $i \in \{0,\ldots,n-1\}$:

$$\mathrm{E}(T_i) = \mathrm{E}(T_i^+) + \mathrm{E}(T_{i+1}^+) + \cdots + \mathrm{E}(T_{n-1}^+).$$

Die erwartete Zeit, um von i zu $i+1$ Einsen zu kommen, lässt sich leicht ausdrücken, wenn man die Wahrscheinlichkeiten p_i^- bzw. p_i^+ kennt, dass der Static EA von einem Zustand mit i Einsen in einer Mutation zu einem Zustand mit $i-1$ bzw. $i+1$ Einsen kommt. Dazu unterscheidet man nach der ersten erfolgten Mutation: erhöht diese die Anzahl der Einsen, so beträgt die erwartete Laufzeit genau Eins; senkt sie sie um Eins, so beträgt die erwartete Laufzeit $1 + \mathrm{E}(T_{i-1}^+) + \mathrm{E}(T_i^+)$; bleibt die Anzahl der Einsen gleich, so ist die erwartete Laufzeit $1 + \mathrm{E}(T_i^+)$. Dies ergibt:

Lemma 3.6.3 *Sei p_i^+ bzw. p_i^- für $i \in \{0,\ldots,n-1\}$ die Wahrscheinlichkeit, dass der* Static EA *die Anzahl der Einsen in einer Mutation von i auf $i+1$ bzw. $i-1$ verändert. Dann gilt für alle $i \in \{0,\ldots,n-1\}$:*

$$\mathrm{E}(T_i^+) = \frac{1}{p_i^+} + \frac{p_i^-}{p_i^+} \cdot \mathrm{E}(T_{i-1}^+).$$

Beweis: Setzen wir die oben angestellten Betrachtungen bzgl. der ersten erfolgten Mutation mit der Methode der bedingten Erwartungswerte um, so folgt:

$$\begin{aligned}
& \mathrm{E}(T_i^+) && = p_i^+ \cdot 1 + p_i^- \cdot \bigl(1 + \mathrm{E}(T_{i-1}^+) + \mathrm{E}(T_i^+)\bigr) + \\
& && \quad \bigl(1 - p_i^+ - p_i^-\bigr) \cdot \bigl(1 + \mathrm{E}(T_i^+)\bigr) \\
\Longleftrightarrow\quad & p_i^+ \cdot \mathrm{E}(T_i^+) && = 1 + p_i^- \cdot \mathrm{E}(T_{i-1}^+) \\
\Longleftrightarrow\quad & \mathrm{E}(T_i^+) && = \frac{1}{p_i^+} + \frac{p_i^-}{p_i^+} \cdot \mathrm{E}(T_{i-1}^+).
\end{aligned}$$

□

Lemma 3.6.3 ermöglicht es, die erwartete Zeit, von (einem Zustand mit) i zu (einem Zustand mit) $i+1$ Einsen zu kommen, durch die Größen p_i^+, p_i^- und die erwartete Zeit, von $i-1$ zu i Einsen zu kommen, auszudrücken. Da wir die Wahrscheinlichkeit, dass die Anzahl der Einsen im Mutationsschritt des Static EA erhöht bzw. gesenkt wird, leicht ausrechnen können, können wir diese bei Kenntnis der Zielfunktion f auch leicht mit der Wahrscheinlichkeit verbinden, dass die Mutation akzeptiert wird. Die Berechnung von p_i^+ und p_i^- ist also kein Problem.

Unser Ziel ist es, $\mathrm{E}(T_i^+)$ nur mithilfe der p_i^+- bzw. p_i^--Werte auszudrücken. Setzen wir Lemma 3.6.3 rekursiv in sich ein, so kommen wir diesem Ziel einen Schritt näher:

Lemma 3.6.4 *Sei p_i^+ bzw. p_i^- für $i \in \{0, \ldots, n-1\}$ die Wahrscheinlichkeit, dass der* Static EA *die Anzahl der Einsen in einer Mutation von i auf $i+1$ bzw. $i-1$ verändert. Dann gilt für alle $i \in \{0, \ldots, n-1\}$ und alle $j \in \{1, \ldots, i\}$:*

$$\mathrm{E}(T_i^+) = \left(\sum_{k=0}^{j-1} \frac{\prod_{l=0}^{k-1} p_{i-l}^-}{\prod_{l=0}^{k} p_{i-l}^+} \right) + \frac{\prod_{l=0}^{j-1} p_{i-l}^-}{\prod_{l=0}^{j-1} p_{i-l}^+} \cdot \mathrm{E}(T_{i-j}^+).$$

Beweis: Diese Gleichung kann per Induktion über $j \in \{1, \ldots, i\}$ bewiesen werden. Dabei wird der Induktionsanfang für $j = 1$ von Lemma 3.6.3 gezeigt.

Der Induktionsschluss von j auf $j+1$ erfolgt dann durch bloßes Einsetzen von Lemma 3.6.3:

$$\begin{aligned}
\mathrm{E}(T_i^+) &= \left(\sum_{k=0}^{j-1} \frac{\prod_{l=0}^{k-1} p_{i-l}^-}{\prod_{l=0}^{k} p_{i-l}^+} \right) + \frac{\prod_{l=0}^{j-1} p_{i-l}^-}{\prod_{l=0}^{j-1} p_{i-l}^+} \cdot \mathrm{E}\left(T_{i-j}^+\right) \\
&= \left(\sum_{k=0}^{j-1} \frac{\prod_{l=0}^{k-1} p_{i-l}^-}{\prod_{l=0}^{k} p_{i-l}^+} \right) + \frac{\prod_{l=0}^{j-1} p_{i-l}^-}{\prod_{l=0}^{j-1} p_{i-l}^+} \cdot \left(\frac{1}{p_{i-j}^+} + \frac{p_{i-j}^-}{p_{i-j}^+} \cdot \mathrm{E}\left(T_{i-j-1}^+\right) \right) \\
&= \left(\sum_{k=0}^{j-1} \frac{\prod_{l=0}^{k-1} p_{i-l}^-}{\prod_{l=0}^{k} p_{i-l}^+} \right) + \frac{\prod_{l=0}^{j-1} p_{i-l}^-}{\prod_{l=0}^{j} p_{i-l}^+} + \frac{\prod_{l=0}^{j} p_{i-l}^-}{\prod_{l=0}^{j} p_{i-l}^+} \cdot \mathrm{E}\left(T_{i-j-1}^+\right) \\
&= \left(\sum_{k=0}^{j} \frac{\prod_{l=0}^{k-1} p_{i-l}^-}{\prod_{l=0}^{k} p_{i-l}^+} \right) + \frac{\prod_{l=0}^{j} p_{i-l}^-}{\prod_{l=0}^{j} p_{i-l}^+} \cdot \mathrm{E}\left(T_{i-(j+1)}^+\right)
\end{aligned}$$

□

Setzt man $j = i$, so erhält man damit einen Ausdruck für $\mathrm{E}(T_i^+)$ in Abhängigkeit der p_l^+- und p_l^--Werte und $\mathrm{E}(T_0^+)$. Da letzterer aber gleich $1/p_0^+$ ist, folgt:

Korollar 3.6.5 *Die erwartete Zeit* $\mathrm{E}(T_i^+)$, *bis der* Static EA *beginnend in einem Punkt mit $i \in \{0, \ldots, n-1\}$ Einsen einen Punkt mit $i+1$ Einsen zum ersten Mal erreicht, ist*

$$\mathrm{E}(T_i^+) = \left(\sum_{k=0}^{i-1} \frac{\prod_{l=0}^{k-1} p_{i-l}^-}{\prod_{l=0}^{k} p_{i-l}^+} \right) + \frac{\prod_{l=0}^{i-1} p_{i-l}^-}{\prod_{l=0}^{i-1} p_{i-l}^+} \cdot \frac{1}{p_0^+} = \sum_{k=0}^{i} \frac{1}{p_k^+} \cdot \prod_{l=k+1}^{i} \frac{p_l^-}{p_l^+}.$$

Damit haben wir nun einige Gleichungen bewiesen, die beim Abschätzen der erwarteten Laufzeiten des Static EA und Dynamic EA sehr nützlich sein werden. In den folgenden zwei Abschnitten werden wir mithilfe dieser Erkenntnisse für zwei Zielfunktionen zeigen, dass der Dynamic EA mit einer in t sinkenden bzw. steigenden Funktion $\alpha(t)$ jeweils schneller ist als der Static EA für beliebige Wahl von α.

3.6.2 Eine Analyse für sinkende Temperatur

Wir wollen in diesem Abschnitt eine Zielfunktion vorstellen, von der wir beweisen werden, dass der Static EA unabhängig von der Wahl des Parameters α zu ihrer Optimierung exponentielle erwartete Laufzeit benötigt, während der Dynamic EA mit einer geeigneten Funktion $\alpha : \mathbb{N}_0 \mapsto [1, \infty[$ nur polynomielle erwartete Laufzeit braucht. Dabei wird $\alpha(t)$ eine monoton wachsende Funktion sein, was, da $\alpha(t)$ umgekehrt proportional in T einfließt, einer monoton sinkenden Temperatur entspricht, wie es auch in den meisten Anwendungen von Simulated Annealing gewählt wird.

Welche Eigenschaften sollte eine Zielfunktion haben, für die der Static EA stets exponentielle erwartete Laufzeit hat? Da sowohl der Static EA als auch der Dynamic EA die Anzahl der Einsen nur um Eins steigern bzw. senken kann und wir nur symmetrische Funktionen betrachten, lässt sich ein Lauf des Algorithmus durch einen zufallsgesteuerten Lauf auf der $||x||_1$-Achse des Funktionsgraphen veranschaulichen. Dabei startet der Algorithmus nach der Tschernoff-Ungleichung (A.9) mit hoher Wahrscheinlichkeit in der Nähe eines Punkts x mit $||x||_1 = n/2$ und kann sich von da aus nur jeweils einen Schritt nach rechts oder links auf der $||x||_1$-Achse bewegen. Die Wahrscheinlichkeit einer (nicht unbedingt erfolgreichen) Mutation nach rechts bzw. links steigt mit sinkendem bzw. wachsendem $||x||_1$-Wert, da die Wahrscheinlichkeit, eine Null zu mutieren, mit sinkender Anzahl von Nullen sinkt. Dabei ist zusätzlich noch der Einfluss der Selektion zu beachten: bewirkt die Mutation keine Verringerung des Funktionswerts, so wird sie auf jeden Fall akzeptiert; bewirkt sie eine Verringerung, so wird dies je unwahrscheinlicher, desto größer die Verringerung ist. Gesucht ist nun die erwartete Zeit, um zum ersten Mal den Endpunkt der $||x||_1$-Achse bei $||x||_1 = n$ zu erreichen.

Eine Zielfunktion mit hoher erwarteter Laufzeit des Static EA sollte deshalb so aufgebaut sein, dass für einen konstanten Anteil aller Punkte auf der $||x||_1$-Achse (wobei Punkte an Stelle $||x||_1 = i$ einen Anteil von $\binom{n}{i}/2^n$ haben) die Funktion nach rechts hin stark abfallende, aber auch stark ansteigende Abschnitte hat. Denn der Parameter α steuert, mit wie großer Wahrscheinlichkeit Verschlechterungen akzeptiert werden. In nach rechts abfallenden Gebieten sollte α deshalb möglichst klein sein, damit Verschlechterungen mit hoher Wahrscheinlichkeit akzeptiert werden, in ansteigenden Gebieten jedoch möglichst groß. Wenn es sehr unterschiedliche Gebiete gibt, so liegt die Vermutung nahe, dass ein über den gesamten Lauf konstanter Wert schlecht sein muss.

Damit mit einem monoton wachsenden $\alpha(t)$ für eine solche Funktion nur eine polynomielle erwartete Laufzeit resultiert, sollten die abfallenden Gebiete für die meisten Punkte vor, d. h. links von den ansteigenden Gebieten sein. Wir werden sehen, dass die im Folgenden definierte Funktion Valley diese Anforderungen erfüllt:

Definition 3.6.6 *Die Funktion* Valley $: \{0,1\}^n \mapsto \mathbb{R}$ *ist definiert gemäß*

$$\text{Valley}(x) := \begin{cases} n/2 - ||x||_1 & \text{, falls } ||x||_1 \leq n/2, \\ 7 \cdot n^2 \ln(n) - n/2 + ||x||_1 & \text{, falls } ||x||_1 > n/2. \end{cases}$$

In Abbildung 3.4 ist die Funktion Valley für $n = 50$ veranschaulicht.

Wir wollen nun zeigen, dass die obigen anschaulichen Überlegungen für das Verhalten des Static EA für Valley zutreffen und dieser dort unabhängig vom Wert von α eine exponentielle erwartete Laufzeit hat:

Theorem 3.6.7 *Die erwartete Laufzeit des* Static EA *mit Parameter* $\alpha \in [1, \infty[$ *zur Optimierung von* Valley *beträgt*

$$\Omega\left(\left(\sqrt{\frac{\alpha}{4}}\right)^n + \left(\frac{1}{\alpha} + 1\right)^n\right),$$

was für jede Wahl von α *exponentiell in* n *wächst.*

Beweis: Wir betrachten nur den Fall, dass Static EA in einem Punkt mit höchstens $n/2 - 1$ Einsen initialisiert. Da n o. B. d. A. gerade ist, hat dies eine Wahrscheinlichkeit von $1/2 - o(1)$. In allen anderen Fällen schätzen wir die Laufzeit mit Null nach unten ab.

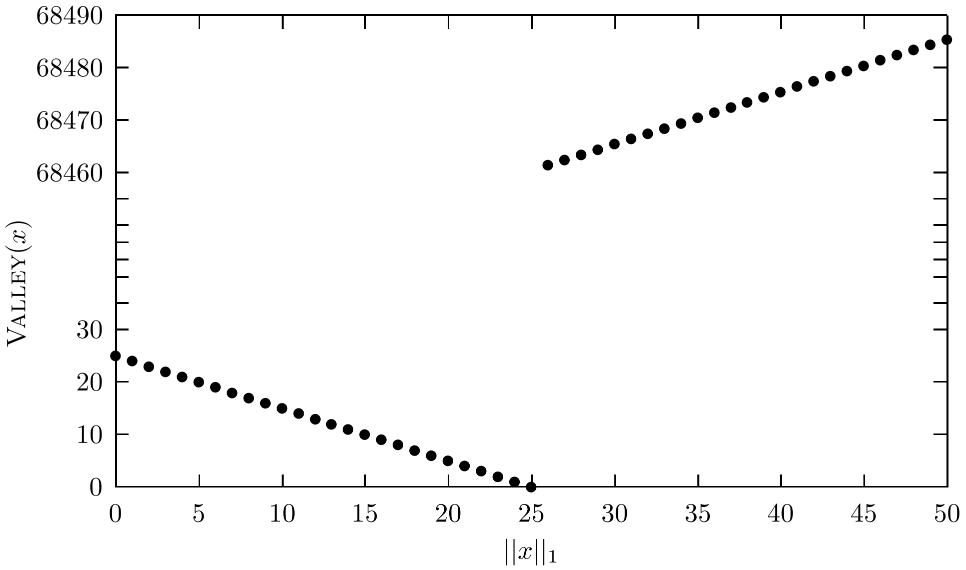

Abbildung 3.4: Die Funktion VALLEY für $n = 50$.

Da VALLEY in dem Bereich der Punkte x mit $||x||_1 < n/2$ bei Erhöhung der Anzahl der Einsen um Eins sinkt, gilt für die Wahrscheinlichkeiten p_l^+ und p_l^- für $l \in \{0, \ldots, n/2 - 1\}$

$$p_l^+ = \frac{n-l}{\alpha \cdot n} \text{ und } p_l^- = \frac{l}{n}.$$

Setzen wir diese Größen in Korollar 3.6.5 ein, so erhalten wir folgenden Ausdruck für $\mathrm{E}(T_i^+)$ mit $i \in \{0, \ldots, n/2 - 1\}$:

$$\begin{aligned}
\mathrm{E}(T_i^+) &= \sum_{k=0}^{i} \frac{1}{p_k^+} \cdot \prod_{l=k+1}^{i} \frac{p_l^-}{p_l^+} = \sum_{k=0}^{i} \frac{\alpha \cdot n}{n-k} \cdot \prod_{l=k+1}^{i} \frac{l}{n} \cdot \frac{\alpha \cdot n}{n-l} \\
&= \sum_{k=0}^{i} \alpha^{i-k+1} \cdot \frac{n}{n-k} \cdot \frac{i! \cdot (n-i-1)!}{k! \cdot (n-k-1)!} = \sum_{k=0}^{i} \alpha^{i-k+1} \cdot \frac{\binom{n}{k}}{\binom{n-1}{i}}.
\end{aligned}$$

Setzen wir $i = n/2 - 1$, so erhalten wir folgende untere Schranke für $\mathrm{E}(T_{n/2-1}^+)$:

$$\mathrm{E}(T_{n/2-1}^+) = \sum_{k=0}^{n/2-1} \alpha^{n/2-k} \cdot \frac{\binom{n}{k}}{\binom{n-1}{n/2-1}} \geq \frac{\alpha^{n/2}}{\binom{n-1}{n/2-1}} \geq \frac{\alpha^{n/2}}{2^n}.$$

Also ist die erwartete Anzahl von Schritten, um von $n/2 - 1$ zu $n/2$ Einsen zu kommen gleich $\Omega((\sqrt{\alpha/4})^n)$. Wenn α mindestens gleich $4 + \varepsilon$ für eine Konstante $\varepsilon > 0$ ist, so benötigt dieser Schritt exponentiell in n wachsende erwartete Zeit. Dies entspricht der Anschauung, dass bei einem großem Wert von α, d. h. einer kleinen Wahrscheinlichkeit, Verschlechterungen zu akzeptieren, der letzte Schritt vor dem Übergang zur rechten Hälfte sehr lange benötigt. Denn an diesem Punkt ist die Wahrscheinlichkeit einer Mutation nach rechts nur noch unwesentlich größer die einer Mutation nach links. Da aber Schritte nach rechts eine Verschlechterung bedeuten, werden sie weit seltener akzeptiert.

Ist α relativ klein, so betrachten wir den Übergang von $n - 1$ zu n Einsen. Dieser benötigt im Erwartungsfall $\mathrm{E}(T_{n-1}^+)$ Schritte. Da für $l \in \{n/2+2, \ldots, n-1\}$

Einsen ein Schritt nach links eine Verschlechterung um Eins und einer nach rechts eine Verbesserung um Eins bedeutet, gilt für die Übergangswahrscheinlichkeiten des STATIC EA:

$$p_l^+ = \frac{n-l}{n} \text{ und } p_l^- = \frac{l}{n \cdot \alpha}.$$

Setzt man dies in Lemma 3.6.4 mit $j = n/2 - 2$ ein, so erhalten wir eine untere Abschätzung von:

$$\begin{aligned}
\mathrm{E}(T_{n-1}^+) &\geq \sum_{k=0}^{n/2-3} \frac{\prod_{l=0}^{k-1} p_{n-1-l}^-}{\prod_{l=0}^{k} p_{n-1-l}^+} = \sum_{k=0}^{n/2-3} \frac{\prod_{l=n-k}^{n-1} p_l^-}{\prod_{l=n-k-1}^{n-1} p_l^+} \\
&= \sum_{k=0}^{n/2-3} \frac{\prod_{l=n-k}^{n-1} \frac{l}{n\cdot\alpha}}{\prod_{l=n-k-1}^{n-1} \frac{n-l}{n}} = \sum_{k=0}^{n/2-3} \frac{n}{\alpha^k} \cdot \frac{(n-1)!}{(n-k-1)! \cdot (k+1)!} \\
&= \sum_{k=0}^{n/2-3} \frac{\binom{n}{k+1}}{\alpha^k} = \alpha \cdot \sum_{k=1}^{n/2-2} \frac{\binom{n}{k}}{\alpha^k} \geq \alpha \cdot \left(\frac{\left(\frac{1}{\alpha}+1\right)^{n-4}}{2} - 1 \right) \\
&= \Omega\left(\left(\frac{1}{\alpha} + 1 \right)^n \right).
\end{aligned}$$

Ist α höchstens gleich einer von n unabhängigen Konstanten, so wächst dieser Ausdruck exponentiell in n.

Initialisiert der (1+1) EA in einem Punkt mit $i < n/2$ Einsen, so ist $\mathrm{E}(T_i) \geq \mathrm{E}(T_{n/2-1}^+) + \mathrm{E}(T_{n-1}^+)$. Da dieses mit konstanter Wahrscheinlichkeit eintritt, gilt:

$$\mathrm{E}(T) = \Omega\left(\left(\sqrt{\frac{\alpha}{4}} \right)^n + \left(\frac{1}{\alpha} + 1 \right)^n \right).$$

□

Die Funktion VALLEY erfüllt also die an sie gestellten Erwartungen. Der Beweis stützt die anschauliche Vorstellung, dass zum Optimum hin abfallende Gebiete nur mit einem kleinen α-Wert durchlaufen werden können, während für ansteigende Gebiete ein großer α-Wert nötig ist. Dabei basiert der Beweis des letzteren grundlegend darauf, dass bei dem letzten Schritt zum Optimum, d. h. von $n-1$ zu n Einsen, mit einer $(n-1)$-fach höheren Wahrscheinlichkeit eine Eins zu Null als eine Null zu Eins mutiert wird. Wegen dieser starken Tendenz zu Verschlechterungen sollte die Wahrscheinlichkeit, diese zu akzeptieren, so gering wie möglich sein, da der aktuelle Punkt des STATIC EA sonst immer wieder „nach links abrutscht".

Kommen wir nun zu dem Beweis, dass VALLEY vom DYNAMIC EA mit geeigneter Funktion $\alpha(t)$ in polynomieller Zeit optimiert wird. Dazu wird die „inverse Temperatur" $\alpha(t)$ im Laufe des Algorithmus ansteigen, um in einer ersten Phase die rechte Hälfte des Suchraums, d. h. einen Punkt $x \in \{0,1\}^n$ mit $||x||_1 \geq n/2 + 1$ zu erreichen. In der zweiten Phase wird dann mit hoher Wahrscheinlichkeit das Optimum erreicht, da die Akzeptanzwahrscheinlichkeit von Verschlechterungen immer geringer wird und es durch den hohen Funktionsunterschied zwischen einem Punkt mit $n/2 + 1$ und $n/2$ Einsen sehr unwahrscheinlich ist, von der rechten in die linke Hälfte zurückzurutschen. Diese Argumentation wird nun formal bewiesen:

Theorem 3.6.8 *Mit einer Wahrscheinlichkeit von $1 - O(n^{-n})$ ist die Laufzeit des* DYNAMIC EA *mit dem Selektionsschema $\alpha(t) := 1 + t/s(n)$ zur Optimierung von* VALLEY *gleich $O(n \cdot s(n))$, wobei $s(n)$ ein beliebiges Polynom mit $s(n) \geq 2\exp(1) \cdot n^4 \log(n)$ ist. Setzen wir $\alpha(t) := 1$ für $t > 2^n$, so ist auch die erwartete Laufzeit des* DYNAMIC EA *auf* VALLEY *gleich $O(n \cdot s(n))$.*

Beweis: Die Grundidee ist schon angerissen worden: wir zeigen, dass mit Wahrscheinlichkeit $1 - O(n^{-n})$ ein Punkt der rechten Hälfte mit mindestens $n/2 + 1$ Einsen nach einer ersten Phase der Länge $s(n)/n + 2\exp(1) \cdot n^3 \log(n)$ angenommen wird und nach der anschließenden zweiten Phase der Länge $n \cdot s(n) - (s(n)/n + 2\exp(1) \cdot n^3 \log(n))$ das Optimum erreicht wird. Anschließend schätzen wir die erwartete Anzahl der Schritte ab, falls eines dieser Ereignisse nicht eintritt.

1. Von der ersten Phase der Länge $s(n)/n+2\exp(1)\cdot n^3 \log(n)$ betrachten wir die ersten $s(n)/n$ Schritte nicht. In den restlichen Schritten ist $1 + 1/n \leq \alpha(t) \leq 1 + 2/n$. Wir betrachten nur diese Schritte, da ansonsten die Akzeptanzwahrscheinlichkeit von Verschlechterungen so groß ist, dass der Dynamic EA mit zu hoher Wahrscheinlichkeit von der rechten in die linke Hälfte mutieren kann. Da wir eine obere Abschätzung suchen, nehmen wir an, dass der aktuelle Punkt nach $s(n)/n$ Schritten höchstens $n/2$ Einsen hat.

   Wir wollen nun die erwartete Zeit nach oben abschätzen, bis von einem Punkt x mit $||x||_1 \leq n/2$ ein Punkt x' mit $||x'||_1 > n/2$ in der rechten Hälfte erreicht wird. Dazu benutzen wir folgende Gleichung aus Theorem 3.6.7, die für alle $i \in \{0, \ldots, n/2-1\}$ und konstanten $\alpha \in [1, \infty[$ gilt:

$$\begin{aligned} E(T_i^+) &= \sum_{k=0}^{i} \alpha^{i-k+1} \cdot \frac{\binom{n}{k}}{\binom{n-1}{i}} = \sum_{k=0}^{i} \alpha^{k+1} \cdot \frac{\binom{n}{i-k}}{\binom{n-1}{i}} \\ &= \sum_{k=0}^{i} \alpha^{k+1} \cdot \frac{n!}{(i-k)! \cdot (n-i+k)!} \cdot \frac{i! \cdot (n-1-i)!}{(n-1)!} \\ &= \sum_{k=0}^{i} \alpha^{k+1} \cdot \frac{\binom{i}{k}}{\binom{n-i+k}{k}} \cdot \frac{n}{n-i}. \end{aligned}$$

   Da in der ersten Phase $\alpha(t) \leq 1+2/n$ ist, erhalten wir durch Ersetzung von α durch $1+2/n$ in obigem Ausdruck eine obere Schranke für $E(T_i^+)$. Da $E(T_i^+)$ mit steigendem i steigt, liefert obiger Ausdruck für $E(T_{n/2-1}^+)$ und $\alpha = 1+2/n$ eine obere Schranke für $E(T_i^+)$, wenn $i \in \{0, \ldots, n/2-1\}$ ist:

$$E(T_i^+) \leq \sum_{k=0}^{n/2-1} \left(1 + \frac{2}{n}\right)^{k+1} \cdot \frac{\binom{n/2-1}{k}}{\binom{n/2+1+k}{k}} \cdot \frac{n}{n/2+1} \leq 2 \cdot \sum_{k=0}^{n/2-1} \exp(1) = \exp(1) \cdot n.$$

   Benutzen wir nun Lemma 3.6.3, so erhalten wir folgende obere Schranke für $E(T_{n/2}^+)$:

$$E(T_{n/2}^+) = \frac{1}{(n/2)/n} + \frac{(n/2)/n}{(n/2)/n} \cdot E(T_{n/2-1}^+) \leq 2 + \exp(1) \cdot n.$$

   Damit können wir die erwartete Anzahl von Schritten, um von $i \in \{0, \ldots, n/2\}$ Einsen zum ersten Mal zu mehr als $n/2$ Einsen zu gelangen, für $n \geq 3$ durch

$$E(T_i^+) + \cdots + E(T_{n/2}^+) \leq (\exp(1) \cdot n) \cdot \frac{n}{2} + (2 + \exp(1) \cdot n) \leq \exp(1) \cdot n^2$$

   nach oben abschätzen.

   Wie groß ist die Wahrscheinlichkeit höchstens, dass während der gesamten ersten Phase der Länge $2\exp(1)\cdot n^3 \log(n)$ kein Punkt mit mehr als $n/2$ Einsen erreicht wird? Dazu teilen wir die Phase in $n \log(n)$ Teilphasen der Länge $2\exp(1) \cdot n^2$ auf. Dass in einer Teilphase ein Punkt der rechten Hälfte erreicht wird, hat nach der Markoff-Ungleichung (A.8) und der obigen Abschätzung eine Wahrscheinlichkeit von mindestens $1/2$, unabhängig vom Startpunkt der Phase. Dass in den $n\log(n)$ Teilphasen niemals ein Punkt der rechten Hälfte erreicht wird, hat also höchstens Wahrscheinlichkeit $1/2^{n\log(n)} = n^{-n}$.

Natürlich impliziert das Erreichen der rechten Hälfte nicht, dass am Ende der ersten Phase auch ein Punkt der rechten Hälfte aktueller Punkt ist. Doch ist die Wahrscheinlichkeit, dass der DYNAMIC EA in der ersten Phase von der rechten in die linke Hälfte wechselt, sehr gering: da wir die ersten $s(n)/n$ Schritte vernachlässigen, gilt in der ersten Phase ja $\alpha(t) \geq 1 + 1/n$. Da der Funktionswert von VALLEY für einen Punkt mit $n/2+1$ Einsen um $7\cdot n^2\ln(n)$ größer ist als für einen mit $n/2$ Einsen, ist die Wahrscheinlichkeit einer erfolgreichen Mutation von der rechten in die linke Hälfte höchstens gleich

$$\frac{n/2+1}{n} \cdot \frac{1}{(1+1/n)^{7\cdot n^2\ln(n)}} < \frac{1}{\exp(4\cdot n\ln(n))}$$

Damit ist die Wahrscheinlichkeit, in den polynomiell vielen Schritten der beiden Phasen mit $t > s(n)/n$ nach Erreichen der rechten Hälfte wieder in die linke zu gelangen, von der Größenordnung $O(n^{-n})$. Insgesamt wissen wir somit, dass der DYNAMIC EA nach der ersten Phase mit Wahrscheinlichkeit $1 - O(n^{-n})$ in der rechten Hälfte ist.

2. Betrachten wir nun die zweite Phase, für die wir zeigen wollen, dass in ihr mit Wahrscheinlichkeit $1 - O(n^{-n})$ das Optimum $(1,\dots,1)$ von VALLEY erreicht wird. Dabei gehen wir analog zu der Abschätzung der ersten Phase vor. Zuerst betrachten wir nur die letzten $s(n)$ Schritte der zweiten Phase mit $t \geq (n-1)\cdot s(n)$. Dann ist $n \leq \alpha(t) \leq n+1$. Nun wollen wir die erwartete Zeit bis zum Erreichen des Optimums nach oben abschätzen, wozu wir die Werte $\mathrm{E}(T_i^+)$ für $i \in \{n/2+1,\dots,n-1\}$ nach oben abschätzen.

   Unter Zuhilfenahme des im Beweis von Theorem 3.6.7 hergeleiteten Ausdrucks kann $\mathrm{E}(T_{n/2-1}^+)$ für die zweite Phase folgendermaßen abgeschätzt werden:

$$\mathrm{E}(T_{n/2-1}^+) \leq \sum_{k=0}^{n/2-1} (n+1)^{n/2-k} \cdot \frac{\binom{n}{k}}{\binom{n-1}{n/2-1}} \leq \sum_{k=0}^{n/2-1} \binom{n}{k} \cdot n^{n-k} < (1+n)^n.$$

   Diese Abschätzung ist notwendig, um hiermit $\mathrm{E}(T_{n/2}^+)$ und dann $\mathrm{E}(T_{n/2+1}^+)$ abzuschätzen:

$$\mathrm{E}(T_{n/2}^+) = \frac{1}{(n/2)/n} + \frac{(n/2)/n}{(n/2)/n} \cdot \mathrm{E}(T_{n/2-1}^+) < 2 + (1+n)^n.$$

   Für $n \geq 3$ gilt dann nach Lemma 3.6.3:

$$\begin{aligned}\mathrm{E}(T_{n/2+1}^+) &\leq \frac{1}{(n/2-1)/n} + \frac{(n/2+1)/(n\cdot n^{7\cdot n^2\ln(n)})}{(n/2-1)/n} \cdot \mathrm{E}(T_{n/2}^+)\\ &< \frac{2n}{n-2} + \frac{n+2}{2n^{7n^2\ln(n)+1}} \cdot \frac{2n}{n-2} \cdot (2+(1+n)^n) < 7.\end{aligned}$$

   Benutzen wir nun Lemma 3.6.4 für $j = i - n/2 - 1$, so erhalten wir für alle $i \in \{n/2+2,\dots,n-1\}$:

$$\begin{aligned}\mathrm{E}(T_i^+) &= \left(\sum_{k=0}^{i-n/2-2} \frac{\prod_{l=0}^{k-1} p_{i-l}^-}{\prod_{l=0}^{k} p_{i-l}^+}\right) + \frac{\prod_{l=0}^{i-n/2-2} p_{i-l}^-}{\prod_{l=0}^{i-n/2-2} p_{i-l}^+} \cdot \mathrm{E}(T_{n/2+1}^+)\\ &< \left(\sum_{k=0}^{i-n/2-2} \frac{\prod_{l=i-k+1}^{i} p_l^-}{\prod_{l=i-k}^{i} p_l^+}\right) + \frac{\prod_{l=n/2+2}^{i} p_l^-}{\prod_{l=n/2+2}^{i} p_l^+} \cdot 7.\end{aligned}$$

Für alle Punkte x mit $l \in \{n/2+2, \ldots, n-1\}$ Einsen bedeutet eine Steigerung der Zahl der Einsen eine Verbesserung und eine Senkung eine Verschlechterung des Funktionswerts um Eins. Also gilt

$$p_l^+ = \frac{n-l}{n} \text{ und } p_l^- = \frac{l}{\alpha \cdot n}.$$

Daraus folgt für $i \in \{n/2+2, \ldots, n-1\}$:

$$\begin{aligned}
\mathrm{E}(T_i^+) &< \left(\sum_{k=0}^{i-n/2-2} \frac{\prod_{l=i-k+1}^{i} l/(n \cdot n)}{\prod_{l=i-k}^{i} (n-l)/n} \right) + \frac{\prod_{l=n/2+2}^{i} l/(n \cdot n)}{\prod_{l=n/2+2}^{i} (n-l)/n} \cdot 7 \\
&= \left(\sum_{k=0}^{i-n/2-2} \frac{n^{-2k} \cdot i!/(i-k)!}{n^{-k-1} \cdot (n-i+k)!/(n-i-1)!} \right) + \\
&\quad \frac{n^{-2i+n+2} \cdot i!/(n/2+1)!}{n^{-i+n/2+1} \cdot (n/2-2)!/(n-i-1)!} \cdot 7 \\
&= \left(\sum_{k=0}^{i-n/2-2} n^{1-k} \cdot \frac{\binom{i}{k}}{(n-i) \cdot \binom{n-i+k}{k}} \right) + \\
&\quad \frac{(n/2-1) \cdot \binom{n}{n/2+1} \cdot n^{n/2+1}}{(n-i) \cdot \binom{n}{i} \cdot n^i} \cdot 7.
\end{aligned}$$

Um den letzten Ausdruck nach oben abzuschätzen, betrachten wir das Verhältnis von Zähler und Nenner für $i \in \{0, \ldots, n-2\}$:

$$\begin{aligned}
& (n-i) \cdot \binom{n}{i} \cdot n^i \leq (n-(i+1)) \cdot \binom{n}{i+1} \cdot n^{i+1} \\
\iff \quad & \frac{n-i}{n-i-1} \cdot \frac{n!}{i! \cdot (n-i)!} \cdot \frac{(i+1)! \cdot (n-i-1)!}{n!} \leq n \\
\iff \quad & \frac{i+1}{n-i-1} \leq n.
\end{aligned}$$

Da der letzte Ausdruck für alle $i \in \{0, \ldots, n-2\}$ gilt, können wir den Bruch durch Eins nach oben abschätzen und erhalten somit für $i \geq n/2+2$:

$$\mathrm{E}(T_i^+) < \left(\sum_{k=0}^{i-n/2-2} n^{1-k} \cdot \frac{\binom{i}{k}}{(n-i) \cdot \binom{n-i+k}{k}} \right) + 7.$$

Da dieser Ausdruck mit steigendem i wächst, gilt die obere Abschätzung für $\mathrm{E}(T_{n-1}^+)$ für alle $\mathrm{E}(T_i^+)$ mit $i \in \{n/2+2, \ldots, n-1\}$ und $n \geq 3$:

$$\begin{aligned}
\mathrm{E}(T_{n-1}^+) &< n \cdot \left(\sum_{k=0}^{n/2-3} \left(\frac{1}{n}\right)^k \cdot \frac{\binom{n-1}{k}}{\binom{k+1}{k}} \right) + 7 \leq n \cdot \left(\sum_{k=0}^{n/2-3} \left(\frac{1}{n}\right)^k \binom{n}{k} \right) + 7 \\
&\leq n \cdot \left(1 + \frac{1}{n}\right)^n + 7 \leq \exp(1) \cdot n + 7 \leq 2\exp(1) \cdot n.
\end{aligned}$$

Damit kann nun die erwartete Anzahl von Schritten, bis der DYNAMIC EA in der zweiten Phase von $i \geq n/2+1$ Einsen zum Optimum gelangt, durch

$$\mathrm{E}(T_i^+) + \cdots + \mathrm{E}(T_{n-1}^+) \leq \frac{n}{2} \cdot 2\exp(1) \cdot n = \exp(1) \cdot n^2$$

nach oben abgeschätzt werden. Analog zur Analyse der ersten Phase ist damit die Wahrscheinlichkeit, in $2\exp(1) \cdot n^2$ Schritten der zweiten Phase von der rechten Hälfte aus nicht das Optimum erreicht zu haben, höchstens gleich $1/2$. Interpretieren wir die mindestens $s(n) \geq 2\exp(1) \cdot n^4 \log(n)$ Schritte der zweiten Phase als $n^2 \log(n)$ Subphasen der Länge $2\exp(1) \cdot n^2$, so ist die Wahrscheinlichkeit, in der zweiten Phase nicht das Optimum erreicht zu haben, höchstens gleich n^{-n^2}. Dies gilt natürlich nur unter der Annahme, dass kein Punkt der linken Hälfte erreicht wird. Da aber in einer der beiden Phasen nur mit Wahrscheinlichkeit $O(n^{-n})$ von der rechten in die linke Hälfte gewechselt wird, haben wir bewiesen, dass nach Ablauf der zweiten Phase mit Wahrscheinlichkeit $1 - O(n^{-n})$ das Optimum erreicht wurde.

Um nun die erwartete Laufzeit von Dynamic EA auf Valley abzuschätzen, brauchen wir die zusätzliche Voraussetzung, dass $\alpha(t) = 1$ für $t \geq 2^n$ ist. Denn könnte $\alpha(t)$ beliebig groß werden, könnte die Wahrscheinlichkeit, eine Verschlechterung zu akzeptieren, beliebig klein werden, so dass die erwartete Zeit, vom lokalen Optimum $(0, \dots, 0)$ zum globalen Optimum $(1, \dots, 1)$ zu kommen, beliebig groß werden könnte. Auch wenn es nur Wahrscheinlichkeit $O(n^{-n})$ hat, am Ende der zweiten Phase an diesem Punkt zu sein, kann dies doch zum dominierenden Term in der erwarteten Laufzeit werden.

Wenn jedoch $\alpha(t) = 1$ für $t > 2^n$ ist, so wird die Folge der vom Dynamic EA besuchten Punkte zu einem reinen *Random Walk* auf dem $\{0,1\}^n$, da die Zielfunktion Valley keinen Einfluss mehr ausübt. Die erwartete Wartezeit, um dann zum globalen Optimum zu gelangen, beträgt $O(2^n)$ (siehe Garnier, Kallel und Schoenauer (1999)). Da die Wahrscheinlichkeit, in $n \cdot s(n)$ Schritten das Optimum nicht erreicht zu haben, aber $O(n^{-n})$ ist, kann dieses höchstens einen additiven Term $o(1)$ zum Erwartungswert beitragen. □

3.6.3 Eine Analyse für steigende Temperatur

Eine Funktion, die vom Dynamic EA mit einer in t sinkenden Funktion $\alpha(t)$, d. h. mit einer steigenden Temperatur effizient optimiert wird, aber vom Static EA unabhängig von α nicht, sollte zu Beginn zum Optimum (das wir o. B. d. A. wieder bei $(1, \dots, 1)$ annehmen) hin ansteigen, so dass dieser Teil mit einer geringen Temperatur am schnellsten optimiert wird. Dann sollte jedoch eine Region folgen, die nur mit einer Verschlechterung durchlaufen werden kann, so dass eine hohe Temperatur nützlich ist. Wenn der Grad der nötigen Verbesserung bzw. Verschlechterung hoch genug ist, so sollte jede feste Wahl von α eine große erwartete Laufzeit zur Folge haben.

Diese Beschreibung einer Funktion erinnert an die Funktionen Jump_m, für die in Theorem 3.4.26 bewiesen wurde, dass sie in erwarteter Laufzeit $\Theta(n^m + n\log(n))$ vom (1+1) EA mit Mutationsstärke $1/n$ optimiert werden. Auch diese Funktionen steigen für alle $x \in \{0,1\}^n$ mit $||x||_1 \leq n - m$ mit zunehmender Anzahl von Einsen in x an, fallen dann aber bis zum Optimum $(1, \dots, 1)$ ab. Dabei wird sich der Wert $m = 2$ als für unseren Beweis ausreichend erweisen. Damit die notwendigen Verbesserungen bzw. Verschlechterungen stark genug ausfallen, wird die Funktion Jump_2 modifiziert:

Definition 3.6.9 *Die Funktion* $\text{ModJump}_2 : \{0,1\}^n \mapsto \mathbb{R}$ *ist definiert als*

$$\text{ModJump}_2(x) := \begin{cases} ||x||_1 & \text{, falls } ||x||_1 \leq n-3, \\ 3n^2 + n & \text{, falls } ||x||_1 = n-2, \\ 3n^2 & \text{, falls } ||x||_1 = n-1, \\ 3n^2 + n + 1 & \text{, falls } ||x||_1 = n. \end{cases}$$

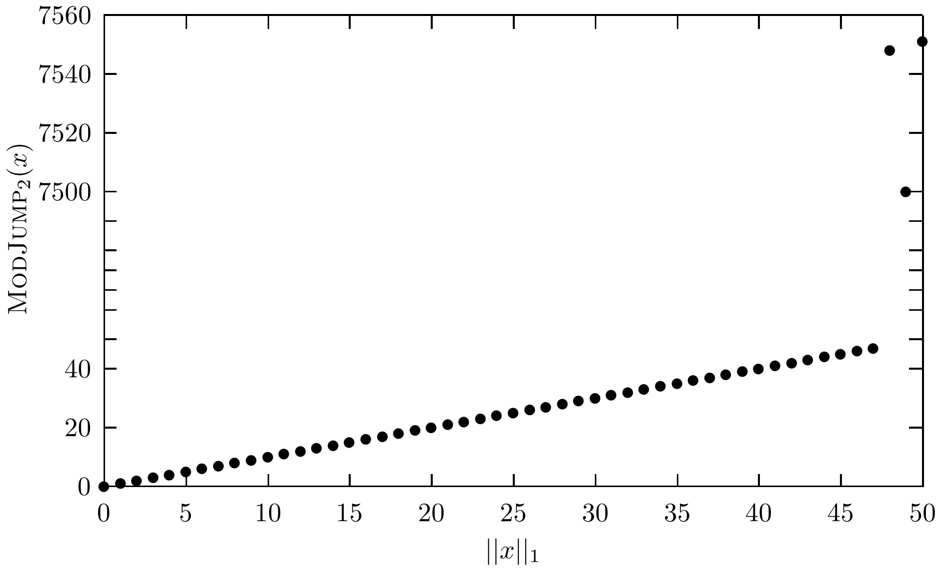

Abbildung 3.5: Die Funktion ModJump_2 für $n = 50$.

In Abbildung 3.2 ist die Funktion ModJump_2 für $n = 50$ veranschaulicht. Da für alle $l \in \{0, \ldots, n-3\}$ gilt

$$p_l^+ = \frac{n-l}{n} \text{ und } p_l^- = \frac{l}{\alpha \cdot n},$$

folgt unter Verwendung von Korollar 3.6.5:

Lemma 3.6.10 *Für alle $i \in \{0, \ldots, n-3\}$ ist die erwartete Zeit $\mathrm{E}(T_i^+)$, bis der* Static EA *auf* ModJump_2 *von einem Zustand mit i Einsen zu einem mit $i+1$ Einsen gekommen ist, gleich*

$$\left(\alpha^i \cdot \binom{n-1}{i}\right)^{-1} \cdot \sum_{k=0}^{i} \binom{n}{k} \cdot \alpha^k$$

Beweis: Setzen wir die Werte für p_l^+ und p_l^- in Korollar 3.6.5 ein, so ergibt sich

$$\begin{aligned}
\mathrm{E}(T_i^+) &= \sum_{k=0}^{i} \frac{1}{p_k^+} \cdot \prod_{l=k+1}^{i} \frac{p_l^-}{p_l^+} \\
&= \sum_{k=0}^{i} \frac{n}{n-k} \cdot \prod_{l=k+1}^{i} \frac{l}{\alpha \cdot (n-l)} \\
&= \sum_{k=0}^{i} \frac{n}{n-k} \cdot \frac{i! \cdot (n-i-1)!}{\alpha^{i-k} \cdot k! \cdot (n-k-1)!} \\
&= \frac{i! \cdot (n-i-1)!}{\alpha^i \cdot (n-1)!} \cdot \sum_{k=0}^{i} \frac{n!}{n-k} \cdot \frac{\alpha^k}{k! \cdot (n-k-1)!} \\
&= \left(\alpha^i \cdot \binom{n-1}{i}\right)^{-1} \cdot \sum_{k=0}^{i} \binom{n}{k} \alpha^k.
\end{aligned}$$

□

Mit der Formulierung dieses Lemmas lässt sich die exponentielle untere Schranke für den Static EA und beliebige Wahl von α nun folgendermaßen zeigen:

Theorem 3.6.11 *Die erwartete Laufzeit des* Static EA *mit Parameter* $\alpha \in [1, \infty[$ *zur Optimierung von* ModJump$_2$ *beträgt*

$$\Omega\left(n \cdot \alpha^n + \frac{(1+1/\alpha)^n}{n^2}\right),$$

was für jede Wahl von α exponentiell in n wächst.

Beweis: Betrachten wir zuerst die erwartete Zeit $\mathrm{E}(T^+_{n-3})$, um die Anzahl der Einsen von $n-3$ auf $n-2$ zu erhöhen. Dann gilt für alle $j \le n-3$, dass $\mathrm{E}(T_j) \ge \mathrm{E}(T^+_{n-3})$ ist. Dass in einem Punkt mit höchstens $n-3$ Einsen initialisiert wird, hat die exponentiell hohe Wahrscheinlichkeit $1 - O(n^2/2^n)$. Da gilt

$$\mathrm{E}(T^+_{n-3}) = \frac{\sum_{k=0}^{n-3}\binom{n}{k}\alpha^k}{\alpha^{n-3}\cdot\binom{n-1}{n-3}} \ge \left(\alpha^{n-3}\cdot n^2\right)^{-1}\cdot\sum_{k=0}^{n-3}\binom{n-3}{k}\alpha^k = \frac{(1+1/\alpha)^{n-3}}{n^2},$$

ist $\mathrm{E}(T) = \Omega\left((1+1/\alpha)^n/n^2\right)$. Wenn α höchstens gleich einer Konstanten ist, so wächst dies exponentiell schnell in n.

Ist α jedoch sehr groß, d. h. die Akzeptanzwahrscheinlichkeit einer Verschlechterung sehr gering, so sollte nicht dieser Übergang lange Wartezeit erfordern, sondern der von $n-2$ zu $n-1$ Einsen, da dort eine Verschlechterung des Funktionswerts um n akzeptiert werden muss. Die Wahrscheinlichkeit einer solchen Mutation beträgt $2/(n\cdot\alpha^n)$. Also ist die Wartezeit, bis dieses Ereignis eintritt, geometrisch mit diesem Parameter verteilt, weshalb die erwartete Laufzeit des Static EA gleich $\Omega(n\cdot\alpha^n)$ ist, da dieser Übergang ebenfalls für einen exponentiell großen Anteil aller initialen Punkte erfolgen muss. □

Nun werden wir zeigen, dass der Dynamic EA mit einem über die Zeit sinkenden Selektionschema $\alpha(t)$ die Zielfunktion ModJump$_2$ effizient optimieren kann. Dabei wird sich diese Funktion nicht beliebig nahe an Eins annähern können, denn dieses würde einer beliebig hohen Akzeptanzwahrscheinlichkeit von Verschlechterungen entsprechen. Eine zumindest kleine Bevorzugung von Verbesserungen ist aber eine sinnvolle Voraussetzung, wenn wir zugrundelegen, dass in der Umgebung von guten Punkten stets weitere gute Punkte liegen.

Theorem 3.6.12 *Mit einer Wahrscheinlichkeit von* $1-O(\exp(-n))$ *ist die Laufzeit des* Dynamic EA *mit dem Selektionsschema*

$$\alpha(t) := \max\left\{1+\frac{1}{n}, \frac{s(n)}{\max(t,1)}\right\}$$

zur Optimierung von ModJump$_2$ *gleich* $O(s(n))$, *wobei* $s(n)$ *ein beliebiges Polynom mit* $s(n) \ge 8\exp(1)\cdot n^3\log(n)$ *ist. Die erwartete Laufzeit des* Dynamic EA *mit diesem Selektionsschema auf* ModJump$_2$ *ist* $O(s(n))$.

Beweis: Wie beim Beweis von Theorem 3.6.8 teilen wir den Lauf von Dynamic EA der Länge $s(n)+2\cdot n^3 = O(s(n))$ in zwei Teile auf und zeigen, dass mit exponentiell großer Wahrscheinlichkeit nach der ersten Phase ein Punkt mit einer bestimmten Mindestanzahl von Einsen erreicht wurde und nach der zweiten Phase das Optimum. Zusätzlich wird die Laufzeit für den Fall abgeschätzt, dass in einer der Phasen das Ziel nicht erreicht wurde. Dabei hat die erste Phase die Länge $s(n)/n$, in der ein Punkt mit mindestens $n-2$ Einsen erreicht werden soll, und die zweite Phase die Länge $s(n)+2\cdot n^3 - s(n)/n$.

1. In der ersten Phase ist $\alpha(t) \ge n$, d. h. Verschlechterungen werden mit relativ kleiner Wahrscheinlichkeit akzeptiert. Setzen wir diese Abschätzung in Lemma

3.6.10 ein, so bekommen wir die folgende obere Abschätzung von $\mathrm{E}(T_i^+)$ für alle $i \in \{0, \ldots, n-3\}$:

$$\begin{aligned}
\mathrm{E}(T_i^+) &\leq \left(n^i \cdot \binom{n-1}{i}\right)^{-1} \cdot \sum_{k=0}^{i} \binom{n}{k} \cdot n^k \\
&= \frac{i! \cdot (n-1-i)!}{(n-1)!} \cdot \sum_{k=0}^{i} \frac{n!}{k! \cdot (n-k)!} \cdot n^{k-i} \\
&= \sum_{k=0}^{i} \frac{i!}{k! \cdot (i-k)!} \cdot \frac{n \cdot (n-1-i)! \cdot (i-k)!}{(n-k)!} \cdot n^{k-i} \\
&\leq \sum_{k=0}^{i} \binom{i}{k} \cdot \frac{n}{n-k} \cdot n^{k-i} \\
&\leq \frac{n}{n-i} \cdot \left(1 + \frac{1}{n}\right)^i .
\end{aligned}$$

Also beträgt die erwartete Zeit, um von einem Punkt mit maximal $n-3$ Einsen zu einem Punkt mit $n-2$ Einsen zu kommen, höchstens

$$\sum_{i=0}^{n-3} \frac{n}{n-i} \cdot \left(1 + \frac{1}{n}\right)^i \leq \exp(1) \cdot n \cdot \sum_{i=0}^{n-3} \frac{1}{n-i} \leq 2\exp(1) \cdot n \ln(n).$$

Da $s(n)/n \geq 8\exp(1) \cdot n^2 \ln(n)$ ist, können wir die erste Phase als mindestens $2n$ Subphasen der Länge $4\exp(1) \cdot n \ln(n)$ auffassen. In jeder Subphase ist die Wahrscheinlichkeit, keinen Punkt mit $n-2$ Einsen erreicht zu haben, nach der Markoff-Ungleichung (A.8) höchstens $1/2$. Also ist die Wahrscheinlichkeit, in allen $2n$ Subphasen keinen Punkt mit mindestens $n-2$ Einsen erreicht zu haben, höchstens gleich 4^{-n}.

Die Funktion ModJump$_2$ verschlechtert sich bei einem Übergang von einem Zustand mit $n-2$ zu einem mit $n-3$ Einsen um mehr als $3n^2$. Da während des ganzen Laufs des Dynamic EA $\alpha(t) \geq 1 + 1/n$ ist, beträgt die Wahrscheinlichkeit, einen solchen Schritt während der betrachteten $s(n) + 2 \cdot n^3$ Schritte durchzuführen, höchstens

$$\left(s(n) + 2 \cdot n^3\right) \cdot \frac{n-2}{n \cdot (1+1/n)^{3n^2}} \leq \exp(-n + O(\ln(n))),$$

was ebenfalls exponentiell klein ist. Also ist der Dynamic EA zu Beginn der zweiten Phase mit exponentiell schnell gegen Eins konvergierender Wahrscheinlichkeit in einem Punkt mit mindestens $n-2$ Einsen.

2. Für die zweite Phase müssen wir nun nur noch die Wahrscheinlichkeit abschätzen, mit der beginnend von einem Zustand mit mindestens $n-2$ Einsen in den $s(n) + 2 \cdot n^3 - s(n)/n$ Schritten der zweiten Phase das Optimum nicht erreicht wird. In den letzten $2 \cdot n^3$ Schritten der zweiten Phase ist $\alpha(t) = 1 + 1/n$. Somit ist die Wahrscheinlichkeit, in zwei Schritten von einem Zustand mit $n-2$ Einsen aus das Optimum zu erreichen, dann gleich

$$\left(\frac{2}{n} \cdot \alpha^{-n}\right) \cdot \frac{1}{n} = \frac{2}{n^2} \cdot \left(1 - \frac{1}{n}\right)^{-n} \geq \frac{2\exp(1)}{n^2}.$$

Also ist die Wahrscheinlichkeit, dass in den letzten $2 \cdot n^3$ Schritten das Optimum unter der Voraussetzung nicht erreicht wird, dass in einem Zustand mit mindestens $n-2$ Einsen gestartet wird, höchstens gleich

$$\left(1-\frac{2\exp(1)}{n^2}\right)^{n^3} \leq \exp(-n).$$

Die Summe der Wahrscheinlichkeiten der betrachteten Fehler, d. h. in der ersten Subphase keinen Zustand mit mindestens $n-2$ Einsen, in der zweiten Phase nicht das Optimum erreicht zu haben oder in der gesamten Zeit eine Mutation von $n-2$ zu $n-3$ Einsen gemacht zu haben, ist von der Größenordnung $O(\exp(-n))$. Somit ist die erste Aussage bewiesen.

Um zu zeigen, dass auch die erwartete Laufzeit von der Größenordnung $O(s(n))$ ist, zeigen wir, dass die Laufzeit des DYNAMIC EA im Fehlerfall, d. h. wenn eine der oben getroffenen Annahmen nicht eingetroffen ist, durch $O(2^n)$ abgeschätzt werden kann. Da die Wahrscheinlichkeit aller Annahmen gleich $O(-\exp(n))$ ist, erhöht dies die erwartete Laufzeit nur um eine additive Konstante. Dabei stützt sich der Beweis darauf, dass der Lauf des DYNAMIC EA durch eine Irrfahrt nach „oben abgeschätzt“ wird.

Im Fehlerfall betrachten wir nur die Schritte mit $t > s(n)$. Dann ist $\alpha(t) = 1 + 1/n$ und somit gilt für alle $l \in \{0, \ldots, n-3\}$:

$$p_l^+ = \frac{n-l}{n} \text{ und } p_l^- = \frac{l}{(1+1/n)\cdot n}.$$

Für alle anderen Werte von l gilt

$$p_{n-2}^+ = \frac{2}{(1+1/n)^n \cdot n},\ p_{n-2}^- = \frac{n-2}{(1+1/n)^{3n^2+3}\cdot n},\ p_{n-1}^+ = \frac{1}{n} \text{ und}$$

$$p_{n-1}^- = \frac{n-1}{(1+1/n)^n \cdot n}.$$

Da nach Korollar 3.6.5 die erwartete Anzahl von Schritten, um von einem Zustand mit i zu einem mit $i+1$ Einsen zu kommen, gleich

$$\mathrm{E}(T_i^+) = \sum_{k=0}^{i} \frac{1}{p_k^+} \cdot \prod_{l=k+1}^{i} \frac{p_l^-}{p_l^+}$$

ist, sinkt diese Größe mit steigenden Werten von p_l^+ oder sinkenden von p_l^-/p_l^+ bzw. p_l^-. Wir wollen zeigen, dass $\mathrm{E}(T_i^+)$ für alle $i \in \{0, \ldots, n-1\}$ jeweils durch $\mathrm{E}(\tilde{T}_i^+)$, den entsprechenden Wert für $\alpha(t) = 1$, nach oben abgeschätzt werden kann. Da $\alpha(t) = 1$ bedeutet, dass der DYNAMIC EA eine rein zufällige Irrfahrt durchführt, also jeden Übergang, egal ob Verbesserung oder Verschlechterung, stets akzeptiert, folgt für die entsprechenden Übergangswahrscheinlichkeiten für $i \in \{0, \ldots, n-1\}$

$$\tilde{p}_i^+ = \frac{n-i}{n} \text{ und } \tilde{p}_i^- = \frac{i}{n}.$$

Somit ist für $i \in \{0, \ldots, n-3\}$ die erwartete Zeit, von i nach $i+1$ Einsen zu kommen, für $\alpha(t) = 1 + 1/n$ nach oben durch die einer rein zufälligen Irrfahrt beschränkt, da $p_i^+ = \tilde{p}_i^+$ und $p_i^- < \tilde{p}_i^-$. Dies war auch zu erwarten: eine Verringerung der Wahrscheinlichkeit, einen Schritt nach links zu tun, wird die erwartete Zeit, nach rechts zu gehen, nicht erhöhen. Für $i = n-2$ ist die Lage nicht so offensichtlich. Da hier

$$p_{n-2}^+ = \frac{2}{n\cdot(1+1/n)^n} \text{ und } p_{n-2}^- = \frac{n-2}{n\cdot(1+1/n)^{3n^2+3}}$$

ist, ist $p^+_{n-2} > \exp(-1) \cdot \tilde{p}^+_{n-2}$ und

$$\frac{p^-_{n-2}}{p^+_{n-2}} = \frac{n-2}{n \cdot (1+1/n)^{3n^2+3}} \cdot \frac{n \cdot (1+1/n)^n}{2} = \frac{n-2}{2 \cdot (1+1/n)^{3n^2-n+3}} < \frac{\tilde{p}^-_{n-2}}{\tilde{p}^+_{n-2}}.$$

Also ist $\mathrm{E}(T^+_{n-2}) < \exp(1) \cdot \mathrm{E}(\tilde{T}^+_{n-2})$. Da nach Lemma 3.6.3 der Erwartungswert von T^+_{n-1} gleich

$$\frac{1}{p^+_{n-1}} + \frac{p^-_{n-1}}{p^+_{n-1}} \cdot \mathrm{E}(T^+_{n-2})$$

ist, folgt mit $p^+_{n-1} = \tilde{p}^+_{n-1}$ und $p^-_{n-1} < \tilde{p}^-_{n-1}$, dass $\mathrm{E}(T^+_{n-1}) \leq \exp(1) \cdot \mathrm{E}(\tilde{T}^+_{n-1})$ und somit $\mathrm{E}(T_i) \leq \exp(1) \cdot \mathrm{E}(\tilde{T}_i)$ für alle $i \in \{0, \ldots, n-1\}$ ist. Anwendung von Korollar 3.6.5 liefert für die erwartete Zeit einer zufälligen Irrfahrt, von einem Zustand mit i Einsen zum Optimum zu gelangen, die obere Schranke

$$\begin{aligned}
\mathrm{E}(\tilde{T}_0) &= \sum_{i=0}^{n-1} \sum_{k=0}^{i} \frac{1}{p^+_k} \cdot \prod_{l=k+1}^{i} \frac{p^-_l}{p^+_l} \\
&= \sum_{i=0}^{n-1} \sum_{k=0}^{i} \frac{n}{n-k} \cdot \prod_{l=k+1}^{i} \frac{l}{n-l} \\
&= \sum_{i=0}^{n-1} \sum_{k=0}^{i} \frac{n}{n-k} \cdot \frac{i!}{k!} \cdot \frac{(n-i-1)!}{(n-k-1)!} \\
&= \sum_{i=0}^{n-1} \sum_{k=0}^{i} \frac{i! \cdot (n-1-i)!}{(n-1)!} \cdot \frac{n!}{k! \cdot (n-k)!} \\
&= \sum_{i=0}^{n-1} \frac{1}{\binom{n-1}{i}} \cdot \sum_{k=0}^{i} \binom{n}{k} < 2^n \cdot \sum_{i=0}^{n-1} \frac{1}{\binom{n-1}{i}} \\
&= 2^n \cdot \left(2 + \sum_{i=1}^{n-2} \frac{1}{\binom{n-1}{i}}\right) \leq 2^n \cdot \left(2 + \sum_{i=1}^{n-2} \frac{1}{n-1}\right) < 2^n \cdot 3.
\end{aligned}$$

Da die Wahrscheinlichkeit dieses Falls von der Größenordnung $O(\exp(-n))$ ist, beträgt sein Anteil am Erwartungswert nur $o(1)$. □

3.7 Analyse einer natürlichen Funktion

Alle bisher in diesem Kapitel betrachteten Zielfunktionen fallen in eine von zwei Klassen: entweder es sind Funktionen, die speziell für den jeweiligen Zweck, z. B. für zwei Algorithmen als ein trennendes Beispiel zu dienen, konstruiert waren. Darunter fallen die Funktionen VALLEY und MODJUMP, die zwar vom DYNAMIC EA in polynomieller erwarteter Laufzeit optimiert werden, für die der STATIC EA jedoch stets exponentielle erwartete Laufzeit benötigt. Diese Funktionen, wozu auch LONGPATH und JUMP_m gehören, sind nicht als natürlich zu bezeichnen, da sie weder einem bekanntem Problem entspringen noch typische Beispiele bestimmter Funktionsklassen sind.

Funktionen wie ONEMAX, BINVAL und LEADINGONES, die typische Vertreter linearer bzw. unimodaler Funktionen sind, gehören zu der anderen Klasse. Diese Funktionen entspringen jedoch nicht einem bekanntem Problem, sondern werden im Gegenteil als oftmals zu leicht angesehen, um relevante Aussagen zu einem Algorithmus zu ermöglichen. So lassen sich lineare Funktionen natürlich deterministisch in linearer Zeit optimieren, wozu aber bekannt sein muss, dass eine lineare Funktion vorliegt. Auch wenn dieser Punkt oftmals unterschlagen wird, so sind doch Probleme von besonderem Interesse, für die kein effizientes Suchverfahren bekannt ist, da dies der Situation bei der praktischen Anwendung von evolutionären Algorithmen entspricht.

Im Folgenden werden wir eine Funktion vorstellen, die wir als natürlich betrachten, da sie von einem der bekanntesten Probleme aus der Komplexitätstheorie stammt. Weiterhin ist diese Funktion trotz ihrer einfachen Beschreibung als ein Polynom dritten Grades in dem Sinne schwierig zu optimieren, dass die besten bekannten Algorithmen für dieses Problem scheitern, obwohl die Lösung vom Menschen leicht zu finden ist. Dass diese Faktoren ausreichen, um eine Funktion „natürlich“ zu nennen, kann sicherlich bestritten werden. Doch scheint keine formale Definition für den Begriff „natürlich“ möglich zu sein, weshalb wegen der angeführten Eigenschaften der Funktion, die die bisher betrachteten Funktionen nicht haben, eine Untersuchung interessant erscheint. Diese Funktion wird sich für den (1+1) EA als sehr schwierig erweisen und damit in einer Reihe mit TRAP, LONGPATH und PEAK stehen.

Weiterhin wird diese Funktion die Eigenschaft haben, dass sie nicht nur vom (1+1) EA mit hoher Wahrscheinlichkeit beweisbar nicht effizient optimiert werden kann, sondern auch von anderen mutationsgestützen evolutionären Algorithmen. Wir werden begründet vermuten, dass kein „vernünftiges“ Suchverfahren dies kann, auch wenn wir keinen formalen Beweis für diese informal beschriebene Algorithmenklasse liefern.

3.7.1 Die Funktion MAXCOUNT

Die Funktion MAXCOUNT : $\{0,1\}^n \mapsto \mathbb{R}$, die wir in diesem Abschnitt einführen, leitet sich aus einer Instanz des Problems MAXSAT ab, eines der bekanntesten NP-harten Probleme (Garey und Johnson (1979)). Sei dieses Problem hier der Vollständigkeit wegen eingeführt:

Definition 3.7.1 *Für $n \in \mathbb{N}$ seien $x_1, \ldots, x_n$ boolesche Variablen und $a \in \{0,1\}^n$ eine* Variablenbelegung. *Ein* Literal *ist eine positive x_i bzw. eine negative Variable $\overline{x_i}$, die von einer Variablenbelegung a erfüllt wird, falls $a_i = 1$ bzw. $a_i = 0$ ist. Eine* Klausel *ist eine Menge von Literalen, die genau dann von einer Variablenbelegung erfüllt wird, falls diese mindestens eines ihrer Literale erfüllt. Die Eingabe von MAXSAT ist eine Menge von Klauseln und die Ausgabe ist eine Variablenbelegung, die maximal viele Klauseln in der Eingabe erfüllt.*

Die folgende Eingabe von MAXSAT wurde in Papadimitriou (1994) als ein für viele Algorithmen für MAXSAT besonders schwieriges Beispiel angegeben. Sie besteht aus den $n + n \cdot (n-1) \cdot (n-2)$ Klauseln

1. $\{x_i\}$ für $i \in \{1, \ldots, n\}$ und
2. $\{x_i, \overline{x_j}, \overline{x_k}\}$ für paarweise verschiedene $i, j, k \in \{1, \ldots, n\}$.

Jede Klausel dieser Eingabe für MAXSAT ist eine so genannte *Horn-Klausel*, d. h. eine Klausel, die jeweils höchstens ein positives Literal enthält. Deshalb erfüllt die Variablenbelegung $(1, \ldots, 1)$ alle Klauseln und ist in diesem Fall die eindeutige Lösung von MAXSAT für diese Eingabe. Dennoch ist diese Instanz für viele Algorithmen schwierig, da in den meisten Klauseln eine positive Variable zwei negativen Variablen gegenübersteht. Eine zufällige Wahl einer nicht erfüllten Klausel und Neusetzung des Werts einer der Literale in ihr wird also tendenziell die Zahl der Nullen in der Variablenbelegung erhöhen. Dies führt dazu, dass auch der beste bekannte (in Bezug auf die erwartete Worst-Case-Laufzeit) Algorithmus für MAXSAT (Schöning (1999)) zur Lösung dieser Eingabe exponentielle erwartete Zeit braucht.

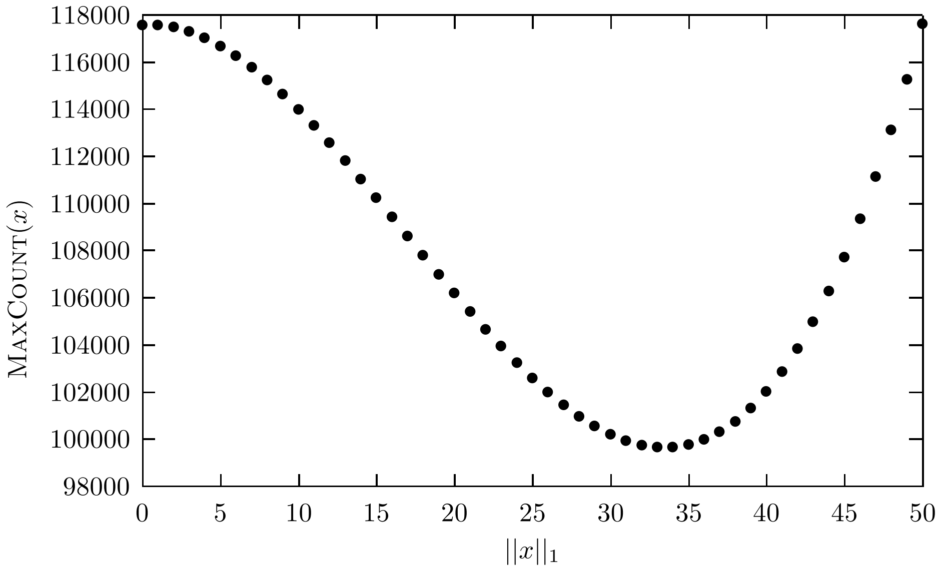

Abbildung 3.6: Die Funktion MaxCount für $n = 50$.

Um zu zeigen, dass sich diese Argumentation in einen formalen Beweis für den (1+1) EA und andere evolutionäre Algorithmen umsetzen lässt, müssen wir diese Eingabe in eine Zielfunktion $f : \{0,1\}^n \mapsto \mathbb{R}$ umformen. Dabei wird die Umformung so geschehen, dass für alle $a \in \{0,1\}^n$ der Wert $f(a)$ gleich der Anzahl der von der Variablenbelegung a erfüllten Klauseln in dieser Eingabe ist. Dies wird Klausel für Klausel geschehen, wobei mit $x_1, \ldots, x_n$ nun auch die Variablen der Funktion f bezeichnet werden:

1. Die Klauseln $\{x_i\}$ werden in den Term x_i umgeformt. Genau dann, wenn eine Belegung die Klausel erfüllt, liefert der Term den Wert Eins.

2. Die Klauseln $\{x_i, \overline{x_j}, \overline{x_k}\}$ werden in den Term

$$1 - (1 - x_i) \cdot x_j \cdot x_k$$

umgeformt. Da dieser Term genau dann gleich Null ist, wenn die Variablenbelegung die zugehörige Klausel nicht erfüllt, sonst jedoch nur den Wert Eins annehmen kann, ist die gewünschte Äquivalenz gesichert.

Summiert man all diese Terme auf, so erhält man eine Funktion MaxCount, deren Funktionswert genau gleich der Anzahl der erfüllten Klauseln der besprochenen Eingabe für MAXSAT ist, wenn das Argument von MaxCount die Variablenbelegung ist:

Definition 3.7.2 *Die Funktion* MaxCount : $\{0,1\}^n \mapsto \mathbb{N}$ *ist definiert als*

$$\text{MaxCount}(x) := \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \sum_{\substack{k=1 \\ i \neq k \neq j}}^{n} (1 - (1 - x_i) \cdot x_j \cdot x_k).$$

In Abbildung 3.6 ist die Funktion MaxCount für $n = 50$ veranschaulicht. Da die Eingabe für MAXSAT keiner Variablen eine besondere Rolle zuweist, ist die Funktion MaxCount symmetrisch. Es gibt also eine Funktion MaxCount* :

$\{0,\ldots,n\} \mapsto \mathbb{N}$, so dass für alle $x \in \{0,1\}^n$ die Werte von $\textsc{MaxCount}(x)$ und $\textsc{MaxCount}^*(||x||_1)$ gleich sind. Zur Untersuchung von MaxCount wird sich die Funktion $\textsc{MaxCount}^*$ als oftmals einfacher zu beschreiben erweisen. Wenn $s = ||x||_1$ ist, so können wir $\textsc{MaxCount}^*$ folgendermaßen herleiten:

$$\begin{aligned}
\textsc{MaxCount}(x) &= \sum_{1\le i\le n} x_i + n(n-1)(n-2) - 2(n-2)\cdot \sum_{1\le j<k\le n} x_j x_k + \\
&\quad 6\cdot \sum_{1\le i<j<k\le n} x_i x_j x_k \\
&= s + n(n-1)(n-2) - 2(n-2)\cdot \binom{s}{2} + 6\cdot \binom{s}{3} \\
&= s + n(n-1)(n-2) - (n-2)s^2 + (n-2)s + s^3 - 3s^2 + 2s \\
&= s^3 - (n+1)\cdot s^2 + (n+1)\cdot s + n(n-1)(n-2) \\
&=: \textsc{MaxCount}^*(s).
\end{aligned}$$

Obwohl schon MaxCount unsere Anforderungen an eine natürliche Funktion erfüllt, wollen wir hier noch eine Verallgemeinerung von MaxCount, genannt $\textsc{MaxCount}_\alpha$ für $\alpha \in \mathbb{N}$ betrachten. Grund dafür ist, dass das Verhältnis des Funktionswerts des Optimums $(1,\ldots,1)$ von MaxCount gegenüber dem des lokalen Optimums $(0,\ldots,0)$ gegen Eins konvergiert:

$$\frac{\textsc{MaxCount}^*(n)}{\textsc{MaxCount}^*(0)} = \frac{n + n\cdot(n-1)\cdot(n-2)}{n\cdot(n-1)\cdot(n-2)} = 1 + O\left(\frac{1}{n^2}\right).$$

Somit ist die Qualität des lokalen Optimums asymptotisch fast so gut wie die des globalen Optimums. Um MaxCount durch eine kleine Änderung so zu modifizieren, dass der Unterschied zwischen den Funktionswerten des lokalen und des globalen Optimums deutlicher wird, kann man auf die ursprüngliche Eingabe von MAXSAT zurückgehen. Nehmen wir jede Klausel $\{x_i\}$ für $i \in \{1,\ldots,n\}$ nicht nur einmal, sondern α-mal in die Eingabe auf, so werden von der optimalen Variablenbelegung $(1,\ldots,1)$ nun im Vergleich zur bisherigen Eingabe $(\alpha-1)\cdot n$ Klauseln mehr erfüllt, während diese Zahl für die Variablenbelegung $(0,\ldots,0)$ gleich bleibt. Wählen wir den Wert von $\alpha \in \mathbb{N}$ nur so groß, dass die Zahl der Klauseln mit je zwei negativen Literalen weiterhin überwiegt, wird auch diese Eingabe schwierig bleiben.

Formen wir diese neue Eingabe in die Funktionen $\textsc{MaxCount}_\alpha : \{0,1\}^n \mapsto \mathbb{N}$ bzw. $\textsc{MaxCount}^*_\alpha : \{0,\ldots,n\} \mapsto \mathbb{N}$ um, so erhalten wir:

Definition 3.7.3 *Sei $\alpha \in \mathbb{N}$. Die Funktionen $\textsc{MaxCount}_\alpha : \{0,1\}^n \mapsto \mathbb{N}$ und $\textsc{MaxCount}^*_\alpha : \{0,\ldots,n\} \mapsto \mathbb{N}$ sind definiert gemäß*

$$\textsc{MaxCount}_\alpha(x) := \alpha\cdot \sum_{1\le i\le n} x_i + \sum_{i=1}^{n} \sum_{\substack{j=1\\ j\ne i}}^{n} \sum_{\substack{k=1\\ i\ne k\ne j}}^{n} (1-(1-x_i)\cdot x_j\cdot x_k)$$

und

$$\textsc{MaxCount}^*_\alpha(s) := s^3 - (n+1)\cdot s^2 + (n+\alpha)\cdot s + n\cdot(n-1)\cdot(n-2).$$

Natürlich gilt weiterhin, dass $\textsc{MaxCount}_\alpha(x) = \textsc{MaxCount}^*_\alpha(||x||_1)$ für alle $x \in \{0,1\}^n$ ist.

Unser Beweis, dass die Laufzeit des (1+1) EA, aber auch anderer evolutionärer Algorithmen, auf $\textsc{MaxCount}_\alpha$ für noch zu bestimmende Werte $\alpha \in \mathbb{N}$ mit hoher Wahrscheinlichkeit exponentiell groß ist, wird sich auf zwei Eigenschaften stützen:

1. Für Konstanten $\varepsilon_1 < 1/2 < \varepsilon_2$ gilt für jedes $x, y \in \{0,1\}^n$ mit $||x||_1, ||y||_1 \in]\varepsilon_1 \cdot n, \varepsilon_2 \cdot n[$, dass $\textsc{MaxCount}_\alpha(x) \geq \textsc{MaxCount}_\alpha(y)$ gleichbedeutend zu $||x||_1 \leq ||y||_1$ ist.
2. Für jede Konstante $\varepsilon > 0$ ist die Wahrscheinlichkeit, dass der betrachtete Algorithmus einen Punkt mit mindestens $(1/2 + \varepsilon) \cdot n$ Einsen auswertet, exponentiell klein.

Da die erste Eigenschaft die zweite implizieren wird, soll zuerst der größte Wert von α bestimmt werden, so dass die erste Eigenschaft erfüllt ist. Deshalb berechnen wir die Stellen des lokalen Maximums und Minimums von $\textsc{MaxCount}^*_\alpha$, um so den Bereich zu bestimmen, in dem $\textsc{MaxCount}^*_\alpha$ monoton fallend ist. Dazu wird der Definitionsbereich von $\textsc{MaxCount}^*_\alpha$ auf die reellen Zahlen erweitert, um die erste und zweite Ableitung nach s bestimmmen zu können:

$$\begin{aligned}(\textsc{MaxCount}^*_\alpha)' &= 3 \cdot s^2 - 2 \cdot (n+1) \cdot s + (n + \alpha) \text{ und}\\ (\textsc{MaxCount}^*_\alpha)'' &= 6 \cdot s - 2 \cdot (n+1)\end{aligned}$$

Somit sind die Nullstellen der ersten Ableitung

$$s_1 = \frac{n+1}{3} - \sqrt{\left(\frac{n+1}{3}\right)^2 - \frac{n+\alpha}{3}} \text{ und } s_2 = \frac{n+1}{3} + \sqrt{\left(\frac{n+1}{3}\right)^2 - \frac{n+\alpha}{3}}.$$

Da $(\textsc{MaxCount}^*_\alpha(s_1))'' < 0$ und $(\textsc{MaxCount}^*_\alpha(s_2))'' > 0$, ist s_1 Maximalstelle, s_2 Minimalstelle von $\textsc{MaxCount}^*_\alpha$ und $\textsc{MaxCount}^*_\alpha$ in dem Intervall von s_1 bis s_2 monoton fallend. $\textsc{MaxCount}_\alpha$ erfüllt die erste Eigenschaft, wenn gilt:

$$\begin{aligned}\left(\frac{1}{6} + \varepsilon\right) \cdot n &\leq \sqrt{\left(\frac{n+1}{3}\right)^2 - \frac{n+\alpha}{3}}\\ \iff \left(\frac{1}{36} + \frac{\varepsilon}{3} + \varepsilon^2\right) \cdot n^2 &\leq \frac{n^2}{9} + \frac{2n}{9} + \frac{1}{9} - \frac{n}{3} - \frac{\alpha}{3}\\ \iff \frac{\alpha}{3} &\leq \left(\frac{1}{12} - \frac{\varepsilon}{3} - \varepsilon^2\right) \cdot n^2 - \frac{n}{9} + \frac{1}{9}\\ \Longleftarrow \alpha &\leq \left(\frac{1}{4} - \varepsilon - 3\varepsilon^2 - \frac{1}{3n}\right) \cdot n^2.\end{aligned}$$

Wenn also α für eine Konstante $\varepsilon' \in]0, 1/4[$ gleich $(1/4 - \varepsilon') \cdot n^2$ gesetzt wird, so ist die Minimalstelle von $\textsc{MaxCount}^*_\alpha$ größer als $(1/2 + \varepsilon) \cdot n$ für eine Konstante $\varepsilon > 0$ und alle n, die größer als ein $n_0 \in \mathbb{N}$ sind. Da wir ohnehin nur nach einer asymptotischen Aussage streben, wird diese letzte Einschränkung nicht weiter erwähnt. Damit ergibt sich das lokale Maximum s_1 von $\textsc{MaxCount}^*_\alpha$ als

$$s_1 = \frac{n+1}{3} - \sqrt{\left(\frac{n+1}{3}\right)^2 - \frac{n + (1/4 - \varepsilon') \cdot n^2}{3}} \leq \frac{n+1}{3} - \sqrt{\frac{n^2}{9} - \frac{n^2}{12}} = \frac{n}{6} + \frac{1}{3}.$$

(Hierbei gilt die Ungleichung wieder nur für $n \in \mathbb{N}$ einer bestimmten Mindestgröße.) Zwar muss nun $s = 0$ nicht mehr lokales Optimum von $\textsc{MaxCount}^*_\alpha$ sein, doch da schon dessen Funktionswert gleich $n \cdot (n-1) \cdot (n-2)$ ist, sind die Funktionswerte sowohl des globalen als auch des lokalen Optimums von der Größenordnung $\Theta(n^3)$. Um den Quotienten der Funktionswerte des globalen und lokalen Optimums zu berechnen, reicht es also aus, nur jeweils kubisch in n wachsende Terme zu berücksichtigen. Unter dieser Einschränkung gilt

$$\begin{aligned}\textsc{MaxCount}^*_{(1/4-\varepsilon')\cdot n^2}(n) &\propto n^3 - n^3 + \left(\frac{1}{4} - \varepsilon'\right) \cdot n^3 + n^3\\ &= \left(\frac{5}{4} - \varepsilon'\right) \cdot n^3.\end{aligned}$$

Da bei Betrachtung nur linearer Terme $s_1 \propto n \cdot (1/3 - \sqrt{1/9 - (1/4 - \varepsilon')/3}) =: n \cdot \beta$ ist, gilt

$$\textsc{MaxCount}^*_{(1/4-\varepsilon')\cdot n^2}(s_1) \propto \left(1 + \left(\frac{1}{4} - \varepsilon'\right) \cdot \beta - \beta^2 + \beta^3\right) \cdot n^3.$$

Der Quotient

$$\frac{5/4 + \varepsilon'}{1 + (1/4 - \varepsilon') \cdot \beta - \beta^2 + \beta^3}$$

ist für $\varepsilon' = 0$ mit einem Wert $27/22 > 1{,}227$ minimal. Durch geeignete Wahl von $\varepsilon' > 0$ kann man also erreichen, dass das globale Optimum bis zu 22,7 % besser als das lokale Optimum wird. Somit ergibt sich

Korollar 3.7.4 *Für jedes konstante $\alpha \in]0, 1/4[$ ist $\textsc{MaxCount}^*_{\alpha \cdot n^2}$ für eine Konstante $\varepsilon > 0$ und alle $n \in \mathbb{N}$ größer als ein $n_0 \in \mathbb{N}$ in dem Intervall von $n/6$ bis $(1/2 + \varepsilon) \cdot n$ monoton fallend. Wenn der Quotient der Funktionswerte des globalen und lokalen Maximums gleich γ ist, so nähert sich γ für $\alpha \nearrow 1/4$ dem Wert $27/22$ beliebig nahe an.*

Somit wird die Funktion $\textsc{MaxCount}^*_{\alpha \cdot n^2}$ für einen exponentiell großen Anteil der Punkte aus $\{0, 1\}^n$ nur irreführende Hinweise geben, da in diesem Bereich Punkte mit weniger Einsen einen höheren Funktionswert haben, obwohl das Optimum der Punkt $s = n$ ist.

Diese Eigenschaft von $\textsc{MaxCount}^*_{\alpha \cdot n^2}$ und damit von $\textsc{MaxCount}_{\alpha \cdot n^2}$ ist entscheidend dafür, dass der (1+1) EA zur Optimierung von $\textsc{MaxCount}_{\alpha \cdot n^2}$ exponentielle Zeit braucht. Denn mit exponentiell hoher Wahrscheinlichkeit wird der (1+1) EA mit höchstens $(1/2 + \varepsilon/2) \cdot n$ Einsen initialisieren (Anwendung der Tschernoff-Ungleichung (A.9)). Da er keine Verschlechterungen akzeptiert, ist zum Erreichen des globalen Optimums $(1, \ldots, 1)$ notwendig, dass er eine Mutation durchführt, in der mindestens $n \cdot \varepsilon/2$ Einsen gleichzeitig mutieren. Da die Wahrscheinlichkeit hierfür exponentiell klein ist, ist die erwartete Wartezeit und damit die erwartete Laufzeit des (1+1) EA auf $\textsc{MaxCount}_{\alpha \cdot n^2}$ für jeden konstanten Wert von α aus $]0, 1/4[$ exponentiell groß.

Dieser dem Beweis von Lemma 3.4.4 für $\textsc{Trap}$ ähnlichen Argumentation folgt hier kein formal geführter Beweis, da der Nachweis einer unteren Schranke für die Laufzeit im Folgenden für eine allgemeine Klasse von mutationsbasierten evolutionären Algorithmen geführt wird. Dabei betrachten wir alle mutationsbasierten evolutionären Algorithmen zur Maximierung von Funktionen $f : \{0, 1\}^n \mapsto \mathbb{R}$, die folgende Eigenschaften haben:

1. Die Initialisierung der Punkte erfolgt gleichverteilt aus $\{0, 1\}^n$.

2. Die Populationsgröße ist polynomiell in n beschränkt.

3. Zur Erzeugung neuer Punkte aus $\{0, 1\}^n$ wird bit-weise Mutation benutzt, d. h. für eine Mutationsstärke $p_m \in]0, 1/2]$ wird der Nachfolger eines Punkts $x \in \{0, 1\}^n$ dadurch bestimmt, dass jedes Bit unabhängig von den Anderen mit Wahrscheinlichkeit p_m negiert wird. Dabei lassen wir auch variierende Mutationsstärken zu, solange sie für alle Bits gleich sind.

4. Die Selektion kann zur Auswahl der zu mutierenden oder der in die nächste Generation zu übernehmenden Punkte dienen, wobei nur gelten muss, dass für alle $x, y \in \{0, 1\}^n$ mit $f(x) < f(y)$ die Wahrscheinlichkeit, dass x gewählt wird, nicht größer ist, als dass y gewählt wird.

Natürlich ist diese Auswahl der Kriterien subjektiv, doch scheint sie aus folgenden Gründen sinnvoll:

1. Da wir allgemeine evolutionäre Algorithmen betrachten, ist keine Teilmenge des Suchraums von vorneherein als besonders erfolgsversprechend ausgezeichnet. Somit ist eine gleichverteilte Initialisierung die naheliegendste Wahl.

2. Ist die Größe der Population nicht mehr polynomiell, so wird schon die Erzeugung der initialen Generation im schlechtesten Fall (alle Punkte sind verschieden) super-polynomiellen Platz und Zeit benötigen. Damit wäre der Ressourcenverbrauch des Algorithmus für jede Zielfunktion super-polynomiell, was nicht sinnvoll wäre.

3. Dass nur mutationsbasierte evolutionäre Algorithmen betrachtet werden, soll nicht bedeuten, dass Rekombination als unerheblich angesehen wird. Jedoch macht die Rekombination die Analyse wesentlich schwieriger, da Abhängigkeiten zwischen den verschiedenen Individuen berücksichtigt werden müssten. Obwohl $\textsc{MaxCount}_{\alpha\cdot n^2}$ für evolutionäre Algorithmen mit Rekombination schwierig zu optimieren sein scheint, wird dies hier deshalb nicht bewiesen.

   Unsere Einschränkung auf bit-weise Mutation scheint willkürlich. Jedoch ist sie die bei der Behandlung von Bitstrings am häufigsten verwendete Mutation. Anhand eines späteren Beispiels werden wir sehen, wie man die Schwierigkeit von $\textsc{MaxCount}_{\alpha\cdot n^2}$ auch für evolutionäre Algorithmen mit anderen Mutationsoperatoren nachweisen kann.

4. Die letzte Eigenschaft der Selektion spiegelt den Gedanken wider, dass die zu optimierende Zielfunktion in der Nähe von guten Punkten überdurchschnittlich viele andere gute Punkte besitzt, sich also eine Suche in der Nähe der bisher gefundenen guten Punkte auszahlen wird. In diesem Zusammenhang spielt natürlich auch die schon bei der Einführung des (1+1) EA diskutierte Beschränkung auf Mutationsstärken aus dem Intervall $]0, 1/2]$ eine wichtige Rolle: dadurch wird nicht bevorzugt in der Nähe von Komplementen existierender Punkte gesucht.

Unter diesen Voraussetzungen wird ein evolutionärer Algorithmus zur Optimierung von $\textsc{MaxCount}_{\alpha\cdot n^2}$ mit einem gemäß Korollar 3.7.4 gewählten Wert von α exponentielle Zeit benötigen: dazu sei $\varepsilon > 0$ die sich aus α nach diesem Korollar ergebende Konstante. Nach der Initialisierung folgt aus den ersten beiden Eigenschaften des betrachteten Algorithmus mit der Tschernoff-Ungleichung (A.9), dass mit exponentiell hoher Wahrscheinlichkeit alle Suchpunkte höchstens $(1/2+\varepsilon/2)\cdot n$ Einsen haben. Dass nun ein Punkt mit mindestens $(1/2+\varepsilon)\cdot n$ Einsen erzeugt wird (was notwendige Bedingung zum Erreichen des globalen Optimums ist), erfordert, dass von einem Punkt mit höchstens $(1/2 + \varepsilon/2) \cdot n$ Einsen durch eine Folge von Mutationen und Selektionen die Zahl der Einsen um $n \cdot \varepsilon/2$ gesteigert wird. Da Selektionen nach Eigenschaft 4 und Korollar 3.7.4 Punkte mit weniger Einsen bevorzugen, wenn deren Anzahl zwischen $n/6$ und $(1/2+\varepsilon)\cdot n$ liegt, und Mutationen mit hoher Wahrscheinlichkeit nur wenige Bits verändern, ist die Wahrscheinlichkeit, einen Punkt mit $(1/2 + \varepsilon) \cdot n$ Einsen zu erreichen, sehr gering.

Dass sich diese Ideen formal korrekt beweisen lassen, zeigt das folgende Theorem:

Theorem 3.7.5 *Die Wahrscheinlichkeit, dass ein mutationsbasierter evolutionärer Algorithmus das Optimum von* $\textsc{MaxCount}_{\alpha\cdot n^2} : \{0,1\}^n \mapsto \mathbb{N}$ *mit konstantem* $\alpha \in]0, 1/4[$ *nach* $\exp(o(\sqrt{n}))$ *Schritten erreicht hat, beträgt* $\exp(-\Omega(\sqrt{n}))$.

Beweis: Da α eine Konstante aus $]0, 1/4[$ ist, folgt nach der Korollar 3.7.4 vorangehenden Argumentation die Existenz eines $\varepsilon > 0$, so dass $\textsc{MaxCount}_{\alpha\cdot n^2}(x) > \textsc{MaxCount}_{\alpha\cdot n^2}(y)$ für alle $x, y \in \{0,1\}^n$ mit $||x||_1, ||y||_1 \in\ ]n/6, (1/2 + \varepsilon) \cdot n]$

und $||x||_1 < ||y||_1$ ist. Im folgenden sei ein *Erfolg* eingetreten, wenn der evolutionäre Algorithmus einen Punkt mit mindestens $(1/2 + \varepsilon) \cdot n$ Einsen in seiner Population hat. Da dies eine notwendige Bedingung für das Erreichen des Optimums $(1, \ldots, 1)$ ist, folgt die Behauptung, wenn wir zeigen, dass die Wahrscheinlichkeit, nach $\exp(o(\sqrt{n}))$ Schritten einen Erfolg gehabt zu haben, gleich $\exp(-\Omega(\sqrt{n}))$ ist.

Für die Initialisierung ist nach der Tschernoff-Ungleichung (A.9) die Wahrscheinlichkeit, bei der gleichverteilten Auswahl einen Punkt mit mindestens $(1/2+\varepsilon/2) \cdot n$ Einsen zu wählen, nach oben durch $\exp(-n \cdot \varepsilon^2/6)$ beschränkt. Da die Populationsgröße polynomiell ist, ist die Wahrscheinlichkeit, schon in der Initialisierung einen Erfolg zu haben, gleich $\exp(-\Omega(n))$. Ebenso kann man die Wahrscheinlichkeit, dass die Anzahl der Einsen höchstens $(1/2 - \varepsilon/2) \cdot n$ Einsen ist, nach oben durch $\exp(-n \cdot \varepsilon^2/6)$ beschränken. Indem wir im Folgenden annehmen, dass die Anzahl der Einsen aller Punkte der initialen Generation zwischen $(1/2 - \varepsilon/2) \cdot n$ und $(1/2+\varepsilon/2) \cdot n$ liegt, machen wir also nur mit exponentiell geringer Wahrscheinlichkeit einen Fehler.

Wir werden nun zwei vereinfachende Annahmen machen, die jeweils die Wahrscheinlichkeit eines Erfolgs nur erhöhen. Wenn diese auch dann nur $\exp(-\Omega(\sqrt{n}))$ beträgt, haben wir die Behauptung nachgewiesen.

Die erste Annahme ist, dass alle Punkte mit weniger als $(1/2 + \varepsilon/2) \cdot n$ Einsen zu Punkten mit genau $(1/2 + \varepsilon/2) \cdot n$ Einsen verändert werden. Da die Mutationsstärke p_m des betrachteten evolutionären Algorithmus höchstens 1/2 ist, kann dies die Wahrscheinlichkeit, Punkte mit mehr Einsen zu erreichen, und somit die eines Erfolgs nicht verkleinern.

Die zweite Annahme ist, dass die Wahrscheinlichkeit der Selektion für alle Punkte, deren Anzahl von Einsen zwischen $(1/2 + \varepsilon/2) \cdot n$ und $(1/2 + \varepsilon) \cdot n$ Einsen liegt, gleich ist. Punkte mit weniger als $(1/2 + \varepsilon/2) \cdot n$ Einsen werden nach der ersten Annahme zu Punkten mit genau $(1/2 + \varepsilon/2) \cdot n$ Einsen. Wird also ein neuer Punkt erzeugt, so wird, wenn zwischen ihm und seinem Vorgänger selektiert wird, gleichverteilt ausgewählt. Da die Selektion zuvor Punkte mit weniger Einsen in diesem Bereich entweder ebenfalls gleichberechtigt oder bevorzugt ausgewählt hat, ist die Wahrscheinlichkeit eines Erfolges damit ebenfalls gestiegen.

Wenn unter diesen Annahmen ein Erfolg eintritt, so muss es eine Folge von Punkten $x_0, \ldots, x_t \in \{0, 1\}^n$ geben, so dass

1. $||x_0||_1 = (1/2 + \varepsilon/2) \cdot n$,
2. $\forall i \in \{1, \ldots, t-1\}$: $||x_i||_1 \in \{(1/2 + \varepsilon/2) \cdot n + 1, \ldots, (1/2 + \varepsilon) \cdot n - 1\}$,
3. $||x_t||_1 = (1/2 + \varepsilon) \cdot n$ und
4. x_{i+1} jeweils durch Mutation aus x_i hervorgegangen ist ($i \in \{0, \ldots, t-1\}$).

Wir werden nun zeigen, dass die Wahrscheinlichkeit einer solchen Folge exponentiell klein ist (ein ähnlicher Ansatz wurde von Rabani, Rabinovich und Sinclair (1995) benutzt, um die Anzahl von Generationen zu bestimmen, nach denen sich in einem rein rekombinationsbasierten evolutionären Algorithmus ein stationärer Zustand einstellt). Dazu ist die Beobachtung entscheidend, dass die Folge der Punkte $x_0, \ldots, x_{t-1}$ mindestens $(1/2 + \varepsilon/2) \cdot n \cdot t$ Einsen enthält. Weiterhin ist es für einen Erfolg notwendig, dass in dieser Folge mindestens $n \cdot \varepsilon/2$ mehr Nullen als Einsen mutieren, weil ansonsten die Anzahl der Einsen nicht um $n \cdot \varepsilon/2$ von x_0 zu x_t zunehmen kann.

Die Wahrscheinlichkeit eines Erfolgs kann durch die Wahrscheinlichkeit des Ereignisses nach oben abgeschätzt werden, dass höchstens $ntp_m/2$ Einsen oder mindestens $ntp_m/2$ Nullen mutieren. Denn dass dies nicht eintritt, ist eine hinreichende Bedingung dafür, dass kein Erfolg eintritt. Wenn die Zufallsvariablen X^1 bzw. X^0

die Zahl der mutierenden Einsen bzw. Nullen angeben, so wollen wir

$$P(X^1 \leq \frac{ntp_m}{2}) \text{ und } P(X^0 \geq \frac{ntp_m}{2})$$

nach oben abschätzen. Da sich X^1 bzw. X^0 als Summe von mindestens $(1/2+\varepsilon/2)\cdot n\cdot t$ bzw. höchstens $(1/2-\varepsilon/2)\cdot n\cdot t$ unabhängigen $\{0,1\}$-wertigen Zufallsvariablen mit Erwartungswert p_m ergeben, liefert die Tschernoff-Ungleichung (A.9)

$$P\left(X^1 \leq \frac{ntp_m}{2}\right) = P\left(X^1 \leq \left(1-\frac{\varepsilon}{1+\varepsilon}\right)\cdot\frac{(1+\varepsilon)\cdot ntp_m}{2}\right) \leq \exp\left(-\frac{\varepsilon^2\cdot ntp_m}{4(1+\varepsilon)}\right).$$

Analog folgt:

$$P\left(X^0 \geq \frac{ntp_m}{2}\right) = P\left(X^0 \geq \left(1+\frac{\varepsilon}{1-\varepsilon}\right)\cdot\frac{(1-\varepsilon)\cdot ntp_m}{2}\right) \leq \exp\left(-\frac{\varepsilon^2\cdot ntp_m}{6(1+\varepsilon)}\right).$$

Beide Ereignisse haben also eine Wahrscheinlichkeit von $\exp(-\Omega(ntp_m))$. Ob diese Größe exponentiell schnell in n sinkt, hängt von dem Wert von p_m ab. Ist $p_m = \Omega((t\sqrt{n})^{-1})$, so ist die Wahrscheinlichkeit beider Ereignisse jeweils $\exp(-\Omega(\sqrt{n}))$. Ist p_m jedoch kleiner, also von der Größenordnung $o((t\sqrt{n})^{-1})$, so ist $ntp_m/2 = o(\sqrt{n})$, d. h. die erwartete Anzahl mutierender Nullen ist $o(\sqrt{n})$. Um einen Erfolg zu haben, müssen jedoch mindestens $n\cdot\varepsilon/2$ Nullen mutieren. Dies hat dann nach der Tschernoff-Ungleichung (A.9) ebenfalls Wahrscheinlichkeit $\exp(-\Omega(n))$. Somit ist die Erfolgswahrscheinlichkeit von der Größenordnung $\exp(-\Omega(\sqrt{n}))$. Selbst wenn der evolutionäre Algorithmus $\exp(o(\sqrt{n}))$ Generationen durchläuft, bleibt die Erfolgswahrscheinlichkeit von dieser Größenordnung. □

Da der (1+1) EA die Anforderungen, die wir an mutationsbasierte evolutionäre Algorithmen gestellt haben, erfüllt, folgt direkt aus Theorem 3.7.5:

Korollar 3.7.6 *Mit einer Wahrscheinlichkeit* $\exp(-\Omega(\sqrt{n}))$ *hat der* (1+1) EA *nach* $\exp(o(\sqrt{n}))$ *Schritten das Optimum der Funktion* $\text{MaxCount}_{\alpha\cdot n^2}: \{0,1\}^n \mapsto \mathbb{R}$ *mit beliebigem konstantem* $\alpha \in]0,1/4[$ *gefunden. Somit ist die erwartete Laufzeit des* (1+1) EA *auf* $\text{MaxCount}_{\alpha\cdot n^2}$ *exponentiell groß.*

Die Funktion $\text{MaxCount}_{\alpha\cdot n^2}$ scheint für alle konstanten Werte von $\alpha \in]0,1/4[$ für eine noch allgemeinere Klasse von Algorithmen schwierig zu sein: denn wie beim Beweis von Theorem 3.7.5 gezeigt, ist der Anteil der Punkte mit mindestens $(1/2-\varepsilon)\cdot n$ und höchstens $(1/2+\varepsilon)\cdot n$ für jede Konstante $\varepsilon > 0$ exponentiell groß. Zu jedem konstantem $\alpha \in]0,1/4[$ gibt es aber nach Korollar 3.7.4 eine Wahl von ε, so dass in dem angesprochenen Bereich Punkte mit weniger Einsen stets einen höheren Zielfunktionswert haben als solche mit mehr Einsen. Um also das Optimum $(1,\ldots,1)$ zu erreichen, müßte der Algorithmus einen exponentiell kleinen Teil des Suchraums untersuchen, obwohl alle Hinweise der Funktion in die andere Richtung zeigen. Dies sollte eigentlich kein sinnvolles allgemeines Suchverfahren tun.

Natürlich kann eine solche allgemeine Aussage nicht bewiesen werden. Wir versuchen, sie trotzdem zu erhärten, indem wir zeigen, dass auch der Dynamic EA zur Optimierung von $\text{MaxCount}_{\alpha\cdot n^2}$ mit hoher Wahrscheinlichkeit exponentiell viele Schritte benötigt und dies unabhängig von dem Selektionsschema:

Theorem 3.7.7 *Sei* $\alpha \in]0,1/4[$ *eine Konstante. Mit einer Wahrscheinlichkeit von* $1-\exp(-\Omega(n))$ *hat jeder* Dynamic EA *nach* $\exp(o(n))$ *Schritten das Optimum der Funktion* $\text{MaxCount}_{\alpha\cdot n^2}$ *nicht erreicht.*

Beweis: Sei $\varepsilon > 0$ die nach Korollar 3.7.4 existierende positive Konstante, so dass $\text{MaxCount}^*_{\alpha\cdot n^2}$ in dem Bereich von $n/6$ bis $(1/2+\varepsilon)\cdot n$ monoton sinkt. Die Initialisierung kann wie im Beweis von Theorem 3.7.5 behandelt werden, so dass die

Wahrscheinlichkeit, in einem Punkt mit mindestens $(1/2 - \varepsilon/2) \cdot n$ und höchstens $(1/2 + \varepsilon/2) \cdot n$ Einsen zu initialisieren, exponentiell groß ist. Analog nehmen wir wiederum an, dass bei der Initialisierung dieser Fall eingetreten ist, und dass als ein *Erfolg* schon das Erreichen eines Punkts mit $(1/2 + \varepsilon) \cdot n$ Einsen gilt.

Der Dynamic EA wählt nun gleichverteilt zufällig ein Bit des aktuellen Punkts $x \in \{0,1\}^n$ aus, negiert dieses und übernimmt den Nachfolger $y \in \{0,1\}^n$ mit Wahrscheinlichkeit $p(x,y) \in [0,1]$. Dabei ist $p(x,y) = 1$, falls $||x||_1 \geq ||y||_1$, und ansonsten echt kleiner als Eins. Somit vergrößern wir die Wahrscheinlichkeit, die Zahl der Einsen zu erhöhen, wenn wir $p(x,y)$ stets auf Eins setzen.

Da mit höchstens $(1/2+\varepsilon/2)\cdot n$ Einsen initialisiert wird, muss es zum Erreichen eines Erfolges eine Folge $x_0, \ldots, x_t \in \{0,1\}^n$ von aufeinanderfolgen Punkten des Dynamic EA geben, so dass

1. $||x_0||_1 = (1/2 + \varepsilon/2) \cdot n$,
2. $\forall i \in \{1, \ldots, t-1\}$: $||x_i||_1 \in \{(1/2+\varepsilon/2)\cdot n + 1, \ldots, (1/2+\varepsilon)\cdot n - 1\}$ und
3. $||x_t||_1 = (1/2 + \varepsilon) \cdot n$ ist.

Für alle Punkte $x_0, \ldots, x_{t-1}$ ist die Wahrscheinlichkeit, im Mutationsschritt eine Null zur Mutation auszuwählen, höchstens $1/2 - \varepsilon/2$. Die Wahrscheinlichkeit einer solchen Folge wird also überschätzt, wenn wir stets genau die Wahrscheinlichkeit $1/2 - \varepsilon/2$ annehmen. Da sich die Zahl der Einsen am Ende der Folge um $n \cdot \varepsilon/2$ erhöht hat, muss es mindestens $t/2 + n \cdot \varepsilon/4$ Mutationen von Nullen zu Einsen in der Folge geben. Die erwartete Anzahl von Null zu Eins mutierender Bits ist jedoch nur höchstens $t/2 - t \cdot \varepsilon/2$. Mit der Tschernoff-Ungleichung (A.9) kann man zeigen, dass die Wahrscheinlichkeit eines Erfolges sehr klein ist.

Dazu muss natürlich $t \geq n \cdot \varepsilon/4$ sein, da es in einer Folge kürzerer Länge nicht mindestens $n \cdot \varepsilon/4$ Mutationen von Nullen geben kann. Sei also $t \geq n \cdot \varepsilon/4$. Wenn X^0 die Anzahl der Mutationen von Nullen in t Schritten ist, so gilt

$$\mathrm{P}\left(X^0 \geq \frac{t}{2} + \frac{n \cdot \varepsilon}{4}\right) \leq \mathrm{P}\left(X^0 \geq \frac{t}{2}\right) \leq \mathrm{P}\left(X^0 \geq \left(1 + \frac{\varepsilon}{1-\varepsilon}\right) \cdot \left(\frac{t}{2} - \frac{t\varepsilon}{2}\right)\right).$$

Letzterer Ausdruck ist nach der Tschernoff-Ungleichung (A.9) exponentiell klein, d. h. $\exp(-\Omega(n))$. Somit ist auch nach $\exp(o(n))$ Schritten die Wahrscheinlichkeit, dass eine solche für einen Erfolg notwendige Folge aufgetreten ist, exponentiell klein.

□

Somit kennen wir für zwei Typen von evolutionären Algorithmen Beweise, dass ihre Laufzeit zur Optimierung von $\text{MaxCount}_{\alpha \cdot n^2}$ mit exponentiell großer Wahrscheinlichkeit exponentiell groß ist. Für andere Algorithmen haben wir Gründe angeführt, weshalb auch für sie diese Funktion schwierig sein sollte.

Dass es eine solche Funktion gibt, ist nicht überraschend. Doch die Funktion $\text{MaxCount}_{\alpha \cdot n^2}$ beschreibt direkt eine Instanz eines der bekanntesten Probleme aus der Komplexitätstheorie, MAXSAT. Weiterhin ist diese Funktion ein Polynom dritten Grades. Beides spricht dafür, dass man $\text{MaxCount}_{\alpha \cdot n^2}$ als ein natürliches Problem bezeichnen kann, wenn man sie z. B. mit den Funktionen Valley oder Jump_m vergleicht. Sieht man dieses Resultat im Kontext des NFL-Theorems und der Diskussion um die Grenzen und Möglichkeiten allgemeiner Suchverfahren, so ergibt sich, dass auch im beschriebenen Sinne natürliche Funktionen falsche Hinweise geben können, so dass allgemeine Suchverfahren scheitern müssen.

Kapitel 4

Zum Entwurf evolutionärer Algorithmen

In Kapitel 2 haben wir gezeigt, dass jedes allgemeine Suchverfahren und damit jeder evolutionäre Algorithmus nicht auf allen Zielfunktionen effizient arbeiten kann. Jedes Suchverfahren wird die Informationen, die es durch Funktionsauswertungen über die Zielfunktion bekommt, nutzen, um anhand dieser die nächsten auszuwertenden Punkte ggf. randomisiert zu bestimmen. Nur wenn die Zielfunktion so beschaffen ist, dass diese Punkte den Suchprozess in der Regel „zum Optimum hinführen", wird das Suchverfahren schnell das Optimum erreichen können. Natürlich stellt sich somit die Frage, welche Zielfunktionen welche Suchverfahren schnell zum Optimum hinführen.

In Kapitel 3 haben wir verschiedene Funktionen bzw. Funktionsklassen daraufhin untersucht, ob sie von einem einfachen evolutionären Algorithmus, dem (1+1) EA, bzw. Varianten hiervon schnell optimiert werden. Hierbei wurde die erwartete Laufzeit des (1+1) EA asymptotisch genau analysiert. Dabei haben wir oft gesehen, dass die anschauliche Vorstellung von „guten Hinweisen", d. h. erfolgreichen Mutationen, die den Hamming-Abstand zum Optimum verringern, eine Einschätzung der Laufzeit bzw. die Konstruktion von Funktionen mit bestimmten Eigenschaften leichter machen kann.

In diesem Kapitel soll nicht die Analyse eines bestimmten evolutionären Algorithmus, sondern das konträre Problem, zu einer Zielfunktion einen passenden evolutionären Algorithmus zu entwerfen, der ihren Hinweisen „folgen kann", im Vordergrund stehen. Diese Probleme sind insofern konträr, als die Ausgangspunkte verschieden sind: bei der Analyse war es der (1+1) EA, hier wird es die Zielfunktion sein. Jedoch beeinflussen sich beide Ansätze natürlich stark.

Beim Entwurf muss davon ausgegangen werden, dass über die Zielfunktion mehr Wissen als im Black-Box Szenario vorhanden ist. Denn nur wenn wir dieses *problemspezifische Wissen* haben, können wir hoffen, einen passenden Algorithmus für die Zielfunktion finden zu können. Natürlich kann es auch bei willkürlicher Wahl eines Algorithmus sein, dass dieser zufällig für die Zielfunktion geeignet ist. Auf diesen Zufall wollen wir uns aber nicht verlassen. In diesem Kapitel werdeen Richtlinien vorgeschlagen, mit denen zielgerichtet aus Wissen über die Zielfunktion ein evolutionärer Algorithmus konstruiert werden kann.

Dabei gehen wir davon aus, dass sich dieses Wissen in Form einer Metrik auf dem Suchraum darstellen lässt, so dass bzgl. der Metrik nicht weit entfernte Punkte nur einen geringen Unterschied im Zielfunktionswert aufweisen. Weiteres zusätzliches Wissen ist nicht notwendig. Unter diesen Voraussetzungen werden wir Richtlinien angeben, wie sich Mutation und Rekombination bzgl. dieser Metrik im Suchraum auswirken sollen. Ein evolutionärer Algorithmus, der diese Richtlinien erfüllt, wird deshalb *Metrik-basierter evolutionärer Algorithmus* (*MBEA*) genannt.

Um die Zweckmäßigkeit der einen MBEA definierenden Richtlinien an einem Beispiel darlegen zu können, wird ein System der genetischen Programmierung entworfen, dass diesen genügt. Die genetische Programmierung stellt ein Teilparadigma evolutionärer Algorithmen dar, in dem die aufgestellten Richtlinien oftmals nicht eingehalten werden. Das entworfene System benutzt *Ordered Binary Decision Diagrams* (*OBDDs*), eine besonders effiziente Datenstruktur zu Darstellung und Manipulation boolescher Funktionen. Diese ermöglicht effiziente genetische Operatoren, die den MBEA-Richtlinien genügen. Durch einen empirischen Vergleich des so entworfenen Systems mit bestehenden GP-Systemen und einem System, das OBDDs benutzt, jedoch nicht den MBEA-Richtlinien entspricht, zeigt sich zumindest für die getesteten Benchmark-Probleme die Überlegenheit des MBEA-konformen GP-Systems.

Die MBEA-Richtlinien stellen einen Vorschlag dar, wie sich evolutionäre Algorithmen auf einer Zielfunktion verhalten sollen, damit sie von der Funktion zum Optimum geleitet werden können. Dabei stellen sie eine Formalisierung der Wirkungsweise von Mutation und Rekombination dar, die, wenn auch nicht explizit ausgesprochen, den meisten evolutionären Algorithmen zugrundeliegt. Sie sind durch die Metrik auf die Zielfunktion abgestimmt. Dies ist bei den Standardformen evolutionärer Algorithmen (siehe Abschnitt 1.2) nicht der Fall, wo implizit oft angenommen wird, dass die Zielfunktion bestimmte Eigenschaften, wie Stetigkeit, besitzt. Arbeitet die Zielfunktion nicht originär auf einem der dort verwendeteten Suchräume, muss erst eine Umkodierung stattfinden, die diese Eigenschaften oftmals zerstört. In diesem Fall werden die Standardformen evolutionärer Algorithmen häufig schlecht arbeiten.

Natürlich ist das Problem des Entwurfs evolutionärer Algorithmen zentral, doch bestehen die bisherigen Ansätze hauptsächlich darin, sich an einer der Standardformen evolutionärer Algorithmen zu orientieren, die Probleme erfolgreich gelöst haben, die dem eigenen „ähnlich“ sind, ohne dies näher zu formalisieren. Da die theoretischen Grundlagen evolutionärer Algorithmen zu schwach sind, können sie nur in Teilaspekten den Entwurf erleichtern und auch das in der Regel nur, wenn eine der Standardformen evolutionärer Algorithmen benutzt wird.

Die empfehlenswerte Berücksichtigung von Problemwissen in evolutionären Algorithmen wird häufig durch die Anpassung der genetischen Operatoren an das Problem bewerkstelligt (siehe z. B. Tao und Michalewicz (1998)). Eine formal definierte Grundlage, nach welchen Prinzipien dies erfolgen kann, fehlt aber. Die MBEA-Richtlinien sollen einen Schritt in diese Richtung bilden, wobei nicht der Anspruch der Vollständigkeit oder einer theoretischen Begründung erhoben wird.

4.1 Metrik-basierte evolutionäre Algorithmen

In diesem Abschnitt stellen wir Richtlinien vor, an denen sich der Entwurf evolutionärer Algorithmen orientieren kann. Dabei werden diese nicht nur umgangssprachlich, sondern formal spezifiziert, um so konkret wie möglich zu sein. Dazu ist zuerst eine formale Beschreibung der Module eines evolutionären Algorithmus notwendig.

4.1.1 Eine formale Beschreibung evolutionärer Algorithmen

Notwendig, aber auch hinreichend zur Anwendung eines evolutionären Algorithmus ist die Kenntnis einer *Zielfunktion* $f : G \mapsto W$, so dass ein Element $g \in G$ gefunden werden soll, dessen Zielfunktionswert $f(g)$ möglichst groß ist (dass der Suchraum nicht mehr wie bisher als S bezeichnet wird, wird weiter unten begründet). Dies setzt voraus, dass die Menge W zumindest partiell geordnet ist, es also eine reflexive und transitive Relation $R \subseteq W \times W$ gibt, so dass $(w_1, w_2) \in R$ impliziert, dass w_2 mindestens so groß wie w_1 ist.

Wir nehmen sogar stets an, dass W total geordnet ist, d. h. für jedes Paar $(w_1, w_2) \in W \times W$ mit $w_1 \neq w_2$ gilt entweder $(w_1, w_2) \in R$ oder $(w_1, w_2) \in R$. Die Aufgabe, für die wir evolutionäre Algorithmen untersuchen, besteht darin, ein globales Optimum $g^* \in G$ von f zu finden, so dass für alle $g \in G$ gilt $f(g^*) \geq f(g)$.

Ein evolutionärer Algorithmus verwaltet in diskreten Schritten, die im Folgenden mit $t \in \mathbb{N}_0$ durchnummeriert werden, jeweils eine Population, d. h. eine Multimenge $P_t \subseteq A$. Dabei ist die Größe von P_t eine fest gewählte von t unabhängige Konstante N. Zu Beginn wird die Startpopulation P_0 auf zufällige Weise initialisiert. Aus der Population P_t wird dann durch Anwendung von Mutation, Rekombination und Selektion eine Nachkommenpopulation P'_t generiert. Aus der Vereinigung von P_t und P'_t wird per Selektion die neue Population P_{t+1} ausgewählt und eine neue Generation beginnt.

Dabei haben Mutation, Rekombination und Selektion den folgenden Aufbau: wenn (P_m, Ω_m), (P_r, Ω_r) und (P_s, Ω_s) drei Wahrscheinlichkeitsräume sind, so lässt sich die Mutation M als eine Abbildung

$$m : G \times \Omega_m \mapsto G$$

beschreiben: aus einem Element aus G erzeugt sie unter zufälliger Einflussnahme ein neues Element aus G, wobei $\mathrm{P}_m(m(g) = g') \in [0, 1]$ die Wahrscheinlichkeit bezeichnet, mit der g zu dem Element g' mutiert wird:

$$\mathrm{P}_m(m(g) = g') := \mathrm{P}_m(\{\omega \in \Omega_m \mid m(g, \omega) = g'\}).$$

Analog lässt sich die Rekombination r als eine Abbildung

$$r : G \times G \times \Omega_r \mapsto G$$

beschreiben. Hierbei wird aus zwei Elementen aus G unter zufälligem Einfluss ein neues generiert. Ähnlich zur Mutation sei $\mathrm{P}_r(r(g_1, g_2) = g') \in [0, 1]$ die Wahrscheinlichkeit, mit der aus den Eltern g_1 und g_2 der Nachkomme g' durch den Rekombinationsoperator r generiert wird:

$$\mathrm{P}_r(r(g_1, g_2) = g') := \mathrm{P}_r(\{\omega \in \Omega_r \mid r(g_1, g_2, \omega) = g'\}).$$

Die Selektion s generiert aus einer Population von Individuen mithilfe ihrer Fitness und zufälligen Entscheidungen eine neue Population. Dabei fließen nur die Fitnesswerte der Individuen der Population ein, die Fitnessfunktion an sich bzw. die Fitnesswerte von Individuen, die nicht in der Population sind, spielen keine Rolle:

$$s : \{\{(g_1, f(g_1)), \ldots, (g_{N_1}, f(g_{N_1}))\} \mid \{g_1, \ldots, g_{N_1}\} \subseteq G\} \times \Omega_s \mapsto$$

$$\{\{g_1, \ldots, g_{N_2}\} \mid \{g_1, \ldots, g_{N_2}\} \subseteq \{g_1, \ldots, g_{N_1}\}\}.$$

(Hierbei sind die Mengen von Suchpunkten genau wie die Populationen als Multimengen aufzufassen, da es nicht ausgeschlossen ist, dass ein Element öfter als einmal in der Population auftaucht bzw. selektiert wird.)

Die Werte von N_1 und N_2 hängen von den verschiedenen Situationen ab, in denen die Selektion auftritt. Allen Varianten der Selektion ist jedoch gemein, dass in ihnen die Zielfunktion eine entscheidende Rolle spielt, wohingegen dies bei den anderen hier erwähnten genetischen Operatoren Mutation und Rekombination in der Regel nicht der Fall ist. Dabei werden Individuen mit größerem Zielfunktionswert eine höhere Wahrscheinlichkeit haben, durch Selektion gewählt zu werden, als solche mit niedrigerem Zielfunktionswert.

Diese knappe Formalisierung der drei wichtigsten Module evolutionärer Algorithmen muss zwangsläufig unvollständig sein, da es viele Varianten gibt, die sich nicht in das beschriebene formale Konzept pressen lassen. Weil sich aber alle in dieser Arbeit besprochenen evolutionären Algorithmen durch diese Operatoren beschreiben lassen, wählen wir diese Formalisierung, um die Richtlinien für evolutionäre Algorithmen zu beschreiben. (Für eine wesentlich genauere Formalisierung siehe Bäck (1996).)

4.1.2 Die MBEA-Richtlinien

Bevor nun die Richtlinien, die insgesamt das Konzept eines Metrik-basierten evolutionären Algorithmus bilden, genau vorgestellt und motiviert werden, muss zuerst die zugrundeliegende Metrik definiert werden. Es ist klar, dass Zielfunktionen, die eigentlich auf einem anderen Raum als G definiert sind, stets auf diesen umkodiert werden können, wobei sich die Form der Zielfunktion jedoch entscheidend ändern kann. Deshalb ist es zur Klarheit sinnvoll, die Begriffe des *Genotyp*- und *Phänotyp*-Raums einzuführen. Der Genotyp-Raum G ist die Menge der Kodierungen der Elemente des Suchraums, während der Phänotyp-Raum P die Menge der abstrakten Objekte des Suchraums ist. Eine solche Unterscheidung macht z. B. ebenfalls Sinn, wenn ein Element des Suchraums mehrere Kodierungen hat. Die Zielfunktion wird, wenn das Problem nur abstrakt gegeben ist, auf dem Phänotyp-Raum definiert sein. Da der evolutionäre Algorithmus aber auf dem Genotyp-Raum arbeitet, muss erst eine Umkodierung stattfinden.

Dazu ein Beispiel: angenommen, das *Travelling Salesperson Problem* (*TSP*) (Papadimitriou (1994)) soll gelöst werden. Eine Instanz besteht dann aus der Menge der Wegkosten zwischen allen Kombinationen von zwei beliebigen der insgesamt n Städte. Die Zielfunktion wird einer Rundreise von Städten die Summe der Wegkosten zwischen den benachbarten Städten in dieser Rundreise zuordnen. Eine mögliche Kodierung, d. h. ein möglicher Genotyp-Raum ist die Menge

$$G := \{(\pi_1, \ldots, \pi_n) \,|\, \pi_1, \ldots, \pi_n \in \{1, \ldots, n\} \text{ und } \{\pi_1\} \cup \cdots \cup \{\pi_n\} = \{1, \ldots, n\}\}$$

der Permutationen der Zahlen aus $\{1, \ldots, n\}$, wobei π_i die Nummer der i-ten besuchten Stadt auf der Rundreise ist. Doch diese Darstellung hat den Nachteil, dass z. B. ein k-Punkt-Crossover zwischen zwei Permutationen in der Regel keine Permutation ergeben wird. Man kann jedoch zur Darstellung auch Elemente aus

$$G' := \{1, \ldots, n\} \times \{1, \ldots, n-1\} \times \cdots \times \{1, 2\} \times \{1\}$$

verwenden, wobei ein $g \in G'$ folgendermaßen in eine Permutation $\pi \in G$ umgeformt werden kann: beginne mit $M := \{1, \ldots, n\}$ und $i := 1$ und wähle das g_i-te Element aus M als π_i. Streiche g_i aus M, erhöhe i um Eins und wiederhole diesen Vorgang, bis $i = n$ ist. Mit dieser Darstellung wird jedes k-Punkt-Crossover wieder ein Element aus G' erzeugen.

In beiden Fällen ist der Phänotyp-Raum gleich, jedoch wird er auf unterschiedliche Weisen kodiert, d. h. jeweils ein anderer Genotyp-Raum gewählt. Obwohl eine formale Beschreibung der Zielfunktion des TSP natürlich in beiden Fällen gleich ist, wenn sie die Kodierung der Permutation nicht in Betracht zieht, kann es ganz praktische Gründe geben, einen Genotyp-Raum einem anderen vorzuziehen. (Ein vielleicht noch deutlicheres Beispiel wird bei der Besprechung der genetischen Programmierung vorgestellt, die Darstellung von booleschen Funktionen). Die Kodierung der Elemente des Phänotyp-Raums P sei durch die *Kodierungsfunktion* $h : P \mapsto G$ beschrieben.

Wir gehen davon aus, dass die Zielfunktion auf dem Phänotyp-Raum definiert und eine *Metrik* $m_P : P \times P \mapsto \mathbb{R}_0^+$ auf dem Phänotyp-Raum bekannt ist, d. h. für m_P gilt:

1. $\forall p_1, p_2 \in P : m_P(p_1, p_2) = 0 \iff p_1 = p_2$,
2. $\forall p_1, p_2 \in P : m_P(p_1, p_2) = m_P(p_2, p_1)$,
3. $\forall p_1, p_2, p_3 \in P : m_P(p_1, p_2) + m_P(p_2, p_3) \geq m_P(p_1, p_3)$.

Eine Metrik kann als ein sinnvolles Abstandsmaß angesehen werden. Wir verlangen von der Metrik, dass sie auf die Zielfunktion $f : P \mapsto \mathbb{R}$ so abgestimmt ist, dass bzgl. m_P benachbarte Punkte einen nur geringen Zielfunktionsunterschied aufweisen können, d. h. gelten soll:

$$\forall p_1, p_2 \in P : |f(p_1) - f(p_2)| \leq m_P(p_1, p_2). \qquad (\Diamond)$$

(Diese Voraussetzung ist weniger restriktiv als es scheinen mag, da mit m_P auch jede Funktion $c \cdot m_P$ für $c > 0$ eine Metrik auf P bildet.)

Dies ist keine Richtlinie, da dieses Wissen entweder bekannt ist oder nicht, sondern eine Voraussetzung, damit die folgenden Richtlinien sinnvoll sein können. Denn mithilfe einer solchen Metrik m_P ist erst eine genauere Formalisierung dessen, was man unter einer gute Hinweise gebenden Zielfunktion verstehen kann, möglich. Wertet der Algorithmus die Zielfunktion an bestimmten Punkten aus, so wird durch die Funktionswerte im Zusammenspiel mit dem Algorithmus eine Wahrscheinlichkeitsverteilung über die nächsten auszuwertenden Punkte vorgegeben. Nur wenn die Zielfunktion so beschaffen ist, dass mit relativ hoher Wahrscheinlichkeit viele dieser Punkte den Algorithmus zum Optimum hinführen, wird sie effizient von ihm optimierbar sein. Ob aber die Auswahl bestimmer Punkte zum Optimum hinführt, hängt wiederum von dem Algorithmus ab.

Die Metrik m_P wird nun genutzt, um Richtlinien für Mutation und Rekombination aufzustellen, so dass der evolutionäre Algorithmus den von m_P definierten Nachbarschaftsbegriff berücksichtigt und somit auf die Zielfunktion abgestimmt arbeitet. Zuvor muss jedoch die Metrik d_P auf den Genotyp-Raum G übertragen werden, da Mutation und Rekombination auf G operieren. Ist die Kodierungsfunktion $h : P \mapsto G$ bijektiv, so ist

$$d_G : G \times G \mapsto \mathbb{R}_0^+ \text{ mit } d_G(g_1, g_2) := d_P(h^{-1}(g_1), h^{-1}(g_2)) \qquad (\triangle)$$

eine Metrik auf G, wie leicht zu zeigen ist. Damit die Auswirkungen der Richtlinien für Mutation und Rekombination in G auf die Zielfunktion formalisiert werden können, muss die Übertragung der Zielfunktion f auf $f' : G \to \mathbb{R}$ gemäß $f'(g) := f(h^{-1}(g))$ betrachtet werden. Diese definiert den Zielfunktionswert eines Elements aus dem Genotyp-Raum. Damit folgt für die Metrik d_G, dass für alle $g_1, g_2 \in G$ gilt:

$$|f'(g_1) - f'(g_2)| = |f(h^{-1}(g_1)) - f(h^{-1}(g_2))| \leq d_P(h^{-1}(g_1), h^{-1}(g_2)) = d_G(g_1, g_2).$$

Ist die Kodierungsfunktion bijektiv, überträgt sich somit die Eigenschaft der Metrik d_P, dass benachbarte Elemente ähnliche Funktionswerte haben, auf die Metrik d_G. Deshalb wird die erste Richtlinie gerade dies fordern:

Richtlinie 4.1.1 (K 1) *Die Kodierungsfunktion $h : P \mapsto G$ ist bijektiv.*

Der Mutationsoperator in evolutionären Algorithmen sollte es möglich machen, beliebige Punkte des Genotyp-Raums zu erreichen, damit es keine Bereiche gibt, die von vorneherein von der Suche ausgeschlossen sind. Dies wird die erste Richtlinie an die Mutation ausdrücken:

Richtlinie 4.1.2 (M 1) *Die Mutation $m : G \times \Omega_m \mapsto G$ erfüllt*

$$\forall g_1, g_2 \in G : \mathrm{P}_m(m(g_1) = g_2) > 0.$$

Die zweite Richtlinie drückt die Vorstellung aus, dass die Mutation kleine Veränderungen gegenüber großen bevorzugen soll. Implizit steht dahinter natürlich, dass kleine Veränderungen auch nur den Zielfunktionswert wenig verändern und somit in der Nachbarschaft guter Punkte Verbesserungen finden. Erst durch die Metrik d_G und ihren Zusammenhang zur Zielfunktion lässt sich diese Vorstellung auch durch die folgende Richtlinie erfüllen:

Richtlinie 4.1.3 (M 2) *Die Mutation* $m : G \times \Omega_m \mapsto G$ *erfüllt*

$$\forall g_1, g_2, g_3 \in G \text{ mit } d_G(g_1, g_2) < d_G(g_1, g_3) : \mathrm{P}_m(m(g_1) = g_2) > \mathrm{P}_m(m(g_1) = g_3).$$

Die dritte Richtlinie für Mutationen drückt aus, dass die Mutation abgesehen von der Entfernung zum aktuellen Punkt keine Suchrichtung bevorzugen sollte, da in evolutionären Algorithmen allein die Selektion für die Steuerung des Suchprozesses verantwortlich ist. Da Punkte mit unterschiedlichem Abstand vom Ausgangspunkt nach Richtlinie M 2 verschiedene Erzeugungswahrscheinlichkeiten haben, sollten somit alle Punkte mit gleichem Abstand gleich behandelt werden:

Richtlinie 4.1.4 (M 3) *Die Mutation* $m : G \times \Omega_m \mapsto G$ *erfüllt*

$$\forall g_1, g_2, g_3 \in G \text{ mit } d_G(g_1, g_2) = d_G(g_1, g_3) : \mathrm{P}_m(m(g_1) = g_2) = \mathrm{P}_m(m(g_1) = g_3).$$

Die implizit oft hinter der Rekombination stehende Idee ist, dass der Nachkomme zweier Eltern von diesen die guten Eigenschaften übernimmt. Das setzt voraus, dass er zu diesen gewisse Ähnlichkeiten besitzt, er also zu den Eltern keinen zu großen Abstand bzgl. d_G besitzt. Als Obergrenze für den Abstand wählen wir den Abstand der Eltern voneinander, da ansonsten der Nachkomme mit mindestens einem Elternteil weniger Ähnlichkeit hat als die Eltern untereinander:

Richtlinie 4.1.5 (R 1) *Die Rekombination* $r : G \times G \times \Omega_r \mapsto G$ *erfüllt*

$$\forall g_1, g_2, g_3 \in G \text{ mit } \mathrm{P}_r(r(g_1, g_2) = g_3) > 0 : \max(d_G(g_1, g_3), d_G(g_2, g_3)) \leq d_G(g_1, g_2).$$

Wie bei der Mutation soll aber auch die Rekombination dem Suchprozess keine weitergehende Richtung geben. Deshalb ist es sinnvoll zu fordern, dass keiner der Elternteile bevorzugt wird, indem z. B. die Nachkommen mit höherer Wahrscheinlichkeit aus seiner Nähe stammen:

Richtlinie 4.1.6 (R 2) *Die Rekombination* $r : G \times G \times \Omega_r \mapsto G$ *erfüllt*

$$\forall g_1, g_2 \in G, \forall \alpha \geq 0 : \mathrm{P}_r(d_G(g_1, r(g_1, g_2)) = \alpha) = \mathrm{P}_r(d_G(g_2, r(g_1, g_2)) = \alpha).$$

Erfüllt ein evolutionärer Algorithmus all diese Richtlinien, so bildet er einen Metrik-basierten evolutionären Algorithmus:

Definition 4.1.7 *Ein evolutionärer Algorithmus, der die Richtlinien K 1, M 1, M 2, M 3, R 1 und R 2 zu einer Metrik* d_G *erfüllt, die gemäß* ($\triangle$) *aus einer die Voraussetzung* ($\Diamond$) *erfüllenden Metrik* d_P *hervorgegangen ist, heißt* Metrik-basierter evolutionärer Algorithmus (MBEA).

Diese insgesamt sechs Richtlinien stellen nur einen Vorschlag dar, sie erheben keinen Anspruch auf Vollständigkeit oder darauf, die Rolle von Mutation und Rekombination vollkommen neu zu interpretieren. Im Gegenteil, sie spiegeln gebräuchliche Vorstellungen von der Rolle dieser Operatoren wider. Dies ist wichtig, um einen Vorzug evolutionärer Algorithmen, die intuitive Verständlichkeit der Operatoren zu bewahren. Jedoch werden diese Vorstellungen selten so konkret benannt.

Die Rekombination bekommt durch diese Richtlinien die Rolle eines Operators, der den Suchraum primär zwischen den bisher gefundenen Elementen erkundet, während die Mutation zwar bevorzugt in der Nähe der bisherigen Punkte sucht, aber im Vergleich zur Rekombination verstärkt Abweichungen in noch nicht besuchte Regionen des Suchraums erlaubt. Dadurch, dass die Metriken d_G bzw. d_P diese Abstände messen und direkt mit der Zielfunktion verbunden sind, erhoffen wir uns, dass ein MBEA effizienter als andere arbeitet. Denn in der Nähe von Punkten mit hohem Funktionswert werden sich nur Punkte mit nicht viel schlechterem Funktionswert befinden, weshalb die Wahrscheinlichkeit besonders hoch sein sollte, dort noch bessere Punkte zu finden. Dabei bezieht sich der Begriff „Effizienz“ auf die Anzahl der Funktionsauswertungen, die der Algorithmus ausführen muss. Der Zeit- und Speicheraufwand der genetischen Operatoren und des gesamten Algorithmus werden von diesen Richtlinien nicht direkt angesprochen.

Diese Hoffnung wird im Folgenden durch Experimente untersucht. Dazu wird ein die MBEA-Richtlinien erfüllendes GP-System entworfen und bestehenden GP-Systemen entgegengesetzt. Die genetische Programmierung wird aus den Paradigmen evolutionärer Algorithmen ausgewählt, weil gerade hier die Auswirkungen von Mutation und Rekombination auf die Elemente des Phänotyp-Raums und damit deren Zielfunktionswerte oftmals sehr schlecht nachvollziehbar sind. Um dieses zu erläutern, wird im nächsten Abschnitt ein Überblick über die genetische Programmierung gegeben.

4.2 Genetische Programmierung

In der genetischen Programmierung werden die Elemente des Suchraums im Gegensatz zu anderen Paradigmen evolutionärer Algorithmen als Programme interpretiert, d. h. der Phänotyp-Raum P ist die Menge von Funktionen $f : A \to B$ und es soll eine dieser Funktionen gefunden werden, die ein durch die Zielfunktion vorgegebenes Kriterium optimiert. Zwar können auch „herkömmliche“ evolutionäre Algorithmen, wie der (1+1) EA, für diese Aufgabe eingesetzt werden, indem eine Funktion z. B. durch eine binärwertige Kodierung ihrer Funktionstabelle dargestellt wird. Diese Darstellung wäre aber sehr platzaufwendig, da jede Funktion somit $\lceil \log(|B|) \rceil^{|A|}$ Bits benötigen würde, und zudem wenig anschaulich.

Da Funktionen repräsentiert werden, liegt es aufgrund des alltäglichen Umgangs mit Programmen (und den damit repräsentierten Funktionen) nahe, einen Genotyp-Raum G zu wählen, dessen Elemente nicht alle gleiche Länge haben. Dieses sollte den Ressourcenbedarf des evolutionären Algorithmus verringern, erfordert aber andere Mutations- und Rekombinationsoperatoren als für die Standardvarianten $G = \{0,1\}^n$ und $G = \mathbb{R}^n$. Die erste Übersicht über Systeme der genetischen Programmierung wurde in Koza (1992) gegeben. Die darin vorgestellte Repräsentationsform und die entsprechenden genetischen Operatoren haben sich zu den Standardwahlen in der genetischen Programmierung entwickelt. Um deutlich zu machen, dass diese genetischen Operatoren nur auf dem Genotyp-Raum den anschaulichen Anforderungen genügen und deshalb nicht den MBEA-Richtlinien genügen, seien sie hier recht detailliert vorgestellt (für eine Übersicht über andere Möglichkeiten genetischer Programmierung siehe Banzhaf, Nordin, Keller und Francone (1998)).

Da die im Weiteren betrachtete Problemstellung das Lernen boolescher Funktionen aus (implizit) gegebenen Eingabe-Ausgabe-Paaren ist, wird die Beschreibung des Einsatzgebiets und der Darstellungsform auf diesen Problemkreis beschränkt, obwohl sich mit genetischer Programmierung auch andere Problemtypen behandeln lassen (Koza (1992) und Koza (1994)).

4.2.1 Das Lernen boolescher Funktionen

Das zu lösende Problem hat in der genetischen Programmierung oft die Gestalt des folgenden Suchproblems: aus gegebenen Paaren von Eingabe- und Ausgabedaten eines unbekannten funktionalen Zusammenhangs f^* soll dieser Zusammenhang gefolgert werden. Formaler gesprochen ist bei der Beschränkung auf funktionale Zusammenhänge $f^* : \{0,1\}^n \mapsto \{0,1\}$ die Menge der *Trainingseingaben* $T_E \subseteq \{0,1\}^n$ mit der *Trainingsmenge*

$$T = \{(x, f^*(y)) \,|\, x \in T_E\}$$

gegeben, so dass eine Darstellung einer *mit T konsistenten* Funktion $f : \{0,1\}^n \mapsto \{0,1\}$, d. h. mit $f(x) = y$ für alle *Trainingspaare* $(x, y) \in T$, gesucht wird. In diesem Fall spricht man auch davon, *eine boolesche Funktion zu lernen.*

Der Extremfall, dass die Trainingsmenge *vollständig*, d. h. $T_E = \{0,1\}^n$ ist, hat zwar keine praktische Anwendung, ist aber ein Standardproblem zum Vergleich der Qualität verschiedener GP-Systeme. In diesem Fall nehmen wir an, dass die Trainingsmenge nur implizit mittels einer Black-Box (siehe Kapitel 2) gegeben ist, die bei Eingabe einer Funktion von $f : \{0,1\}^n \mapsto \{0,1\}$ mit der Zahl der Paare $(x, y) \in T$ mit $f(x) = y$ antwortet. Dies entspricht genau dem von uns vorausgesetzten Black-Box Szenario: einem Element des Suchraums S (der in diesem Fall aus allen Funktionen $f : \{0,1\}^n \to \{0,1\}$ besteht) kann nur sein Zielfunktionswert zugeordnet werden.

Unter dieser Voraussetzung kann das Suchproblem auch bei einer vollständigen Trainingsmenge T nicht nur als reines Benchmark-Problem sinnvoll sein: zwar kann man eine Repräsentation einer vollständig definierten Funktion leicht mit 2^n verschiedenen Anfragen an die Black-Box bestimmen und diese Anzahl ist im schlimmsten Fall über alle Funktionen $f : \{0,1\}^n \mapsto \{0,1\}$ auch notwendig, doch ist es interessant, Heuristiken zu finden, die in der Praxis oftmals mit weniger Anfragen auskommen.

Weitaus interessanter und praktischer ist natürlich der Fall einer *unvollständigen* Trainingsmenge $T_E \subset \{0,1\}^n$. Denn obwohl in dieser Situation das Auffinden einer Funktion, die die Trainingspaare wiedergibt, nicht schwieriger geworden ist, ist dies hier nicht das eigentliche Ziel. Vielmehr soll die zugrundeliegende Funktion $f^* : \{0,1\}^n \mapsto \{0,1\}$ auf den Eingaben, die nicht in T_E enthalten sind, möglichst genau approximiert werden. Es soll also eine *gut generalisierende* Funktion gefunden werden, die mit f^* für möglichst viele Eingaben übereinstimmt. Um in diesem klassischen Lernproblem aus der Struktur der Trainingsdaten Einsicht in die zugrundeliegende Funktion f^* gewinnen zu können, nehmen wir in diesem Fall an, dass auf die Trainingsmenge T explizit zugegriffen werden kann.

Ist die Menge der in Frage kommenden funktionalen Zusammenhänge auf keine Weise eingeschränkt, f^* also mit gleicher Wahrscheinlichkeit eine der $2^{2^n-|T|}$ zu T konsistenten Funktionen, so folgt leicht, dass die durchschnittliche Zahl (über diese zugrundeliegenden Funktionen) der nicht korrekt wiedergegebenen Eingaben für alle zu T konsistenten Funktionen g gleich

$$\frac{1}{2^{2^n-|T|}} \cdot \sum_{i=0}^{2^n-|T|} \binom{2^n-|T|}{i} \cdot i$$

ist (siehe Abschnitt 5.1). Eine überdurchschnittliche Leistung kann also von einem GP-System nur auf einer eingeschränkten Menge von zu lernenden Funktion erwartet werden. Doch gelten hier natürlich dieselben Anmerkungen wie bei der Diskussion des NFL-Theorems und realistischer Optimierungs-Szenarien (Kapitel 2): die zu findende Funktion wird, damit sie überhaupt effizient repräsentierbar sein kann, nur aus einem sehr kleinen Teil aller möglichen Funktionen stammen können, z. B. der Menge aller Funktionen mit polynomiell beschränkter Kolmogoroff-Komplexität.

Eine häufig benutzte Strategie, um auf den Eingaben aus $\{0,1\}^n \setminus T_E$ die unbekannte Funktion möglichst gut anzunähern, ist es, eine möglichst einfache Funktion zu suchen, die die Trainingsdaten korrekt widerspiegelt. Dieses *Occam's Razor*-Prinzip, dass einfache Lösungen besser generalisieren als komplizierte, ist ein in vielen Naturwissenschaften befolgtes, und deshalb schon scheinbar natürliches Prinzip. Wie jedoch gesehen, wird es ohne Einschränkung der zu lernenden Funktionen nicht besser sein als jedes andere. Ist jedoch die zu lernende Funktion von einfacher Struktur, so ist eine überdurchschnittliche Qualität bei Anwendung dieses Prinzips nicht ausgeschlossen.

Ein für die Anwendung von Occam's Razor entscheidender Faktor ist die Art und Weise, wie die Einfachheit einer Funktion gemessen wird. In vielen Anwendungen wird die Größe der Darstellung der Funktion benutzt: je kleiner diese ist, desto einfacher sei die Funktion. Somit werden je nach benutzter Darstellungsform die Auswirkungen von Occam's Razor unterschiedlich sein, weshalb in diesem Fall die Repräsentation von besonderer Wichtigkeit ist.

4.2.2 Die Standard-Repräsentation in der genetischen Programmierung

Als Standard-Repräsentation in der genetischen Programmierung haben sich die so genannten *S-Expressions* durchgesetzt, wohl auch, weil diese in den ersten erfolgreichen Experimenten eingesetzt wurden (siehe Koza (1992)). Wir werden diese hier nur in einer zur Darstellung boolescher Funktionen geeigneten Form definieren, eine Erweiterung ist aber offensichtlich:

Definition 4.2.1 *Sei $F = \{f_1, \ldots, f_m\}$ eine Menge von booleschen Funktionen, d. h. $f_i : \{0,1\}^{n_i} \mapsto \{0,1\}$ für Werte $n_i \in \mathbb{N}$ $(i \in \{1, \ldots, m\})$. Sei T die Menge der booleschen Variablen $\{x_1, \ldots, x_n\}$ mitsamt den Terminalen* 0 *und* 1*. Eine* S-Expression über F und T *ist ein Baum, so dass jedes Blatt mit einem Element von T und jeder innere Knoten mit einem Element von F markiert ist und genau n_i Nachfolger hat, wenn er mit f_i markiert ist.*

Zu einer S-Expression müssen also stets die Mengen F und T mit angegeben werden, über die die S-Expression definiert ist. Eine S-Expression wird für eine Eingabe $a \in \{0,1\}^n$ ausgewertet, indem die mit x_i markierten Blätter den Wert $a_i \in \{0,1\}$ erhalten und die Knoten des Baums dann von den Blättern zu seiner Wurzel hin ausgewertet werden. Dabei ist der Wert eines inneren Knotens gleich dem Wert der ihn markierenden Funktion angewandt auf die in seinen Nachfolgern stehenden Argumente in der durch den Baum gegebenen Reihenfolge. Der Wert des Wurzelknotens entspricht dann dem Funktionswert. Die Auswertung benötigt also im schlechtesten Fall lineare Zeit in der Größe der S-Expression. Da n boolesche Variablen vorkommen können, stellt die S-Expression eine Funktion $f : \{0,1\}^n \mapsto \{0,1\}$ dar.

Bei der Wahl von F ist die *Vollständigkeit* der Funktionsmenge wichtig, d. h. ob sich jede Funktion $f : \{0,1\}^n \mapsto \{0,1\}$ durch eine S-Expression über F darstellen lässt. Für die häufige Wahl $F = \{AND, OR, NOT\}$ ist dies der Fall, da sich so die disjunktive Normalform (Papadimitriou (1994)) jeder Funktion durch eine S-Expression über F darstellen lässt.

Abhängig von der Wahl der Funktionsmenge F können boolesche Funktionen S-Expressions ganz unterschiedlicher Größe und verschieden leichter Auffindbarkeit durch den evolutionären Prozess besitzen. Die Wahl von F ist also ein entscheidender Punkt beim Entwurf von GP-Systemen, was das Problem einer guten Wahl von F aufwirft, jedoch auch die Möglichkeit eröffnet, Wissen über das zu optimierende Problem in den Entwurf einfließen zu lassen.

Eine in Koza (1992) erwähnte, in Koza (1994) detaillierter besprochene Erweiterung von S-Expressions besteht in dem Einsatz von *Automatically Defined Functions (ADFs)*. Dabei handelt es sich um die Aufnahme von speziellen Funktionssymbolen in die Menge F, deren Funktionalität in jedem Individuum jeweils durch eine getrennte S-Expression dargestellt wird. Grundgedanke ist, dass sich in einer ADF eine Subfunktion herausbildet, die an vielen Stellen in der Funktionsdarstellung fitness-steigernd genutzt werden kann. Dabei werden die genetischen Operatoren jeweils getrennt in den die eigentliche Funktion bzw. die ADFs darstellenden S-Expressions arbeiten. Eine Änderung in einer ADF kann sich also in der dargestellten Funktion an vielen Stellen auswirken.

Natürlich muss eine S-Expression nicht als Baum abgespeichert werden: sind Teilbäume isomorph zueinander, so können sie durch einen Teilbaum dargestellt werden, dessen Wurzel mehrere eingehende Kanten hat (siehe Handley (1994) und Ehrenburg (1996)). Dies spart Speicherplatz und kann auch zu einer schnelleren Auswertung benutzt werden. Jedoch machen die genetischen Operatoren, die standardmäßig für S-Expressions verwendet werden, von dieser zusätzlichen Möglichkeit keinen Gebrauch.

4.2.3 Die Standard-Operatoren in der genetischen Programmierung

Die üblichen genetischen Operatoren für S-Expressions (siehe Koza (1992)) versuchen, bestehende Individuen durch eine Mutation recht wenig zu verändern bzw. durch Rekombination Individuen entstehen zu lassen, die zu den Eltern relativ ähnlich sind. Doch, und dieser Unterschied ist für ihre Auswirkungen erheblich, sie orientieren sich dabei nur an der Form der S-Expressions, d. h. an den Auswirkungen im Genotyp-Raum. Die dargestellten Funktionen können sich aber teils erheblich ändern (bei einer Mutation) oder nur geringe Ähnlichkeit mit ihren Eltern aufweisen (Rekombination). Um dies zu verdeutlichen, seien die wichtigsten Formen von Initialisierung, Mutation und Rekombination mit S-Expressions erläutert (siehe Koza (1992)):

- Die Initialisierung wirft im Vergleich zu evolutionären Algorithmen mit Suchräumen fester Dimension das Problem auf, dass S-Expressions beliebig groß werden können, wenn die Mengen F und T nicht leer sind. Deshalb wird aus Effizienzgründen eine Maximaltiefe d der zu erzeugenden S-Expressions von vorneherein festgelegt. Die Initialisierung wird dann von der Wurzel aus erfolgen, indem für den aktuellen Knoten ein Element aus $F \cup T$ ausgewählt wird und daraufhin rekursiv, falls der Knoten mit einer Funktion mit n_i Argumenten markiert wurde, die n_i Nachfolger erzeugt werden. Ist die Tiefe des aktuellen Knotens gleich d, so wird die Wahl der Markierung auf T eingeschränkt, so dass die Maximaltiefe garantiert ist. Anschaulich klar und durch Experimente belegt (Koza (1992)) ist, dass dieses Verfahren keine Gleichverteilung auf der Menge aller durch S-Expressions über F und T darstellbaren Funktionen erzeugt.

- Die Mutation einer S-Expression erfolgt in der Regel durch zufällige Auswahl eines Knotens der S-Expression und Ersetzung des dort beginnenden Teilbaums durch einen neuen, gemäß des zur Initialisierung beschriebenen Verfahrens erzeugten, zufälligen Teilbaums. Dabei wird wiederum darauf geachtet, dass die neue S-Expression nicht die zulässige Maximaltiefe überschreitet. Die Mutation wird in den meisten Systemen der genetischen Programmierung, wie bei den als Vorbild dienenden genetischen Algorithmen, nur als sekundärer Operator verstanden, also nur selten bzw. mit geringer Wahrscheinlichkeit eingesetzt, obwohl diesbezügliche empirische Ergebnisse unterschiedliche Schlüsse zulassen (Luke und Spector (1998)).

- Der Hauptoperator zur Erzeugung neuer Individuen ist in den meisten GP-Systemen mit S-Expressions die Rekombination: hierbei wird in den beiden Eltern jeweils ein Knoten zufällig ausgewählt und die an den ausgewählten Knoten beginnenden Teilbäume ausgetauscht. Somit werden zwei Kinder pro Mutation erzeugt. Wenn eines der Kinder die Maximaltiefe d überschreitet, wird es durch sein entsprechendes Elternteil ersetzt. Im Gegensatz zu genetischen Algorithmen mit uniformem oder k-Punkt Crossover wird die Rekombination eines Individuums mit sich selbst also nicht zwangsläufig zu einer Reproduktion entarten.

Die Selektion in GP-Systemen erfolgt in den meisten Fällen fitness-proportional. Dabei wird die Selektion wie in genetischen Algorithmen nur zur Auswahl der Eltern für Mutation und Rekombination benutzt, jedoch nicht zur Bestimmung der nächsten Generation aus den Eltern und Kindern, da die Kinder die Eltern stets vollständig ersetzen. Die Zielfunktion F wird, wie besprochen, einer S-Expression S die Zahl der Paare aus der Trainingsmenge zuordnen, die die von S dargestellte Funktion $f_S : \{0,1\}^n \mapsto \{0,1\}$ korrekt wiedergibt:

$$F(S) := |\{(x,y) \in T \mid f_S(x) = y\}|.$$

Unser Ziel ist natürlich eine Maximierung von F.

Eine Mutation verändert die beteiligte S-Expression also nur recht wenig, wenn man die Anzahl der veränderten Knoten als Maßstab wählt. Denn da die Wurzel des zu ersetzenden Teilbaums gleichverteilt ausgewählt wird, liegt sie mit großer Wahrscheinlichkeit weit unten in der S-Expression. Somit wird der neu eingefügte Teilbaum aufgrund der Tiefenbeschränkung in der Regel nicht sehr tief und damit groß sein. Die Auswirkungen der Ersetzung eines Teilbaums auf die dargestellte Funktion können aber recht groß sein: auch das Ergebnis eines tief beginnenden Teilbaums kann sich bis zu der Wurzel auswirken, wenn die Geschwister aller Vorgänger des Teilbaums jeweils das Neutralelement der Funktion des Vorgängers zurückgeben. Im Regelfall wird dies mit wachsender Tiefe des ausgewählten Knotens unwahrscheinlicher, lässt sich aber nur schwer abschätzen. Die Mutation scheint somit in ihrer Wirkung auf die beteiligten Funktionen kleine Änderungen zu bevorzugen (dabei messen wir den Unterschied zweier Funktionen anhand der Zahl verschieden abgebildeter Eingaben), obwohl sie auch das Erreichen jedes Punkts des Suchraums erlaubt. Für die dritte MBEA-Richtlinie R 3, die Symmetrie um den Elter herum, kann man leicht Situationen bilden, in denen sie nicht erfüllt wird. Die MBEA-Richtlinien für die Mutation werden jedoch tendenziell von dem beschriebenen Mutationsoperator eingehalten, auch wenn sich Ausnahmen finden lassen.

Die Rekombination wird in ihren Auswirkungen viel mehr gegen die den MBEA-Richtlinien zugrundeliegende Vorstellung, dass die Nachkommen ihren Eltern ähnlich sind, verstoßen: auch wenn sich die zwei Eltern sehr ähnlich sind (als Funktion oder als S-Expression betrachtet), werden die Nachkommen in ihrem funktionalen Verhalten sehr weit von ihnen abweichen können. Denn schon wenn in einer S-Expression ein Teilbaum durch eine Kopie eines anderen aus derselben S-Expression ersetzt wird, kann die resultierende Funktion stark von der ursprünglichen abweichen. Somit ist z. B. nicht gesichert, dass der Nachkomme auf den Eingaben mit den Eltern übereinstimmt, wo diese dies tun.

Nun kann zu Recht behauptet werden, dass es Funktionen gibt, wo dieses Verhalten eine schnelle Optimierung erst ermöglicht. Doch scheint es schwierig, ein solches GP-System für eine gegebene Funktion zu finden, wenn die Auswirkungen von Mutation und Rekombination so schwer abzuschätzen sind. Die unklaren und kaum formal zu beschreibenden Auswirkungen auf die dargestellten Funktionen erschweren somit den Entwurf von GP-Systemen, da selbst kaum anschaulich argumentiert werden kann, wann ein GP-System für ein Problem besser als ein anderes geeignet ist. Um zu zeigen, dass die MBEA-Richtlinien die Leistung von GP-Systemen erhöhen können, brauchen wir natürlich ein Maß für die Leistung.

4.2.4 Eine Bewertungsmöglichkeit von GP-Systemen

Ziel eines GP-Systems ist es, die Zielfunktion F zu maximieren. Um zu bewerten, wie groß der Aufwand zum Finden des Optimums ist, hat sich der so genannte *Computational Effort* (Koza (1992)) durchgesetzt, der einen Schätzwert für die Zahl der zu verarbeitenden Individuen angibt, bis mit hoher Wahrscheinlichkeit ein Optimum gefunden wurde. Bevor wir diesen genauer definieren, sei noch einmal darauf hingewiesen, dass er sich nur auf eine Menge von experimentellen Daten, jedoch auf keine theoretischen Analysen des Systems stützt und deshalb mit den Resultaten aus Kapitel 3 in keiner Weise zu vergleichen ist.

Grundlage ist eine Menge von L Läufen des GP-Systems, die jeweils bis zu einer Maximalzahl von G Generationen durchgeführt werden. Da angenommen wird, dass der maximale Wert der Zielfunktion, d. h. die Größe der Trainingsmenge, bekannt ist, kann man für jede Generation $g \in \{1, \ldots, G\}$ den Anteil $A(g) \in [0, 1]$ der Läufe bestimmen, die bis zur g-ten Generation ein Optimum gefunden haben. Dieser Anteil wird als Schätzwert für die Wahrscheinlichkeit benutzt, mit der das GP-System in den ersten g Generationen das Optimum findet, was im Folgenden als *Erfolgswahrscheinlichkeit* bezeichnet wird.

Bezeichnen wir eine Wahrscheinlichkeit von $z \in [0, 1]$, das Optimum gefunden zu haben, als ausreichend. Nimmt man nun die kleinste Generation $g \in \{1, \ldots, G\}$ mit $A(g) \geq z$, so betrachtet man $\mu + \lambda \cdot g$ als eine Abschätzung für die Zahl der Individuen, die das GP-System bearbeiten muss, damit es mit einer Wahrscheinlichkeit von z das Optimum in einem Lauf findet. Dabei ist μ bzw. λ die Zahl der Eltern bzw. der Kinder.

Damit wird aber die Möglichkeit einer Multistart-Variante (siehe Definition 3.3.6) noch ausser Acht gelassen, wobei das GP-System W-mal jeweils bis zu genau G Generationen unabhängig wiederholt wird (und kein anschließender unbegrenzter Lauf wie in Definition 3.3.6 durchgeführt wird). In diesem Fall würde sich als Wahrscheinlichkeit, dass die Multistart-Variante nach g Generationen ein Optimum gefunden hat, der Wert

$$1 - (1 - A(g))^W$$

ergeben. Wenn schon nach relativ wenigen Generationen ein recht hoher Anteil der L Läufe das Optimum gefunden hat, so ist zu erwarten, dass relativ wenige Wiederholungen ausreichen, um die Erfolgswahrscheinlichkeit auf $z \in [0, 1]$ zu heben, und somit mit insgesamt weniger Individuen ein Optimum gefunden werden kann. Wie groß W sein muss, ergibt sich durch folgende einfache Rechnung:

$$\begin{array}{lcl} & 1 - (1 - A(g))^W & \geq z \\ \iff & W & \geq \log_{1-A(g)}(1 - z) = \dfrac{\log(1 - z)}{\log(1 - A(g))}. \end{array}$$

Dabei wird vorausgesetzt, dass $A(g) > 0$ ist, da ansonsten eine Multistart-Variante die Erfolgswahrscheinlichkeit nie verbessern kann. Somit ergibt sich nach der Bearbeitung von $\mu + \lceil \log(1 - z)/\log(1 - A(g)) \rceil \cdot g \cdot \lambda$ Individuen eine Erfolgswahrscheinlichkeit von mindestens z. Den geringsten Aufwand erhält man mit dem diesen Ausdruck minimierenden Wert von g. Der dann angenommene Wert ist der Computational Effort, ein Schätzwert für die benötigte Anzahl an Individuen, bis ein Optimum mit Wahrscheinlichkeit z gefunden wird:

Definition 4.2.2 *Sei das untersuchte GP-System mit jeweils μ Eltern und λ Kindern genau L-mal bis zur Generation G gelaufen und $A(g) \in \{0, \frac{1}{L}, \ldots, 1\}$ ($g \in \{1, \ldots, G\}$) der Anteil an diesen L Läufen, in denen ein Optimum in den ersten g Generationen gefunden wurde. Dann ist der* Computational Effort *des GP-Systems zur Wahrscheinlichkeit z definiert als*

$$\mu + \lambda \cdot \min \left\{ \left\lceil \frac{\log(1-z)}{\log(1-A(g))} \right\rceil \cdot g \mid g \in \{1, \ldots, G\} \text{ mit } A(g) > 0 \right\}.$$

In allen folgenden Auswertungen wird, wie in der Literatur üblich, z als 0,99 gewählt. Ein Vorteil des Computational Efforts ist, dass er ein GP-System unabhängig von der konkreten Implementierung und der Geschwindigkeit des Rechners, auf dem es ausgeführt wurde, bewertet. Er misst ausschließlich die Qualität des Zusammenspiels der genetischen Operatoren. Insofern liefert er eine maschinen-unabhängige Vergleichsmöglichkeit und ist somit der reinen Messung des Zeitbedarfs überlegen. Jedoch kann die Populationsgröße eine entscheidende Rolle spielen.

Dass der Zeitaufwand keine Rolle spielt, kann aber auch im Hinblick auf die Praxisrelevanz von Nachteil sein, da ein zeitaufwendigerer Rekombinations-Operator, der das GP-System mit weniger Individuen ein Optimum finden lässt, stets besser bewertet wird, als eine zeitlich schnellere Rekombination, die die Anzahl benötigter Individuen des GP-Systems erhöht. In einer praktischen Situation kann der zweite Operator aber durchaus besser sein, wenn der Geschwindigkeitsgewinn den höheren „Individuenverbrauch“ mindestens kompensiert.

Weiterhin ist zu kritisieren, dass die Zahl L der durchgeführten Läufe beliebig gewählt werden kann. Natürlich führt eine höhere Zahl von Wiederholungen mit größerer Sicherheit zu einem Wert $A(g)$, der der Wahrscheinlichkeit, ein Optimum in g Generationen gefunden zu haben, nahe kommt. Doch eine Mindestzahl, nach der sich $A(g)$ z. B. mit einer hohen Sicherheit nur um $\varepsilon > 0$ von der wahren Wahrscheinlichkeit unterscheidet, kann nur bei genauerer Kenntnis des zugrunde liegenden stochastischen Prozesses angegeben werden (wenn z. B. dessen Varianz bekannt ist). Insofern gibt es keine Sicherheit, dass der Computational Effort die wahre Mindestanzahl an Individuen, um mit z Prozent Wahrscheinlichkeit ein Optimum zu finden, angibt.

Auch ist der Computational Effort bei kleiner Anzahl der Läufe eher ein *Worst-Case*-Maßstab, reagiert also sehr empfindlich auf die Verteilung der Läufe mit der größten Dauer. Wenn z. B. bei $z = 0{,}99$ weniger als 100 Läufe durchgeführt werden, so wird die Generationenzahl, zu der alle Läufe erfolgreich sind, große Auswirkungen auf den Computational Effort haben, da bei Abbruch zu einer kleineren Generationenzahl schon mindestens zwei Wiederholungen durchgeführt werden müssen, um die Erfolgswahrscheinlichkeit auf 0,99 zu erhöhen. Da sich der Computational Effort als Vergleichsmaßstab für die Qualität eines GP-Systems durchgesetzt hat, wird er auch von uns in den folgenden Abschnitten zur Bewertung bzw. zum Vergleich eingesetzt. Dabei seien wir uns der erwähnten Kritikpunkte immer bewusst.

4.2.5 Nachteile von GP mit S-Expressions

Wenn in den nächsten Abschnitten GP-Systeme mit einer anderen Darstellungsform vorgestellt werden, so ist dies natürlich in Nachteilen der Repräsentation mit S-Expressions begründet. Deshalb sollen hier die wichtigsten Nachteile aufgezählt werden:

- Die Darstellung mit S-Expressions ist nicht eindeutig, d. h. zu einer Funktion $f : \{0,1\}^n \mapsto \{0,1\}$ gibt es für jede Terminalmenge T und jede vollständige Funktionsmenge F beliebig viele S-Expressions, die die Funktion f darstellen.

Denn ist S eine beliebige S-Expression für f, so kann eine größere f darstellende S-Expression erzeugt werden, indem z. B. in der S-Expression von AND (die nach Vollständigkeit existiert) die Variablenknoten durch S ersetzt werden.

Nun muss dies ja kein Problem sein, wenn es einen effizienten Algorithmus zur Minimierung einer gegebenen S-Expression geben würde. Doch da S-Expressions über einer vollständigen Funktionsmenge F äquivalent zu booleschen Formeln sind, ist das Problem, zu einer S-Expression S und einer Zahl $k \in \mathbb{N}$ zu entscheiden, ob es eine zu S äquivalente S-Expression S' mit höchstens k Knoten gibt, NP-vollständig, wie sich leicht durch eine polynomielle Transformation von SAT ergibt (Garey und Johnson (1979)). Somit gibt es unter der Annahme $P \neq NP$ keinen polynomiellen Algorithmus zur Minimierung von S-Expressions.

Dies hat den Nachteil, dass man bei der Repräsentation mit S-Expressions einen recht hohen Speicherplatzbedarf haben kann. Da gerade im Bereich der genetischen Programmierung aufgrund des oftmals sehr großen Suchraums extrem große Populationsgrößen nicht selten sind (in Bennett, Koza, Yu und Mydlowec (2000) wird z. B. eine Population von 10 Millionen Individuen verwendet), ist eine möglichst kleine Darstellungsform natürlich von Vorteil.

Neben diesem Problem der Ressourcenverschwendung folgt aus der nicht eindeutigen Darstellung mit S-Expressions auch, dass die Größe einer Funktion, wenn sie anhand der Knoten ihrer S-Expression gemessen wird, nicht eindeutig ist. Dies hat natürlich Auswirkungen auf das Occam's Razor-Prinzip, dessen Kernpunkt ist, eine möglichst „einfache“ Funktion zu finden. Die Einfachheit einer Funktion wird in allen Ansätzen über die Größe der Darstellung der Funktion gemessen und ist somit von der Darstellungsform abhängig. Wenn diese nicht eindeutig ist, so kann eine schon gefundene Funktion, die eine sehr kleine Beschreibung besitzt, im Laufe der Suche verloren gehen, nur weil ihre gefundene Darstellung relativ groß war. Eine Umsetzung des Occam's Razor Prinzips ist also zumindest weniger sinnvoll, wenn die benutzte Darstellungsform nicht eindeutig ist.

- Die besprochenen genetischen Operatoren Mutation und Rekombination für S-Expressions stellen sehr einfache Möglichkeiten dar, neue S-Expressions zu erzeugen, die mit dem bzw. den alten S-Expressions relativ große Ähnlichkeit besitzen. Jedoch bezieht sich die Ähnlichkeit nur auf die syntaktische Form der S-Expressions, da sich z. B. ein Eltern-Individuum von seinem durch Mutation erzeugten Nachfolger nur in dem an dem ausgewählten Knoten beginnenden Teilbaum unterscheiden kann.

  Dies kann sich jedoch auf die dargestellte Funktion recht gravierend auswirken, wozu natürlich vorher noch ein Maß für die Ähnlichkeit zweier boolescher Funktionen festgelegt sein muss. Ein sinnvolles Maß hierfür ist die Zahl der Eingaben, an denen die beiden Funktionen übereinstimmen (siehe auch Abschnitt 4.4). Benutzt man dieses, so kann der Austausch eines einzigen Knotens eine maximal verschiedene Funktion zur Folge haben. Zwar wird dies nicht der typische Fall sein, da der Wert relativ tief im Baum liegender Knoten in den meisten Fällen nur für relativ wenige Eingaben Auswirkungen auf das Gesamtergebnis haben wird. Doch geben einem diese heuristischen Argumente keine Garantie, dass z. B. bei der Mutation kleine Änderungen des Funktionsverhaltens öfter auftreten als große oder bei der Rekombination die Nachkommen Ähnlichkeiten im Funktionsverhalten mit ihren Eltern aufweisen. Wie schon erwähnt wirkt sich dies negativ auf einen systematischen und gezielten Entwurf von GP-Systemen aus.

Der zweite Punkt ist also die Motivation, gerade ein MBEA-konformes GP-System zu konstruieren und dieses gegen bestehende GP-Systeme zu testen. Dieses wird seine Individuen nicht mit S-Expressions darstellen, sondern mit OBDDs, einer im nächsten Abschnitt beschriebenen Datenstruktur, für die auch der erste Kritikpunkt an S-Expressions nicht gilt.

4.3 Ordered Binary Decision Diagrams

In diesem Abschnitt werden die so genannten *Ordered Binary Decision Diagrams* (*OBDDs*) vorgestellt und ihre wichtigsten Vorzüge für die Anwendung in der genetischen Programmierung beschrieben. Für eine umfassende Erläuterung von OBDDs und anderer BDD-Varianten sei auf Wegener (2000) verwiesen.

Unter OBDDs versteht man gerichtete Graphen mit folgender syntaktischer Struktur (Bryant (1986)):

Definition 4.3.1 *Sei* $n \in \mathbb{N}$ *und* $\pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$ *eine Permutation auf* $\{1, \ldots, n\}$. *Ein* π-Ordered Binary Decision Diagram (OBDD) *ist ein gerichteter Graph,*

1. *der eine Quelle und zwei Senken besitzt, wobei letztere mit 0 bzw. 1 markiert sind,*
2. *dessen innere Knoten, d. h. alle Nicht-Senken, mit einer der booleschen Variablen* $x_1, \ldots, x_n$ *markiert sind,*
3. *dessen innere Knoten* v *jeweils zwei ausgehende Kanten haben, die* 0- *bzw.* 1-Kante, *die zum* 0- *bzw.* 1-Nachfolger *des inneren Knotens führen, welche mit* $v \rightarrow 0$ *bzw.* $v \rightarrow 1$ *abgekürzt werden, und*
4. *dessen Kanten die Variablenordnung* π *beachten, d. h. wenn eine Kante von einem mit* x_i *zu einem mit* x_j *markierten Knoten führt* ($i, j \in \{1, \ldots, n\}$), *so ist* $\pi^{-1}(i) < \pi^{-1}(j)$.

Ein OBDD *ist ein* π*-OBDD zu einer beliebigen Variablenordnung* π.

Ein OBDD ist also stets nur zu einer gegebenen Ordnung π der Variablen definiert. Oft wird eine Permutation π durch die Angabe des Vektors $(\pi(1), \ldots, \pi(n))$ repräsentiert; so bedeutet $\pi = (n, \ldots, 1)$ z. B., dass nur Kanten von x_i nach x_j zeigen dürfen, wenn $i > j$ ist. Die Variablenordnung spielt für die Form und Größe eines OBDDs zu einer gegebenen Funktion eine entscheidende Rolle, wie wir noch an Beispielen sehen werden. Zuvor soll geklärt werden, wie ein OBDD D eine Funktion $f_D : \{0,1\}^n \mapsto \{0,1\}$ repräsentiert:

Algorithmus 4.3.2
Eingabe: *ein* π*-OBDD* D *mit* n *Variablen und ein Eingabevektor* $a \in \{0,1\}^n$.
Ausgabe: *der Wert* $f_D(a) \in \{0,1\}$.

1. *Sei* v, *der aktuelle Knoten, gleich der Quelle von* D.
2. *Falls* v *mit* x_i ($i \in \{1, \ldots, n\}$) *markiert ist, so setze* v
   (a) *gleich dem 1-Nachfolger* $v \rightarrow 1$ *von* v, *falls* $a_i = 1$, *oder*
   (b) *gleich dem 0-Nachfolger* $v \rightarrow 0$ *von* v, *falls* $a_i = 0$ *ist,*

   und wiederhole Schritt 2.
3. *Falls* v *eine* s*-Senke ist, gib* s *aus.*

Die Auswertungsprozedur durchläuft das OBDD also von der Wurzel aus, indem an jedem Knoten der Nachfolger aufgesucht wird, der dem Wert der Eingabe an der durch die Markierung des Knotens gegebenen Variablen entspricht. Dies geschieht so lange, bis eine Senke erreicht wird. Deren Markierung gibt den Funktionswert der von dem OBDD dargestellten Funktion auf der Eingabe an.

Die Bedingung 4 der Definition 4.3.1 unterscheidet ein OBDD von einem BDD und garantiert, dass die Auswertung stets in linearer Zeit in der Anzahl der Variablen durchgeführt werden kann. Dies ist ein erster Vorteil von OBDDs im Vergleich zu S-Expressions, bei denen die Auswertung lineare Zeit in der Größe der S-Expression erfordern kann. Da Funktionen, in denen alle Variablen Einfluss auf den Funktionswert haben, nur durch S-Expressions dargestellt werden können, die alle Variablen beinhalten, ist die Auswertungszeit von S-Expressions im schlechtesten Fall stets mindestens so hoch wie die von OBDDs.

Als Größe $|D|$ eines OBDDs D definieren wir die Anzahl der inneren Knoten von D, da jedes nicht-triviale OBDD stets beide Senken enthalten muss. Wie man sich leicht überlegen kann, gibt es zu den meisten Funktionen $f : \{0,1\}^n \mapsto \{0,1\}$ auch bei Berücksichtigung von Isomorphien nicht nur ein OBDD. Die Eindeutigkeit der OBDD-Darstellung wird erst durch den Übergang zu *reduzierten* OBDDs sichergestellt.

Ein reduziertes OBDD ist ein OBDD, so dass für keinen Knoten der Null- gleich dem Einsnachfolger ist und es keine zwei Knoten mit der gleichen Markierung gibt, deren Null- bzw. Einsnachfolger jeweils übereinstimmen. Ein gegebenes OBDD kann reduziert werden, indem die folgenden Reduktionsregeln so oft wie möglich angewendet werden:

1. Gibt es einen Knoten v, dessen Null- und Einsnachfolger gleich sind, so lasse alle in v eingehenden Kanten auf diesen Nachfolger zeigen und lösche v (*deletion rule*)

2. Gibt es zwei Knoten v und w mit der gleichen Markierung, deren Null- bzw. Einsnachfolger jeweils gleich sind, so lasse alle in v eingehenden Kanten auf w zeigen und lösche v (*merging rule*).

Die erschöpfende Anwendung dieser zwei Regeln erzeugt ein reduziertes OBDD, das unter allen OBDDs, die dieselbe Funktion darstellen, die geringste Anzahl von Knoten hat und bis auf Isomorphie eindeutig ist (Wegener (2000)). Somit gibt es für OBDDs eine eindeutige Minimalform, die sich aus einem OBDD D in Linearzeit bzgl. der Größe von D berechnen lässt, indem die obigen Reduktionsregeln von den Senken zu der Quelle hin durchgeführt werden.

Zusätzlich besteht ein Zusammenhang zwischen der syntaktischen Struktur eines reduzierten OBDDs und seiner Semantik, d. h. der dargestellten Funktion. Dazu sei für ein $k \in \{1,\ldots,n\}$, eine Indexmenge $\{i_1,\ldots,i_k\} \subseteq \{1,\ldots,n\}$ und einen Vektor $a \in \{0,1\}^k$ die Funktion

$$f_{|x_{i_1}=a_1,\ldots,x_{i_k}=a_k} : \{0,1\}^{n-k} \mapsto \{0,1\}$$

definiert, die sich aus f durch Konstantsetzung der Variablen x_{i_j} durch den Wert a_j für alle $j \in \{1,\ldots,k\}$ ergibt. Dann heißt eine Funktion f von einer Variablen x_i *essentiell abhängig*, wenn $f_{|x_i=0} \neq f_{|x_i=1}$ ist. Das reduzierte π-OBDD für f enthält für jeden Index $i \in \{1,\ldots,n\}$ genau so viele mit x_i markierte Knoten wie es von x_i essentiell abhängige Funktionen $f_{|x_{\pi(1)}=a_1,\ldots,x_{\pi(i-1)}=a_{i-1}} : \{0,1\}^{n-i+1} \mapsto \{0,1\}$ für beliebige Vektoren $a \in \{0,1\}^{i-1}$ gibt (Wegener (2000)). Anschaulich gesprochen besitzt ein reduziertes π-OBDD also viele mit x_i markierte Knoten, wenn es viele von x_i abhängige Subfunktionen gibt, die sich durch Festlegung der gemäß π vor x_i getesteten Variablen ergeben. Somit hat die Größe eines OBDDs eine eindeutige Beziehung zu der durch das OBDD dargestellten Funktion.

Die Anwendung der zwei oben angesprochenen Regeln zur Reduzierung setzt natürlich voraus, dass das ggf. nicht-reduzierte OBDD schon existiert. Um nur das reduzierte OBDD aufzubauen, kann man ein OBDD stets von den Senken zur Quelle hin aufbauen: dann wird ein Knoten v nur aufgebaut, wenn seine Nachfolgerknoten schon existieren, so dass leicht überprüft werden kann, ob beide Nachfolgerknoten die gleiche Funktion darstellen (denn dann werden sie durch denselben Knoten repräsentiert) oder ob es schon einen Knoten mit derselben Markierung und denselben Null- und Einsnachfolgern gibt (was in einer dynamischen Datenstruktur, wie z. B. einer Hash-Tabelle abgespeichert werden kann). Da dieses gegenüber der Anwendung der Reduktionsregeln Speicherplatz sparen kann, wenn das nicht-reduzierte OBDD im Vergleich zum reduzierten OBDD wesentlich größer ist, wird diese Methode in allen im Folgenden beschriebenen GP-Systemen mit OBDDs verwandt, ohne wegen seiner rein technischen Natur weiter erwähnt zu werden.

Gerade für den Einsatz in der genetischen Programmierung bietet sich der Einsatz von *Shared Binary Decision Diagrams* (*SBDDs*) an, einer Modifikation von OBDDs: ein π-SBDD ist wie ein π-OBDD definiert, hat aber statt einer Quelle $m \in \mathbb{N}$ Quellen. Jede dieser Quellen repräsentiert eine Funktion $f_i : \{0,1\}^n \mapsto \{0,1\}$ $(i \in \{1,\ldots,m\})$. Somit repräsentiert ein SBDD eine Funktion $f : \{0,1\}^n \mapsto \{0,1\}^m$. Von einer schlichten Vereinigung von m OBDDs unterscheidet das SBDD, dass gleiche Substrukturen der an verschiedenen Quellen beginnenden OBDDs gemeinsam benutzt werden können. Wendet man die beiden Reduktionsregeln erschöpfend oft auf ein SBDD an, ergibt sich ein reduziertes SBDD, das unter allen SBDDs, die die Funktionenfamilie $(f_1,\ldots,f_m)$ darstellen, minimale Knotenanzahl hat. In reduzierten SBDDs werden gleiche Substrukturen also so weit wie möglich zwischen OBDDs geteilt, wodurch zusätzlich Speicherplatz gespart werden kann. In der genetischen Programmierung sollten deshalb die verschiedenen Individuen, sowohl Eltern als auch Kinder, in einem reduzierten SBDD gespeichert werden.

Für den Entwurf genetischer Operatoren, die die MBEA-Richtlineien erfüllen, ist es von Vorteil, dass sich viele Operationen auf OBDDs in polynomieller Zeit bzgl. der Größe der beteiligten OBDDs durchführen lassen. Eine wichtige, weil Grundlage für spätere Rekombinations- und Mutationsoperatoren bildende Operation ist die *Synthese* zweier π-OBDDs D_1 und D_2 (siehe Algorithmus 4.4.8): hierbei soll für eine boolesche Funktion $\otimes : \{0,1\}^2 \mapsto \{0,1\}$ das π-OBDD für die Funktion $f_{D_1} \otimes f_{D_2}$, definiert als

$$(f_{D_1} \otimes f_{D_2})(x) := f_{D_1}(x) \otimes f_{D_2}(x),$$

berechnet werden. Unter der beschriebenen Voraussetzung, dass die Variablenordnungen der OBDDs D_1 und D_2 gleich sind, gibt es einen Algorithmus mit Laufzeit $O(|D_1| \cdot |D_2|)$, wobei das entstehende OBDD dieselbe Variablenordnung beachtet (Wegener (2000)). Somit liefert die Synthese gerade die bei S-Expressions fehlende Möglichkeit, zu durch OBDDs gegebenen Funktionen neue Funktionen zu generieren, deren Semantik sich gut nachvollziehbar aus der Semantik der Eltern ergibt. So können sie sich in ihrem Funktionsverhalten z. B. nicht sehr von dem des alten OBDD unterscheiden bzw. Gemeinsamkeiten bestehender OBDDs übernehmen.

Weiterhin bilden OBDDs implizit TeilOBDDs, die an vielen Stellen im OBDD genutzt werden, d. h. die viele eingehende Kanten haben. Dies ergibt sich insbesondere beim Einsatz von reduzierten OBDDs, wo gleiche Subfunktionen ja stets nur durch ein TeilOBDD dargestellt werden. Wird nun das an diesem Knoten beginnende TeilOBDD geändert, so wird sich dies an vielen Stellen in der dargestellten Funktion auswirken. Deshalb ergeben sich bei OBDDs implizit die erweiterten Möglichkeiten, die bei genetischer Programmierung mit S-Expressions erst explizit durch den Einsatz von ADFs (siehe Abschnitt 4.2) geschaffen werden.

Wie diese Vorteile von OBDDs zur Konstruktion eines GP-System eingesetzt werden können, das den MBEA-Richtlinien genügt, wird in Abschnitt 4.4 gezeigt.

4.3.1 Bestehende GP-Systeme mit OBDDs

In diesem Abschnitt soll ein Überblick über den bisherigen Einsatz von OBDDs im Bereich der genetischen Programmierung gegeben werden. Dabei sollen die Schwächen und Vorzüge der bisherigen Ansätze als Ausgangspunkt für Verbesserungen ausfindig gemacht werden.

Das erste GP-System mit OBDDs wird von Yanagiya (1994) vorgestellt. Hier werden reduzierte π-OBDDs zur Darstellung benutzt (wobei die Variablenordnung π fest voreingestellt ist) die als ein π-SBDD gespeichert werden. Um zu großen Speicherplatzverbrauch zu vermeiden, darf jedes OBDD nur eine bestimmte *Breite*, d. h. Höchstzahl von Knoten haben, die mit derselben Markierung versehen sind. Wird versucht, einen Knoten mit einer Markierung zu erzeugen, für die diese Maximalzahl schon erreicht ist, wird dieser Knoten nicht erzeugt und in diesen eingehende Kanten zeigen auf einen zufällig gewählten Knoten mit der nächsthöheren Markierung gemäß π. Das Ziel in Yanagiya (1994) ist es, eine anhand ihrer vollständigen Trainingsmenge gegebene Funktion zu finden. Das System geht dazu folgendermaßen vor:

In der Initialisierung werden S-Expressions auf die dort übliche Weise erzeugt und in reduzierte OBDDs transformiert. Während also die Initialisierung auf der Ebene der S-Expressions arbeitet, gehen Mutation und Rekombination explizit auf die OBDD-Darstellung ein: die Mutation eines OBDDs wählt gleichverteilt zwei verschiedene Variablen x_i und x_j aus und vertauscht die Rollen dieser Variablen. Dies wird nicht durch das explizite Vertauschen der Markierungen gemacht, was die Variablenordnung ändern würde, sondern durch eine Folge von Synthese-Operationen. Danach wird eine der Variablen zufällig ausgewählt und zufällig durch die Konstante 0 oder 1 ersetzt.

Die Rekombination ähnelt dem Synthesealgorithmus 4.4.8 für OBDDs: rekursiv werden beide OBDDs von der Quellen zu den Senken durchlaufen, wobei die Variablenordnung bei der Auswahl der jeweils durchlaufenen Nachfolger berücksichtigt und ein Knoten der jeweils kleineren Markierung zurückgegeben wird. Werden zwei verschiedene Senken erreicht, so wird mit Wahrscheinlichkeit 1/2 die eine oder die andere zurückgegeben. Wird in dem rekursiven Durchlauf in beiden OBDDs der gleiche Knoten erreicht, so wird dieser zuerst mutiert und dann zurückgegeben. Dabei erfolgt die Mutation stets nur mit einer geringen Wahrscheinlichkeit zwischen 0 und 1/10, die erhöht wird, wenn sich für eine bestimmte Anzahl von Generationen der beste Zielfunktionswert nicht um ein bestimmtes Verhältnis verbessert hat.

Für weitere Details sei auf Yanagiya (1994) verwiesen. Es sei festgestellt, dass dieses GP-System im Vergleich zu GP-Systemen mit S-Expressions den MBEA-Richtlinien wesentlich näher kommt: eine Mutation wird häufig nur relativ wenige Ausgaben der Funktion ändern und das Resultat einer Rekombination wird für die Eingaben, an denen die Eltern übereinstimmen, auch dieses Ergebnis liefern. Jedoch kann die in die Rekombination einfließende Mutation diesen Effekt wieder zerstören. Weiterhin muss stets die Beschränkung der Breite berücksichtigt werden, die eine genaue Abschätzung der Wirkung der Operatoren sehr schwierig macht.

Das System wurde in $L = 30$ Läufen für die MUX_{11}-Funktion getestet (für die Funktionsdefinitionen siehe Abschnitt 4.4). Da die Populationsgröße 4.000 ist und nach spätestens 40 Generationen in allen Läufen das Optimum gefunden wurde, kann man den Computational Effort durch 160.000 nach oben abschätzen. Dieser wird in Yanagiya (1994) nicht genauer angegeben. Für die MUX_{20}-Funktion wurde das System in $L = 5$ Läufen mit einer Populationsgröße von 8.000 getestet. Nach spätestens 424 Generationen wurde in alle Läufen das Optimum gefunden. Damit ist dieses das erste GP-System, das die 20-Multiplexer-Funktion finden konnte. Den Computational Effort für die MUX_{20}-Funktion kann man durch 3.392.000 nach oben abschätzen, wobei man die geringe Zahl von fünf Läufen berücksichtigen sollte.

Das GP-System von Yanagiya (1994) zeigt zum ersten Mal, wie OBDDs in der genetischen Programmierung genutzt werden können, wobei neben der Effizienz der Darstellung zumindest zum Teil die Möglichkeit genutzt wird, Operationen mit gut nachvollziehbaren funktionalen Auswirkungen zu definieren. Das in Abschnitt 4.4 vorgestellte MBGP-System wird sich deshalb insbesondere den Rekombinationsoperator als Vorbild nehmen, wobei aber die implizite Mutation und die Breitenbeschränkung weggelassen wird.

Das zweite existierende GP-System, welches die Individuen durch OBDDs darstellt, ist in Sakanashi, Higuchi, Iba und Kakazu (1996) beschrieben. Hier werden jedoch auch Strukturen zugelassen, die keine OBDDs nach Definition 4.3.1 darstellen: so können z. B. auf Pfaden von der Quelle zu einer Senke Variablen in verschiedenen Reihenfolgen auftreten, d. h. es wird mit Entscheidungsbäumen gearbeitet. Dadurch, dass die Zielfunktion Strukturen, die auf vielen Pfaden alle Variablen in derselben Reihenfolge testen, einen höheren Wert zuordnet, sollen im Verlauf der Suche OBDDs nach Definition 4.3.1 gefunden werden. Stärker wird aber das Ziel gewichtet, mit der vollständigen Trainingsmenge eine maximale Übereinstimmung zu erhalten.

Dadurch, dass keine OBDDs, sondern binäre Entscheidungsbäume zur Darstellung genutzt werden, können die von Koza (1992) vorgeschlagenen Mutations- und Rekombinationsoperatoren für S-Expressions direkt übernommen werden. Empirische Ergebnisse des sich so ergebenden Systems sind für die $\textsc{Par}_n$-Funktion mit $n \in \{3, \ldots, 6\}$, $\textsc{Mux}_6$ und $\textsc{Mux}_{11}$ beschrieben. Dabei sind jedoch nur für $\textsc{Par}_3$ und $\textsc{Mux}_6$ stets korrekte OBDDs entstanden, die wie in den anderen Läufen die Trainingsdaten komplett wiedergeben. Da zudem nur über 10 Läufe gemittelt und nur die durchschnittliche Generation, in der ein alle Trainingsbeispiele korrekt wiedergebender Entscheidungsbaum gefunden wurde, angegeben ist, lässt sich der Computational Effort nicht abschätzen.

Dieses System versucht, die Vorteile von OBDDs zu nutzen, ohne auf die von S-Expressions gewohnten genetischen Operatoren zu verzichten. Da diese jedoch die OBDD-Struktur nicht beachten, entstehen nur in den wenigsten Fällen reduzierte OBDDs, wodurch der Effizienzgewinn der Darstellung verloren gehen kann.

Die bestehenden GP-Systeme mit OBDDs schlagen also verschiedene Wege ein: das in Yanagiya (1994) beschriebene versucht neben der effizienten Darstellung auch die Vorteile von gezielt auf OBDDs abgestimmten genetischen Operationen zu nutzen, während das in Sakanashi, Higuchi, Iba und Kakazu (1996) beschriebene System sowohl Effizienz der Darstellung als auch die Möglichkeit gezielterer genetischer Operatoren zugunsten einer Übernahme der Operatoren für S-Expressions nicht stets ausnutzt.

4.4 Ein MBGP-System mit OBDDs

In diesem Abschnitt zeigen wir, wie ein GP-System aussehen kann, dass die MBEA-Richtlinen erfüllt. Es wird deshalb MBGP-System genannt. Zum Vergleich dazu entwerfen wir ein GP-System mit OBDDs, das versucht, die von GP-Systemen mit S-Expressions bekannten Operatoren sehr direkt umzusetzen, dabei jedoch stets bei syntaktisch korrekten OBDDs bleibt. Dieses wird OBDD-GP-System genannt. Somit wird auch das zweite System von der effizienten Darstellung mit OBDDs profitieren. Der Vergleich soll erkennen helfen, ob eine etwaige Leistungssteigerung des MBGP-Systems gegenüber herkömmlichen GP-Systemen mit S-Expressions nur an der Verwendung von OBDDs liegt. Nur wenn das MBGP-System auch besser als das OBDD-GP-System abschneidet, haben wir einen Hinweis, dass die Einhaltung der MBEA-Richtlinien leistungssteigernd ist.

Hauptaugenmerk soll auf die Verbesserung des Suchprozesses gelegt werden,

d. h. die Verringerung des Computational Efforts. Der Zeit- und Speicheraufwand der Systeme wird nicht näher empirisch untersucht, da er von der genutzten Hardware und Implementierungsdetails abhängt. Jedoch wird versucht, den Speicher- und Zeitbedarf der verschiedenen Komponenten der GP-Systeme asymptotisch abzuschätzen. Während die effiziente Speichernutzung und Durchführbarkeit der Operatoren keinen positiven Einfluss auf die mittels des Computational Efforts abschätzbare Güte des Suchprozesses haben, sollte dies für die MBEA-konformen genetischen Operatoren anders sein. Dabei werden sich beide GP-Systeme im empirischen Vergleich für die gewählten Aufgaben gegenüber bisherigen Systemen mit S-Expressions als überlegen erweisen.

4.4.1 Die zu lernenden Funktionen und die Metrik

Das Ziel des GP-Systems wird es sein, eine Darstellung einer implizit gegebenen Funktion $f^* : \{0,1\}^n \mapsto \{0,1\}$ zu lernen. Dabei gehen wir wie in Abschnit 4.2 beschrieben davon aus, dass auf die vollständige Trainingsmenge T nicht direkt zugegriffen werden, sondern nur indirekt über eine Zielfunktion $F : \{f \mid f : \{0,1\}^n \mapsto \{0,1\}\} \mapsto \{0,\ldots,2^n\}$, definiert als

$$F(f) := |\{x \in \{0,1\}^n \mid f(x) = y \wedge (x,y) \in T\}|.$$

Sowohl in dieser Variante einer vollständigen als auch einer unvollständigen Trainingsmenge in Kapitel 5 werden als zugrundeliegende Funktionen die *Multiplexer*- und *Parity*-Funktion benutzt:

Definition 4.4.1 *Sei $k \in \mathbb{N}$ und $n = 2^k + k$. Die n-Multiplexer-Funktion* $\textsc{Mux}_n : \{0,1\}^n \mapsto \{0,1\}$ *ist definiert als*

$$\textsc{Mux}_n(a_0,\ldots,a_{k-1},d_0,\ldots,d_{2^k-1}) := d_{|a|} \ \textit{mit}\ |a| := \sum_{i=0}^{k-1} 2^i \cdot a_i.$$

Definition 4.4.2 *Sei $n \in \mathbb{N}$. Die n-Parity-Funktion* $\textsc{Par}_n : \{0,1\}^n \mapsto \{0,1\}$ *ist definiert als*

$$\textsc{Par}_n(x_0,\ldots,x_{n-1}) := \left(\sum_{i=0}^{n-1} x_i\right) \text{ MOD } 2.$$

Während für die Auswertung der Multiplexer-Funktion nur die *Adressvariablen* $a_0,\ldots,a_{k-1}$ und die dadurch adressierte *Datenvariable* getestet werden müssen, ist zur Bestimmung des Funktionswerts der Parity-Funktion für jede Eingabe der Test aller Variablen notwendig.

Da reduzierte OBDDs zu einer festen Variablenordnung betrachtet werden, ist die Kodierungsfunktion $h : P \mapsto G$ bijektiv. Somit überträgt sich eine Metrik d_P auf dem Phänotyp-Raum P wie in Abschnitt 4.1 besprochen auf eine Metrik d_G auf dem Genotyp-Raum G. Als Metrik d_P auf der Menge der Funktionen $P = \{f : \{0,1\}^n \mapsto \{0,1\}\}$, so dass die Fitness zweier Funktionen höchstens um ihren Abstand voneinander abweichen kann, bietet sich die Anzahl der Eingaben an, auf denen die Funktionen unterschiedliche Ausgaben haben:

$$d_P^*(f_1,f_2) := |\{x \in \{0,1\}^n \mid f_1(x) \neq f_2(x)\}|.$$

Es ist klar, dass d_P^* eine Metrik ist. Ziel unserer Konstruktion der MBEA-konformen Operatoren muss also sein, bzgl. d_P^* bzw. der dadurch definierten Metrik d_G^* auf dem Genotyp-Raum der reduzierten OBDDs die MBEA-Richtlinien zu erfüllen.

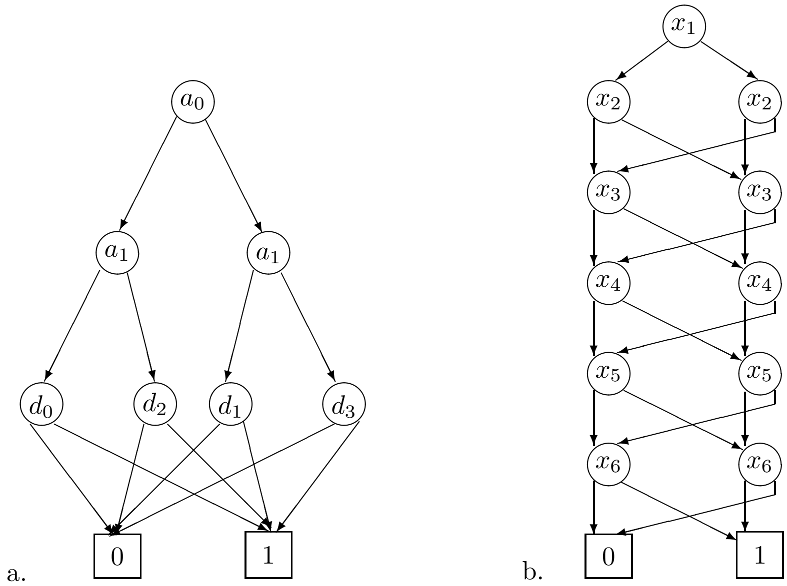

Abbildung 4.1: Zwei reduzierte OBDDs für die Funktionen MUX_6 (a.) und PAR_6 (b.). Dabei ist die linke bzw. rechte ausgehende Kante stets die zum 0- bzw. 1-Nachfolger.

4.4.2 Repräsentation

Die Individuen unseres GP-Systems werden durch die in Abschnitt 4.3 definierten reduzierten OBDDs repräsentiert. Dabei wird die beschriebene Reduzierung der OBDDs während ihrer Erzeugung benutzt, wodurch zwar eine dynamische Datenstruktur (in diesem Fall eine Hash-Tabelle) angelegt werden muss, jedoch zu keinem Zeitpunkt nicht-reduzierte OBDDs im Speicher stehen. Da verschiedene OBDDs gemeinsame Substrukturen gemeinsam nutzen, müsste genauer von einem SBDD gesprochen werden. Weil dieses jedoch gegenüber der separaten Abspeicherung der einzelnen OBDDs keinen Gewinn an zusätzlichen Möglichkeiten für Operatoren liefert, werden wir wie bisher stets von OBDDs als Repräsentationsform sprechen.

Die Variablenordnung π aller OBDDs in einem Lauf des GP-Systems wird stets gleich sein. Dies erst ermöglicht in vielen Fällen die effiziente Durchführbarkeit von wichtigen Operationen, wirft aber auch das Problem der richtigen Wahl von π auf. Denn abhängig von der Variablenordnung kann es für eine Funktion reduzierte OBDDs polynomieller bzw. nur exponentieller Größe geben (Wegener (2000)). Ein leicht nachzuvollziehendes Beispiel ist die Multiplexer-Funktion MUX_n mit $n = k + 2^k$: stehen in der Reihenfolge π die Adressvariablen vor den Datenvariablen, so hat das reduzierte π-OBDD genau $2^k - 1 + 2^k = O(n)$ Knoten; stehen jedoch die Datenvariablen vor den Adressvariablen, so besitzt jedes π-OBDD mehr als $2^{2^k} = \Omega(2^n)$ Knoten, da keine der 2^{2^k} Kombinationsmöglichkeiten der Datenvariablen „vergessen“ werden darf, bevor eine Adressvariable getestet wurde. Für die Parity-Funktion ergibt sich unabhängig von der gewählten Variablenordnung stets ein OBDD mit $2 \cdot n - 1$ Knoten (siehe Abbildung 4.1).

Die Variablenordnung π wird in dem GP-System stets auf eine vorgegebene optimale Ordnung für die zu lernende Funktion gesetzt, d. h. $\pi = (a_0, \ldots, a_{k-1}, d_0, \ldots, d_{2^k-1})$ für MUX_n und $\pi = (x_0, \ldots, x_{n-1})$ für PAR_n. Natürlich ist dies für den praktischen Einsatz nicht durchzuführen, da dann die zu lernende Funktion und somit eine für sie gute Variablenordnung nicht bekannt ist (selbst bei Kenntnis eines OBDDs ist das Problem, ob es eine Variablenordnung gibt, die zu einem äquivalenten OBDD einer gegebenen Maximalgröße führt, NP-vollständig (Bollig und Wegener (1996)) und sogar die Approximation in einer konstanten Genauigkeit ist nicht in polynomieller Zeit möglich, falls NP $\neq$ P ist (Sieling (1998))). Die Variablenordnung wird zwar die Größe der Darstellung und somit die Effizienz der Operatoren stark beeinflussen, doch wird sie den Suchprozess, d. h. die Wahrscheinlichkeit, mit der eine Funktion aus einer Mutation oder Rekombination entspringt, nicht augenscheinlich negativ beeinflußen. Da wir uns auf den Computational Effort stützen, wird das Variablenordnungsproblem hier nicht weiter betrachtet.

4.4.3 Initialisierung

Für die Initialisierung ist die naheliegendste Wahl stets die Gleichverteilung in dem betrachteten Suchraum. Eine gleichverteilte Auswahl unter allen Funktionen $f : \{0,1\}^n \mapsto \{0,1\}$ kann dadurch erreicht werden, dass für jede Eingabe aus $\{0,1\}^n$ gleichverteilt zufällig die Ausgabe 0 oder 1 gewählt wird und dann ein reduziertes π-OBDD für die so spezifizierte Funktion konstruiert wird. Da jedes reduzierte π-OBDD genau eine Funktion darstellt und jede Funktion genau ein reduziertes π-OBDD besitzt, erzeugt dieses Vorgehen auch eine Gleichverteilung auf der Menge der reduzierten π-OBDDs.

Problematisch an dieser Lösung ist, dass zu jeder Variablenordnung π ein exponentiell großer Anteil aller Funktionen von $\{0,1\}^n$ nach $\{0,1\}$ nur π-OBDDs exponentieller Größe besitzen, was man durch Abzählargumente, wie in der Diskussion der Kolmogoroff-Komplexität in Kapitel 2 präsentiert, leicht nachvollziehen kann. Dies gilt für jede Darstellungsform boolescher Funktionen, da ihre Anzahl doppelt exponentiell in der Dimension n des Eingaberaums ist. Daraus folgt natürlich, dass auch jeder Algorithmus zur gleichverteilt zufälligen Initialisierung mit exponentiell großer Wahrscheinlichkeit exponentielle Zeit und exponentiellen Platz benötigt.

Eine schnellere Initialisierung ist also nur möglich, wenn nicht alle Ausgabewerte gleichverteilt unabhängig voneinander gewählt werden. Eine Möglichkeit hierfür ist die folgende: wähle als Wurzel des pseudozufälligen π-OBDDs einen Knoten mit Markierung $x_{\pi(1)}$, der aktueller Knoten wird. Wenn der aktuelle Knoten Markierung $x_{\pi(i)}$ hat, so bestimme zufällige Zahlen $\delta_0, \delta_1 \in \{1, \ldots, n-i+1\}$ und erzeuge mit demselben Vorgehen rekursiv pseudozufällige OBDDs als Null- bzw. Einsnachfolger des aktuellen Knotens, deren Wurzeln mit $x_{\pi(i+\delta_0)}$ bzw. $x_{\pi(i+\delta_1)}$ markiert sind. Falls ein Index größer als n ist, so wähle statt eines inneren Knotens gleichverteilt die 0- oder 1-Senke aus, wodurch die rekursive Anwendung in diesem Zweig stoppt. Somit lässt sich dieser Operator folgendermaßen zusammenfassen:

Algorithmus 4.4.3 (Pseudozufällige Initialisierung) *Sei Δ_i eine Zufallsvariable mit Werten in $\{1, \ldots, n-i+1\}$. Die Wurzel eines pseudozufälligen π-OBDDs wird durch den Aufruf von* `Erzeuge_Pseudozufaelliges_OBDD(1)` *zurückgegeben, wobei die Prozedur folgendermaßen definiert ist:*

`Erzeuge_Pseudozufaelliges_OBDD(i)`*:*

1. *Falls $i > n$, gib mit Wahrscheinlichkeit 1/2 die Nullsenke zurück, ansonsten die Einssenke. Beende den rekursiven Aufruf.*
2. *Seien $\delta_0, \delta_1 \in \{1, \ldots, n-i+1\}$ unabhängig voneinander gemäß Δ_i gezogen.*
3. *Setze $v_0 :=$* `Erzeuge_Pseudozufaelliges_OBDD(i` $+ \delta_0$`)`.
4. *Setze $v_1 :=$* `Erzeuge_Pseudozufaelliges_OBDD(i` $+ \delta_1$`)`.
5. *Gib einen Knoten mit Markierung $x_{\pi(i)}$, 0-Nachfolger v_0 und 1-Nachfolger v_1 zurück.*

Je nach Verteilung von Δ_i werden sich OBDDs unterschiedlicher erwarteter Größe ergeben: ist Δ_i konstant gleich 1, so erzeugt `Erzeuge_Pseudozufaelliges_OBDD(1)` ein OBDD einer vollkommen gleichverteilt ausgewählten Funktion und benötigt deshalb mit hoher Wahrscheinlichkeit exponentielle Laufzeit. Je mehr die Verteilung von Δ_i größere Werte bevorzugt, desto kleiner wird das resultierende OBDD im Erwartungswert werden.

Für die von uns in den Experimenten betrachteten Werte von n aus dem Bereich von 3 bis 12 wird für Δ_i eine geometrische Verteilung mit Parameter $p = 1/4$ gewählt, die so modifiziert wurde, dass sie keine Werte größer als $n-i+1$ annehmen kann:

Definition 4.4.4 *Für $i \in \{1, \ldots, n\}$ hat die Zufallsvariable Δ_i die Verteilung:*

$$\forall j \in \{1, \ldots, n-i+1\} : \mathrm{P}(\Delta_i = j) = \begin{cases} p \cdot (1-p)^{j-1} & \text{, falls } j \in \{1, \ldots, n-i\}, \\ (1-p)^{n-i} & \text{, falls } j = n-i+1. \end{cases}$$

Da

$$\sum_{j=1}^{n-i} p \cdot (1-p)^{j-1} = p \cdot \frac{1-(1-p)^{n-i}}{1-(1-p)} = 1-(1-p)^{n-i}$$

ist dies eine korrekte Wahrscheinlichkeitsverteilung. Sowohl im MBGP- als auch im OBDD-GP-System wird Algorithmus 4.4.3 zur jeweils unabhängigen Initialisierung aller Individuen eingesetzt.

4.4.4 Mutation

Die Grundidee einer Mutation einer S-Expression ist die Ersetzung eines an einem Knoten beginnenden Teilstücks der S-Expression durch ein neues, zufällig gewähltes Teilstück. Bei der Erzeugung des neuen Teilstücks wird dabei nur die einzuhaltende Maximaltiefe der S-Expression berücksichtigt. Überträgt man diese Idee auf ein π-OBDD D, indem man einen Knoten v des OBDDs zufällig auswählt und das dort beginnende TeilOBDD durch ein zufällliges ersetzt, so muss man zwei Probleme lösen:

1. Die Markierungen der Knoten des zufällig erzeugten TeilOBDDs müssen so gesetzt sein, dass das neue OBDD die Variablenordnung π berücksichtigt.

2. Da mit reduzierten OBDDs gearbeitet wird, haben diese in der Regel keine Baum-Struktur. Es kann also Kanten geben, die auf Knoten in dem an v beginnenden TeilOBDD zeigen, jedoch nicht von einem Knoten aus diesem TeilOBDD ausgehen. Eine Ersetzung des an v beginnenden TeilOBDDs erfordert, dass diese Kanten geeignet „umgebogen“ oder von den Knoten des TeilOBDDs Kopien erzeugt werden.

Beide Probleme können nicht getrennt voneinander gelöst werden. Der hier vorgeschlagene Mutationsoperator wählt zuerst gleichverteilt eine Zahl $j \in \{1, \ldots, |D|\}$ aus. In einer anschließenden Tiefensuche, bei der jeweils gleichverteilt zufällig zuerst der Null- bzw. Einsnachfolger besucht wird, wird der j-te besuchte Knoten mitsamt eines Pfades von der Quelle zu ihm zurückgegeben. An diesem Knoten v wird dann das zu ersetzende TeilOBDD beginnen. Die oben angesprochenen Punkte werden dann wie folgt gelöst:

1. Wenn $x_{\pi(i)}$ die Markierung von v ist, so wird das neue TeilOBDD eine Wurzel mit Markierung $x_{\pi(i-1+\delta)}$ besitzen, wobei δ nach der Verteilung der Zufallsvariablen Δ_{i-1} (Definition 4.4.4) gewählt ist. Das TeilOBDD wird durch Aufruf von `Erzeuge_Pseudozufaelliges_OBDD`$(\mathtt{i}-1+\delta)$ erzeugt. Somit hat

die Quelle des neuen TeilOBDDs eine gemäß π mindestens so große Markierung wie v, weshalb alle Kanten zu v auf die neue Wurzel umgebogen werden könnten, ohne die Variablenordnung π zu verletzen.

2. Sei mit $D(v)$ das an v beginnende TeilOBDD bezeichnet. Dann werden alle Kanten, die von Knoten außerhalb von $D(v)$ ausgehen und zu einem Knoten in $D(v)$ führen, der nicht der Knoten v ist, weiterhin auf diesen Knoten verweisen. Jeweils mit Wahrscheinlichkeit 1/2 wird entweder nur die Kante zu v, die auf dem bei der Auswahl von v festgelegten Pfad liegt, zu der Quelle des neuen TeilOBDDs umgebogen oder alle Kanten, die in dem bisherigen OBDD auf v zeigen. (Im letzteren Fall wird v, falls er keine eingehenden Kanten mehr besitzt, aus dem Speicher gelöscht.) Obwohl das OBDD mit allen anderen OBDDs in einem SBDD gespeichert wird, bleiben alle anderen Individuen von der Mutation unberührt.

   Denkt man sich das OBDD als einen binären Entscheidungsbaum, so wird also mit Wahrscheinlichkeit 1/2 entweder das TeilOBDD $D(v)$ nur am Ende des gewählten Pfads zu v durch ein neues TeilOBDD ersetzt oder es werden alle Vorkommen von $D(v)$ ersetzt. Während erstere Möglichkeit versucht, die Veränderung durch die Mutation möglichst gering zu lassen, berücksichtigt die zweite Möglichkeit die implizite Rolle von TeilOBDDs als ADFs, wie in Abschnitt 4.3 besprochen. Denn falls das neue TeilOBDD eine Subfunktion darstellt, die an möglichst vielen Stellen in der Funktion sinnvoll eingesetzt werden kann, könnte eine simultane Ersetzung an allen Kanten zu v nützlich sein.

Natürlich gibt es andere Möglichkeiten, die auf S-Expressions übliche Mutation auf OBDDs zu übertragen, doch ist diese relativ naheliegend und wird deshalb im Folgenden betrachtet. Diese Mutation basiert auf der syntaktischen Struktur der OBDDs: die Auswirkungen auf die Struktur sind klar nachzuvollziehen, während dies für diejenigen auf die dargestellte Funktion nicht gilt. Der Operator sei hier schematisch zusammengefasst:

Algorithmus 4.4.5 (OBDD-GP-**Mutation**)
Eingabe: *ein zu mutierendes π-OBDD D.*
Ausgabe: *ein mutiertes π-OBDD D'.*

1. *Wähle gleichverteilt ein $j \in \{1, \ldots, |D|\}$.*

2. *Wähle durch eine Tiefensuche von der Quelle von D aus den j-ten so besuchten Knoten v von D, wobei jeweils mit Wahrscheinlichkeit 1/2 rekursiv zuerst zum Null- bzw. Einsnachfolger verzweigt wird. Sei $(q, \ldots, v', v)$ der dabei gewählte Weg von der Quelle q von D zu v.*

3. *Erzeuge durch den Aufruf* `Erzeuge_Pseudozufaelliges_OBDD(i − 1 + δ)` *ein pseudozufälliges π-OBDD D^*, wobei $x_{\pi(i)}$ die Markierung von v ist und δ nach der Verteilung von Δ_{i-1} gewählt wurde.*

4. *Bilde eine Kopie D' von D.*

5. *Setze in D' die Kante von v' zu v auf die Wurzel von D^*.*

6. *Mit Wahrscheinlichkeit 1/2 durchlaufe in einer Tiefensuche alle Knoten von D' und setze jede Kante, die auf v zeigt, auf die Wurzel von D^*.*

7. *Gib D' aus.*

Da ein OBDD D mit $|D|$ Knoten genau $2 \cdot |D|$ Kanten besitzt, ist die Laufzeit der Tiefensuche in D von der Größenordnung $O(|D|)$. Neben der Erzeugung des zufälligen OBDDs D' ist dies der zeitaufwendigste Schritt in der syntaktischen Mutation.

Die OBDD-GP-Mutation ermöglicht zwar, dass sich aus einer Mutation jede beliebige Funktion ergeben kann (M 1), jedoch ist nicht klar, ob die Richtlinien M 2 und M 3 erfüllt sind. Sie macht von den Möglichkeiten der effizienten Synthese zweier OBDDs keinen Gebrauch.

Für eine MBEA-Mutation, die die von D repräsentierte Funktion f_D nur gezielt verändern soll, bietet sich die EXOR-Synthese an ($\oplus$ bezeichnet dabei die Exklusiv-Oder-Verknüpfung): denn die Funktion $f_D \oplus f_{D_M}$ unterscheidet sich genau für die Eingaben von f_D, für die f_{D_M} die Ausgabe 1 liefert. Identifiziert man eine Funktion $f : \{0,1\}^n \mapsto \{0,1\}$ mit dem String seiner Ausgabebits für alle 2^n möglichen Eingaben, so kann man eine zu der des (1+1) EA äquivalente Mutation erreichen, indem D_M so ausgewählt wird, dass dieses zu jeder Eingabe unabhängig voneinander mit Wahrscheinlichkeit $1/2^n$ die Ausgabe 1 liefert.

Da somit eine Funktion f_1 durch Mutation genau mit Wahrscheinlichkeit

$$\left(\frac{1}{2^n}\right)^{d_P^*(f_1,f_2)} \cdot \left(1 - \frac{1}{2^n}\right)^{2^n - d_P^*(f_1,f_2)}$$

zu einer Funktion f_2 mutiert wird, hat

- jede Funktion eine positive Wahrscheinlichkeit, durch Mutation gebildet zu werden (M 1),
- eine Funktion mit größerem Abstand von der Ausgangsfunktion echt geringere Wahrscheinlichkeit, durch Mutation gebildet zu werden (M 2), und
- jede Funktion mit gleichem Abstand von der Ausgangsfunktion die gleiche Wahrscheinlichkeit, durch Mutation gebildet zu werden.

Somit erfüllt folgende Mutation alle MBEA-Anforderungen und wird deshalb als MBGP-Mutation bezeichnet:

Algorithmus 4.4.6 (MBGP**-Mutation)**
Eingabe: *ein zu mutierendes π-OBDD D.*
Ausgabe: *ein mutiertes π-OBDD D'.*

1. *Setze $i := -1$ und D_M auf die 0-Senke.*
2. *Wähle den Wert $d \in \mathbb{N}$ unter der geometrischen Verteilung mit Parameter $1/2^n$ (A.5).*
3. *Setze $i := i + d$.*
4. *Falls $i < 2^n$, erzeuge ein π-OBDD $\tilde{D}$, das genau für die i-te Eingabe 1 ausgibt, setze $D_M := D_M \vee \tilde{D}$ und gehe zu Schritt 2.*
5. *Gib das Ergebnis von $D \oplus D_M$ aus.*

Im Erwartungswert wird D_M nur für eine Eingabe den Wert 1 liefern und deshalb erwartete Größe $O(n)$ besitzen. Somit lässt sich die Synthese in erwarteter Zeit $O(|D| \cdot n)$ durchführen, was polynomiell in n ist, solange dies für $|D|$ gilt. Um das OBDD D_M zu erzeugen, ist bei naiver Umsetzung der Mutation des (1+1) EA lineare Zeit in der Zahl der möglicherweise mutierenden Stellen notwendig, d. h. Zeit $\Omega(2^n)$. In der MBGP-Mutation wird geschickter vorgegangen, indem nicht für jede Eingabe, d. h. Stelle im Ausgabestring entschieden wird, ob die entsprechende

Ausgabe 1 ist, sondern nur die nächste Eingabe bestimmt wird, die von D_M auf 1 abgebildet wird.

Identifiziert man die Eingabemenge $\{0,1\}^n$ mit $\{0,\dots,2^n-1\}$ (z. B. über die Binärdarstellung), erhält man eine eindeutige Ordnung auf der Eingabemenge. Hat man für ein beliebiges $i \in \{0,\dots,2^n-1\}$ festgelegt, dass die i-te Eingabe mutiert wird, so wird als nächstes die $(i+d)$-te Eingabe mit Wahrscheinlichkeit

$$\left(1-\frac{1}{2^n}\right)^{d-1}\cdot\frac{1}{2^n}$$

mutieren. Die Zufallsvariable, die den Abstand zu der nächsten zu mutierenden Eingabe angibt, ist also geometrisch mit Parameter $1/2^n$ verteilt (A.5). Dies scheint noch keine wesentliche Vereinfachung zu sein. Ist jedoch μ gleichverteilt aus dem halboffenenen Intervall $[0,1[$ gewählt, so ist der Wert

$$1+\left\lfloor\frac{\ln(1-\mu)}{\ln(1-p)}\right\rfloor$$

geometrisch mit Parameter p verteilt, da

$$\begin{array}{rcl} & 1+\left\lfloor\dfrac{\ln(1-\mu)}{\ln(1-p)}\right\rfloor & = d \\ \Longleftrightarrow & \dfrac{\ln(1-\mu)}{\ln(1-p)} & \in [d-1,d[\\ \Longleftrightarrow & \ln(1-\mu) & \in\,]d\cdot\ln(1-p),(d-1)\cdot\ln(1-p)] \\ \Longleftrightarrow & 1-\mu & \in [(1-p)^d,(1-p)^{d-1}[\end{array}$$

ist. Weil $(1-p)^{d-1}-(1-p)^d = p\cdot(1-p)^{d-1}$, lässt sich der Abstand zu der nächsten mutierenden Eingabe also mittels einer aus $[0,1[$ gleichverteilten Zufallsvariablen in einem Schritt bestimmen, wenn $p=1/2^n$ gesetzt wird. Somit lässt sich die MBGP-Mutation in erwarteter Zeit $O(|D|\cdot n)$ durchführen.

4.4.5 Rekombination

Auch für die Rekombination sollen nun zwei mögliche Umsetzungen auf OBDDs beschrieben werden. Die erste wird sich an den Austausch von Teilausdrücken in S-Expressions, die zweite, MBEA-konforme, an den Synthesealgorithmus für OBDDs anlehnen.

Sei zuerst der Ansatz erläutert, der in Analogie zur Rekombination bei S-Expressions zwei TeilOBDDs auswählt und diese vertauscht. Dies wirft folgende Probleme auf:

1. Wird in jedem π-OBDD jeweils ein Knoten gleichverteilt ausgewählt, so werden diese in der Regel unterschiedliche Markierungen haben, weshalb ein Austausch der dort beginnenden TeilOBDDs zu Strukturen führt, die die Variablenordnung π verletzen. Deshalb muss die Auswahl der Knoten aufeinander abgestimmt werden.

2. Genau wie bei der Mutation müssen Kanten, die von außerhalb in den ausgewählten Teil des OBDDs weisen, ggf. geeignet auf neue Knoten „umgebogen“ werden.

Um das erste Problem zu lösen, wird, nachdem in dem zufällig gewählten ersten π-OBDD D_1, wie bei der Mutation beschrieben, ein Knoten v_1 gleichverteilt ausgewählt wird, in dem zweiten π-OBDD D_2 nur unter den Knoten, deren Markierung nach der Variablenreihenfolge π mindestens so groß ist wie die von v_1, ein Knoten v_2 gleichverteilt ausgewählt. Somit kann jede Kante, die in D_1 auf v_1 zeigt, in dem Nachfolger von D_1 auf v_2 umgebogen werden.

Umgekehrt kann jedoch eine Kante in D_2, die auf v_2 zeigt, nicht immer auf v_1 umgebogen werden, da die Variable von v_1 gemäß π vor der von v_2 stehen kann. Dies lösen wir dadurch, dass nur das durch Ersetzung des bei v_1 beginnenden TeilOBDD erzeugte π-OBDD als Kind entsteht. Eine gleichzeitige Erzeugung zweier Kinder würde erzwingen, dass die ausgewählten Knoten dieselbe Markierung haben müssen, was die Auswahlmöglichkeit derselben stark einschränkt.

Somit lässt sich diese Rekombination folgendermaßen beschreiben:

Algorithmus 4.4.7 (OBDD-GP-**Rekombination)**
Eingabe: *zwei zu rekombinierende π-OBDDs D_1 und D_2.*
Ausgabe: *ein rekombiniertes π-OBDD D'.*

1. *Wähle gleichverteilt ein $j_1 \in \{1, \ldots, |D_1|\}$.*

2. *Wähle durch eine Tiefensuche von der Quelle von D_1 aus den j_1-ten so besuchten Knoten v_1 von D_1, wobei jeweils mit Wahrscheinlichkeit 1/2 rekursiv zuerst zum Null- bzw. Einsnachfolger verzweigt wird. Sei $(q, \ldots, v', v_1)$ der dabei gewählte Weg von der Quelle q von D_1 zu v_1.*

3. *Bestimme durch eine Tiefensuche die Zahl j' der Knoten in D_2, deren Markierung in der Reihenfolge π mindestens gleich x_i, der Markierung von v_1, ist und wähle gleichverteilt eine Zahl $j_2 \in \{1, \ldots, j'\}$ aus. Wähle durch eine Tiefensuche von der Quelle von D_2 aus den j_2-ten Knoten v_2 aus, der zu dieser Gruppe gehört, wobei jeweils mit Wahrscheinlichkeit 1/2 rekursiv zuerst zum Null- bzw. Einsnachfolger verzweigt wird.*

4. *Bilde eine Kopie D' von D_1.*

5. *Ersetze die Kante von v' zu v_1 in D' durch eine Kante nach v_2.*

6. *Mit Wahrscheinlichkeit 1/2 ersetze in einer Tiefensuche in D' jede Kante nach v_1 durch eine Kante nach v_2.*

7. *Gib D' aus.*

Der Zeitbedarf dieses Algorithmus wird von der Tiefensuche bestimmt und beträgt somit $O(\max\{|D_1|, |D_2|\})$. Während die Auswirkungen auf die Struktur der OBDDs gut nachzuvollziehen sind, ist nur schwer abzuschätzen, wie sich die von der OBDD-GP-Rekombination ausgegebene Funktion zu ihren Eltern D_1 und D_2 verhält, ob sie also zu den MBEA-Richtlinien konform ist.

Deshalb soll nun ein Rekombinationsoperator für OBDDs beschrieben werden, der die MBEA-Richtlinien R 1 und R 2 erfüllt. Da sich dieser in seiner Funktionsweise auf den Synthesealgorithmus für π-OBDDs stützt, sei zuerst dieser kurz beschrieben.

Die Aufgabe der Synthese ist es, bei Eingabe zweier OBDDs D_1 und D_2 zu derselben Variablenordnung π und einer booleschen Funktion $\otimes : \{0,1\}^2 \mapsto \{0,1\}$ das π-OBDD D' zu erzeugen, das die Funktion $f_{D_1} \otimes f_{D_2}$ darstellt. Dem besten bekannten Algorithmus liegt folgende Idee zugrunde: bestehen die OBDDs jeweils nur aus einer Senke, so ist die Senke, deren Markierung sich durch Anwendung von $\otimes$ auf die Markierungen der Senken ergibt, das Resultat der Synthese. Ist zumindest eine der Wurzeln der zwei OBDDs ein innerer Knoten, so werden die 0- und 1-Nachfolger, falls vorhanden, rekursiv behandelt, bis sich der erstgenannte Fall ergibt. Dabei wird jeweils ein innerer Knoten mit der gemäß π kleineren Markierung der beiden Knoten zurückgegeben:

Algorithmus 4.4.8
$\texttt{Synthese\_rekursiv}(\mathtt{v_1}, \mathtt{v_2}, \otimes)$
Eingabe: *zwei Knoten v_1 und v_2 zweier π-OBDDs und ein Operator $\otimes : \{0,1\}^2 \mapsto \{0,1\}$.*
Ausgabe: *die Wurzel v' eines π-OBDDs, das $f_{v_1} \otimes f_{v_2}$ repräsentiert.*

1. *Ist v_1 eine a_1-Senke und v_2 eine a_2-Senke, so gib die $\otimes(a_1, a_2)$-Senke zurück.*

2. *Ist v_1 ein innerer Knoten und v_2 eine Senke oder ein innerer Knoten, dessen Markierung gemäß π nach der von v_1 steht, so gib einen Knoten zurück, der dieselbe Markierung wie v_1 trägt und dessen Null- bzw. Einsnachfolger gleich* $\texttt{Synthese\_rekursiv}(\mathtt{v_1} \to \mathtt{0}, \mathtt{v_2}, \otimes)$ *bzw.* $\texttt{Synthese\_rekursiv}(\mathtt{v_1} \to \mathtt{1}, \mathtt{v_2}, \otimes)$ *ist.*

3. *Ist v_2 ein innerer Knoten und v_1 eine Senke oder ein innerer Knoten, dessen Markierung gemäß π nach der von v_2 steht, so gib einen Knoten zurück, der dieselbe Markierung wie v_2 trägt und dessen Null- bzw. Einsnachfolger gleich* $\texttt{Synthese\_rekursiv}(\mathtt{v_1}, \mathtt{v_2} \to \mathtt{0}, \otimes)$ *bzw.* $\texttt{Synthese\_rekursiv}(\mathtt{v_1}, \mathtt{v_2} \to \mathtt{1}, \otimes)$ *ist.*

4. *Sind v_1 und v_2 innere Knoten mit derselben Markierung x_i, so gib einen Knoten mit Markierung x_i zurück, dessen Null- bzw. Einsnachfolger durch* $\texttt{Synthese\_rekursiv}(\mathtt{v_1} \to \mathtt{0}, \mathtt{v_2} \to \mathtt{0}, \otimes)$ *bzw.* $\texttt{Synthese\_rekursiv}(\mathtt{v_1} \to \mathtt{1}, \mathtt{v_2} \to \mathtt{1}, \otimes)$ *gegeben sind.*

Die Korrektheit dieses Algorithmus folgt leicht durch eine Bottom-Up-Induktion (siehe auch Wegener (2000)). Wird diese Prozedur mit den Wurzelknoten von D_1 und D_2 aufgerufen, so gibt sie die Wurzel des π-OBDDs für $f_1 \otimes f_2$ zurück. Die Laufzeit beträgt $O(|D_1| \cdot |D_2|)$, wenn die Ergebnisse schon berechneter Aufrufe $\texttt{Synthese\_rekursiv}(\mathtt{v_1}, \mathtt{v_2}, \otimes)$ abgespeichert und vor einem rekursiven Aufruf mit den aktuellen Parametern verglichen und ggf. nur zurückgegeben werden.

Wie sähe eine geeignete Synthesefunktion $\odot$ aus, damit das Resultat der Synthese die MBEA-Richtlinien an eine Rekombination erfüllt? Der Nachkomme soll in der Nähe der Eltern liegen, d. h. für Eingaben, die von beiden Eltern-OBDDs gleich abgebildet werden, hat der Nachkomme keinen Grund, von diesem Verhalten abzuweichen. Unterscheiden sich die Ausgaben der Eltern-OBDDs, so sollte eine der beiden Ausgaben übernommen werden. Da in der Rekombination die Fitness der Eltern nicht in Betracht gezogen wird, sollten beide Ausgaben gleich wahrscheinlich sein. Damit würde sich ein randomisierter Operator $\odot : \{0,1\}^2 \mapsto \{0,1\}$ gemäß

$$\odot(x_1, x_2) := \begin{cases} x_1 & \text{mit Wahrscheinlichkeit } 1/2, \\ x_2 & \text{sonst,} \end{cases}$$

anbieten, der genau dem uniformen Crossover auf den Strings der Funktionswerte zu entsprechen scheint. Dies ergibt folgenden Rekombinationsalgoritmus:

Algorithmus 4.4.9 (MBGP**-Rekombination)**
Eingabe: *zwei zu rekombinierende π-OBDDs D_1 und D_2.*
Ausgabe: *ein rekombiniertes π-OBDD D'.*

1. *Sei v_1 bzw. v_2 gleich der Quelle von D_1 bzw. D_2.*

2. *Rufe* $\texttt{Synthese\_rekursiv}(\mathtt{v_1}, \mathtt{v_2}, \odot)$ *auf und gib das an dem zurückgegebenen Knoten v' beginnende π-OBDD D' aus.*

Dabei speichern wir die Ergebnisse schon erfolgter Aufrufe nicht in einer dynamischen Datenstruktur, da sonst eine größere Abhängigkeit zwischen den einzelnen rekursiven Aufrufen entsteht, die die Wirkungsweise der MBGP-Rekombination noch schwerer nachvollziehbar machen würde. Jedoch kann dies (siehe unten) einen exponentiellen Zeit- und Platzbedarf bedeuten.

Der Synthesealgorithmus 4.4.8 mit dieser Funktion $\odot$ wird im Allgemeinen nicht wie das uniforme Crossover auf den Strings der Ausgabebits arbeiten. Denn ist z. B. D_1 gleich der konstanten 0-Funktion und D_2 gleich der konstanten 1-Funktion, so wird das Resultat von Algorithmus 4.4.8 jeweils mit Wahrscheinlichkeit 1/2 die 0- oder die 1-Senke sein. Damit werden sich nicht wie beim uniformen Crossover alle Funktionen jeweils mit Wahrscheinlichkeit 2^{-2^n} ergeben. Dies liegt darin begründet, dass der Synthesealgorithmus nur für nicht-randomisierte Funktionen korrekt arbeitet. Für diese ist garantiert, dass die Größe des neuen OBDDs maximal gleich dem Produkt der Größen der beiden beteiligten OBDDs ist, was in dem betrachteten Beispiel nur mit Wahrscheinlichkeit 2^{-2^n+1} der Fall ist.

Wie kann die Verteilung des resultierenden OBDDs in Abhängigkeit von D_1 und D_2 beschrieben werden? Betrachtet man den Synthesealgorithmus, so sieht man, dass eine Auswertung von $\odot$ und damit eine Zufallsentscheidung nur erfolgt, wenn in beiden OBDDs jeweils eine Senke erreicht wurde. Welche Eingaben werden von einer solchen Entscheidung beeinflusst?

Dazu werden wir die von einem OBDD dargestellte Funktion auf eine andere Art und Weise als bisher charakterisieren. Wenn $P = (v_1, \ldots, v_l, s)$ $(l \leq n)$ ein Pfad von der Wurzel v_1 zu der Senke s ist, so gibt die Senke s für die Eingaben $a \in \{0,1\}^n$ die Ausgabe an, die diesem Pfad „gefolgt" sind. Dies bedeutet, dass für alle $i \in \{1, \ldots, l\}$ der Wert a_i gleich dem Index der Kante sein muss, der auf P ggf. beim Verlassen des mit x_i markierten Knotens gefolgt wurde. Somit ist in unserem Zusammenhang ein Pfad P von der Quelle zu einer der Senken dadurch ausreichend beschrieben, dass festgelegt ist, welche Variablen wie getestet werden. Somit können wir P durch einen Vektor $a(P) \in \{0, 1, *\}$ beschreiben, wobei

$$a_i(P) = \begin{cases} * & \text{, falls } P \text{ keine von einem } x_i\text{-Knoten ausgehende Kante enthält,} \\ 0 & \text{, falls } P \text{ eine von einem } x_i\text{-Knoten ausgehende 0-Kante enthält,} \\ 1 & \text{, falls } P \text{ eine von einem } x_i\text{-Knoten ausgehende 1-Kante enthält.} \end{cases}$$

Da in einem OBDD auf jedem Pfad jede Markierung höchstens einmal vorkommt, ist zu einem Pfad P der Vektor $a(P)$ eindeutig definiert. Die Menge $C(a)$ der Eingaben, die einem Pfad $a \in \{0, 1, *\}$ folgt, ist

$$C(a) := \{x \in \{0,1\}^n \,|\, \forall i \in \{1, \ldots, n\} \text{ mit } a_i \neq * : x_i = a_i\}.$$

Erreicht der Synthesealgorithmus 4.4.8 in D_1 über den Vektor a_1 und in D_2 über den Vektor a_2 eine Senke, so wird eine der beiden Senken gleichverteilt ausgewählt. In dem resultierenden OBDD D' wird diese Senke an das Ende des Pfades gesetzt, dem genau für die Eingaben aus $C(a_1) \cap C(a_2)$ gefolgt wird. Denn auf diesem Pfad werden genau die Variablen getestet, die auf a_1 *oder* a_2 getestet wurden. Da im Synthesealgorithmus ein rekursiver Aufruf, der in beiden OBDDs jeweils zum Nachfolger geht, nur erfolgt, wenn jeweils zum 0- bzw. 1-Nachfolger verzweigt wird, werden genau die Eingaben aus $C(a_1) \cap C(a_2)$ an dieser neuen Senke „ankommen".

Wenn für einen Pfad $a \in \{0, 1, *\}$ die Senke $s_D(a)$ erreicht wird, so kann das OBDD D also auch durch die Menge $M(D)$ aller Pfade zu Senken mitsamt der dazugehörigen Markierungen beschrieben werden:

$$M(D) := \{(C(a), s_D(a)) \,|\, a \text{ ist ein Pfad in } D \text{ zur } s_D(a)\text{-Senke } \}.$$

Diese offensichtlich äquivalente Beschreibungsweise ermöglicht eine einfache Beschreibung des funktionalen Verhaltens des OBDDs D':

$$M(D') = \{(C_1 \cap C_2, \odot(s_1, s_2)) \,|\, (C_1, s_1) \in M(D_1), (C_2, s_2) \in M(D_2)\}.$$

Denn nach den obigen Erläuterungen wird der Synthesealgorithmus nur für Paare von Pfaden zu Senken eine Funktionsauswertung von $\odot$ durchführen. Da keine bisherigen Resultate zwischengespeichert werden, können alle Paare von Pfaden dabei vorkommen. Paare von Pfaden, die im Algorithmus nicht gleichzeitig durchlaufen werden, führen zu einer leeren Schnittmenge $C_1 \cap C_2$, so dass genau alle richtigen Paare berücksichtigt werden.

Somit entspricht das Ergebnis der semantischen Rekombination nicht der jeweils unabhängigen Auswahl aller Ausgaben, sondern einer nur teilweise unabhängigen Auswahl: für zwei Pfade in D_1 bzw. D_2, die von mindestens einer Eingabe gemeinsam durchlaufen werden, wird nur einmal entschieden, welche der beiden Senken gewählt wird. Obwohl die Funktion $f_{D'}$ von der Form der OBDDs D_1 und D_2 abhängt und sich nur schwer formal fassen lässt, gilt für Algorithmus 4.4.9 trotzdem, dass er seinen Namen „MBGP-Rekombination" zu Recht trägt:

Lemma 4.4.10 *Der Rekombinationsalgorithmus 4.4.9 erfüllt die Richtlinien R 1 und R 2.*

Beweis: Richtlinie R 1 besagt, dass sich jeder mögliche Nachkomme D' nicht an mehr Eingaben von seinen Eltern D_1 und D_2 unterscheiden darf, als diese es tun. Direkt aus dem Ablauf der Rekombination folgt, dass sich D' nur an Stellen von D_1 oder D_2 unterscheiden kann, an denen diese es tun. Somit ist R 1 erfüllt.

Richtlinie R 2 besagt, dass für alle $\alpha \in \mathbb{N}$ die Wahrscheinlichkeit, dass sich D_1 und D' an α Eingaben unterscheiden, gleich der Wahrscheinlichkeit ist, dass sich D_2 und D' an α Eingaben unterscheiden. Seien die Eltern D_1 und D_2 und der Nachkomme D' als fest angenommen. Die Wahrscheinlichkeit, dass sich aus D_1 und D_2 der Nachkomme D' bildet, ergibt sich als das Produkt der Wahrscheinlichkeiten der Zufallsentscheidungen, die in Algorithmus 4.4.9 getroffen werden, wenn in D_1 und D_2 verschiedene Senken erreicht sind. Da die Entscheidungen jeweils gleichverteilt sind, hat die Folge der genau komplementären Ereignisse dieselbe Wahrscheinlichkeit. Das sich durch sie ergebende OBDD D'' unterscheidet sich von D' genau an den Eingaben, an denen sich D_1 und D_2 unterscheiden. Wenn also $d_G(D_1, D') = \alpha$ ist, so gilt $d_G(D_2, D'') = \alpha$. Da diese Argumentation für beliebige D_1, D_2 und D' stimmt und die Abbildung von D' zu D'' bijektiv bzgl. der Menge aller aus der Mutation möglicherweise resultierenden OBDDs ist, ist Richtline R 2 erfüllt. $\square$

Die nicht vollkommen unabhängige Auswahl hat den Vorteil, Rechenzeit zu sparen, erhält aber trotzdem die MBEA-Richtlinien R 1 und R 2. Man kann jedoch leicht Beispiele finden, bei denen die Rechenzeit der MBGP-Rekombination gleich $\Omega(2^n)$ ist, obwohl die beteiligten OBDDs lineare Größe haben (wenn z. B. D_1 die Parity-Funktion darstellt und D_2 ihr Komplement). Haben die Funktionen aber auf vielen Eingaben dieselbe Ausgabe, wie es typischerweise nach den ersten Generationen in einem GP-System der Fall ist, so wird die Rekombination wesentlich schneller durchgeführt. Denn wird `Synthese_rekursiv`$(\mathtt{v}_1, \mathtt{v}_2, \odot)$ mit $v_1 = v_2$ aufgerufen, so kann v_1 zurückgegeben werden, da stets $f \odot f = f$ ist. Deshalb wird die MBGP-Rekombination in der Regel mit steigender Generationenzahl schneller ablaufen. Aus diesem Grunde wird sie hier trotz des exponentiell hohen Worst-Case-Ressourcenverbrauchs benutzt.

Allgemeiner lässt sich eine beliebige Rekombination, die von den beiden Eltern D_1 und D_2 gleich abgebildete Eingaben im erzeugten OBDD D' auch auf diese Ausgabe abbildet, folgendermaßen beschreiben:

$$f_{D'} = f_{D_Z} \wedge (f_{D_1} \oplus f_{D_2}) \vee (f_{D_1} \wedge f_{D_2}) \,.$$

Dabei ist D_Z ein zufällig erzeugtes OBDD. Wenn dieses so generiert wird, dass es erwartete polynomielle Größe hat, so wird eine Erzeugung von D' durch eine der obigen Formalisierung entsprechenden Folge von Synthese-Schritten erwartete polynomielle Laufzeit haben, wenn dies für die beteiligten OBDDs D_1 und D_2 gilt. Deshalb wird auch D' erwartete polynomielle Größe haben.

Da das OBDD D_Z jedoch stets aufgebaut werden muss, kann die MBGP-Rekombination schneller sein, wenn f_{D_1} und f_{D_2} nur für sehr wenige Eingaben verschieden sind. Dies wird gerade in der Endphase der evolutionären Suche, wenn alle Individuen eine sehr hohe Fitness haben, der Fall sein. Trotz der Schwächen der MBGP-Rekombination bzgl. Beschreibbarkeit und Effizienz ermöglicht sie doch im Vergleich zu der Rekombination von S-Expressions die Erzeugung von Funktionen, die von ihren Vorgängerfunktionen wichtige Eigenschaften übernehmen.

4.4.6 Überblick über die GP-Systeme

Nachdem die einzelnen Module Initialisierung, Mutation und Rekombination vorgestellt sind, soll jetzt ein Überblick gegeben werden, wie diese in den kompletten GP-Systemen genutzt werden. Da dieses Grundgerüst für das OBDD-GP- wie für das MBGP-System gleich aussieht, wird es in einer für beide Varianten passenden Form beschrieben.

Der Selektionsmechanismus wird sich an dem bei Evolutionsstrategien geläufigen (μ, λ)-Schema orientieren. Die Elternpopulation besteht aus μ Individuen, aus denen λ Kinder durch Rekombination und anschließende Mutation generiert werden. Die μ Kinder mit dem höchsten Zielfunktionswert bilden dann die nächste Elterngeneration. Dieser Ablauf wird beendet, wenn eine vorgegebene Zahl G von Generationen erreicht wird. Das Individuum mit dem höchsten Fitnesswert, das während des gesamten Laufs gefunden wurde, wird ausgegeben.

Zusammengefasst ergibt sich also:

Algorithmus 4.4.11
Eingabe: *eine Zielfunktion* $F : \{f : \{0,1\}^n \mapsto \{0,1\}\} \mapsto \{0, \ldots, 2^n\}$ *mit* $F(f) = |\{x \in \{0,1\}^n \,|\, f(x) = f^*(x)\}|$ *für eine unbekannte Funktion* $f^* : \{0,1\}^n \to \{0,1\}$.
Ausgabe: *ein* π*-OBDD* D*, das eine Funktion* $f_D : \{0,1\}^n \mapsto \{0,1\}$ *repräsentiert.*

1. *Setze* $g := 0$. *Initialisiere die Population* P_0*, indem* μ *durch* `Initialisierung` *erzeugte* π*-OBDDs aufgenommen werden.*
2. *Setze* $i := 1$.
3. *Wähle zwei* π*-OBDDs* D_1 *und* D_2 *aus* P_g *gleichverteilt aus.*
4. *Rekombiniere* D_1 *und* D_2 *durch* `Rekombination` *zu* D'.
5. *Mutiere* D' *durch* `Mutation` *und nehme das Resultat in* P_{g+1} *auf.*
6. *Falls* $i < \lambda$*, setze* $i := i + 1$ *und gehe zu Schritt 3.*
7. *Schränke* P_{g+1} *auf die* μ π*-OBDDs mit den höchsten Zielfunktionswerten ein, wobei bei Gleichheit zufällig gewählt wird.*
8. *Falls* $g < G$*, setze* $g := g + 1$ *und gehe zu Schritt 2.*
9. *Gib eines der* π*-OBDDs mit dem höchsten Zielfunktionswert aus, das gefunden wurde.*

Um den Algorithmus 4.4.11 zu vervollständigen, müssen einerseits die Module `Initialisierung`, `Rekombination` und `Mutation` spezifiziert und andererseits die Werte der Parameter μ, λ und G festgelegt werden. Die GP-Systeme OBDD-GP und MBGP ergeben sich durch Verwendung der entsprechenden Module an diesen

	OBDD-GP	MBGP
`Initialisierung`	Algorithmus 4.4.3	Algorithmus 4.4.3
`Mutation`	Algorithmus 4.4.5	Algorithmus 4.4.6
`Rekombination`	Algorithmus 4.4.7	Algorithmus 4.4.9
Anzahl Eltern	$\mu = 15$	$\mu = 15$
Anzahl Kinder	$\lambda = 100$	$\lambda = 100$
Anzahl Generationen	$G = 2000$	$G = 2000$

Tabelle 4.1: Die betrachteten GP-Systeme mit OBDDs.

Stellen: das OBDD-GP-System benutzt die OBDD-GP-Mutation 4.4.5 und die OBDD-GP-Rekombination 4.4.7, das MBGP-System die MBGP-Mutation 4.4.6 und die MBGP-Rekombination 4.4.9. Beide Varianten verwenden jeweils die pseudozufällige Initialisierung 4.4.3. Für das MBGP- und das OBDD-GP-System werden dieselben Parameterwerte $\mu = 15$, $\lambda = 100$ und $G = 2000$ gewählt. Die sich so ergebenden Varianten OBDD-GP und MBGP definieren sich also durch die in Tabelle 4.1 gezeigten Festlegungen:

4.4.7 Empirische Resultate

Die vorgestellten GP-Systeme mit OBDDs werden nun auf den Testfunktionen $\textsc{Par}_n$ für $n \in \{3, \ldots, 12\}$, $\textsc{Mux}_6$ und $\textsc{Mux}_{11}$ untereinander und mit den folgenden GP-Systemen verglichen:

- Ein in Koza (1992) und Koza (1994) untersuchtes GP-System, das S-Expressions über der Funktionsmenge $F = \{AND, OR, NAND, NOR\}$ und der Terminalmenge $T = \{x_1, \ldots, x_n\}$ mit der in Abschnitt 4.2 besprochenen Rekombination verwendet. Dieses dient als ein Beispiel für ein typisches GP-System mit S-Expressions ohne Mutation, weshalb es als `Rek-GP` bezeichnet wird.

- Die Erweiterung `Rek-GP-ADF` des letztgenannten GP-Systems mit S-Expressions um die Verwendung von ADFs (siehe Koza (1992) und Koza (1994)) wie in Abschnitt 4.2 besprochen. Da die OBDD-GP-Operatoren die implizit gegebenen Möglichkeiten von OBDDs, ADF-ähnliche Strukturen zu benutzen, unterstützen, sollte ein Vergleich auch auf S-Expressions basierende GP-Systeme mit ADFs berücksichtigen.

- Weiterhin ziehen wir ein in Chellapilla (1997) vorgestelltes GP-System mit S-Expressions, das keine Rekombination, sondern ausschließlich Mutationsoperatoren verwendet, zum Vergleich mit heran. Bei der Erzeugung neuer Individuen wird zufällig eine von sechs Mutationsvarianten benutzt, die alle syntaktischer Natur sind und schlecht nachvollziehbare Auswirkungen auf die dargestellten Funktionen haben. Dieses System wird als `Mut-GP` bezeichnet.

- Eine Erweiterung dieses Systems (Chellapilla (1998)) um die Verwendung von ADFs wird ebenfalls zum Vergleich verwendet. Dieses wird als `Mut-GP-ADF` bezeichnet.

Diese Systeme stellen die besten bekannten GP-Systeme zum Lernen vollständig definierter boolescher Funktionen dar, die empirisch in Hinblick auf den Computational Effort (Definition 4.2.2) getestet wurden. (Das in Poli, Page und Langdon (1999) vorgestellte GP-System lernt zwar erfolgreich Parity-Funktion mit bis zu $n = 22$ Variablen, doch wird es in diesen Vergleich nicht mitaufgenommen, da es nur für die Parity-Funktion getestet wurde und jeweils nur ein Lauf durchgeführt

	Rek-GP	Rek-GP-ADF	Mut-GP	Mut-GP-ADF
PAR_3	96.000	64.000	63.000	15.600
PAR_4	384.000	176.000	181.500	118.500
PAR_5	6.528.000	464.000	2.100.000	126.000
PAR_6	-	1.344.000	-	121.000
PAR_7	-	1.440.000	-	169.000
PAR_8	-	/	-	321.000
PAR_9	-	/	-	586.500
PAR_{10}	-	/	-	-
PAR_{11}	-	/	-	-
PAR_{12}	-	-	-	-
MUX_6	160.000	-	93.000	-
MUX_{11}	-	-	-	-

Tabelle 4.2: Ein Vergleich bestehender GP-Systeme mit S-Expressions im Hinblick auf ihren Computational Effort (mit $z = 0{,}99$) zum Finden von Testfunktionen. Ein '-' bedeutet, dass das System mit dieser Funktion nicht getestet wurde; ein '/' bedeutet, dass nur je 4 Läufe durchgeführt wurden, weshalb kein Computational Effort berechnet wurde.

wurde). Die empirischen Ergebnisse sind aus der Originalliteratur übernommen und es werden keine neuen Experimente mit diesen Systemen durchgeführt. Der Computational Effort dieser Systeme ist in Tabelle 4.2 zusammengefasst.

Neben der Multiplexer- und Parity-Funktion werden die vorgestellten GP-Systeme mit OBDDs auf der Funktion RAND_8 verglichen. Dabei stellt RAND_8 keine Funktion im herkömmlichen Sinne dar, sondern dient als Test, wie sich die GP-Systeme auf rein zufällig gewählten Funktionen verhalten. Denn in jedem der Läufe mit RAND_8 werden die 2^8 Funktionswerte der Trainingsmenge jeweils gleichverteilt zufällig ausgewählt. Insbesondere dient dies als Test, wie und ob sich die dabei mit hoher Wahrscheinlichkeit entstehenden exponentiell großen OBDDs der zu lernenden Funktionen auf den Computational Effort auswirken.

Zur Berechnung des Computational Efforts der neuen GP-Systeme mit OBDDs wurden jeweils $L = 100$ Läufe der vorgestellten GP-Systeme OBDD-GP und MBGP für jede der 13 Testfunktionen jeweils bis zum Erreichen der größtmöglichen Fitness 2^n oder der Maximalzahl G an Generationen durchgeführt. Die sich ergebenden Daten sind in Tabelle 4.3 zusammengefasst.

Daraus wird klar ersichtlich, dass für die betrachteten Testfunktionen PAR und MUX in dem Bereich $n \in \{3, \ldots, 12\}$ beide GP-Systeme mit OBDDs einen deutlich geringeren Computational Effort haben als die bekannten GP-Systeme mit S-Expressions. Während dies für das MBGP-System aufgrund der Berücksichtigung der MBEA-Richtlinien zumindest zu erhoffen war, ist dies für das OBDD-GP-System überraschender. Denn dieses GP-System ergibt sich ja allein durch Übertragung der auf S-Expressions üblichen genetischen Operatoren auf OBDDs und es scheint keinen offensichtlichen Grund zu geben, warum der dadurch definierte Suchprozess günstiger zum Auffinden von Funktionen sein sollte als der für S-Expressions.

Das MBGP-System schneidet im direkten Vergleich der neuen Systeme besser ab. Dabei muss jedoch berücksichtigt werden, dass der starke Anstieg des Computational Efforts des OBDD-GP-Systems bei den Funktionen PAR_{11} und PAR_{12} darauf zurückzuführen ist, dass der Prozentsatz der Läufe, die nach $G = 2000$ Generationen das Optimum gefunden haben, stark gesunken ist. Hätten wir G hier erhöht, wäre der Computational Effort sicher weniger stark gestiegen. Aufgrund der bes-

	OBDD-GP	MBGP
$\textsc{Par}_3$	4.915	415
$\textsc{Par}_4$	17.815	715
$\textsc{Par}_5$	41.015	1.215
$\textsc{Par}_6$	46.515	2.115
$\textsc{Par}_7$	42.315	4.615
$\textsc{Par}_8$	81.515	8.615
$\textsc{Par}_9$	127.715	17.215
$\textsc{Par}_{10}$	758.415	33.415
$\textsc{Par}_{11}$	17.352.015	77.415
$\textsc{Par}_{12}$	43.958.415	147.015
$\textsc{Mux}_6$	45.915	1.615
$\textsc{Mux}_{11}$	1.584.015	60.515
$\textsc{Rand}_8$	93.915	8.415

Tabelle 4.3: Ein Vergleich der GP-Systeme mit OBDDs im Hinblick auf ihren Computational Effort (mit $z = 0,99$) zum Finden von Testfunktionen.

seren Vergleichbarkeit haben wir diesen besprochenen Nachteil des Computational Efforts in Kauf genommen.

Nichtsdestotrotz ist der Computational Effort des MBGP-System stets um mindestens einen Faktor 10 kleiner als der des OBDD-GP-Systems. Eine Berücksichtigung der MBEA-Richtlinien scheint also bei der Aufgabe des Lernens von Funktionen mit den hier betrachteten Testfunktionen eine Verbesserung des Suchprozesses zu bewirken. Eine darüber hinaus gehende Bestätigung in Form weiterer experimenteller Ergebnisse oder theoretischer Analysen muss aber noch erbracht werden. Hier angemerkt werden soll nur, dass die, wenn auch nicht in allen formalen Aspekten erfolgreiche, Orientierung eines praktisch eingesetzten GP-Systems zur Generierung von Fuzzy-Regeln an den MBEA-Richtlinien zu einer deutlichen Leistungssteigerung gegenüber einem zuvor benutzten GP-System geführt hat (Slawinski, Krone, Hammel, Wiesmann und Krause (1999)).

Anhand der Ergebnisse in Tabelle 4.3 zeigt sich auch, dass der Einsatz von OBDDs in GP-Systemen eine erhebliche Leistungssteigerung bewirken kann. Dies gilt sogar, wenn Funktionen gelernt werden sollen, deren OBDDs sehr groß werden können, wie die Testläufe mit der „Funktion“ $\textsc{Rand}_8$ zeigen, deren Computational Effort nicht wesentlich höher als für die Funktion $\textsc{Par}_8$ ist. (Natürlich wird bei $\textsc{Rand}_8$ der Speicher- und Zeitaufwand des MBGP-Systems exponentiell steigen, was bei den zur Zeit verwendeten Variablenzahlen noch keine Probleme macht.)

Dies ist ein starker Hinweis dafür, dass der Computational Effort des MBGP-Systems relativ unabhängig von der zu findenden Funktion ist. Dies wird natürlich von der Tatsache gestützt, dass dieses, wenn wir keine Rekombination zulassen, wie eine Variante des (1+1) EA mit größerer Population auf OneMax arbeitet: es soll eine Funktion, d. h. ein String der Ausgabebits gefunden werden, der mit der unbekannten Funktion möglichst viele Stellen gemeinsam hat. Da die MBGP-Mutation 4.4.6 wie die bit-weise Mutation des (1+1) EA arbeitet, entspricht das Auffinden boolescher Funktionen in einem Black-Box Szenario dem Maximieren von OneMax in der Dimension 2^n.

Weshalb dann die durchgeführten Experimente, wo doch die Analyse in Kapitel 3 zeigt, dass die erwartete Anzahl der Generationen gleich $\Theta(n \cdot 2^n)$ ist? Zum einen, um anhand empirischer Daten einen Vergleich zu bestehenden GP-Systemen für die gebräuchlichen Werte von n zu haben, zum anderen, da das MBGP-System Rekombination benutzt. Es gibt keine Ergebnisse theoretischer Analyse, ob etwa

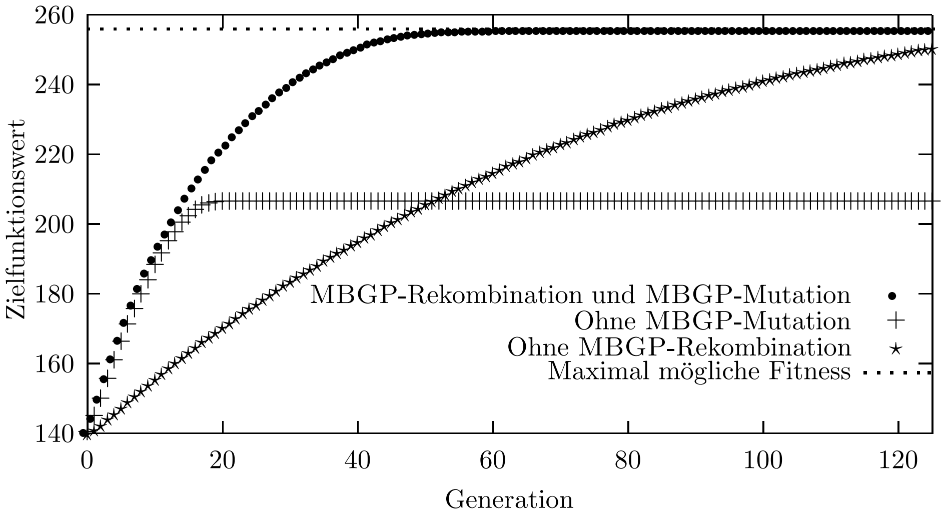

Abbildung 4.2: Das MBGP-System auf $\textsc{Par}_8$ mit Mutation und Rekombination, nur mit Rekombination bzw. nur mit Mutation.

das uniforme Crossover den Suchprozess auf OneMax, wenn nicht um einen in n wachsenden, so doch großen konstanten Faktor beschleunigen kann. Zudem sind die Auswirkungen der MBGP-Rekombination (Algorithmus 4.4.9) komplizierter als die des uniformen Crossovers und von der Struktur der beteiligten OBDDs abhängig, weshalb eine theoretische Analyse noch schwieriger ist.

Dass die Rekombination den Suchprozess beschleunigt, können wir durch empirische Untersuchungen gut nachweisen. Dazu testen wir das MBGP-System auf der Funktion $\textsc{Par}_8$ in drei Varianten:

1. Wie in Tabelle 4.1 besprochen mit MBGP-Mutation und MBGP-Rekombination.

2. Wie in Tabelle 4.1 gezeigt, wobei jedoch die MBGP-Rekombination durch die Reproduktion des ersten Eltern-OBDDs ersetzt wurde.

3. Wie in Tabelle 4.1 gezeigt, wobei jedoch die MBGP-Mutation durch die Reproduktion des Elter-OBDDs ersetzt wurde.

Der durchschnittliche Verlauf über 50 Läufe des Zielfunktionswerts des bis zur Generation g besten gefundenen Individuums ist in Abbildung 4.2 dargestellt. Während die Anwendung der MBGP-Rekombination allein in keinem der 50 Läufe zum Erreichen des Optimums reicht, wird bei alleiniger Anwendung der MBGP-Mutation nach durchschnittlich 154,5 und höchstens 186 Generationen das Optimum gefunden. Werden jedoch beide Operatoren in Verbindung eingesetzt, ergibt sich eine wesentlich bessere Leistung auf $\textsc{Par}_8$: nach durchschnittlich 53,3 und höchstens 73 Generationen wird das Optimum gefunden. Die MBGP-Rekombination senkt den Computational Effort für $\textsc{Par}_8$ in den durchgeführten Experimenten also erheblich, weshalb die Durchführung der Experimente durchaus sinnvoll ist.

Zusammengefasst lässt sich sagen, dass die MBEA-Richtlinien genetische Operatoren und damit einen evolutionären Algorithmus erzwingen, dessen Auswirkungen auf dem Genotyp-Raum klar nachvollziehbar und intuitiv dem evolutionären Konzept entsprechend sind. Dabei ist entscheidend, dass diese Auswirkungen sich durch die Metrik auf die Zielfunktionswerte der beteiligten Individuen übertragen: die Mutation wird bevorzugt kleine Änderungen der Zielfunktion bewirken, während die Rekombination Kinder erzeugen sollte, deren Zielfunktionswerte in der Nähe derer der Eltern liegen. Im Fall der genetischen Programmierung zum Lernen boolescher Funktionen fällt eine Umsetzung von MBEA-konformen Operatoren wesentlich leichter, wenn die gängige Darstellungsform der S-Expressions fallen gelassen und durch eine effizientere Datenstruktur, die OBDDs, ersetzt wird. Die experimentellen Resultate des so entstandenen MBGP-Systems zeigen für die gewählten Testfunktionen eine deutliche Effizienzsteigerung. Durch das hier präsentierte MBEA-Konzept und die demgemäße Umsetzung in das MBGP-System bestätigt sich die anschauliche Vorstellung, dass sowohl die Operatoren eines evolutionären Algorithmus als auch seine Datenstruktur auf das zu lösende Problem abgestimmt sein sollten.

Kapitel 5

Theoretische Qualitätsgarantien für die genetische Programmierung

Nun wenden wir uns der Variante der in Abschnitt 4.2 besprochenen Aufgabe des Lernens von booleschen Funktionen anhand von unvollständigen Trainingsmengen in der genetischen Programmierung zu. Dabei steht weniger die genaue Wiedergabe der Trainingsbeispiele im Vordergrund, sondern vielmehr die Vorhersage der den Trainingsdaten zugrundeliegenden Funktion auf den nicht in der Trainingsmenge enthaltenen Eingaben, kurz das Finden einer *gut generalisierenden* Funktion. Da aber über das Verhalten der Funktion auf diesen Eingaben keinerlei Informationen vorliegen, scheint hier jedes Verfahren keinerlei Verbesserung gegenüber purem Raten der entsprechenden Ausgabewerte erzielen zu können.

In diesem Kapitel zeigen wir, wie unter bestimmten Voraussetzungen an die Auswahl der Trainingsdaten probabilistische Garantien über die Generalisierungsgüten der gefundenen Funktionen gegeben werden können. Es werden also keine Aussagen über die erwartete Laufzeit oder andere Merkmale eines GP-Systems getroffen, sondern es wird eine Methode vorgestellt, mit der Aussagen über die wahrscheinliche Güte der mit den Trainingsdaten übereinstimmenden Lösungen eines GP-Systems getroffen werden können. Diese Methode ergibt sich durch Übertragung des *Occam's Razor* Theorem (Blumer, Ehrenfeucht, Haussler und Warmuth (1990)) und liefert eine Begründung, warum das weit verbreitete *Occam's Razor Prinzip*, eine möglichst „einfache“ Lösung zu obigem Problem zu suchen, eine erfolgsversprechende Möglichkeit darstellt.

Im nächsten Abschnitt wird das Problem des Lernens unvollständig spezifizierter Funktionen formal definiert. Dann wird das Occam's Razor Theorem vorgestellt und gezeigt, wie und in welchem Szenario es Generalisierungsgüten garantieren kann. Um zu zeigen, dass diese Schranken nicht nur zu trivialen Werten führen, wird in Abschnitt 5.3 in einer beispielhaften Anwendung gezeigt, wie für die Lösungen eines GP-Systems nicht-triviale Generalisierungsgüten garantiert werden können.

5.1 Lernen unvollständig definierter Funktionen

Seien hier die Voraussetzungen und notwendigen Definitionen für das Problem des Lernens einer Funktion anhand einer unvollständigen Trainingsmenge in allgemeiner Form angestellt, obwohl wir uns später wiederum auf boolesche Funktionen beschränken (siehe auch Abschnitt 4.2).

Es sei eine Menge $T = \{(a,b) \mid a \in T_A, b \in B\}$ von Trainingsdaten gegeben, wobei $T_A \subset A$ die Menge der Trainingseingaben bezeichnet und A und B zwei beliebige endliche Mengen sind. Dabei nehmen wir an, dass T *konsistent* ist, also für kein $a \in A$ sowohl $(a,b_1) \in T$ als auch $(a,b_2) \in T$ mit $b_1 \neq b_2$ ist. Weiterhin nehmen wir an, dass den Trainingsdaten ein funktionaler Zusammenhang $f^* : A \to B$ zugrunde liegt, also $f^*(a) = b$ für alle $(a,b) \in T$ gilt. Natürlich gibt es sehr viele, genauer $|B|^{|A|-|T|}$, Funktionen $f : A \to B$, die mit der Trainingsmenge T *konsistent* sind, d. h. für die für alle $(a,b) \in T$ gilt $f(a) = b$. Doch allein f^* wird für alle weiteren Eingaben $a \in A \setminus T_A$ mit ggf. weiteren Trainingsbeispielen konsistent sein.

Unser Ziel ist es, diese Funktion f^* möglichst genau zu approximieren, d. h. eine Funktion $f : A \mapsto B$ zu finden, die mit f^* auf möglichst vielen Eingaben übereinstimmt. Jedoch bildet die Trainingsmenge T die einzig verfügbaren Daten, die über f^* vorliegen. Der erste Ansatz, eine gute Approximation von f^* zu finden, besteht darin, eine zu T konsistente Funktion zu finden. Denn die Beschränkung auf zu T konsistente Funktionen schränkt die Möglichkeiten, eine gute Approximation zu finden, nicht ein: jede Funktion $f : A \mapsto B$ wird durch eine Anpassung ihres Verhaltens auf den Eingaben aus T_A an die Trainingsmenge T nur eine bessere Übereinstimmung mit f^* erreichen. Somit können wir o. B. d. A. den Suchprozess auf zu T konsistente Funktionen einschränken.

Die nun eingeführten Konsistenzbegriffe seien zur Klarheit formal definiert:

Definition 5.1.1 *Eine Menge $T \subset \{(a,b) \mid a \in A, b \in B\}$ heißt* konsistent*, wenn es kein $a \in A$ gibt, so dass sowohl $(a,b_1) \in T$ als auch $(a,b_2) \in T$ mit $b_1 \neq b_2$ ist. Eine Funktion $f : A \to B$ heißt zu einer Menge $T \subset \{(a,b) \mid a \in A, b \in B\}$* konsistent*, falls für jedes Element $(a,b) \in T$ gilt $f(a) = b$.*

Natürlich kann eine Funktion nur zu einer konsistenten Menge konsistent sein und eine Trainingsmenge T ist konsistent, wenn eine Funktion zu ihr konsistent ist.

Schränkt man die Menge von Funktionen, aus der f^* stammt, nicht ein, so wird jedes Verfahren zum Auffinden einer guten Approximation von f^* im Durchschnitt über alle f^* gleich gut sein, da:

Lemma 5.1.2 *Sei $T \subset \{(a,b) \mid a \in A, b \in B\}$ eine konsistente Trainingsmenge. Dann gilt für jede zu T konsistente Funktion f, dass sie im Durchschnitt über alle zu T konsistenten Funktionen f^* auf*

$$|B|^{-|A|+|T|} \cdot \sum_{i=0}^{|A|-|T|} \binom{|A|-|T|}{i} \cdot (|B|-1)^{|A|-|T|-i} \cdot i$$

Eingaben aus $A \setminus T$ mit f^* übereinstimmt.

Beweis: Hält man die Funktion f fest und betrachtet alle möglichen Funktionen f^*, die der Trainingsmenge T zugrundeliegen könnten, so ergibt sich die obige Gleichung. Denn genau $\binom{|A|-|T|}{i} \cdot (|B|-1)^{|A|-|T|-i}$ der $|B|^{|A|-|T|}$ zu T konsistenten Funktionen f^* stimmen mit f an genau i weiteren Stellen überein. □

Dieses Resultat kann analog zum NFL-Theorem (Korollar 2.2.3) gesehen werden: alle Systeme, die zu einer gegebenen Trainingsmenge T eine konsistente Funktion ausgeben, werden im Durchschnitt über alle der Trainingsmenge T möglicherweise zugrundeliegenden Funktionen f^* Funktionen der gleichen durchschnittlichen Generalisierungsgüte liefern. Der Versuch, ein Verfahren zu finden, das zu einer Trainingsmenge T eine Funktion ausgibt, die über alle T möglicherweise zugrundeliegenden Funktionen überdurchschnittlich gut generalisiert, ist also zum Scheitern verurteilt.

Implizit nimmt man jedoch an, dass die Trainingsmenge T eine „typische“ Auswahl des Ein-Ausgabeverhaltens von f^* ist. Dieser Begriff scheint sich einer Formalisierung ähnlich wie der einer „natürlichen“ Funktion zu entziehen. Nimmt man jedoch an, dass die Elemente der Trainingseingaben T_A zufällig gemäß der Gleichverteilung auf A gezogen sind, so werden diese das „typische" Verhalten von f^* mit hoher Wahrscheinlichkeit widerspiegeln, wenn wir „typisches" Verhalten als ein solches charakterisieren, das für die meisten der Eingaben gilt.

Dazu ein einfaches Beispiel: wenn f^* bis auf einen kleinen Anteil $\varepsilon > 0$ der Eingaben konstant gleich $b^* \in B$ wäre, so könnte eine feste Trainingsmenge T nur solche Eingaben enthalten, die von f^* auf ein anderes Element als b^* abgebildet werden, was sehr „untypisch" wäre. Werden die Elemente der Trainingsmenge jedoch gleichverteilt unabhängig gewählt, so ist dieser Fall mit einer Wahrscheinlichkeit von $(1-\varepsilon)^{|T|}$ sehr unwahrscheinlich, wenn auch nicht unmöglich.

Diese Argumentation kann auf die gleiche Weise für eine beliebige Wahrscheinlichkeitsverteilung P_A anstelle der Gleichverteilung durchgeführt werden, wenn man annimmt, dass diese auch als Maß für die Generalisierungsgüte einer Funktion benutzt wird. Dieses kann in einem allgemeineren Kontext folgendermaßen definiert werden:

Definition 5.1.3 *Seien A und B endliche Mengen, $F := \{f : A \to B\}$, $\mathrm{P}_A : \mathcal{P}(A) \mapsto [0,1]$ eine Wahrscheinlichkeitsverteilung auf A und $f_1, f_2 \in F$. Der* Fehler $Err_{\mathrm{P}_A}(f_1, f_2)$ von f_1 bzgl. f_2 *ist definiert als*

$$Err_{\mathrm{P}_A}(f_1, f_2) := \mathrm{P}_A(\{a \in A \mid f_1(a) \neq f_2(a)\}).$$

Wenn $f^* \in F$ die zu lernende Funktion ist, so besteht das betrachtete Optimierungsproblem darin, die Funktion $Err_{\mathrm{P}_A}(f, f^*)$ zu minimieren. Da bei einer festen Wahl der Trainingsmenge eine „typische" nicht von einer „untypischen" Trainingsmenge unterschieden werden kann, nehmen wir an, dass die Trainingseingaben für jeden Lauf jeweils unabhängig gemäß P_A gezogen werden. Diese Annahmen bilden das Szenario, das wir im Folgenden betrachten:

Szenario 5.1.4 (Occam's Razor Szenario) *Für jeden Lauf des Systems zur Approximation von f^* werden die Eingaben der Trainingsbeispiele von f^* unabhängig gemäß der Wahrscheinlichkeitsverteilung P_A auf A gewählt.*

Eine oftmals auch im Bereich der genetischen Programmierung (siehe z. B. Zhang und Mühlenbein (1995) oder Zhang und Joung (1999)) angewandte Strategie, um möglichst gut generalisierende Programme zu finden, ist das Occam's Razor Prinzip. Dabei wird versucht, ein Programm mit möglichst kleiner Darstellung zu finden, das die Trainingsbeispiele korrekt wiedergibt. Wenn die zugrundeliegende unbekannte Funktion f^* eine kleine Darstellung hat, so sollte diese Strategie anschaulich bessere Erfolgsaussichten haben als eine rein zufällige Wahl einer konsistenten Funktion. Doch wird dieses Prinzip auch oftmals angewandt, ohne Voraussetzungen über f^* zu machen. Eine Methode, die Nützlichkeit dieses Prinzips im Occam's Razor Szenario nachzuweisen, wird im nächsten Abschnitt vorgestellt.

5.2 Das Occam's Razor Theorem

In diesem Abschnitt zeigen wir, wie sich im Szenario 5.1.4 ein Resultat von Blumer, Ehrenfeucht, Haussler und Warmuth (1990) aus der Lerntheorie, das so genannte *Occam's Razor Theorem*, zur Vorhersage der Generalisierungsgüte von Funktionen anwenden lässt. Bevor dieses formal definiert wird, sei die zugrundeliegende sehr anschauliche Idee vorgestellt. Angenommen, wir kennen einen (randomisierten) Algorithmus, der zu einer beliebigen Trainingsmenge T eine zu T konsistente Funktion f ausgibt, Wir wiederholen diesen Algorithmus L-mal mit jeweils unabhängig gemäß P_A gezogenen Traininingseingaben, die mit den entsprechenden Ausgaben der unbekannten Funktion f^* die Trainingsdaten bilden. Gäbe der Algorithmus stets dieselbe Funktion f aus, so wäre die Wahrscheinlichkeit der Trainingsbeispiele, wenn der Fehler $Err_A(f, f^*)$ von f bzgl. f^* groß wäre, sehr gering, da f zu allen in den L Läufen gezogenen Trainingsbeispielen konsistent ist.

Diese einfache Beobachtung scheint nicht von großem Nutzen zu sein, da der besprochene Fall, dass alle ausgegebenen Funktionen gleich sind, nur sehr selten auftreten mag. Jedoch greift diese Überlegung auch, wenn nicht stets dieselbe Funktion f von dem Algorithmus ausgegeben wird, jedoch stets eine Funktion aus einer recht kleinen Menge $H \subset F$. Auch in diesem Fall wird die Wahrscheinlichkeit der Trainingseingaben, wenn in H auch nur eine konsistente Funktion mit einem großen Fehler vorkommt, sehr gering sein. Wie groß diese in Abhängigkeit von der Fehlerwahrscheinlichkeit $\varepsilon > 0$ und $|H|$ ist, besagt das Occam's Razor Theorem von Blumer, Ehrenfeucht, Haussler und Warmuth (1990), das hier der Vollständigkeit halber mit einem kurzen Beweis angegeben wird:

Theorem 5.2.1 (Occam's Razor Theorem) *Seien A und B beliebige endliche Mengen, $F := \{f : A \mapsto B\}$, $H \subset F$, $\mathrm{P}_A : \mathcal{P}(A) \mapsto [0, 1]$ eine Wahrscheinlichkeitsverteilung auf A, $f^* \in F$ und $\varepsilon \in [0, 1]$. Seien m Trainingseingaben $a_1, \ldots, a_m \in A$ unabhängig voneinander gemäß P_A gewählt. Die Wahrscheinlichkeit, dass es eine Funktion $f \in H$ gibt, die zu allen Trainingsbeispielen $(a_1, f^*(a_1)), \ldots, (a_m, f^*(a_m))$ konsistent ist und einen Fehler $Err_{\mathrm{P}_A}(f, f^*) > \varepsilon$ hat, ist kleiner als $|H| \cdot (1 - \varepsilon)^m$.*

Beweis: Die Wahrscheinlichkeit, dass eine Funktion $f \in H$ mit einem Fehler größer als ε zu allen m Trainingsbeispielen konsistent ist, ist kleiner als $(1 - \varepsilon)^m$. Somit lässt sich die Wahrscheinlichkeit, dass dies für eine der Funktionen aus H gilt, nach oben durch $|H| \cdot (1 - \varepsilon)^m$ abschätzen. $\square$

Es werden also mit hoher Wahrscheinlichkeit nur solche Trainingsbeispiele gewählt, dass alle zu diesen konsistenten Funktionen aus H einen sehr kleinen Fehler haben. Dies ist vollkommen von dem betrachteten Algorithmus zur Generierung konsistenter Funktionen unabhängig. Um aber Occam's Razor Theorem ausnutzen zu können, müsssen wir die Menge H so wählen, dass sie einerseits sehr klein ist, wir aber andererseits einen Weg kennen, konsistente Funktionen aus ihr zu bestimmen. Erst hier spielt der Algorithmus eine Rolle. Denn findet er eine Funktion aus einer vorbestimmten Menge $H \subset F$, so wissen wir, dass es eine konsistente Funktion aus H gibt. Der Algorithmus spielt also eine Indikator-Funktion: gibt er eine Funktion aus H aus, wissen wir, dass H eine konsistente Funktion enthält; anderenfalls ist es unbekannt, da wir keine Leistungsgarantie des Algorithmus voraussetzen. Natürlich ist es für uns umso günstiger, je öfter der Algorithmus konsistente Funktionen aus einer möglichst kleinen Menge ausgibt.

Denn mit Occam's Razor Theorem können wir die Wahrscheinlichkeit einer Wahl von m Trainingsbeispielen, so dass alle konsistenten Funktionen aus H einen Fehler von höchstens $\varepsilon > 0$ haben, durch

$$\max\left(0, 1 - |H| \cdot (1 - \varepsilon)^m\right)$$

nach unten abschätzen. Sei f eine zu den Trainingsbeispielen konsistente Funktion. Natürlich kann f einen größeren Fehler als ε haben, doch sinkt dann die Wahrscheinlichkeit, dass die Trainingsbeispiele diesen Fehler nicht „aufgedeckt“ haben. Wird das System L-mal mit jeweils unabhängig gezogenen Trainingsbeispielen wiederholt, so ist die Wahrscheinlichkeit, dass in allen L Läufen nur solche Trainingsbeispiele gezogen wurden, für die jeweils alle konsistenten Funktionen in H einen Fehler von höchstens $\varepsilon > 0$ haben, mindestens gleich

$$\left(\max\left(0, 1 - |H| \cdot (1 - \varepsilon)^m\right)\right)^L.$$

Damit haben natürlich auch alle konsistenten Funktionen aus H, die unser Algorithmus in L Läufen findet, einen Fehler von höchstens ε. Zusammengefasst ergibt dies:

Korollar 5.2.2 *Sei $H \subset F$ gegeben. Sei ein Algorithmus betrachtet, der zu einer gemäß P_A gezogenen Menge T von m Trainingsdaten von f^* eine zu T konsistente Funktion $f \in F$ ausgibt. Wird dieser Algorithmus L-mal mit unabhängig gezogenen Trainingsbeispielen wiederholt, so ist die Wahrscheinlichkeit, die Trainingsbeispiele so gezogen zu haben, dass alle in den L Läufen gefundenen Funktionen aus H einen Fehler von höchstens von $\varepsilon > 0$ haben, mindestens gleich*

$$\left(\max(0, 1 - |H| \cdot (1-\varepsilon)^m)\right)^L .$$

Wenn also H hinreichend klein und m hinreichend groß ist, so ist es sehr unwahrscheinlich, dass auch nur eine der gefundenen Funktionen aus H einen Fehler größer als $\varepsilon > 0$ hat. Letzteres lässt sich zwar nicht ausschließen, doch ist dies unter den betrachteten Voraussetzungen auch gar nicht möglich, da keinerlei Anforderungen an den Algorithmus oder die zu lernende Funktion f^* gestellt werden.

Direkt aus der Formulierung dieses Korollars bzw. des Occam's Razor Theorems ist klar, dass die Menge H möglichst klein sein sollte, um gute untere bzw. obere Schranken zu bekommen. Somit sollte ein Algorithmus versuchen, konsistente Funktionen aus einem möglichst kleinen Teil der Menge aller Funktionen zu finden. Dies muss nicht zwangsläufig die Qualität des Algorithmus verbessern, sollte aber unsere Möglichkeiten, nach obigem Muster Aussagen über seine Güte zu machen, erweitern. Dem Occam's Razor-Prinzip zu folgen und möglichst Funktionen mit kleiner Darstellung zu suchen, ist dabei ein gangbarer Weg, aber nicht der einzig denkbare. Auch die gegenteilige Vorgehensweise, konsistente Funktionen mit möglichst großer Darstellung zu suchen, kann zu einer Einschränkung des Menge der ausgegebenen Funktionen führen. Somit stützt das Occam's Razor-Theorem nicht nur die Anwendung des Occam's Razor-Prinzips, sondern generell jede Methode, den Suchprozess auf eine möglichst kleine Menge zu beschränken (deren Elemente nicht zwangsläufig eine kleine Darstellung besitzen müssen).

Der Versuch, Funktionen mit möglichst kleiner Darstellung zu finden, hat drei entscheidende Vorteile: Der Erste ist, dass sich eine Funktion umso effizienter behandeln lässt, je kleiner ihre Darstellung ist. Würde man Funktionen mit großer Darstellung suchen, so würde sowohl Speicher- als auch Zeitverbrauch anwachsen. Zweitens gibt es in der Regel weniger Funktionen mit einer kleinen Darstellung als solche mit einer großen Darstellung. Drittens ermöglicht eine Orientierung an der Größe eine bequeme und oftmals ausreichend gute Festlegung der Funktionsmenge H, indem in einem ersten Satz von Testläufen eine Schätzung der Verteilung der Größen der gefundenen Funktionen ermittelt wird. Damit können wir die Menge H als Menge aller Funktionen festlegen, die z. B. höchstens so groß wie der Durchschnitt der in den Testläufen gefundenen Funktionen sind. Ist die Verteilung der Größen der vom Algorithmus ausgegebenen Funktionen relativ stabil, so wird auch in weiteren Läufen häufig eine Funktion aus H gefunden werden.

Da Occam's Razor Theorem und damit auch das daraus folgende Korollar 5.2.2 nur grobe Abschätzungen der Wahrscheinlichkeiten liefern, ist nicht klar, ob für einen konkreten Algorithmus die resultierenden Schranken nicht trivial sind. Im Folgenden werden wir ein GP-System vorstellen, das zu gegebenen Trainingsmengen konsistente Funktionen mit möglichst kleinen OBDDs zu finden versucht. Dieses Beispiel wird zum ersten Mal zeigen, wie wir für ein System der genetischen Programmierung auch für unbekannte Testfunktionen nicht-triviale Generalisierungsgüten garantieren können.

5.3 Ein GP-System mit OBDDs für unvollständige Trainingsmengen

Das in diesem Abschnitt besprochene GP-System Occam-GP wird zur Repräsentation wie die Systeme OBDD-GP und MBGP aus Kapitel 4 reduzierte OBDDs benutzen. Von den in Abschnitt 4.3 besprochenen Vorteilen von OBDDs wird, neben der speichereffizienten Darstellung, insbesondere die eindeutige Beziehung zwischen der Größe eines reduzierten OBDDs und der Struktur der von ihm dargestellten Funktion wichtig sein.

Denn ein jede Anwendung des Occam's Razor Prinzips beeinflussender Faktor ist die gewählte Darstellungsform, wenn die Einfachheit einer Funktion anhand der Größe ihrer Darstellung gemessen wird. Die Verwendung einer nicht eindeutigen Darstellungsform, wie S-Expressions (Definition 4.2.1), würde dabei eine neue Schwierigkeit ins Spiel bringen, da eine Funktion, die eine sehr kleine minimale Darstellung hat, wieder verloren gehen kann, wenn ihre aktuelle Darstellung wesentlich größer ist. In einem GP-System, das nicht mit minimalen Darstellungsformen arbeitet, wird der Suchprozess also zusätzlich davon abhängen, ob eine „kleine" Funktion auch in einer kleinen Darstellung gefunden wurde. Bei der Verwendung von reduzierten OBDDs tritt dieses Problem nicht auf.

Weiterhin wird der Suchraum von Occam-GP explizit zusätzlich eingeschränkt, indem nur OBDDs in die Population aufgenommen werden, die zu den gegebenen Trainingsdaten T konsistent sind. Dies ist ein im Vergleich zu anderen GP-Systemen, die konsistente Lösungen möglichst kleiner Darstellung zu finden versuchen, ungebräuchliches Vorgehen, da diese meistens die Größe eines Individuums und seine Übereinstimmung mit der Trainingsmenge mittels einer gewichteten Summe zu einem Fitnesswert berechnen (siehe z. B. Zhang und Joung (1997)). Das Problem, eine passende Wahl dieser Gewichte zu finden, kann umgangen werden, indem in der Initialisierung explizit konsistente Funktionen konstruiert werden und nach Mutation und Rekombination nicht-konsistente Kinder durch ihre Eltern ersetzt werden. In die Zielfunktion fließt deshalb nur die Größe des Individuums ein, nicht die Anzahl korrekt wiedergegebener Trainingsbeispiele, die ja stets gleich $|T|$ ist.

Das GP-System Occam-GP lehnt sich in seinem Aufbau an das OBDD-GP-System an, wie es in Abschnitt 4.4 vorgestellt wurde. Dies hat den Grund, dass die dort verwendeten genetischen Operatoren (Algorithmen 4.4.5 und 4.4.7) kleinere Änderungen an der Größe der beteiligten OBDDs bevorzugen. Insbesondere die Rekombination des MBGP-Systems hingegen (Algorithmus 4.4.9) kann aus zwei OBDDs ähnlicher Größe ein sehr viel größeres oder kleineres OBDD erzeugen, wenn sich die beiden für relativ viele Eingaben unterscheiden.

Extrembeispiel ist hierfür das schon in Abschnitt 4.4 genannte Paar der Parity-Funktion und ihrem Komplement: beide haben nur je $2 \cdot n - 1$ Knoten, doch wird das resultierende OBDD mit exponentiell großer Wahrscheinlichkeit exponentiell groß werden, da es gleichverteilt aus der Menge aller reduzierten OBDDs ausgewählt wird. Dies ist sogar von der benutzten Variablenordnung unabhängig. Der Austausch eines SubOBDDs, wie von der OBDD-GP-Rekombination (Algorithmus 4.4.7) durchgeführt, wird das OBDD höchstens um die Größe des neu eingefügten SubOBDDs vergrößern. Haben also beide beteiligten OBDDs polynomielle Größe, so gilt dies auch für jedes Resultat der OBDD-GP-Rekombination.

Die OBDD-GP-Mutation (Algorithmus 4.4.5) fügt mit der pseudozufälligen Initialisierung 4.4.3 erzeugte SubOBDDs ein. Sind die OBDDs in der Population relativ klein, so wird dies tendenziell eher zu einer Vergrößerung der OBDDs führen. In der Anfangsphase werden Mutationen aber weder Vergrößerungen noch Verkleinerungen bevorzugen. Die verwendeten genetischen Operatoren Mutation und Rekombination orientieren sich also nur an den MBEA-Richtlinien, erfüllen sie aber wohl nicht. Seien nun die einzelnen Komponenten von OCCAM-GP detaillierter vorgestellt.

Initialisierung

Ein besonderes Augenmerk legen wir auf die Initialisierung. Benutzten wir hier die pseudozufällige Initialisierung (Algorithmus 4.4.3), so wäre die Wahrscheinlichkeit, dass ein erzeugtes OBDD konsistent zu den Trainingsbeispielen ist, sehr klein. Da die Auswahl der Ausgaben nicht für alle Eingaben gleichverteilt zufällig erfolgt, kann ihre Wahrscheinlichkeit nicht als $2^{-|T|}$ nachgewiesen werden, sie wird jedoch vermutlich nicht sehr viel größer sein. Um schon in der Anfangspopulation nur konsistente OBDDs zu haben, wird deshalb die pseudozufällige Initialisierung nicht jeweils solange wiederholt, bis das erzeugte OBDD konsistent ist. Vielmehr werden durch eine Modifikation dieser Initialisierung explizit nur konsistente OBDDs erzeugt.

Grundlegend hierfür ist die Funktion $\texttt{CheckTable}(a, T)$, die zu einem Vektor $a \in \{0, 1, \star\}^n$ und einer konsistenten Trainingsmenge T die Menge

$$\{y \in \{0, 1\} \mid (x, y) \in T \text{ und } \forall i \in \{1, \dots, n\} \text{ mit } a_i \neq \star : x_i = a_i\}$$

ausgibt. Somit bestimmt $\texttt{CheckTable}(a, T)$ genau die Menge der Ausgaben $y \in \{0, 1\}$, für die es eine zu a passende Eingabe $x \in \{0, 1\}^n$ in T gibt. Der Vektor $a \in \{0, 1, \star\}^n$ kann, wie in Abschnitt 4.4 gezeigt, mit einem Teil-Pfad in einem OBDD von der Wurzel zu einem Knoten v identifiziert werden. Somit gibt die Größe C der Ausgabe von $\texttt{CheckTable}(a, T)$ an, ob das ggf. an dem zu a gehörigen Knoten v beginnende TeilOBDD für die Konsistenz mit T beliebig ist ($C = 0$), ob v durch die entsprechende Senke ersetzt werden kann ($C = 1$) oder ob v ein innerer Knoten sein muss, da ansonsten die Konsistenz verletzt wäre ($C = 2$). Die Auswertung von $\texttt{CheckTable}$ benötigt nur Laufzeit $O(n \cdot |T|)$.

Die Erzeugung eines zu T konsistenten π-OBDDs zu gegebener Trainingsmenge T und Variablenordnung π erfolgt durch eine leicht modifizierte Version der rekursiven pseudozufälligen Initialisierung (Algorithmus 4.4.3), genannt $\texttt{BuildOBDD}(i, a)$ ($i \in \{1, \dots, n+1\}$ und $a \in \{0, 1, \star\}$). Diese erzeugt ein konsistentes pseudozufälliges π-TeilOBDD, dessen Wurzel die Markierung $x_{\pi(i)}$ hat bzw. für $i = n + 1$ eine Senke ist, und im kompletten π-OBDD an dem Pfad a beginnt. Durch Aufruf von $\texttt{BuildOBDD}(1, \{\star, \dots, \star\})$ erhält man also ein zu T konsistentes pseudozufälliges π-OBDD:

Algorithmus 5.3.1 ($\texttt{BuildOBDD}(i, a)$)

1. *Falls $i = n + 1$, gib* $\begin{cases} \textit{die Nullsenke aus, falls } \texttt{CheckTable}(a, T) = \{0\}, \\ \textit{die Einssenke aus, falls } \texttt{CheckTable}(a, T) = \{1\}, \\ \textit{ansonsten gleichverteilt eine davon.} \end{cases}$

2. *Setze $a^0 := a$ und $(a^0)_{\pi(i)} := 0$.*

3. *Setze $a^1 := a$ und $(a^1)_{\pi(i)} := 1$.*

4. *Falls $|\texttt{CheckTable}(a, T)| = 2$ ist, gib einen Knoten mit Markierung $x_{\pi(i)}$, Nullnachfolger $\texttt{BuildOBDD}(i + 1, a_0)$ und Einsnachfolger $\texttt{BuildOBDD}(i + 1, a_1)$ zurück.*

5. *Bestimme ansonsten unabhängig zufällige Werte $\delta_0, \delta_1 \in \{1, \dots, n-i+1\}$ nach der Verteilung Δ_i (Definition 4.4.4) und gib einen Knoten mit Markierung $x_{\pi(i)}$, Nullnachfolger $\texttt{BuildOBDD}(i+\delta_0, a_0)$ und Einsnachfolger $\texttt{BuildOBDD}(i+\delta_1, a_1)$ zurück.*

Nur solange der aktuell konstruierte Knoten für die Konsistenz mit zwei Trainingseingaben mit unterschiedlichen Ausgaben wichtig ist, werden sowohl sein Null- als auch sein Einsnachfolger mit der nächsten Variable gemäß π markiert. Haben alle relevanten Trainingsbeispiele jedoch dieselbe Ausgabe oder gibt es keine relevanten Trainingsbeispiele mehr, so wird ein bis auf die für T relevanten Senken zufälliges π-OBDD wie in Algorithmus 4.4.3 konstruiert.

Mutation

Die Occam-GP-Mutation erfolgt exakt wie die OBDD-GP-Mutation (Algorithmus 4.4.5). Ist das erzeute OBDD nicht konsistent zu T, wird es durch seinen Elter ersetzt.

Rekombination

Die Occam-GP-Rekombination erfolgt exakt wie die OBDD-GP-Rekombination (Algorithmus 4.4.7). Ist das erzeugte OBDD nicht konsistent zu T, wird es durch seinen ersten Elter ersetzt.

Selektion

Als Selektionsmechanismus zur Auswahl der nächsten Elternpopulation aus der Vereinigung der jetzigen Eltern- und Kinderpopulation wird die (μ, λ)-Selektion wie im MBGP- bzw. OBDD-GP-System verwandt. Dabei ist der Zielfunktionswert eines reduzierten π-OBDDs D die Anzahl $|D|$ seiner (inneren) Knoten.

Das Occam-GP-System

Nachdem nun alle Teile vorgestellt sind, sei das Occam-GP-System zur Klarheit zusammengefasst:

Algorithmus 5.3.2 (Occam-GP-System)

Eingabe: *eine konsistente Trainingsmenge* $T \subset \{(x, y) \mid x \in \{0,1\}^n, y \in \{0,1\}\}$.
Ausgabe: *ein zu* T *konsistentes* π*-OBDD* D.

1. *Setze* $g := 0$. *Initialisiere die Population* P_g*, indem* μ *durch* `BuildOBDD`$(1, (\star, \ldots, \star))$ *erzeugte* π*-OBDDs aufgenommen werden.*
2. *Setze* $i := 1$.
3. *Wähle zwei* π*-OBDDs* D_1 *und* D_2 *gleichverteilt zufällig aus* P_g *aus.*
4. *Rekombiniere* D_1 *und* D_2 *durch* Occam-GP*-Rekombination und nenne das Ergebnis* D'. *Falls* D' *nicht zu* T *konsistent ist, lösche* D' *und kopiere* D_1 *nach* D'.
5. *Mutiere* D' *durch* Occam-GP*-Mutation und nenne das Ergebnis* D''. *Falls* D'' *zu* T *nicht konsistent ist, lösche* D'' *und nenne* D' *zu* D'' *um.*
6. *Nehme* D'' *in* P_{g+1} *auf.*
7. *Falls* $i < \lambda$*, setze* $i := i + 1$ *und gehe zu Schritt 3.*
8. *Schränke* P_{g+1} *auf die* μ *kleinsten* π*-OBDDs ein, wobei bei Gleichheit zufällig gewählt wird.*
9. *Falls* $g < G$*, setze* $g := g + 1$ *und gehe zu Schritt 2.*
10. *Gib eines der kleinsten* π*-OBDDs aus, die gefunden wurden.*

5.4 Anwendung des Occam's Razor Theorems

In diesem Abschnitt wird gezeigt, wie Occam's Razor Theorem genutzt werden kann, um die Generalisierungsgüte der Ergebnisse des OCCAM-GP-Systems nach unten abzuschätzen. Dabei werden die ausgegebenen OBDDs von OCCAM-GP nur nach ihrer Größe bewertet, da sie stets mit T konsistent sind. Deshalb wird die Menge H stets als Menge aller Funktionen gewählt, die für eine feste Variablenordnung π ein reduziertes π-OBDD mit höchstens k Knoten besitzen.

Um Occam's Razor Theorem bzw. Korollar 5.2.2 benutzen zu können, muss die Größe der Menge H nach oben abgeschätzt werden können. Wir werden dies im Folgenden für die Zahl der Funktionen mit π-OBDDs einer bestimmten Maximalgröße ausführen. Sei $T(n, k)$ die Zahl der booleschen Funktionen auf n Eingabevariablen, die von einem reduzierten π-OBDD bei festgelegter Variablenordnung π mit höchstens k Knoten dargestellt werden können. Ein π-OBDD D auf n Variablen mit genau k Knoten kann in drei Teile eingeteilt werden:

1. Die Wurzel mit Markierung $x_{\pi(i)}$ für ein $i \in \{1, \ldots, n\}$.
2. Das an dem Nullnachfolger der Wurzel beginnende TeilOBDD D_0 auf höchstens $n - i$ Variablen, dessen Knotenanzahl $k_0 \in \{0, \ldots, k - 1\}$ genannt sei.
3. Die an dem Einsnachfolger der Wurzel beginnende Struktur D_1, wobei jedoch alle schon in D_0 enthaltenen Knoten ausgenommen sind. Diese enthält höchstens $n - i$ verschiedene Variablen und höchstens $k - k_0 - 1$ Knoten.

Für das TeilOBDD D_0 gibt es $T(n - i, k_0)$ Möglichkeiten. Die Struktur D_1 ist kein OBDD mehr, da sie Kanten enthalten kann, die zu Knoten in D_0 zeigen, also nicht mehr zu D_1 gehören. Erweitert man die Definition von OBDDs insoweit, dass die Anzahl der Senken gleich $s \geq 2$ ist, so entspricht D_1 einem OBDD auf $n - i$ Variablen mit höchstens $k - k_0 - 1$ Knoten und höchstens $s + k_0$ Senken.

Sei also $T(n, k, s)$ eine obere Schranke der Zahl der reduzierten OBDDs auf n Variablen mit genau k Knoten und s Senken, so kann man aus obiger Überlegung folgende Rekursionsformel schließen

$$T(n, k, s) \leq \sum_{i=1}^{n} \sum_{k_0=0}^{k-1} T(n - i, k_0, s) \cdot T(n - i, k - k_0 - 1, s + k_0).$$

Der Rekursionsanfang ergibt sich durch Testen folgender Fälle in genau dieser Reihenfolge:

1. Falls $k = 0$ ist, so ist $T(n, k, s) = s$, da ein OBDD ohne Knoten eine Senke sein muss.
2. Falls $n = 0$ ist, so ist $T(n, k, s) = 0$, da es kein OBDD mit $k > 1$ Knoten, jedoch ohne Variablen gibt.
3. Falls $k = 1$ ist, so ist $T(n, k, s) = n \cdot s \cdot (s - 1)$, da es n Möglichkeiten für die Markierung des Knotens gibt und $s \cdot (s - 1)$ Möglichkeiten für die Auswahl der beiden Senken.
4. Falls $k = 2$ ist, so ist

   $$T(n, k, s) = \sum_{i=1}^{n-1} 2 \cdot s \cdot (n - i) \cdot s \cdot (s - 1),$$

   da es bei Markierung $x_{\pi(i)}$ des ersten Knotens $(n - i) \cdot s \cdot (s - 1)$ Möglichkeiten für das an dem zweiten Knoten beginnende TeilOBDD gibt, s Möglichkeiten für den zweiten Nachfolger des ersten Knotens und zwei Möglichkeiten, Null- und Einsnachfolger zu ordnen.

Somit kann die Anzahl der reduzierten π-OBDDs mit höchstens k Knoten durch $S(k,n) := \sum_{i=0}^{k} T(n,i,2)$ nach oben abgeschätzt werden, was in den folgenden sich aus Occam's Razor Theorem ergebenden Abschätzungen stets gemacht wird.

Um diese auf ihre Genauigkeit abzuschätzen, wird die den Trainingsmengen zugrundeliegende Funktion entweder die Multiplexer-Funktion Mux (Definition 4.4.1), die Parity-Funktion Par (Definition 4.4.2) oder die Additionsfunktion Add sein:

Definition 5.4.1 *Sei $n \in \mathbb{N}$. Die n-Additions-Funktion* $\text{ADD}_n : \{0,1\}^{2n} \mapsto \{0,1\}$ *ist definiert als*

$$\text{ADD}_n(x_0,\ldots,x_{n-1},y_0,\ldots,y_{n-1}) := \left\lfloor \left(\sum_{i=0}^{n-1}(x_i+y_i)\cdot 2^{n-1-i}\right)/2^n \right\rfloor .$$

Anschaulich gesprochen, gibt ADD_n das $(n+1)$-te Bit der Summe der zwei n-Bit Zahlen $(x_0,\ldots,x_{n-1})$ und $(y_0,\ldots,y_{n-1})$ zurück.

Es werden dann vor jedem Lauf von Occam-GP genau m Trainingseingaben aus $\{0,1\}^n$ zufällig gleichverteilt ausgewählt (dabei werden Duplikate nicht verhindert) und diese jeweils mit ihrem entsprechenden Funktionswert in die Trainingsmenge T aufgenommen. Da für die Multiplexer- und Additions-Funktion die Variablenordnung π entscheidenden Einfluss auf die Größe der darstellenden reduzierten π-OBDDs hat und das Occam-GP-System als ein auf der syntaktischen Ebene arbeitendes System davon beeinflusst wird, wird es auf folgenden fünf Kombinationen von Funktion und Variablenordnung getestet:

- MUX_{20}^{+}: Es wird die Funktion MUX_n mit $n = 20$ und der Variablenordnung $a_0,\ldots,a_3,d_0,\ldots,d_{15}$ verwendet. Diese Variablenordnung ist für MUX_{20} optimal, das reduzierte OBDD für MUX_{20} hat genau 31 Knoten.

- MUX_{20}^{-}: Es wird die Funktion MUX_n mit $n = 20$ und der Variablenordnung $d_0,\ldots,d_{15},a_0,\ldots,a_3$ verwendet. Diese Variablenordnung ist für MUX_{20} die schlechtest mögliche, das reduzierte OBDD für MUX_{20} hat 131069 Knoten.

- ADD_{20}^{+}: Es wird die Funktion ADD_n mit $n = 20$ und der Variablenordnung $x_0,y_0,\ldots,x_9,y_9$ verwendet. Diese ermöglicht einem OBDD einen kleinen Fehler, auch wenn nur die ersten Variablen getestet werden.

- ADD_{20}^{-}: Es wird die Funktion ADD_n mit $n = 20$ und der Variablenordnung $x_9,y_9,\ldots,x_0,y_0$ verwendet. Werden nur die ersten Variablen getestet, so ist mit dieser Variablenordnung ein kleiner Fehler nicht möglich.

- PAR_{20}: Es wird die Funktion PAR_n mit $n = 20$ und der Variablenordnung $x_0,\ldots,x_{19}$ verwendet. Mit dieser wie mit jeder anderen Variablenordnung hat das reduzierte OBDD für PAR_{20} genau 39 Knoten.

In allen Läufen werden $m = 512$ Trainingsbeispiele unabhängig gleichverteilt gezogen. Da es insgesamt $2^{20} = 1048576$ mögliche Eingaben gibt, ist die Wahrscheinlichkeit, dass diese nicht alle verschieden sind, zwar größer als Null, aber vernachlässigbar klein. Die ersten $L = 100$ Läufe dienen nur dazu, anhand der Verteilung der Größen des jeweils kleinsten gefundenen OBDDs die Menge H zu bestimmen: wenn d die durchschnittliche Größe des kleinsten gefundenen OBDDs über die $L = 100$ Läufe ist, so wird H als die Menge der Funktionen mit höchstens $\lfloor d \rfloor$ Knoten gewählt. In den ersten L Läufen ergeben sich die in Tabelle 5.1 gezeigten Ergebnisse, wobei zu Vergleichszwecken noch der durchschnittliche Fehler der jeweils kleinsten gefundenen OBDDs über die $L = 100$ Läufe bzgl. der zugrundeliegenden Funktion angegeben ist. (Der durchschnittliche Fehler für PAR_{20} von genau 50 Prozent erklärt sich dadurch, dass ein OBDD, um auf der Parity-Funktion einen

Funktion	Durchschnittliche OBDD-Größe d	Durchschnittlicher Fehler
MUX_{20}^{+}	51,3	6,9 %
MUX_{20}^{-}	170,8	46,4 %
ADD_{20}^{+}	32,1	4,3 %
ADD_{20}^{-}	100,1	29,6 %
PAR_{20}	184,1	50,0 %

Tabelle 5.1: Die Ergebnisse der ersten $L = 100$ Läufe.

Funktion	$\lfloor d \rfloor$	Anzahl OBDDs mit $\leq \lfloor d \rfloor$ Knoten	Fehler ε
MUX_{20}^{+}	51	45	44,4 %
MUX_{20}^{-}	170	50	89,8 %
ADD_{20}^{+}	32	62	29,5 %
ADD_{20}^{-}	100	50	71,5 %
PAR_{20}	184	48	91,8 %

Tabelle 5.2: Die Ergebnisse der zweiten $L = 100$ Läufe.

kleineren Fehler als 50 Prozent zu haben, mindestens einen Pfad von der Quelle zu einer Senke besitzen muss, auf dem alle Variablen getestet werden. Mit hoher Wahrscheinlichkeit werden die nur 512 Trainingsbeispiele einen solchen Pfad nicht erzwingen, so dass durch die angestrebte Verkleinerung der OBDDs solche Pfade nur mit sehr geringer Wahrscheinlichkeit zustande kommen.)

Da man annehmen kann, dass die Ergebnisse von OCCAM-GP bzgl. der Größe reproduzierbar sind, macht es Sinn, für die zweiten $L = 100$ unabhängig durchgeführten Läufe H jeweils als die Menge aller Funktionen mit reduzierten π-OBDDs mit maximal $\lfloor d \rfloor$ Knoten zu wählen. Nach Korollar 5.2.2 ist die Wahrscheinlichkeit, dass die Trainingseingaben so gewählt werden, dass in allen L Läufen alle Funktionen aus H mit Fehler größer als ε nicht zu der Trainingsmenge T konsistent sind, mindestens gleich

$$\left(\max(0, 1 - |H| \cdot (1-\varepsilon)^m)\right)^L .$$

Von Interesse ist der kleinste Wert von ε, so dass die resultierende Wahrscheinlichkeit mindestens gleich einem als ausreichend empfundenen Wert $z \in [0,1]$ ist, was äquivalent zu

$$\frac{1 - z^{1/L}}{|H|} \geq (1-\varepsilon)^m \iff \varepsilon \geq 1 - \left(\frac{1 - z^{1/L}}{|H|}\right)^{1/m}$$

ist. In den zweiten $L = 100$ Läufen ergeben sich die in Tabelle 5.2 gezeigten Ergebnisse. Aufgeführt sind die Größenschranke $\lfloor d \rfloor$ aus den ersten $L = 100$ Läufen, die Anzahl der OBDDs mit höchstens $\lfloor d \rfloor$ Knoten aus den zweiten $L = 100$ Läufen, die OCCAM-GP gefunden hat, und der nach obiger Formel resultierende kleinste Fehler ε für die gewählte Wahrscheinlichkeit von $z = 0{,}99$.

Somit haben alle 62 in den zweiten 100 Läufen gefundenen OBDDs für ADD_{20}^{+} mit einer Wahrscheinlichkeit von 99 Prozent nur einen Fehler von 29,5 Prozent. In Anbetracht der Tatsache, dass die Informationen, die das OCCAM-GP-System von der Funktion hat, nur aus maximal 51200 verschiedene Trainingsbeispielen von insgesamt 1048576 möglichen Eingaben besteht, ist dies eine sehr gute Garantie, auch wenn die gefundenen OBDDs einen wesentlich geringeren Fehler haben (siehe die auch für die zweiten 100 Läufe repräsentativen Zahlen in Tabelle 5.1).

Ebenso kann auch für die 45 in den zweiten 100 Läufen gefundenen OBDDs für MUX_{20}^{+} gesagt werden, dass die Wahl der Trainingsbeispiele nur eine Wahrscheinlichkeit von höchstens einem Prozent hat, wenn auch nur eine der von diesen repräsentierten Funktion einen Fehler von mehr als 44,4 Prozent hat.

Jedoch sind die Fehlergarantien für alle anderen Funktionen sehr viel größer. Schon für das nächstbeste Ergebnis bei ADD^{-}_{20} ist die Fehlergarantie größer als 50,26 Prozent. Dies ist aber die Fehlerwahrscheinlichkeit, die mit 99 Prozent Wahrscheinlichkeit (über die Wahl der Funktion) aus einer rein zufälligen Auswahl der Funktion resultiert. Denn mit der Tschernoff-Ungleichung (A.9) ergibt sich, dass die Wahrscheinlichkeit, eine zufällige Funktion mit einem Fehler von mindestens $\alpha \in [0, 1]$ zu ziehen, gleich

$$\mathrm{P}(X \geq (1 + (2 \cdot \alpha - 1)) \cdot 2^{19}) \leq \exp\left(-\frac{(2 \cdot \alpha - 1)^2 \cdot 2^{19}}{3}\right)$$

ist (dabei gibt die Zufallsvariable X die Zahl der Fehler an). Für $\alpha \geq 0{,}5026$ ist dieser Fehler kleiner als 0,01. Somit sind die Garantien für alle anderen Funktionen nicht praktisch verwertbar. Dies ließe sich auf zwei Arten beheben: die Menge H der Funktionen könnte noch weiter eingeschränkt werden, indem man z. B. die maximale Größe der 10 Prozent kleinsten OBDDs in den ersten L Läufen misst und diese als Obergrenze für die Knotenanzahl von Funktionen in H wählt. Auch in diesem Fall sollte die Menge der Funktionen aus H in den zweiten L Läufen zumindest nicht leer sein, die gefundenen Gütegarantien aber besser. Zweitens könnten wir versuchen, das OCCAM-GP-System zu verbessern, damit es kleinere OBDDs ausgibt.

Doch zeigen schon die bisherigen Ergebnisse, wie es möglich ist, die Wahrscheinlichkeit des Fehlers von GP-Systemen zum Lernen unvollständig definierter Funktionen theoretisch zu beschränken. Dabei ist wichtig, dass die Testfunktionen MUX, ADD und PAR nur über die Trainingseingaben in OCCAM-GP eingeflossen sind. Diese Methode ist also auch anwendbar und dann erst von Interesse, wenn die Zielfunktionen unbekannt sind. Solange unabhängig gezogene Trainingsbeispiele zur Verfügung stehen und der betrachtete Algorithmus reproduzierbar zu diesen konsistente Funktionen aus einer kleinen Menge ausgibt, lässt sich die Wahrscheinlichkeit, dass diese alle nur einen kleinen Fehler haben, nach unten abschätzen.

Kapitel 6

Zusammenfasssung und Ausblick

Evolutionäre Algorithmen bilden eine Klasse allgemeiner Suchverfahren, die oftmals für schlecht verstandene Optimierprobleme gute Lösungen liefern. Ihre Beschreibung basiert auf Prinzipien der natürlichen Evolution wie Mutation, Rekombination und Selektion, deren Einsichtigkeit und leichte Anwendbarkeit auch ohne tieferes Problemwissen viel zum Erfolg evolutionärer Algorithmen beigetragen haben. Doch bilden sie ein so breites Konzept, dass die Versuche, eine umfassende Theorie aufzubauen, nur geringe Aussagekraft haben, wie die Bemühungen um das Schema-Theorem zeigen. Andere spezifischere Versuche, theoretische Ergebnisse zu erlangen, beruhen oftmals auf unbewiesenen Annahmen oder sind nicht aufeinander aufbauend. Dies behindert natürlich den systematischen Entwurf evolutionärer Algorithmen, der sich aus Mangel an gesicherten Ergebnissen oftmals an historisch gewachsenen Grundformen und empirisch belegten Daumenregeln orientiert.

Die hier vorgestellten Ergebnisse sollen einen kleinen Schritt bilden, diese Mängel zu beheben. Dabei haben wir in Kapitel 2 zuerst geklärt, was allgemeine Suchverfahren zu leisten in der Lage sind, wenn sie im Durchschnitt über alle Funktionen zwischen zwei festen endlichen Mengen betrachtet werden. Wir haben dabei das NFL-Theorem von Wolpert und Macready (1997) mitsamt einem einfachem Beweis rekapituliert, das zeigt, dass eine einschränkungslose Betrachtung aller Funktionen sinnlos ist, da in diesem Szenario alle Suchverfahren im Durchschnitt die gleiche Zahl an unterschiedlichen Funktionsauswertungen durchführen, bis sie ein Optimum gefunden haben. Ein „Free Lunch“, ein Gewinn, ohne dafür an anderer Stelle zu verlieren, ist also nicht möglich. Jedoch ist die Annahme des NFL-Theorems nicht realistisch, da nicht alle Funktionen in der Praxis zu optimieren sind. Eine Formalisierung der „natürlichen“ Probleme und somit eine Widerlegung des NFL-Theorems über diese Funktionsklasse ist aber nicht möglich. Die in Abschnitt 2.3 angeführten Beschränkungen der Komplexität von Funktionen sind aber zumindest notwendige Bedingungen für ihr Vorkommen in praktischen Optimierproblemen.

Auch wenn wir anhand eines kleinen Beispiels gezeigt haben, dass in diesen eingeschränkten Szenarien im Allgemeinen kein NFL-Theorem gelten muss, d. h. verschiedene Suchverfahren verschiedene Güten haben können, haben wir mit dem allgemeinen NFL-Theorem und dem ANFL-Theorem diese Erkenntnis relativiert. Denn das in Abschnitt 2.5 vorgestellte allgemeine NFL-Theorem erweitert die ursprünglich angenommenen Voraussetzungen auf den Fall, dass nur alle Funktionen einer unter Permutationen abgeschlossenen Klasse betrachtet werden. An Beispielen haben wir gesehen, dass komplexitätsbeschränkte Szenarien somit stets Teilklassen enthalten, in denen ein NFL-Theorem gilt. Noch verstärkend hat uns das „Almost

No Free Lunch“ (ANFL)-Theorem gezeigt, dass es zu von einem Suchverfahren effizient zu optimierenden Funktionen nur wenig komplexere gibt, auf denen das Suchverfahren mit hoher Wahrscheinlichkeit ineffizient ist. Dies impliziert natürlich nichts über die durchschnittlichen Kosten eines Suchverfahrens in einem komplexitätsbeschränkten Szenario, zeigt aber, dass es kein Suchverfahren gibt, das in einem solchen Szenario stets effizient ist. Es wird also immer Funktionen geben, wo man hoch verlieren wird, insofern gilt „fast“ ein NFL-Theorem.

Aus diesen Betrachtungen haben wir den Schluss gezogen, dass es wenig sinnvoll ist, über zu große Klassen von Funktionen argumentieren zu wollen. Natürlich werden sinnvolle Average-Case-Aussagen über das Verhalten von Suchverfahren in komplexitätsbeschränkten Szenarien von unseren Resultaten nicht ausgeschlossen, doch scheint die Übertragung des Wissens über beschränkte Komplexität auf solchermaßen verwertbare Aussagen schwierig zu sein. Deshalb könnte es eher sinnvoll sein, andere Szenarien auszumachen, in denen NFL-Resultate zumindest näherungsweise gelten, und somit die Grenzen der Einsetzbarkeit allgemeiner Suchverfahren enger zu ziehen.

Aus den Resultaten zum NFL-Theorem und den bisher vergeblichen Versuchen, eine Theorie evolutionärer Algorithmen für zu große Klassen von Algorithmen finden zu wollen, haben wir für Kapitel 3 den Schluss gezogen, die Analysen auf einen genau definierten evolutionären Algorithmus, den (1+1) EA, und Varianten hiervon zu beschränken. Aufgrund der Vielzahl der schon für diesen einfachen Algorithmus offenen Probleme und der Möglichkeit, wegen der Einschränkung auf Mutation und Selektion Analyseergebnisse nur diesen Operatoren zuzuordnen, wurde gerade dieser Algorithmus gewählt. Dabei stand die mathematisch exakte Abschätzung der Laufzeit, d. h. der Zahl an Funktionsauswertungen, die der (1+1) EA bis zum erstmaligen Finden eines Optimums benötigt, im Zentrum unseres Interesses. Ohne vereinfachende Annahmen wurden asymptotische Analysen durchgeführt, wobei aber auch anschauliche Begründungen des Verhaltens erläutert wurden.

Dabei teilen sich die Resultate in zwei Klassen auf: einerseits wurde die Laufzeit des (1+1) EA für typische Funktionsklassen, wie lineare, unimodale oder Funktionen vom Grad Zwei untersucht, um zu erkennen, wo die Grenzen dieses einfachen evolutionären Algorithmus liegen. Dabei war der Aufwand der Analysen desto größer, je komplizierter die Funktion aufgebaut bzw. je größer die betrachtete Funktionsklasse war: die Analyse für OneMax war sehr einfach, für BinVal schon schwieriger, da diese Funktion verbessernde Mutationen zu vom Optimum entfernteren Punkten ermöglicht („falsche Hinweise“), und für die erwartete Laufzeit von $\Theta(n\log(n))$ für alle linearen Funktionen demgemäß am aufwendigsten. Mit Letzterem wurde eine lange gehegte Vermutung (Mühlenbein (1992)) bewiesen.

Das anschauliche Konzept der schlechten Hinweise erwies sich auch bei der Betrachtung von Funktionen zweiten Grades als hilfreich, denn die aus der Anschauung begründete Funktion Distance, einer nur leicht verschobenen Parabel in der Anzahl der Einsen des Arguments, ließ sich als für den (1+1) EA schwierig zu optimieren beweisen, wenn man die erwartete Laufzeit als Maßstab nimmt. Somit wurde ein Beispiel gefunden, das die aus komplexitätstheoretischen Argumenten geborene Vermutung, dass es eine vom (1+1) EA nur mit exponentiellem Aufwand zu optimierende Funktion vom Grad Zwei gibt, explizit belegt.

Eine ähnliche Rolle spielte die Funktion LongPath für die Klasse aller unimodalen Funktionen. Obwohl diese oftmals als leicht zu optimieren angesehen werden, was zu der Vermutung geführt hat, dass sie in erwarteter Zeit $\Theta(n\log(n))$ vom (1+1) EA optimiert werden können (Mühlenbein (1992)), zeigt LongPath, dass dem nicht so ist, da im Erwartungsfall exponentielle Laufzeit benötigt wird. Die Konstruktion dieser Funktion bestätigt die Anschauung, dass die bei unimodalen Funktionen unausweichlich recht wahrscheinlichen Verbesserungen für eine effiziente Optimierung nicht ausreichend sind, wenn diese nur über einen sehr langen Pfad zum Optimum führen. Dabei spielte entscheidend die Tatsache eine Rolle, dass der (1+1) EA „kurzsichtig“ ist, d. h. nur mit exponentiell kleiner Wahrscheinlichkeit Punkte mit einem Hamming-Abstand von mindestens $\sqrt{n}$ „betrachtet“.

Die andere Klasse von Ergebnissen bestand in dem Nachweis von trennenden Beispielen für zwei Klassen von Algorithmen, d. h. Funktionen, die von Algorithmen der einen Klasse effizient optimiert werden können, während allen der anderen Klasse dies nicht gelingt. Dafür sind die Funktionen Valley und ModJump_2 konstruiert worden, die einfach strukturiert sind und zeigen, wann Simulated Annealing besser als jeder Metropolis-Algorithmus sein kann. Diese gehören zu den bestbekannten lokalen Suchverfahren, lassen sich jedoch problemlos auch als evolutionäre Algorithmen bezeichnen. Der hier vorgestellte Beweis, der sich im Gegensatz zu einem ähnlichen Resultat (Sorkin (1991)) auf sehr einfache Art und Weise erbringen ließ, könnte deshalb einen Schritt zu dem Ziel darstellen, eine natürliche Funktion zu finden, die den Metropolis-Algorithmus und Simulated Annealing solchermaßen trennt (Jerrum und Sinclair (1997)).

Wie die bei „künstlichen“, d. h. kompliziert aufgebauten oder keine Vertreter typischer Probleme darstellenden, Funktionen geübten Techniken und festgestellten Eigenschaften genutzt werden können, um eine „natürliche“ Funktion als schwierig vom (1+1) EA zu optimieren nachzuweisen, zeigte Abschnitt 3.7. Die dort analysierte Funktion MaxCount leitet sich direkt aus einem der bekanntesten Probleme, MAXSAT, der Komplexitätstheorie ab und ist zudem einfach zu beschreiben. Trotzdem konnten wir zeigen, dass zu ihrer Optimierung sowohl der (1+1) EA als auch jeder Vertreter einer großen, jedoch genau definierten Klasse von mutationsbasierten evolutionären Algorithmen, mit hoher Wahrscheinlichkeit exponentielle Laufzeit benötigt.

Dieses Resultat zeigt eine Richtung auf, in die die Analyse evolutionärer Algorithmen gehen kann: die Untersuchung von Funktionen, die Vertreter wichtiger Probleme sind. Die bisherigen Untersuchungen sind hauptsächlich auf Funktionen einfacher Struktur oder speziell konstruierte Beispiele konzentriert. Eine Erweiterung auf wichtige Probleme, deren Komplexität wie z. B. beim Sortierproblem bekannt ist, kann nicht zum Ziel haben zu zeigen, dass evolutionäre Algorithmen mit Spezialverfahren konkurrieren können. Doch könnten solche Ergebnisse eine Übertragung der theoretischen Ergebnisse auf in der Praxis verwendete Probleme erleichtern, da im Gegensatz zu vielen bisher verwendeten Testfunktionen nicht ihre syntaktische, sondern ihre semantische Struktur gut verstanden ist.

Eine andere einzuschlagende Richtung der Analyse evolutionärer Algorithmen ist die Untersuchung komplizierterer und realistischerer Algorithmen. Der (1+1) EA spiegelt zwar die Grundprinzipien Mutation und Selektion wider, doch lässt seine Analyse natürlich keine Schlüsse auf evolutionäre Algorithmen mit Rekombination zu. Diese ist jedoch zusammen mit der Verwendung ganzer Populationen von Suchpunkten ein entscheidendes Prinzip evolutionärer Algorithmen. Herauszufinden, für welche Funktionen erst die Verwendung von Populationen ggf. zusammen mit Rekombination eine effiziente Optimierung ermöglicht, ist zentral. Ein erstes Beispiel für die Nützlichkeit der Rekombination wurde in Jansen und Wegener (1999) gefunden. Dieser richtungsweisenden Arbeit sollten weitere folgen, die Funktionen charakterisieren, bei denen die Verwendung bestimmter Module bzw. Strategien evolutionärer Algorithmen notwendig ist (siehe auch Wegener (2000)).

Aber auch wenn solche Bemühungen fortgeführt werden, wird es bis zur Übertragbarkeit theoretischer Analyseresultate auf den Entwurf evolutionärer Algorithmen noch dauern. Denn die in der Praxis verwendeten Systeme sind oftmals wesentlich komplizierter, wie die Vorstellung der genetischen Programmierung in Abschnitt 4.2 nur angerissen hat. Einen ersten Schritt, solche Systeme gezielt zu entwerfen und eine Möglichkeit zu haben, ihre Auswirkungen abzuschätzen, sollen die in Kapitel 4 vorgestellten MBEA-Richtlinien bilden. Diese sind einerseits exakt formalisiert, so dass ein Beweis, ob sie eingehalten sind, wie beispielhaft vorgeführt, möglich ist. Andererseits spiegeln sie anschauliche Vorstellungen von der Wirkungsweise der genetischen Operatoren wider, um so nicht einen Hauptvorteil evolutionärer Algorithmen, die leichte Verständlichkeit ihres Konzepts, einzuschränken. Dadurch, dass sie den Aufbau eines evolutionären Algorithmus explizit mit der Zielfunktion verbinden, eröffnen sie die Chance, dass entsprechend entworfene evolutionäre Algorithmen nicht nur aufgrund des glücklichen Zusammenpassens von Algorithmus und Zielfunktion effizient arbeiten.

Jedoch sind die MBEA-Richtlinien noch zu allgemein, als dass ihre Auswirkungen theoretisch analysiert werden konnten. Deshalb wurde ihre Nützlichkeit anhand der Konstruktion eines Systems der genetischen Programmierung unter (empirischen) Beweis gestellt. Zur Einhaltung der MBEA-Richtlinien wurde dafür nicht die übliche, mit einigen offensichtlichen Nachteilen behaftete Repräsentation mit S-Expressions benutzt, sondern eine der handhabbarsten Darstellungsformen für boolesche Funktionen, die so genannten OBDDs. Die Verwendung dieser Datenstruktur ermöglichte eine leichte Umsetzbarkeit der Richtlinien, die offensichtliche Erkenntnis stützend, dass die verwendete Datenstruktur auf das Problem zugeschnitten sein muss. Dass schon eine direkte Übertragung von Mutation und Rekombination von S-Expressions auf OBDDs eine höhere Effizienz zur Folge hatte, zeigt, wie kompliziert und wenig verstanden der resultierende Suchprozess ist.

Zur Stützung bzw. Erweiterung oder Umformung der MBEA-Richtlinien bieten sich mehrere Wege an. Natürlich ist es sinnvoll, die Nüzlichkeit der MBEA-Richtlinien in weiteren Problemkreisen zu überprüfen, also eine größere empirische Grundlage zu legen. Weiterhin sollte auch versucht werden, durch eine, wenn notwendig, stärkere Einschränkung der Richtlinien zu theoretischen Aussagen über evolutionäre Algorithmen auf bestimmten Funktionsklassen zu kommen. Dass dies durchführbar ist, zeigt der Beweis der Schwierigkeit von MaxCount in Abschnitt 3.7. Schon dies kann Grenzen und Möglichkeiten der MBEA-konformen Algorithmen aufweisen, auch wenn es nur für wenige Funktionen gelingen mag.

Eine ganz andere Möglichkeit, wie theoretische Ergebnisse hilfreich zur Verwendung evolutionärer Algorithmen eingesetzt werden können, zeigte die Übertragung von Occam's Razor Theorem (Blumer, Ehrenfeucht, Haussler und Warmuth (1990)) in Kapitel 5 auf die genetische Programmierung zum Lernen unvollständig spezifizierter Funktionen. Die Idee, dass in einer kleinen Funktionmenge, die für viele unabhängig zufällig gezogene Trainingsbeispiele konsistente Funktionen enthält, mit hoher Wahrscheinlichkeit nur gut generalisierende Funktionen sind, drückt dieses quantitativ aus und ermöglicht somit eine probabilistische Garantie für die Generalisierungsgüte. Dass diese gut genug ist, nicht-triviale Schranken zu liefern, wurde anhand eines konkreten GP-Systems beispielhaft gezeigt.

Dieses Ergebnis zeigt wie die anderen dieser Arbeit, dass im Bereich evolutionärer Algorithmen eine Vielzahl von interessanten Fragestellungen existiert, deren Lösung für eine Annäherung von theoretisch gesichertem Wissen und empirischer Erfahrung wichtig ist.

Anhang A

Grundlegende mathematische Begriffe

Grundbegriffe der Stochastik

Die hier aufgeführten Definitionen finden sich, wenn nicht anders vermerkt, in Krengel (1991).

Sei Ω eine abzählbare Menge. Dann heißt $\mathcal{P}(\Omega)$, die Menge aller Teilmengen von Ω, die *Potenzmenge* von Ω. Diskrete Wahrscheinlichkeitsräume, die für unsere Modellierungen ausreichen, lassen sich folgendermaßen definieren:

A.1 (Diskreter Wahrscheinlichkeitsraum) *Sei* Ω *eine abzählbare nicht-leere Menge und* $\mathrm{P} : \mathcal{P}(\Omega) \mapsto [0, 1]$ *mit den Eigenschaften*

1. $\mathrm{P}(\Omega) = 1$,
2. $\forall A \in \mathcal{P}(\Omega) : \mathrm{P}(A) \geq 0$ *und*
3. $\forall A_1, A_2, \dots \in \mathcal{P}(\Omega)$: *sind* $A_1, A_2, \dots$ *paarweise disjunkt, so ist*
$$\mathrm{P}\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mathrm{P}(A_i).$$

Dann heißt Ω Ereignismenge, P Wahrscheinlichkeitsverteilung *über* Ω *und das Paar* (Ω, P) diskreter Wahrscheinlichkeitsraum.

Zur Modellierung von zufälligen Ereignissen ist der Begriff der Zufallsvariablen zentral:

A.2 (Zufallsvariable) *Sei* (Ω, P) *ein diskreter Wahrscheinlichkeitsraum und* W *eine beliebige Menge, so heißt eine Abbildung* $Z : \Omega \mapsto W$ *eine* W-wertige Zufallsvariable *über* (Ω, P)*. Die Wahrscheinlichkeit* $\mathrm{P}(Z = w)$*, dass* Z *einen Wert* $w \in W$ *annimmt, ist definiert als*
$$\mathrm{P}(Z = w) := \mathrm{P}(\{\omega \in \Omega \,|\, Z(\omega) = w\}).$$

A.3 (Erwartungswert) *Der Erwartungswert* $\mathrm{E}(Z)$ *einer reellwertigen Zufallsvariablen* $Z : \Omega \mapsto \mathbb{R}$ *über einem diskreten Wahrscheinlichkeitsraum* (Ω, P) *ist definiert als*
$$\mathrm{E}(Z) := \sum_{\omega \in \Omega} Z(\omega) \cdot \mathrm{P}(\{\omega\}).$$

A.4 (Varianz) *Die Varianz* $Var(Z)$ *einer reellwertigen Zufallsvariablen* $Z : \Omega \mapsto \mathbb{R}$ *über einem diskreten Wahrscheinlichkeitsraum* (Ω, P) *ist definiert als*

$$\mathrm{E}((Z - \mathrm{E}(Z))^2).$$

A.5 (Geometrisch verteilte Zufallsvariable) *Eine* $\mathbb{N}$*-wertige Zufallsvariable* Z *heißt* geometrisch verteilt mit Parameter $p \in [0, 1]$, *wenn für alle* $k \in \mathbb{N}$ *gilt, dass*

$$\mathrm{P}(Z = k) = p \cdot (1 - p)^{k-1}$$

ist. Der Erwartungswert von Z *ist* $1/p$.

A.6 (Binomial-verteilte Zufallsvariable) *Eine* $\{0, \ldots, n\}$*-wertige Zufallsvariable* Z *heißt* binomial-verteilt mit Parametern $p \in [0, 1]$ und $n \in \mathbb{N}$, *wenn für alle* $k \in \{0, \ldots, n\}$ *gilt, dass*

$$\mathrm{P}(Z = k) = \binom{n}{k} \cdot p^k \cdot (1 - p)^{n-k}$$

ist. Der Erwartungswert von Z *ist* $n \cdot p$ *und ihre Varianz* $n \cdot p \cdot (1 - p)$.

Binomial-verteilte Zufallsvariablen mit Parametern p und n können gut durch Poisson-verteilte Zufallsvariablen angenähert werden, wenn $p \cdot n$ konstant ist:

A.7 (Poisson-verteilte Zufallsvariable) *Eine* $\mathbb{N}_0$*-wertige Zufallsvariable* Z *heißt* Poisson-verteilt mit Parameter $\lambda \geq 0$, *wenn für alle* $k \in \mathbb{N}_0$ *gilt, dass*

$$\mathrm{P}(Z = k) = \exp(-\lambda) \cdot \frac{\lambda^k}{k!}$$

ist. Der Erwartungswert und die Varianz von Z *sind jeweils gleich* λ.

Ist Z_1 *binomial-verteilt mit Parametern* p *und* n *und* Z_2 *Poisson-verteilt mit Parameter* $\lambda = p \cdot n$, *so gilt*

$$\sum_{k=0}^{n} |\mathrm{P}(Z_1 = k) - \mathrm{P}(Z_2 = k)| \leq 2 \cdot n \cdot p^2.$$

Zur Abschätzung der Verteilung einer Zufallsvariablen aus ihrem Erwartungswert ist folgende Ungleichung wichtig:

A.8 (Markoff-Ungleichung) *Sei* Z *eine reellwertige Zufallsvariable und* ϕ *eine monoton wachsende Funktion* $\phi : [0, \infty[\mapsto [0, \infty[$, *so gilt für alle* $\varepsilon \geq 0$ *mit* $\phi(\varepsilon) > 0$*:*

$$\mathrm{P}(|Z| \geq \varepsilon) \leq \frac{\mathrm{E}(\phi(|Z|))}{\phi(\varepsilon)}.$$

Setzt sich eine Zufallsvariable additiv aus vielen $\{0, 1\}$-wertigen Zufallsvariablen zusammen, so liefert die Tschernoff-Ungleichung oftmals eine sehr gute Abschätzung (siehe Hagerup und Rüb (1990)):

A.9 (Tschernoff-Ungleichung) *Seien* $X_1, \ldots, X_n$ $\{0, 1\}$*-wertige Zufallsvariablen, so dass* $\mathrm{E}(X_i) = \mu_i$ *für alle* $i \in \{1, \ldots, n\}$ *gilt. Ist* $Z = X_1 + \cdots + X_n$ *und* $\mu = \mu_1 + \cdots + \mu_n$, *so gilt für alle* $\varepsilon \in [0, 1]$*:*

$$\mathrm{P}(Z \geq (1 + \varepsilon) \cdot \mu) \leq \exp\left(-\frac{\varepsilon^2 \cdot \mu}{3}\right)$$

und

$$\mathrm{P}(Z \leq (1 - \varepsilon) \cdot \mu) \leq \exp\left(-\frac{\varepsilon^2 \cdot \mu}{2}\right).$$

Grundlegende mathematische Abschätzungen

Herleitungen der folgenden Abschätzungen finden sich in Barner und Flohr (1987).

A.10 (Stirling'sche Formel) *Sei $n \in \mathbb{N}$. Dann gilt:*

$$n! = \left(\frac{n}{\exp(1)}\right)^n \cdot \sqrt{2\pi n} \cdot \exp(\Phi(n)) \text{ mit } 0 < \Phi(n) < \frac{1}{12n}.$$

A.11 (Abschätzung von $(1 + 1/x)^x$) *Sei $x \in \mathbb{R} \setminus \{0, 1\}$. Dann gilt:*

$$\left(1 + \frac{1}{x}\right)^x \leq \exp(1) \leq \left(1 + \frac{1}{x-1}\right)^x.$$

Ersetzt man x durch $-x$, so ergibt sich:

A.12 (Abschätzung von $(1 - 1/x)^x$) *Sei $x \in \mathbb{R} \setminus \{-1, 0\}$. Dann gilt:*

$$\left(1 - \frac{1}{x}\right)^x \leq \exp(-1) \leq \left(1 - \frac{1}{x+1}\right)^x.$$

A.13 (Harmonische Reihe) *Sei $n \in \mathbb{N}$. Dann gilt für die* harmonische Reihe $H(n) := \sum_{i=1}^{n} 1/i$, *dass*

$$\lim_{n \to \infty} H(n) \nearrow \ln(n) + \gamma$$

ist, wobei $\gamma \approx 0,57722$ die Euler'sche Konstante *ist.*

Größenordnungen von Funktionen

Die folgenden Notationen halten sich an Motwani und Raghavan (1995).

A.14 (Größenordnungen) *Seien $f, g : \mathbb{R} \mapsto \mathbb{R}^+$. Dann heißt*

- *$f = O(g)$, wenn es Zahlen $c, n_0 \in \mathbb{R}^+$ gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$ ist,*
- *$f = \Omega(g)$, wenn $g = O(f)$ ist,*
- *$f = \Theta(g)$, wenn $f = O(g)$ und $f = \Omega(g)$ ist,*
- *$f = o(g)$, wenn $\lim_{n \to \infty} f(n)/g(n) = 0$ ist und*
- *$f = \omega(g)$, wenn $g = o(f)$ ist.*

A.15 (Polynomielles und exponentielles Wachstum) *Sei $f : \mathbb{R} \mapsto \mathbb{R}^+$. Dann heißt*

- f polynomiell (beschränkt), *wenn es ein Polynom $p : \mathbb{R} \mapsto \mathbb{R}^+$ mit $f = O(p)$ gibt,*
- f exponentiell (wachsend), *wenn es ein $\varepsilon > 0$ mit $f = \Omega(2^{n^\varepsilon})$ gibt und*
- f exponentiell klein, *wenn es eine exponentielle Funktion $f' : \mathbb{R} \mapsto \mathbb{R}^+$ mit $f = O(1/f')$ gibt.*

Die Menge der polynomiell beschränkten Funktionen wird mit poly(n) bezeichnet.

A.16 (Exponentielle Wahrscheinlichkeiten) *Gebe $f : \mathbb{R} \mapsto [0, 1]$ die Wahrscheinlichkeit eines Ereignisses an. Dann heißt f* exponentiell groß, *wenn es eine exponentielle Funktion $f' : \mathbb{R} \mapsto \mathbb{R}^+$ mit $f = \Omega(1 - 1/f')$ gibt.*

Literatur

D. Ackley (1987). *A Connectionist Machine for Genetic Hillclimbers*. Kluwer Academic, Boston, MA.

S. Arora, Y. Rabani und U. Vazirani (1994). Simulating quadratic dynamical systems is PSPACE-complete. In *Proceedings of the 26th Symposium on the Theory of Computing* (*STOC '94*), 459–467. ACM Press, New York, NY.

T. Bäck (1992). The interaction of mutation rate, selection, and self-adaptation with a genetic algorithm. In R. Männer und B. Manderick (Hrsg.), *Proceedings of the Second Conference on Parallel Problem Solving from Nature* (*PPSN '92*), 85–94. North-Holland, Amsterdam.

T. Bäck (1993). Optimal mutation rates in genetic search. In S. Forrest (Hrsg.), *Proceedings of the Sixth International Conference on Genetic Algorithms* (*ICGA '93*), 2–8. Morgan Kaufmann, San Francisco, CA.

T. Bäck (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, NY.

T. Bäck (1998). An overview of parameter control methods by self-adaptation in evolutionary algorithms. *Fundamenta Informaticae 34*, 51–66.

T. Bäck, D. B. Fogel und Z. Michalewicz (Hrsg.) (1997). *Handbook of Evolutionary Computation*. Institute of Physics Publishing and Oxford University Press, New York, NY.

T. Bäck, U. Hammel und H.-P. Schwefel (1997). Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation 1*(1), 3–17.

W. Banzhaf, P. Nordin, R. E. Keller und F. D. Francone (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA.

M. Barner und F. Flohr (1987). *Analysis I*. Walter de Gruyter, Berlin.

F. H. I. Bennett, J. R. Koza, J. Yu und W. Mydlowec (2000). Automatic synthesis, placement, and routing of an amplifier circuit by means of genetic programming. In J. Miller, A. Thompson, P. Thomson und T. Fogarty (Hrsg.), *Proceedings of the Third International Conference on Evolvable Systems* (*ICES 2000*), 1–10. Springer, Berlin. Lecture Notes in Computer Science 1801.

H.-G. Beyer und G. Rudolph (1997). Local performance measures. In T. Bäck, D. B. Fogel und Z. Michalewicz (Hrsg.), *Handbook of Evolutionary Computation*, B2.4:1–27. Institute of Physics Publishing and Oxford University Press, New York, NY.

A. Blumer, A. Ehrenfeucht, D. Haussler und M. K. Warmuth (1990). Occam's razor. In J. W. Shavlik und T. G. Dietterich (Hrsg.), *Readings in Machine Learning*, 201–204. Morgan Kaufmann, San Francisco, CA.

B. Bollig und I. Wegener (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers 45*, 993–1002.

R. E. Bryant (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers 35*, 677–691.

K. Chellapilla (1997). Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation 1*(3), 209–216.

K. Chellapilla (1998). A preliminary investigation into evolving modular programs without subtree crossover. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba und R. Riolo (Hrsg.), *Proceedings of the Third Genetic Programming Conference* (*GP '98*), 23–31. Morgan Kaufmann, San Francisco, CA.

J. C. Culberson (1998). On the futility of blind search: An algorithmic view of "No Free Lunch". *Evolutionary Computation 6*(2), 109–127.

S. Droste (1997). Efficient genetic programming for finding good generalizing boolean functions. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba und R. L. Riolo (Hrsg.), *Proceedings of the Second Genetic Programming Conference* (*GP '97*), 82–87. Morgan Kaufmann, San Francisco, CA.

S. Droste (1998). Genetic programming with guaranteed quality. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba und R. Riolo (Hrsg.), *Proceedings of the Third Genetic Programming Conference* (*GP '98*), 54–59. Morgan Kaufmann, San Francisco, CA.

S. Droste, T. Jansen und I. Wegener (1998a). On the optimization of unimodal functions with the $(1 + 1)$ evolutionary algorithm. In A. E. Eiben, T. Bäck, M. Schoenauer und H.-P. Schwefel (Hrsg.), *Proceedings of the Fifth Conference on Parallel Problem Solving from Nature* (*PPSN '98*), 13–22. Springer, Berlin. Lecture Notes in Computer Science 1498.

S. Droste, T. Jansen und I. Wegener (1998b). A rigorous complexity analysis of the $(1 + 1)$ evolutionary algorithm for linear functions with boolean inputs. In *Proceedings of the Third IEEE International Conference on Evolutionary Computation* (*ICEC '98*), 499–504. IEEE Press, Piscataway, NY.

S. Droste, T. Jansen und I. Wegener (1998c). A rigorous complexity analysis of the $(1+1)$ evolutionary algorithm for separable functions with boolean inputs. *Evolutionary Computation 6*(2), 185–196.

S. Droste, T. Jansen und I. Wegener (1999). Perhaps not a free lunch but at least a free appetizer. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela und R. E. Smith (Hrsg.), *Proceedings of the First Genetic and Evolutionary Computation Conference* (*GECCO '99*), 833–839. Morgan Kaufmann, San Francisco, CA.

S. Droste, T. Jansen und I. Wegener (2000a). Dynamic parameter control in simple evolutionary algorithms. In *Proceedings of the Sixth Foundations of Genetic Algorithms Workshop* (*FOGA 2000*). Im Druck.

S. Droste, T. Jansen und I. Wegener (2000b). A natural and simple function which is hard for all evolutionary algorithms. In *Proceedings of Third Asia-Pacific Conference on Simulated Evolution and Learning* (*SEAL 2000*), 2704–2709. IEEE Press, Piscataway, NY.

S. Droste, T. Jansen und I. Wegener (2000c). On the analysis of the $(1 + 1)$ evolutionary algorithm. Eingereicht bei Theoretical Computer Science.

S. Droste, T. Jansen und I. Wegener (2000d). Optimization with randomized search heuristics - the (A)NFL theorem, realistic scenarios, and difficult functions. Eingereicht bei Theoretical Computer Science.

S. Droste und D. Wiesmann (2000a). Metric based evolutionary algorithms. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin und T. C. Fogarty (Hrsg.), *Proceedings of the Third European Workshop on Genetic Programming* (*EuroGP 2000*), 29–43. Springer, Berlin. Lecture Notes in Computer Science 1802.

S. Droste und D. Wiesmann (2000b). On the design of problem-specific evolutionary algorithms. Eingereicht bei A. Ghosh und S. Tsutsui (Hrsg.): Theory and Application of Evolutionary Computation : Recent Trends.

H. Ehrenburg (1996). Improved directed acyclic graph evaluation and the combine operator in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel und R. L. Riolo (Hrsg.), *Proceedings of the First Genetic Programming Conference* (*GP '96*), 285–290. MIT Press, Cambridge, MA.

A. E. Eiben und G. Rudolph (1999). Theory of evolutionary algorithms: A bird's eye view. *Theoretical Computer Science 229*(1), 3–9.

W. Feller (1971). *An Introduction to Probability Theory and its Applications.* John Wiley & Sons, New York, NY.

D. B. Fogel (1995). *Evolutionary Computation.* IEEE Press, Piscataway, NY.

S. Forrest und M. Mitchell (1993). Relative building-block fitness and the building block hypothesis. In D. L. Whitley (Hrsg.), *Proceedings of the Second Workshop on Foundations of Genetic Algorithms* (*FOGA '93*), 109–126. Morgan Kaufmann, San Francisco, CA.

M. Garey und D. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, NY.

J. Garnier, L. Kallel und M. Schoenauer (1999). Rigorous hitting times for binary mutations. *Evolutionary Computation 7(2)*, 167–203.

F. Glover und M. Laguna (1998). *Tabu Search.* Kluwer Academic, Boston, MA.

D. E. Goldberg (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA.

T. Hagerup und C. Rüb (1990). A guided tour of Chernoff bounds. *Information Processing Letters 33*(6), 305–308.

S. Handley (1994). On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the First IEEE International Conference on Evolutionary Computation* (*ICEC '94*), 154–159. IEEE Press, Piscataway, NY.

W. E. Hart und R. K. Belew (1991). Optimizing an arbitrary function is hard for the genetic algorithm. In R. K. Belew und L. B. Booker (Hrsg.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (*ICGA '91*), 190–195. Morgan Kaufmann, San Francisco, CA.

J. H. Holland (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI.

J. H. Holland (1992). *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA.

J. Horn, D. Goldberg und K. Deb (1994). Long path problems. In Y. Davidor, H.-P. Schwefel und R. Männer (Hrsg.), *Proceedings of the Third Parallel Problem Solving from Nature* (*PPSN '94*), 149–158. Springer, Berlin. Lecture Notes in Computer Science 866.

T. Jansen (2000). On classifications of fitness functions. In L. Kallel, B. Naudts und A. Rogers (Hrsg.), *Theoretical Aspects of Evolutionary Computing*. Springer, Berlin. Im Druck.

T. Jansen und I. Wegener (1999). On the analysis of evolutionary algorithms – a proof that crossover really can help. In J. Nešetřil (Hrsg.), *Proceedings of the Seventh Annual European Symposium on Algorithms* (*ESA '99*), 184–193. Springer, Berlin. Lecture Notes in Computer Science 1643.

T. Jansen und I. Wegener (2000a). Evolutionary algorithms - how to cope with plateaus of constant fitness and when to reject strings of the same fitness. Eingereicht bei IEEE Transactions on Evolutionary Computation.

T. Jansen und I. Wegener (2000b). On the choice of the mutation probability for the (1+1) EA. In M. Schoenauer (Hrsg.), *Proceedings of the Sixth Conference on Parallel Problem Solving from Nature* (*PPSN 2000*), 89–98. Springer, Berlin.

M. Jerrum und A. Sinclair (1997). The Markov chain Monte Carlo method: an approach to approximate counting an integration. In D. S. Hochbaum (Hrsg.), *Approximation Algorithms for NP-hard Problems*, 482–520. PWS Publishers.

M. Jerrum und G. Sorkin (1998). The Metropolis algorithm for graph bisection. *Discrete Applied Mathematics 82*, 155–175.

J. R. Koza (1992). *Genetic Programming*. MIT Press, Cambridge, MA.

J. R. Koza (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

U. Krengel (1991). *Einführung in die Wahrscheinlichkeitstheorie und Statistik*. Vieweg, Braunschweig.

M. Li und P. Vitányi (1993). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, Berlin.

S. Luke und L. Spector (1998). A revised comparison of crossover and mutation in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba und R. Riolo (Hrsg.), *Proceedings of the Third Genetic Programming Conference* (*GP '98*), 208–213. Morgan Kaufmann, San Francisco, CA.

K. Mehlhorn und S. Näher (1999). *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge.

R. Menke (1997). A revision of the schema theorem. Reihe CI 14/97 des SFB 531. Universität Dortmund.

N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller und E. Teller (1953). Equation of state calculation by fast computing machines. *Journal of Chemical Physics 21*, 1087–1092.

M. Mitchell, S. Forrest und J. H. Holland (1992). The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Proceedings of the First European Conference on Artificial Life*, 245–254. MIT Press, Cambridge, MA.

R. Motwani und P. Raghavan (1995). *Randomized Algorithms*. Cambridge University Press, Cambridge.

H. Mühlenbein (1992). How genetic algorithms really work: I. mutation and hillclimbing. In H.-P. Schwefel, R. Männer und B. Manderick (Hrsg.), *Proceedings of the Second Conference on Parallel Problem Solving from Nature* (*PPSN '92*), 15–26. Springer, Berlin.

A. I. Oyman, H.-G. Beyer und H.-P. Schwefel (1998). Where elitists start limping evolution strategies at ridge functions. In A. E. Eiben, T. Bäck, M. Schoenauer und H.-P. Schwefel (Hrsg.), *Proceedings of the Fifth Conference on Parallel Problem Solving from Nature* (*PPSN '98*), 34–43. Springer, Berlin. Lecture Notes in Computer Science 1498.

C. H. Papadimitriou (1994). *Computational Complexity*. Addison-Wesley, Reading, MA.

R. Poli und W. B. Langdon (1998). Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation 6*(3), 231–252.

R. Poli, J. Page und W. B. Langdon (1999). Smooth uniform crossover, submachine code GP and demes: A recipe for solving high-order boolean parity problems. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela und R. E. Smith (Hrsg.), *Proceedings of the First Genetic and Evolutionary Computation Conference* (*GECCO '99*), 1162–1169. Morgan Kaufmann, San Francisco, CA.

Y. Rabani, Y. Rabinovich und A. Sinclair (1995). A computational view of population genetics. In *Proceedings of the 27th ACM Symposium on the Theory of Computing* (*STOC '95*), 83–92. ACM, New York, NY.

Y. Rabinovich, A. Sinclair und A. Wigderson (1992). Quadratic dynamical systems. In *Proceedings of the 33rd Symposium on Foundations of Computer Science* (*FOCS '92*), 304–313. IEEE Press, Piscataway, NY.

Y. Rabinovich und A. Wigderson (1991). An analysis of a simple genetic algorithm. In L. B. Belew und R. K. Booker (Hrsg.), *Proceedings of the Fourth International Conference on Genetic Algorithms* (*ICGA '91*), 215–221. Morgan Kaufmann, San Francisco, CA.

I. Rechenberg (1994). *Evolutionsstrategie '94*. Frommann-Holzbog, Stuttgart.

G. Rudolph (1997). *Convergence Properties of Evolutionary Algorithms*. Verlag Dr. Kovač, Hamburg.

G. Rudolph (1998). Finite markov chain results in evolutionary computation: A tour d'horizon. *Fundamenta Informaticae 35*, 67–89.

H. Sakanashi, T. Higuchi, H. Iba und Y. Kakazu (1996). An approach for genetic synthesizer of binary decision diagram. In *Proceedings of the Second IEEE International Conference on Evolutionary Computation* (*ICEC '96*), 559–564. IEEE Press, Piscataway, NY.

R. Salomon (1996). Reevaluating genetic algorithm performance under coordinate rotation of benchmark functions. *BioSystems 39*(3), 263–278.

U. Schöning (1999). A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science* (*FOCS '99*), 410–414. IEEE Press, Piscataway, NY.

H.-P. Schwefel (1995). *Evolution and Optimum Seeking*. John Wiley & Sons, Inc., New York, NY.

J. L. Shapiro und A. Prügel-Bennett (1995). Maximum entropy analysis of genetic algorithm operators. In *Evolutionary Computing: AISB Workshop*, 14–24. Springer, Berlin. Lecture Notes in Computer Science 993.

D. Sieling (1998). The nonapproximability of OBDD minimization. Eingereicht bei Information and Computation.

A. Sinclair (1993). *Algorithms for Random Generation & Counting*. Birkhäuser, Boston, MA.

T. Slawinski, A. Krone, U. Hammel, D. Wiesmann und P. Krause (1999). A hybrid evolutionary search concept for data-based generation of relevant fuzzy rules in high dimensional spaces. In *Proceedings of the Eighth International Conference on Fuzzy Systems* (*FUZZ-IEEE '99*), 1432–1437. IEEE Press, Piscataway, NJ.

G. Sorkin (1991). Efficient simulated annealing on fractal energy landscapes. *Algorithmica 6*, 367–418.

G. Tao und Z. Michalewicz (1998). Inver-over operator for the TSP. In A. E. Eiben, T. Bäck, M. Schoenauer und H.-P. Schwefel (Hrsg.), *Proceedings of the Fifth Conference on Parallel Problem Solving from Nature* (*PPSN '98*), 803–812. Springer, Berlin. Lecture Notes in Computer Science 1498.

P. van Laarhoven und E. Aarts (1987). *Simulated Annealing: Theory and Practice.* Kluwer Academic, Boston, MA.

M. D. Vose (1999). *The Simple Genetic Algorithm.* MIT Press, Cambridge, MA.

I. Wegener (2000). *Branching Programs and Binary Decision Diagrams - Theory and Applications.* SIAM. (Im Druck).

I. Wegener und C. Witt (2000). On the behaviour of the (1+1) evolutionary algorithm on quadratic pseudo-boolean functions. Eingereicht bei Evolutionary Computation.

D. H. Wolpert und W. G. Macready (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation 1*(1), 67–82.

M. Yanagiya (1994). Efficient genetic programming based on binary decision diagrams. In *Proceedings of the First International IEEE Conference on Evolutionary Computation* (*ICEC '94*), 234–239. IEEE Press, Piscataway, NY.

B.-T. Zhang und J.-G. Joung (1997). Enhancing robustness of genetic programming at the species level. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba und R. L. Riolo (Hrsg.), *Proceedings of the Second Genetic Programming Conference* (*GP '97*), 336–342. Morgan Kaufmann, San Francisco, CA.

B.-T. Zhang und J.-G. Joung (1999). Genetic programming with incremental data inheritance. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela und R. E. Smith (Hrsg.), *Proceedings of the First Genetic and Evolutionary Computation Conference* (*GECCO '99*), 1217–1224. Morgan Kaufmann, San Francisco, CA.

B.-T. Zhang und H. Mühlenbein (1995). Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation 3*(1), 17–38.